ABSTRACT

The task of formulating queries is greatly facilitated when they can be generated automatically from some given data values, schema concepts or both (e.g., names of particular entities and XML tags). This automation is the basis of various database applications, such as keyword search and interactive query formulation. Usually, automatic query generation is realized by finding a set of small tree patterns that contain some given labels. More formally, the computational problem at hand is to find top-$k$ patterns, that is, $k$ minimum-weight tree patterns that contain a given bag of labels, conform to the schema, and are non-redundant. A plethora of systems and research papers include a component that deals with this problem. This paper presents an algorithm for this problem, with complexity guarantees, that allows nontrivial schema constraints and, hence, avoids generating patterns that cannot be instantiated. Specifically, this paper shows that for schemas with certain types of neighborhood constraints, the problem is fixed-parameter tractable (FPT), the parameter being the size of the given bag of labels. As machinery, an adaptation of Lawler-Murty’s procedure is developed. This adaptation reduces a top-$k$ problem, over an infinite space of solutions, to a prefix-constrained optimization problem. It is shown how to cast the problem of top-$k$ patterns in this adaptation. A solution is developed for the corresponding prefix-constrained optimization problem, and it uses an algorithm for finding a (single) minimum-weight tree pattern. This algorithm generalizes an earlier work by handling leaf constraints (i.e., which labels may, must or should not be leaves). It all boils down to a reduction showing that, under a language for neighborhood constraints, finding top-$k$ patterns is FPT if a certain variant of exact cover is FPT.

1. INTRODUCTION

Manual query formulation can be highly laborious and time consuming when users lack close familiarity with the schema. SAP schemas, for instance, may have thousands of relations [10]. Various paradigms tackle this problem. Some of them, like tools for schema exploration and semi-automatic query formulation [14, 24], are aimed at facilitating the task of fully specifying a query in a language such as SQL. Others, such as schema-free (or flexible) queries [3, 16], and keyword search over structured data [2, 9, 11, 21], are aimed at supporting under-specified queries. Within the realization of these paradigms, a standard component is an algorithm that extracts from the schema small tree patterns that connect a given set of items (such as the name of a relation, the value of an attribute, an XML tag, an XML CDATA value, etc.).

Take, for example, keyword search over structured data. 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 that cover all the 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 at hand is to find these small subtrees. In the graph approach (e.g., [2, 5, 11]), subtrees are extracted directly from the data while ignoring the schema. The pattern approach [9, 17, 21, 24], which is the one relevant to this paper, deploys two steps. First, a set of candidate patterns is generated; then, search results are obtained by evaluating the patterns as queries over the data graph (e.g., finding subtrees that are isomorphic to those patterns).

Efficiency in producing a set of candidate patterns is crucial, as these 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), with the exception of the work by Talukdar et al. [21] that we discuss in the next paragraph. It is not obvious that these algorithms scale up to the sizes of real-world schemas. Most of the algorithms (e.g., [17, 18, 20, 24]) follow DISCOVER [9] 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. This generation of the complete patterns is grossly inefficient, as the incremental construction can blow up 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 of the database like elementary one-to-many and many-to-one relationships. More complex constraints are imposed by XML schemas, where regular expressions qualify the possible sets of sub-elements.

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 patterns, graph search

*The work of this author was supported by the Israel Science Foundation (Grant No. 1632/12).
The only algorithm that generates patterns with a nontrivial guarantee on the running time is that of Talukdar et al. [21]. Their algorithm has been applied to a schema of around 400 relations taken from bioinformatic data sources. Their experiments show that generating the patterns can be a bottleneck in an online scenario (e.g., more than 10 seconds for 5 patterns). Their algorithm views the schema as a graph of labels, and applies a top-k Steiner-tree algorithm to extract small subtrees of the graph, which are then used as patterns. As we illustrated in [13], this approach has two significant (and inherent) drawbacks. First, it does not allow patterns with repeated labels (although, conceivably, this problem could be solved by augmenting the graph with label repetitions). Second, it does not consider basic constraints on the graph (as those formerly mentioned), which may require generating a large number of patterns to produce just a few useful ones.

To sum up thus far, a plethora of systems and research papers include a component that extracts, from a given schema, small tree patterns that connect a given set of labels. An algorithm for effectively realizing such a component should avoid useless patterns that violate basic database constraints, be able to handle repeated labels, and be efficient. Quite surprisingly, to the best of our knowledge there has not been any algorithm with complexity guarantees that can either handle nontrivial schema constraints or support repeated labels. In this paper, we present such an algorithm.

In a previous paper [13], we introduced schemas for data graphs that allow to express neighborhood constraints. For illustration, consider the graph at the top of Figure 1 that specifies (by edges) which pairs of labels can be neighbors. A neighborhood constraint can state that an emp node (representing an employee) can have either a regular or an exec neighbor (representing the type of the employee), but not both; and a manager neighbor is allowed only if an exec neighbor exists. Formally, a constraint maps a label to a (possibly infinite) set comprising finite bags of labels. A schema \( S \) is a collection of constraints \( S(\sigma) \), one for each label \( \sigma \), and it also specifies weights on the nodes and edges. For a node \( v \) of a pattern, where \( v \) is labeled with \( \sigma \), the neighboring labels of \( v \) (including their multiplicities) must form a subbag of a bag in \( S(\sigma) \).

In [13], we investigated the complexity of languages for specifying neighborhood constraints. We also developed an algorithm for MINPATTERN, which is the following problem. Given a schema \( S \) and a bag \( \Lambda \) of labels (derived from a user’s query), find a minimum-weight \((S, \Lambda)\)-pattern—a tree \( t \) such that the bag of labels in \( t \) forms a superbag of \( \Lambda \), and each of \( t \) ’s nodes satisfies the constraint for its label. Of course, in all practical implementations mentioned earlier, the need is for multiple (i.e., \( k \)) minimum-weight \((S, \Lambda)\)-patterns, rather than just one. In this paper, we consider TOPPATTERNS, which is the problem of generating \( k \) minimum-weight \((S, \Lambda)\)-patterns, unlike the case of a (single) minimum-weight \((S, \Lambda)\)-pattern, it is not even clear how to define the meaning of “\( k \) minimum-weight \((S, \Lambda)\)-patterns.” We proposed a definition in [13], but the complexity of TOPPATTERNS was left as an open question. The current paper resolves this question by presenting an algorithm for TOPPATTERNS.

The complexity of the problem TOPPATTERNS depends on the language used for expressing the neighborhood constraints. However, even for the simplest conceivable languages (e.g., each \( S(\sigma) \) is a finite set that is specified explicitly), and even if we want just one pattern (rather than \( k > 1 \)), TOPPATTERNS is NP-hard [13]. But this hardness assumes that the given bag \( \Lambda \) of labels is large, whereas in practice the size of \( \Lambda \) is similar to that of the user’s query. In turn, a user’s query is typically very small, and in particular, significantly smaller than the schema. Hence, we view the size of \( \Lambda \) as a parameter, and focus on parameterized complexity [4, 7]. A problem is deemed tractable if it is Fixed-Parameter Tractable (FPT), which basically means that the running time is polynomial, except that the parameter can arbitrarily affect the constant of the polynomial (but not on the degree). We showed [13] that MINPATTERN is FPT when neighborhood constraints are regular expressions or “circular-arc graphs” (which we do not define in this paper). In this paper we show the same for TOPPATTERNS. In particular, the main contribution is the first provably efficient algorithm for extracting top-\( k \) tree queries from a schema with nontrivial expressiveness (e.g., neighborhood constraints as regular expressions). Next, we review the algorithm and the involved challenges.

The produced top-\( k \) minimum-weight \((S, \Lambda)\)-patterns should be non-redundant. Otherwise, we may get the second-best pattern \( t_2 \) just by adding some irrelevant part to a minimum-weight \((S, \Lambda)\)-pattern \( t_1 \). Applying \( t_2 \) to a data graph can only yield relevant answers that are also produced by \( t_1 \); hence, \( t_2 \) is useless.) Out of all the \((S, \Lambda)\)-patterns, the non-redundant ones are just a small fraction. Thus, an effective algorithm should avoid redundant patterns. In our case, non-redundancy is not merely the absence of a proper subtrie that is also an \((S, \Lambda)\)-pattern (which is a conventional definition for top-\( k \) Steiner trees [5, 11]), because answering some queries may involve a sequence of joins in which the same entity appears several times (e.g., “find the exec and the regular employees that report to the same exec”). In Section 2, we give the precise definition of non-redundancy.

Thus, the task is to find the top-\( k \) minimal-weight \((S, \Lambda)\)-patterns among the non-redundant ones; moreover, no two of the top-\( k \) can be isomorphic to each other. There does not seem to be any obvious general way of tackling this optimization problem (in particular, when the goal is an FPT algorithm). We are aware of very few general top-\( k \) techniques that are capable of enforcing nontrivial constraints. Our candidate is the procedure of Lawler [15] and Murty [19] that formulates the top-\( k \) problem in terms of finding \( k \) least-cost assignments for a given set of Boolean variables (or equivalently \( k \) least-cost subsets of a given set). However, there is no clear way of formulating TOPPATTERNS under that abstraction, because the patterns come from a space that is conceptually infinite; they are built from nodes that are created on the fly (rather than nodes of a given graph, as for example in [5, 11]), and they could be larger than the input (even if the bag \( \Lambda \) is small).

In Section 4, we adapt Lawler-Murty’s procedure and complexity result to a new abstraction, where the space of solutions (the \( k \) least cost of which are desired) is the infinite set of all finite strings over a finite alphabet. To apply this abstraction to TOPPATTERNS, we begin with the assumption that \( \Lambda \) is a set (rather than a general bag). In Section 5, we devise a serialization of a non-redundant \((S, \Lambda)\)-pattern into a string over a finite alphabet. This serialization acts as canonization, in the sense that isomorphism of patterns is equivalent to the equality of their corresponding strings. With serialization at hand, the adapted Lawler-Murty’s procedure reduces TOPPATTERNS to prefix optimization, that is, the problem of finding a minimum-weight, non-redundant \((S, \Lambda)\)-pattern, such that its serialization contains a given string as a prefix. In Section 6, we further reduce prefix optimization into a variant of MINPATTERN with leaf constraints that specify which labels may, must or should not be leaves. In Section 7, we adapt the algorithm for MINPATTERN [13].

---

1This algorithm is based on that of [11]. The latter was originally designed for the graph approach mentioned earlier.

2This schema language is essentially the core of DTDs, except that we do not require one-unambiguity.

3Lawler [15] generalized Yen’s algorithm [23] for finding \( k \)-shortest simple paths in a graph. Another general technique is that of Hamacher and Queyranne [8], which is very similar to that of Lawler-Murty.
to handle leaf constraints. This sequence of adaptations and reductions is our algorithm for TopPatterns. In Section 8, we discuss how to generalize the algorithm (and complexity result) to the case where \( \Lambda \) is a general bag.

Our algorithm for TopPatterns is generic in the sense that it does not depend on a specific language of neighbor constraints. Instead, the algorithm is a reduction of TopPatterns to a generalization of the exact-cover problem, denoted by MinLBE\( X \) and defined in Section 2.8. Our algorithm is FPT if there is an FPT solver of MinLBE\( X \) for the given language of neighborhood constraints. By adapting known results [13], it follows that our algorithm is FPT for the aforementioned constraint languages (regular expressions and circular-arc max graphs). It should be noted that some subtle changes in the latter cause MinPattern to become \( W[1]\)-hard [13], thereby implying that an FPT algorithm for TopPatterns is unlikely to exist [7].

2. FORMAL SETTING

In this section, we describe the formal setting of this work and give some basic notation.

2.1 Bag Notation

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). 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 \( \mu_b(1) = 2 \).

We use \( \cup \) to denote bag union. Containment of bags takes multiplicities into account. Thus, 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 \( x \in X \).

For a set \( B \) of bags, we define the containment closure of \( B \), denoted by \( \langle B \rangle \), as the set of bags

\[
\langle B \rangle \overset{\text{def}}{=} \bigcup_{b \in B} \{b' \mid b' \subseteq b\} ,
\]

that is, \( \langle B \rangle \) comprises all the bags of \( B \) and their subbags.

2.2 Labels and Regular Expressions

We fix an infinite set \( \Sigma \) of labels. The set of all the finite bags of labels is denoted by \( \mathcal{F}_\Sigma \). We assume that there is a total order \( \prec \) on \( \Sigma \). To simplify our complexity analysis, we further assume that \( \sigma_1 \prec \sigma_2 \) can be tested in constant time (given \( \sigma_1 \) and \( \sigma_2 \)).

Regular expressions over \( \Sigma \) are defined by the language

\[
e := \sigma \mid e \cdot e \mid e? \mid ee \mid e \lor e
\]

where \( \sigma \in \Sigma \) and \( e \) is the empty string. A regular expression \( e \) defines the language \( L(e) \), which is the set of all the strings over \( \Sigma \) that match the expression \( e \). We define the bag language \( L^*(e) \) of \( e \) in the standard way (e.g., as in [1]): \( L^*(e) \) is the set of all the bags \( b \in \mathcal{F}_\Sigma \) such that the elements of \( b \) can be ordered to form a word in \( L(e) \) (in other words, for some string \( x \in L(e) \), every label has the same multiplicity in both \( x \) and \( b \)).

2.3 Graphs, Schemas and Patterns

We consider undirected graphs with labeled nodes. Section 8 extends our results to the directed type of problem we study. For a graph \( g \), the sets of nodes and edges are denoted by \( V(g) \) and \( E(g) \), respectively. Note that an edge of \( E(g) \) is a set \((u, v) \subseteq V(g) \) where \( u \neq v \). For a node \( v \in V(g) \), the label of \( v \) is denoted by \( \lambda^0(v) \). If \( U \) is a subset of \( V(g) \), then \( \lambda^0(U) \) denotes the bag that is obtained from \( U \) by replacing each node \( u \) with its label \( \lambda^0(u) \) (that is, the multiplicity of each label \( \sigma \) is \(|u \in U \mid \lambda^0(u) = \sigma| \)). Note that \( \lambda^0(v) \in \Sigma \) and \( \lambda^0(U) \in \mathcal{F}_\Sigma \). For a node \( v \), the set of neighbors of \( v \) is denoted by \( nbr^0(v) \). Usually, the graph \( g \) is clear from the context, and then we may write just \( \lambda(v) \), \( \lambda(U) \) and \( nbr(v) \) instead of \( \lambda^0(v) \), \( \lambda^0(U) \) and \( nbr^0(v) \), respectively.

A tree is a connected and acyclic graph. A leaf is a node that is incident to at most one edge. A non-leaf node is internal. We use leaves \( t \) and internal \( t \) to denote the set of all the leaves and the set of all the internal nodes, respectively, of the tree \( t \).

Graphs represent data and the valid ones are those that satisfy the constraints imposed by the schema. We use lb-constraints ("lb" stands for "label bags"), which are a specific type of neighborhood constraints. Formally, an lb-constraint is a (possibly infinite) subset of \( \mathcal{F}_\Sigma \). A schema (or graph schema) \( S \) is a mapping over a finite set of labels, denoted by \( dom(S) \). The schema \( S \) 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 \( v \in V(g) \) it holds that \( \lambda(v) \in dom(S) \) and \( \lambda(nbr(v)) \in S(\lambda(v)) \). That is, the bag of labels appearing in the neighbors of \( v \) is an element of the lb-constraint to which the label of \( v \) is mapped.

For complexity analysis, we assume that an lb-constraint is represented by an lb-specification, which is a finite string in the language of the computational model at hand. In [13], several languages of lb-specifications are discussed. In one of them, a regular expression \( e \) specifies the lb-constraint \( L^*(e) \); this is the language we use in our examples throughout the paper.

A tree is called a pattern (a.k.a. twig pattern) when it denotes an expression that is evaluated over a graph. Next, we discuss how to define validity of patterns. Suppose, for example, that a schema \( S \) maps the label \( \sigma \) to the lb-constraint \( \{\tau_1, \tau_2\} \). Thus, if a graph \( g \) conforms to \( S \), then a node labeled with \( \sigma \) must have either two or three neighbors, such that exactly one is labeled with \( \tau_1 \) and the others—with \( \tau_2 \). However, a query may refer to only some neighbors; for example, "find edges, such that one end-point is labeled with \( \sigma \) and the other—with \( \tau_2 \)." We can express this query by the pattern consisting of \( v_1 \) and \( v_2 \) connected by an edge and labeled with \( \sigma \) and \( \tau_2 \) respectively. This pattern does not conform to the schema \( S \), because the node labeled with \( \sigma \) has only one neighbor. However, as an expression over a graph that conforms to \( S \), this pattern does not violate the lb-constraint \( S(\sigma) \), because the bag comprising the labels of the neighbors of \( v_1 \) is contained in some bag of \( S(\sigma) \). This example motivates the definition of an S-pattern given below.

Let \( S \) be a schema. 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, let \( \Lambda \in \mathcal{F}_\Sigma \) be a finite bag of labels, and let \( t \) be a tree. We say that \( t \) is an S-pattern if \( t \models [S] \), a \( \Lambda \)-pattern if \( \Lambda \subseteq \lambda(V(t)) \), and an \( (S, \Lambda) \)-pattern if \( t \) is both an \( S \)-pattern and a \( \Lambda \)-pattern.

**Example 2.2.** Let \( s' \) be the graph at the top of Figure 1. We define the schema \( S' \) as follows. The labels of \( dom(S') \) are the eight nodes of \( s' \) (i.e., \( \mathit{exec \_ regular} \), etc.). For all \( \sigma \in dom(S') \), the lb-constraint \( S'(\sigma) \) comprises all the finite bags over the neighbors of \( \sigma \) in the graph \( s' \). For example, \( S'(\mathit{dept}) \) is the set that consists of every bag that consists of \( \mathit{manages} \), \( \mathit{worksIn} \) and \( \mathit{memberOf} \), each appearing zero or more times. Let \( \Lambda \) be the bag at the bottom of Figure 1, namely \( \{\mathit{exec \_ project \_ regular}\} \). Every pattern in Figure 2 is an \( (S', \Lambda) \)-pattern (that describes a relationship involving a regular employee, an executive, and a project).
Let $S$ be a schema with the same domain as $S'$, but now assume that the lb-specifications are given by the regular expressions in the middle of Figure 1. Hence, $S(\sigma)$ is $L^0(\sigma)$, where $\sigma$ is the regular expression assigned to $\sigma$ in the middle of Figure 1. For example, $S(\text{emp}) = L^0(\text{emp})$, where $e$ is the expression $\text{regular | (exec manages)}$. For each $\Lambda$-pattern $\Lambda$, we define the regular expression $\text{regular | (exec manages)}$. In particular, note that none of the bags of $S(\text{emp})$ contain both regular and exec, and hence, $\Lambda$ is not an $S$-pattern. Similarly, none of the bags of $S(\text{emp})$ contain both regular and manages, and therefore, $\Lambda$ is not an $S$-pattern as well. On the other hand, the reader can verify that each $\Lambda$ is an $(S, \Lambda)$-pattern.

### 2.4 Non-Redundant Patterns

To define a non-redundant pattern, we fix a label $\star \in \Sigma$ that is assumed to be never used in the schema $S$ and the bag $\Lambda$ of labels. An output designation of a $\Lambda$-pattern $\Lambda$ is a tree $t'$ that is obtained as follows. We choose $|\Lambda|$ nodes of $t$ whose labels form the same bag as $\Lambda$ and designate them as the output; we then replace the label of every non-output node with $\star$. A $\Lambda$-pattern is basic if it has no proper subtree that is also a $\Lambda$-pattern (i.e., removing any leaf results in a non-$\Lambda$-pattern). A $\Lambda$-pattern is non-redundant if some output designation converts it into a basic $\Lambda$-pattern.\footnote{In [13], basic and non-redundant $\Lambda$-patterns are called non-redundant and weakly non-redundant, respectively.}

**Example 2.3**. Consider again the $\Lambda$-patterns $t_1, \ldots, t_4$ of Figure 2. Each of $t_1, t_2$ and $t_4$ is basic and, hence, non-redundant. The pattern $t_3$ is not basic, because if we remove the project node at the top left (which is a leaf), the result is still a $\Lambda$-pattern. However, $t_3$ is non-redundant, because it becomes basic if the output designation includes the left project node. Note that if we choose the right project node for the output, then $t_3$ does not become basic. Given the bag $\Lambda = \{\text{exec, regular}\}$, only $t_1$ is a non-redundant (and also basic) $\Lambda$-pattern, while each of $t_2, t_3$ and $t_4$ is a redundant $\Lambda$-pattern.

An easy observation is that a $\Lambda$-pattern $t$ is non-redundant if and only if no label has a higher multiplicity among the leaves of $t$ than in $\Lambda$. For later use, we record this observation as a proposition.

**Proposition 2.4.** Let $\Lambda$ be a bag of labels. A tree $t$ is a non-redundant $\Lambda$-pattern if and only if $\lambda(\text{leaves}(t)) \subseteq \Lambda \subseteq \lambda(\text{V}(t))$.

### 2.5 Weights

When extracting patterns from schemas, some labels may be preferred over others, and some relationships could be viewed as stronger than others. To accommodate that, a schema may assign weights to labels and edges. Formally, a weight function for a schema $S$ is a mapping $W : \text{dom}(S) \cup (\text{dom}(S) \times \text{dom}(S)) \rightarrow [0, \infty]$ that is symmetric over $\text{dom}(S) \times \text{dom}(S)$. By symmetric we mean that $W(\sigma, \tau) = W(\tau, \sigma)$ for all $\sigma$ and $\tau$ in $\text{dom}(S)$.

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

$$W(g) \equiv \sum_{v \in \text{V}(g)} W(\lambda(v)) + \sum_{(u,v) \in \text{E}(g)} W(\lambda(u), \lambda(v)).$$

A weighted schema comprises a schema $S$ and a weight function $W$ for $S$, and is denoted by $S^W$. However, when $W$ is irrelevant, we may omit it from $S^W$. In particular, “$S$-pattern” and “$S^W$-pattern” have the same meaning.

**Example 2.5.** Consider again the schema $S$ of Figure 1. The figure also defines a weight function $W$, so the figure actually defines a weighted schema $S^W$. Let $W'$ be the same as $W$, except that $W'(\text{project, pMember}) = 2$ and $W'(\text{pMember, project}) = 2$. Now, consider the trees $t_1, t_2$ and $t_3$ of Figure 2. The reader can easily verify that $W(t_1) = 4 + 1 + 2 = 7, W(t_2) = 1 + 2 + 2 = 5, W(t_3) = 4 + 2 + 3 + 2 = 10$.

In [13], we studied the problem $\text{MINPATTERN}$, namely, finding a minimal pattern.

**Problem 2.6.** The problem $\text{MINPATTERN}$ accepts as input a weighted schema $S^W$ and a bag $\Lambda \in \mathcal{F}_\Sigma$. The goal is to find a minimal $(S, \Lambda)$-pattern.

### 2.6 Top-k Patterns

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

1. No two distinct members of $T$ are isomorphic.
2. Each $(S, \Lambda)$-pattern in $T$ is non-redundant.
3. For all non-redundant $(S, \Lambda)$-patterns $t'$, either $t'$ is isomorphic to some $t \in T$, or $W(t') \geq W(t)$ for all $t \in T$. 

**Figure 1:** Schema $S$ (given by regular expressions) and bag $\Lambda$

**Figure 2:** Non-redundant $\Lambda$-patterns $t_i$ under $S$ (Figure 1), and $t_i'$ under $S'$ (Example 2.2), for $\Lambda = \{\text{exec, project, regular}\}$
If there are only \( k' < k \) non-redundant and pairwise non-isomorphic \((S, \Lambda)\)-patterns, then a top-\( k \)-set is defined to be a top-\( k' \)-set.

**Example 2.7.** We continue with our running example, where the schema \( S^W \) and the bag \( \Lambda \) are those of Figure 1. Now, consider the patterns \( t_1 \) in Figure 2. The reader can verify that \( t_1 \) is the minimal \((S, \Lambda)\)-pattern. Hence, \( \{t_1\} \) is the unique top-1 set. Similarly, the second-minimal \((S, \Lambda)\)-pattern is \( t_2 \), and hence, \( \{t_1, t_2\} \) is a top-2 set. Finally, each of \( t_2 \) and \( t_4 \) is a third-minimal \((S, \Lambda)\)-pattern, and therefore both \( \{t_1, t_2, t_3\} \) and \( \{t_1, t_3, t_4\} \) are top-3 sets. Of course, if we used some other weight function \( W \), then these top-\( k \)-sets could be different. □

Our main contribution is an algorithm for TopPatterns, which generalizes MinPattern (finding a minimum-weight \((S, \Lambda)\)-pattern) to the problem of finding a top-\( k \)-set of \((S, \Lambda)\)-patterns.

**Problem 2.8.** The problem TopPatterns accepts as input a weighted schema \( S^W \), a bag \( \Lambda \in \mathcal{F}_S \) and a natural number \( k \). The goal is to find a top-\( k \)-set of \((S, \Lambda)\)-patterns. □

### 2.7 Complexity Measure

The complexity of TopPatterns (MinPattern being a special case) depends on the language for lb-specifications in use. However, even for extremely simple languages, finding a minimum-weight \((S, \Lambda)\)-pattern is already NP-hard [13]. Nevertheless, in practical scenarios the input bag \( \Lambda \) is typically very small, because it corresponds to a user query. Thus, our main yardstick for efficiency is fixed-parameter tractability [4, 7], 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 \( f(m) \) is a computable function (e.g., \( 2^m \)), \( p(m) \) is a polynomial, and \( |S^W| \) is the size of the representation of \( S^W \). (Usually, we do not formally define \( |S^W| \), but rather assume a simple encoding.) A weaker yardstick for efficiency, in the spirit of data complexity [22], is polynomial time under the assumption that \( |\Lambda| \) is fixed.

For simplicity of analysis, we assume that each arithmetic operation (which is always addition in this paper) has a unit cost.

### 2.8 Labeled-Bag Cover

In the remainder of the paper we give an FPT algorithm for TopPatterns. The algorithm is a reduction to a variant of the problem minimum labeled-bag cover [13]. This problem, which we denote by MinLBC, is the following. The input consists of a bag \( \Gamma \), an lb-constraint \( B \), and a set \( C \) of triples \( \Delta = (b, \tau, w) \), where \( b \) is a bag, \( \tau \in \Sigma \) is the label of \( \Delta \), and \( w \in [0, \infty) \) is the weight of \( \Delta \). A legal cover (of \( \Gamma \) by \( C \)) is a bag \( \{\Delta_1, \ldots, \Delta_n\} \), such that each \( \Delta_i \) is a triple \( (b_i, \tau_i, w_i) \) in \( C \), \( \Gamma \subseteq \bigcup_{i=1}^n b_i \) and \( \{\tau_1, \ldots, \tau_n\} \in [B] \); 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. The problem minimum labeled-bag exact cover, which we denote by MinLBE, is the same as MinLBC, except that the legal cover \( \{\Delta_1, \ldots, \Delta_n\} \) is now required to be such that \( \Gamma \) is equal to (and not just a subbag of) \( \bigcup_{i=1}^n b_i \).

In [13], it was shown that MinLBC, parameterized by \( |\Gamma| \), is FPT if lb-specifications are in the language of regular expressions. Essentially the same algorithm (with very minor changes) works for MinLBE. Hence, we have the following theorem.\(^3\)

**Theorem 2.9.** If lb-specifications are regular expressions, then MinLBE can be solved in time polynomial in the input size and \( 2^{|\Gamma|} \) (hence, MinLBE is FPT when parameterized by \( |\Gamma| \)).

\(^3\)The same result holds for the case where lb-specifications are given as circular-arc \textit{max} graphs [13].

### 3. MAIN RESULT

The main result of this paper is that TopPatterns is FPT whenever MinLBE is FPT.

**Theorem 3.1.** The following holds for any language of lb-specifications. If MinLBE (parameterized by \( |\Gamma| \)) is FPT, then TopPatterns (parameterized by \( |\Lambda| \)) is FPT in \( |S^W| \) and \( k \).

By combining Theorem 3.1 with Theorem 2.9, we get the following result.\(^5\)

**Corollary 3.2.** If lb-specifications are regular expressions, then TopPatterns (parameterized by \( |\Lambda| \)) is FPT in \( |S^W| \) and \( k \).

We now discuss the implication of Theorem 3.1 on the complexity of TopPatterns when \( \Lambda \) is of a fixed size (rather than a parameter). Let \( c \) be a natural number. The definition of c-bounded TopPatterns is the same as TopPatterns, except that we make the assumption that \( |\Lambda| \leq c \) (note that there is no restriction on the size of the schema \( S \)). Similarly, \( |\Gamma| \leq c \) in c-bounded MinLBE. For a language of lb-specifications, containment checking is the problem of testing whether \( b \in \mathcal{F}_S \) and an lb-constraint \( B \) represented by an lb-specification. The problem c-bounded containment checking is the same as containment checking, except that \( |b| \leq c \) is assumed. It is straightforward to show that a polynomial-time procedure for c-bounded containment checking is sufficient for efficiently solving c-bounded MinLBE. We get the following corollary of Theorem 3.1, relating c-bounded TopPatterns to c-bounded containment checking.

**Corollary 3.3.** Let \( c \) be a fixed natural number. For a language of lb-specifications, if c-bounded containment checking is in polynomial time, then c-bounded TopPatterns can be computed in polynomial time in \( |S^W| \) and \( k \).

In the following sections, we first assume that \( \Lambda \) is a set rather than a general bag, that is, the multiplicity of each label in \( \Lambda \) is 1. In Section 8, we show how to extend our results to a general \( \Lambda \).

### 4. ADAPTING LAWER-MURTY’S REDUCTION

Lawler-Murty’s reduction [15, 19] is a general technique for top-k problems (Lawler [15] generalized Yen’s algorithm [23] for finding \( k \)-shortest simple paths between two given nodes in a graph). Essentially, that technique efficiently reduces a top-k problem to a constrained optimization problem. The latter is the problem of finding the best (namely, top-1) solution under some constraints. In order to apply Lawler-Murty’s reduction to a specific top-k problem, we should provide an efficient algorithm for the constrained optimization problem.

In Yen’s algorithm [23] for \( k \)-shortest paths as well as in some other cases (e.g., [5, 11]), the top-k results can be viewed as substructures of the input, and so can the generated constraints. For example, in some applications [5, 11, 23] the input includes a graph, the results are paths or subtrees of that graph, and the constraints are inclusions and exclusions of edges. As another example, in Lawler’s generalization [15] the input contains Boolean variables, the results are (scored) truth assignments to these variables, and constraints are partial truth assignments. That, however, is no longer true when applying Lawler-Murty’s reduction to the problem of extracting the top-k patterns from a schema—there could be infinitely many patterns, and there is no bound on their size (in particular, a result could be larger than the input). In turn, those patterns give
rise to constraints of an unbounded size (and again, a constraint could be larger than the input). Therefore, we cannot (or at least it is not clear how to) represent and handle constraints in the same manner that Lawler-Murty’s reduction does. In the next section, we adapt Lawler-Murty’s reduction to the case of this paper.

4.1 Adaptation

We first describe an abstraction of top-\(k\) problems. Given an input \(x\), the top-\(k\) objects to be found are not parts of \(x\) (e.g., subtrees) or assignments to Boolean variables given in \(x\). Instead, each instance of the input is an encoding of infinitely many objects and their weights. Formally, an instance \(x\) comprises a finite alphabet \(\Gamma_x\) and a weight function \(w_x : \Gamma_x^* \to \mathbb{R} \cup \{\infty\}\), where \(\Gamma_x^*\) is the set of all finite strings over \(\Gamma_x\). A top-strings problem is defined over a set \(P\) of instances as follows. Given an instance \(x \in P\) and a natural number \(k\), the goal is to find a set \(T \subseteq \Gamma_x^*\) with the following two properties:

1. \(|T| = k\).
2. \(w_x(t) \leq w_x(s)\) for all \(t \in T\) and \(s \in \Gamma_x^* \setminus T\).

As for computational complexity, each instance \(x \in P\) is represented by some finite string, and we denote by \(|x|\) the length of this string. We assume the following. First, the alphabet \(\Gamma_x\) can be computed in polynomial time in \(|x|\). Second, the weight \(w_x(s)\), where \(s\) is a string in \(\Gamma_x^*\), can be computed in polynomial time in \(|x|\) and \(|s|\) (the latter is the length of the string \(s\)).

**Comment 4.1.** We allow \(\infty\) as a possible weight in order to model strings that are of no interest. Alternatively, we could assume that \(w_x\) is undefined for such strings. However, how every string has a weight (even though it could be \(\infty\)) results in a simpler abstraction.

If \(s\) and \(t\) are strings, then we denote by \(s \preceq_{\text{prfx}} t\) the fact that \(s\) is a prefix of \(t\). Note that we do not assume that the weight function \(w_x\) is monotone. That is, it could be the case that \(w_x(s) > w_x(t)\) even if \(s \preceq_{\text{prfx}} t\).

A constrained optimization problem is an essential part of Lawler-Murty’s reduction, and we now describe its abstraction. The goal of a prefix-constrained optimization problem (or just prefix optimization for short) is to find a minimum-weight string that has a given prefix \(u\). More formally:

**Problem 4.2.** Let \(P\) be a set of instances. Prefix optimization over \(P\) is the following problem. Given \(x \in P\) and \(u \in \Gamma_x^*\), find a string \(t \in \Gamma_x^*\) such that:

1. \(u \preceq_{\text{prfx}} t\).
2. \(w_x(t) \leq w_x(s)\) for all \(s \in \Gamma_x^*\) with \(u \preceq_{\text{prfx}} s\).

An algorithm that solves the prefix optimization over \(P\) is called a prefix optimizer for \(P\). In particular, a prefix optimizer \(A\) gets as input an instance \(x \in P\) and a string \(u \in \Gamma_x^*\), and outputs a string \(A(x, u) \in \Gamma_x^*\). To simplify our complexity analysis, we assume that \(A\) returns the weight of \(A(x, u)\) in addition to the string \(A(x, u)\) itself (hence, there is no need to explicitly account for the cost of computing the weight).

In our abstraction of Lawler-Murty’s reduction, a top-strings problem is solved by means of a prefix optimizer. The role of the prefix \(u\) is to encode the constraints that the solution \(t = A(x, u)\) must satisfy. (Note that the size \(u\) is not limited by that of \(x\).) The time needed to solve a top-strings problem is determined by two factors: the running time of the prefix optimizer \(A\) and the size of the string \(A(x, u)\). We use the monotone functions \(f\) and \(g\) in order to state upper bounds on the former and the latter, respectively. The size of \(A(x, u)\) affects the running time, because \(A(x, u)\) may be used to create prefix constraints that subsequent solutions should satisfy. The formal result is given by the next theorem.

**Theorem 4.3 (Adapted Lawler-Murty’s Reduction).** Let \(A\) be a prefix optimizer for \(P\), such that the running time of \(A\) is bounded by \(f(|x|, |u|)\) and \(|A(x, u)| < g(|x|) + |u|\). Then in \(O(k^2 \cdot |\Gamma_x| \cdot g(|x|) \cdot f(|x|, k \cdot g(|x|)))\) time, we can solve the top-strings problem over \(P\).

In the following section, we formulate TopPatterns as a top-strings problem. Later we show that TopPatterns is FPT by devising a prefix optimizer and applying Theorem 4.3.

5. Serializing Patterns

In this section, we formulate TopPatterns as a top-strings problem. The set \(P\) of instances comprises all pairs \((S^W, \Lambda)\), where \(S^W\) is a weighted schema and \(\Lambda\) is a set of labels. For \(x = (S^W, \Lambda)\), we define the alphabet \(\Gamma_x\) to be \(\text{dom}(S) \cup \{\prec\}\), where \(\prec\) is a special symbol that is not in \(\Sigma\). Later, we will show how the symbol \(\prec\) will be used for representing the structure of the tree (rather than the labels on its nodes). The strings in \(\Gamma_x^*\) represent the objects to be generated by TopPatterns, namely, non-redundant (\(S, \Lambda\))-patterns. Therefore, we should define how a non-redundant (\(S, \Lambda\))-pattern is serialized, that is, encoded as a string over \(\Gamma_x\). Actually, we define a function \(\text{ser}\) that maps every non-redundant \(\Lambda\)-pattern \(t\) to a string over \(\Gamma_x\), provided that \(t\) has only labels of \(\text{dom}(S)\). If a string \(s\) represents a non-redundant (\(S, \Lambda\))-pattern \(t\) (i.e., \(s = \text{ser}(t)\)), then its weight \(w_x(s)\) is \(|t|\); otherwise, \(w_x(s) = \infty\).

The serialization \(\text{ser}\) needs to be sound in the sense that two non-redundant \(\Lambda\)-patterns are mapped to the same string if and only if they are isomorphic; that is, \(\text{ser}(t_1) = \text{ser}(t_2)\) if and only if \(t_1\) and \(t_2\) are isomorphic \(\Lambda\)-patterns. The serialization should also be complete in the sense that \(\text{ser}(t)\) is defined for every non-redundant \(\Lambda\)-pattern \(t\). However, \(\text{ser}\) is not necessarily surjective, that is, some strings of \(\Gamma_x^*\) might not correspond to any \(\Lambda\)-pattern and, hence, their weight is infinity. Finally, we need to devise the mapping \(\text{ser}\) so that it will be possible to efficiently solve the prefix optimization.

Next, we explain the mapping \(\text{ser}\). Suppose that \(t\) is a non-redundant \(\Lambda\)-pattern with \(q + 1\) leaves. By our assumption (in Section 3) that \(\Lambda\) is a set, Proposition 2.4 implies that the labels of all the leaves are different from one another and all of them belong to \(\Lambda\). Recall that \(\prec\) is a total order over \(\Sigma\). Hence, all the leaves of \(t\) can be arranged in a unique sequence \((v_0, v_1, \ldots, v_q)\), such that \(\lambda(v_{i+1}) \prec \lambda(v_i)\) for all \(i \in \{1, \ldots, q\}\). We denote this sequence by \(I(t)\) and use it to construct \(\text{ser}(t)\) in two steps. Let \(p_i (1 \leq i \leq q)\) be the unique shortest path of \(t\) from \(v_{i-1}\) to \(v_i\). An ascending-leaf traversal of \(t\) is done as follows. We start by traversing the nodes of \(p_i\) from \(v_0\) to \(v_1\). Then, we continue from the first unvisited node of \(p_{i}\) to \(v_i\), and so on. The traversal ends at leaf \(v_q\) of \(p_q\). For a node \(u\) of \(t\), we define \(\text{time}(u) = i\) if \(u\) is the \(i\)th node to be visited. Note that \(\text{time}(v_0) = 1\) and \(\text{time}(v_q)\) is the number of nodes in \(t\). Observe that the ascending-leaf traversal is unique, and each node is visited exactly once. Hence, \(\text{time}(u)\) is fully determined by \(t\) and \(u\).

**Example 5.1.** Consider the schema \(S\) and the set \(\Lambda\) of Figure 1. In our examples, we assume that \(\prec\) is the dictionary order; particularly, \(\text{dept} \prec \text{exec} \prec \text{project} \prec \text{regular}\). Consider the trees of Figure 3 (where \(t_1\) and \(t_2\) are also in Figure 2). The trees
that belongs to every
are isomorphic if and only if
ser
starts with
ser
this is not the case for
In particular,
followed by the label
each node
is obtained by iterating over the nodes
and for each
u

∆(u) = time(u) − time(u′). Finally, the string
ser(t) is obtained by iterating over the nodes
u in time(u) and, for each
u
writing down
∆(u) = 1 occurrences of
followed by the label
λ(u).
(Recall that
is the special symbol
that belongs to every
but not to
.) As becomes apparent in
next example, if a label in
ser(t) is followed by a
, then its corresponding node in
is a leaf.

**Example 5.2.** Consider again the trees of Figure 3.

\[ ser(t_1) = \text{exec, emp, pMember, project, pMember, emp, regular} \]

In particular,
\( ser(t_1) \) does not contain
because
is a path. But this is not the case for
\( ser(t_3) \).

\[ ser(t_3) = \text{exec, emp, pMember, project, pMember, project, \ <\, emp, regular} \]

For
\( t_5 \) we have:

\[ ser(t_5) = \text{dept, pMember, project, pMember, emp, exec, \ <\, \ <\, \ <\, pMember, project, \ <\, emp, regular} \]

Finally, consider the trees
\( t_6 \) and
\( t_7 \) of Figure 3. While
\( ser(t_6) \) starts with
exec,
the string
\( ser(t_7) \) starts with
depth. The reason is that
depth
and
depth
is a leaf in
but not in
.

Clearly,
ser
is complete, that is, defined for every non-redundant
A-pattern
. The following proposition shows that it is also sound.

**Proposition 5.3.** Two non-redundant
A-patterns
\( t_1 \) and
\( t_2 \) are isomorphic if and only if
\( ser(t_1) = ser(t_2) \).

In order to obtain an algorithm for
TopPatterns
by applying
Theorem 4.3,
we need to solve the following prefix-optimization problem.

**Problem 5.4.** For instances
\( x = (S^W, \Lambda) \),
prefix optimization is the following problem. Given a weighted schema
\( S^W \), a set
\( \Lambda \) of labels and a string
\( u \in \Gamma^+ \), find a minimum-weight,
non-redundant
\( (S, \Lambda) \)-pattern
such that
\( u \preceq_{\text{prfx}} ser(t) \), or determine that no such
\( t \) exists.

**Example 5.5.** Let
\( S^W \) be our usual schema (Figure 1), and let
\( \Lambda \) be the set \{dept, exec, project\}. Consider again the patterns of Figure 3.

Let
\( u = \text{exec, emp} \). Then
\( t_6 \) satisfies
\( u \preceq_{\text{prfx}} ser(t_6) \). But for
\( t_2 \) it is not true that
\( u \preceq_{\text{prfx}} ser(t_2) \), since
\( ser(t_2) \) begins with
depth.
It can be shown that
\( t_6 \) is a minimal
\( (S, \Lambda) \)-pattern
\( t \) satisfying
\( u \preceq_{\text{prfx}} ser(t) \); therefore,
\( t_6 \) is a solution to the prefix optimization for the input
\( S^W, \Lambda \) and
\( u \).

Let
\( v = \text{dept, worksIn, emp, exec, \ <\} \). The tree
\( t_7 \) satisfies
\( v \preceq_{\text{prfx}} ser(t_7) \). But, now
\( v \preceq_{\text{prfx}} ser(t'_3) \) is also true, but
\( t'_3 \) is not a
\( S \)-pattern because the
worksIn node has two
depth
eighbors.

Next suppose that
\( w = \text{dept, worksIn, emp, exec, \ <\} \). Then
\( w \preceq_{\text{prfx}} ser(t'_5) \), but 
\( t'_5 \) is a
\( S \)-pattern. Actually, it is easy to show that no
\( (S, \Lambda) \)-pattern
\( t \) satisfies
\( w \preceq_{\text{prfx}} ser(t) \).

In the following two sections, we present an algorithm for the prefix optimization over instances
\( x = (S^W, \Lambda) \).

6. **Prefix Optimization by Leaf Constraints**

This section shows how to reduce the prefix optimization (stated in Problem 5.4) to the task of finding a minimal
\( (S, \Lambda) \)-pattern under
leaf constraints.
So, for this section, we fix the input
\( S^W \),
\( \Lambda \) and
\( u \). Recall that the goal is to find a minimum-weight, non-redundant
\( (S, \Lambda) \)-pattern
\( t \) such that
\( u \preceq_{\text{prfx}} ser(t) \), or determine that no such
\( t \) exists.

### 6.1 Consistency

We first discuss the issue of the consistency of the prefix
\( u \). We say that
\( u \) is a consistent prefix
\( \Lambda \) if there exists a non-redundant
A-pattern
\( t \) such that
\( u \preceq_{\text{prfx}} ser(t) \). Note that consistency does not take the schema
\( S \) into account. In particular, consistency of
\( u \) does not necessarily imply that there exists a non-redundant
\( (S, \Lambda) \)-pattern
\( t \), such that
\( u \preceq_{\text{prfx}} ser(t) \).

Testing consistency can be done by trying to construct a subtree of a non-redundant
\( (S, \Lambda) \)-pattern
\( t \) such that
\( u \preceq_{\text{prfx}} ser(t) \), or determine that no such
\( t \) exists.

---

6 If no such
\( t \) exists, then formally we return
\( u \), which has an infinite weight, and which can be represented by some special symbol (like
null).

7 Minimum weight does not guarantee non-redundancy, because some noes and edges may have zero weight.

8 The pattern we are looking for must be non-redundant, because
\( ser \) is well defined only for non-redundant patterns.
an ordinary label and continues until the next one. Thus, \( s_i \) begins with an ordinary label followed by zero or more occurrences of \(<\). For each \( s_i \), we create a node \( v_i \) that has the ordinary label that appears in \( s_i \). For all \( i \), we define\(^9\) \( \text{index}(v_i) = i \) and set \( \text{cnt}(v_i) \) to the number of \(<\) in \( s_i \). We say that \( v_i \) is a definite leaf if \( i = 1 \) or \( \text{cnt}(v_i) > 0 \). We create an edge \([v_i, v_j]\), where \( i < j \), if \( \text{index}(v_i) = \text{index}(v_j) + \text{cnt}(v_{j-1}) - 1 \). Figure 4 shows examples of trees created in this way.

The construction may fail for three main reasons. First, it is not well defined if there is a \( j > 1 \), such that \( \text{index}(v_i) - \text{cnt}(v_{i-1}) - 1 \leq 0 \). Second, it creates a tree \( t \) that is inconsistent with \( u \). This happens if there are \( v_i \) and \( v_j \), such that \( \text{index}(v_i) = \text{index}(v_j) - \text{cnt}(v_{j-1}) - 1 \) and \( v_i \) is a definite leaf. It also happens if \( \text{cnt}(v_n) > 0 \), but for every \( v \) that is not a definite leaf, \( \text{index}(v) > \text{index}(v_n) - \text{cnt}(v_n) - 1 \).

The last reason for failure is inconsistency with \( \Lambda \). This happens if one of the following holds. First, either \( t \) is not a \( \Lambda \)-pattern or \( \text{cnt}(v_n) > 0 \), but some definite leaf has the largest (under \(<\) label of \( \Lambda \). Second, the label of some definite leaf is not in \( \Lambda \). Third, there are two definite leaves \( u \) and \( v \), such that \( \text{index}(v) > \text{index}(u) \), but \( \Lambda(v) < \Lambda(u) \).

If the construction does not fail, then we can add a path \( p \) from a certain node \( v \) of \( \text{tree}(u) \), such that \( v \) is not a definite leaf, and the path \( p \) includes all the labels of \( \Lambda \) that are missing from \( \text{tree}(u) \) and ends in a leaf labeled with the largest element of \( \Lambda \). Hence, we succeed if and only if \( u \) is a consistent prefix for \( \Lambda \).

In the sequel, \( v_{\text{last}} \) denotes the node of \( \text{tree}(u) \) with the largest index (i.e., \( v_n \) and \( k_{\text{last}} \) denotes \( \text{cnt}(v_n) \) (i.e., the number of \(<\) at the end of \( u \)).

### 6.2 General Reduction

The reduction we present in this section is general in the sense that it does not depend on a particular language of \( b \)-specifications.

Suppose that \( u \) is a consistent prefix for \( \Lambda \). The goal is to expand \( \text{tree}(u) \) into a minimum-weight \((S, \Lambda)\)-pattern \( t_{\text{min}} \), such that \( u \preceq_{\text{prefix}} \text{ser}(t_{\text{min}}) \). Expanding \( \text{tree}(u) \) in one particular way is done by choosing some nodes of \( \text{tree}(u) \) and adding a subtree to each one of them. Obviously, there are many ways of choosing the nodes and subtrees, and we should consider all of them in order to find \( t_{\text{min}} \). Conceptually, we do it in two stages. Firstly, we create all possible subtrees for each node \( v \) of \( \text{tree}(u) \) and, secondly, we consider all combinations of those subtrees.

There are several constraints that should be satisfied by each expansion of \( \text{tree}(u) \). We take care of some of those constraints (such as conformity to the schema \( S \)) in the first stage, as explained in the next section. In the second stage, the main concern is that the resulting expansion will be a \( \Lambda \)-pattern. We should choose subtrees so that together with \( \text{tree}(u) \) they form an exact cover of \( \Lambda \). We do it by using a \textsc{MinLbexc} solver (see Section 2.8).

The input to the \textsc{MinLbexc} solver is obtained as follows. For each node \( v \) of \( \text{tree}(u) \), we pick a unique label \( l_v \) and create triples of the form \((L, l_v, w)\), where \( L \subseteq \Lambda \). Each triple \((L, l_v, w)\) corresponds to a minimum-weight \((S, \Lambda)\)-pattern \( t \) that satisfies some constraints (as explained later), and \( w \) is the weight of \( t \). Thus, \( L \) is the set of labels of \( t \) that \( v \) can contribute to an exact cover of \( \Lambda \) if we add \( t \) to \( \text{tree}(u) \) (by merging its root\(^{10}\) with node \( v \)). Note that the labels \( l_v \) are needed only for the \textsc{MinLbexc} solver, and we assume that they do not appear \( \Gamma_v^* \), where \( x \) is the instance \((S^{10}, \Lambda) \).

We first explain intuitively how the triples \((L, l_v, w)\) are created

---

\(^9\)The idea is that in an ascending-leaf traversal of the generated tree, \( \text{time}(v) = \text{index}(v) \).

\(^{10}\)In this section, we use the term “root” informally. We fully define it in the next section.
rule that applies to definite leaves. If \( \mathfrak{v}_{\text{last}} \) is not a definite leaf, then only the third rule applies to it. Applying this rule requires that \( \lambda(v) \neq \xi \), because \( \xi \) should be contributed to the exact cover by a triple that either the third or fourth rule creates.

**Rule 2.** For each \( v \) that is neither \( \mathfrak{v}_{\text{last}} \) nor a definite leaf, we create the triples \( (L, l_v, \perp) \) for all nonempty subsets \( L \subseteq \Lambda - D \) such that \( \xi \not\in L \).

This rule says that nodes of \( \text{tree}(u) \) that are definitely internal (i.e., not leaves) can be merged with the root of a new \( L \)-pattern if its leaves are labeled by neither \( \xi \) nor the label of any definite leaf.

**Rule 3.** If \( \mathfrak{v}_{\text{last}} \) is not a definite leaf, we create \( (L, l_v, \perp) \) for every \( L \) such that \( \xi \in L \subseteq \Lambda - D \). If \( \lambda(\mathfrak{v}_{\text{last}}) = \xi \), then we also create the triple \( \{\xi\}, l_{\mathfrak{v}_{\text{last}}} \), because \( \mathfrak{v}_{\text{last}} \) can contribute \( \xi \) just by itself.

This rule creates the triples corresponding to patterns that could be merged with \( \mathfrak{v}_{\text{last}} \). Hence, they cannot contribute any label of \( D \) to the exact cover. However, they must contain the next leaf \( \mathfrak{v}_{\text{next}} \) to be visited. Therefore, the label \( \xi \) chosen for \( \mathfrak{v}_{\text{next}} \) must appear in \( L \), since \( (L, l_v, \perp) \) corresponds to a non-redundant \( L \)-pattern \( \xi \). We later show how to handle the constraint that \( \xi \) should be the label of a leaf in \( L \).

**Rule 4.** If \( \mathfrak{v}_{\text{last}} \) is a definite leaf (i.e., \( k_{\text{last}} > 0 \)), then we consider every node \( v \), such that \( v \) is not a definite leaf and \( \text{index}(v) < \text{index}(\mathfrak{v}_{\text{last}}) - k_{\text{last}} \). We create for \( v \) the triples \( (L, l_v, \perp) \) for all \( L \) such that \( \xi \in L \subseteq \Lambda - D \).

This rule creates the triples corresponding to patterns that contain \( \mathfrak{v}_{\text{next}} \). As explained earlier, these patterns can be merged only with some of the non-leaf nodes of \( \text{tree}(u) \). Similarly to the previous rule, the leaves of \( L \) cannot have any label of \( D \), but one of them must be labeled with \( \xi \).

When \( L = \{\lambda(v)\} \), the rules may create both \( (L, l_v, \perp) \) and \( (L, l_v, 0) \). In this case, the former is discarded.

After constructing the triples, we replace \( \perp \) with the appropriate weight in each \( (L, l_v, \perp) \). We do it by finding the corresponding pattern \( t_{\xi} L \) (see Section 7). The triple \( (L, l_v, \perp) \) is updated to \( (L, l_v, w) \), where \( w = W(t_{\xi} L) - W(\lambda(v)) \). We subtract \( W(\lambda(v)) \) from the weight of \( t_{\xi} L \), because \( v \) and the root of \( t_{\xi} L \) are going to be merged and, hence, the weight of these two nodes should be counted only once. Let \( C^\xi \) be the resulting set of triples.

The input to the MinLBExC solver comprises \( C^\xi \), \( \Lambda \) (i.e., the set to be covered) and the lb-constraint \( B \) that we now explain. Each node \( v \) is merged with at most one subtree; hence, the exact cover should have at most one triple for \( v \). So, we specify \( B \) by the regular expression \( l_v ^n \_r ^{n-2} _r ^{n-2} \_r ^{n-2} _r ^{n-2} \), where \( v_1 \ldots v_n \) are the nodes of \( \text{tree}(u) \). The four rules guarantee that only the definite leaves can contribute their labels to the exact cover; moreover, \( \xi \) will be the label of the next leaf (after the last definite leaf of \( \text{tree}(u) \)) in the ascending-leaf traversal of the expansion \( t_{\xi} \) of \( \text{tree}(u) \).

The algorithm is given in Figure 5. Line 1 finds the set \( P \) of all labels that are greater than the label of every definite leaf. Line 2 creates the unique label for each variable; note that these labels are used only for finding the exact cover, and the labels of \( \text{tree}(u) \) are unchanged. The loop of line 3 iterates over all the labels \( \xi \) of \( P \). For each \( \xi \), line 4 creates the triples according to the four rules. The loop of line 5 iterates over all the triples of the form \( (L, l_v, \perp) \) (i.e., the weight is undefined). Line 6 uses the algorithm of Section 7 to find a minimum-weight pattern \( t_{\xi} L \) that corresponds to \( (L, l_v, \perp) \). Line 7 tests if \( t_{\xi} L \) exists. If so, line 8 updates the weight; otherwise the triple is discarded in line 10. Line 11 finds a minimum-weight exact cover. Line 12 tests if the cover exists. If so, for every \( t_{\xi} L \) that corresponds to a triple in the exact cover, line 13 merges the root of \( t_{\xi} L \) with node \( v \) of \( \text{tree}(u) \). The resulting tree is assigned to \( t_{\xi} \). Line 14 returns a minimum-weight pattern among all the \( t_{\xi} \) that were found.

In the next section, we precisely define the trees \( t_{\xi} L \).

### 6.3 Constrained Minimal Patterns

Some of the constraints that the construction of each \( t_{\xi} \) should satisfy are incorporated either in the rules for creating the triples or in the regular expression \( B \). Now, we discuss the rest of these constraints.

First, we need to verify that \( \text{tree}(u) \) is an \( S \)-pattern; if not, it cannot be expanded into an \( (S, \Lambda) \)-pattern. Our main result (Theorem 3.1) is for all the languages of lb-specifications, such that MinLBExC is FPT. Therefore, we have to check that \( \text{tree}(u) \) is an \( S \)-pattern by reducing this problem to MinLBExC. Thus, for each node \( v \) of \( \text{tree}(u) \), we construct the following instance of MinLBExC. There is a triple \((\{\sigma\}, \{\sigma, 1\}, v)\) for each occurrence of a label \( \sigma \) in the bag \( \text{lb}(\text{nbr}(v)) \). The bag to be covered is \( \text{lb}(\text{nbr}(v)) \) and the lb-constraint is \( S(\sigma(v)) \). Clearly, there is an exact cover if and only if \( \text{lb}(\text{nbr}(v)) \subseteq [S(\sigma(v))] \).

Recall that we add a tree \( t_{\xi} L \) to \( \text{tree}(u) \) at node \( v \). To make this operation well defined, we require \( t_{\xi} L \) to have a node \( v_0 \) that is designated as the root. We do not include \( \text{nbr}(v) \) among the leaves, even if it has only one neighbor. The label of \( v_0 \), denoted by \( l_r \), is \( \lambda(v) \) since \( v \) has to be merged with \( v_0 \).

The algorithm that finds \( t_{\xi} L \) creates an \( S \)-pattern. Thus, we only need to make sure that when merging the root \( v_0 \) of \( t_{\xi} L \) with \( v \), the resulting node \( \hat{v} \) still satisfies \( \text{lb}(\text{nbr}(\hat{v})) \subseteq [S(\lambda(v))] \). So, we associate \( v_0 \) with the lb-constraint \( B_{\hat{v}} \), rather than \( S(l_r) \). We get \( B_{\hat{v}} \) as follows. Let \( b = \text{lb}(\text{nbr}(v)) \) (i.e., the bag of labels of the neighbors of \( v \) in \( \text{tree}(u) \)). \( B_{\hat{v}} \) is obtained from \( B \) by

\[
B_{\hat{v}} \triangleq \{b' \mid b \cup b' \in B\}.
\]

Finally, \( B_{\hat{v}} \) is \( S(l_r) \).

Recall that in the third and fourth rules of the previous section, \( \xi \) is required to be the label of a leaf in \( t_{\xi} L \). Moreover, by the definition of \( \xi \), all labels \( l \in \Lambda \), such that \( l < \xi \), can appear only in non-leaf nodes of \( t_{\xi} L \). Thus, we use two sets \( \text{LF}, \text{nLF} \subseteq L \).
of leaf constraints (where LF and nLF stand for “leaf” and “non-leaf,” respectively). The set LF consists of the labels that must appear in leaf nodes. The set nLF comprises the labels that may appear only in non-leaf nodes. The pattern \( t_1^L[L] \) should satisfy LF \( \subseteq \lambda(\text{leaves}(t_1^L[L])) \subseteq L \setminus \text{nLF} \). In summary, \( \text{nLF} = \{ t \in \Lambda \wedge t \prec \xi \} \) and LF = \( \{ t \in L \} \) (i.e., LF is empty if \( \xi \) is not in \( L \); otherwise, LF is the singleton \( \{ \xi \} \)). Formally, the problem leaf-constrained minimal pattern, denoted by MinLCPattern, generalizes MinPattern when the bag of labels is a set. The input consists of a weighted schema \( S^W \), a set \( L \) of labels, a root label \( r \), an lb-constraint \( B_r \), and leaf constraints LF, nLF \( \subseteq \Lambda \). The goal is to find a satisfactory \((S, L)\)-pattern of a minimal weight. We assume that LF \( \cap \text{nLF} = \emptyset \) or else there is no such an \((S, L)\)-pattern.

For every label \( \pi \) in \( \text{dom}(S) \), the algorithm has an array \( T^\pi \). An index of \( T^\pi \) consists of a label \( \sigma \) in \( \text{dom}(S) \) and a nonempty subset \( N \) of \( L \). Throughout the execution, \( T^\pi[\sigma, N] \) is either \( \emptyset \) (i.e., null) or a tree \( t \). Let \( \pi / t \) denote the tree that is obtained by creating a new root \( v_\pi \) with the label \( \pi \) and adding an edge from \( v_\pi \) to root(\( t \)) (which denotes the root of \( t \)). We say that \( t \) is \((\pi, \sigma, N)\)-proper if all of the following hold.

- The root of \( t \) is labeled with \( \sigma \).
- Both \( t \) and \( \pi / t \) are \((S, N)\)-patterns.
- LF \( \cap N \subseteq \lambda(\text{leaves}(\pi / t)) \subseteq N \setminus \text{nLF} \).

During the execution of the algorithm, if \( T^\pi[\sigma, N] \) is non-null, then it is \((\pi, \sigma, N)\)-proper. Note that in the special case where \( N = \{ \sigma \} \), the tree \( t \) may contain only one node, but not when \( \sigma \in \text{nLF} \) or else we have that \( \text{leaves}(\pi / t) \cap \text{nLF} \neq \emptyset \).

We create trees by combining smaller ones. Each entry of the \( T^\pi \) may be updated several times during the execution of the algorithm. The priority queue \( Q \) stores all the triplets \((\pi, \sigma, N)\). Priority in \( Q \) is determined by the weight of the tree \( T^\pi[\sigma, N] \); a higher weight means a lower priority. The weight of \( \emptyset \) is infinity; hence, \((\pi, \sigma, N)\) has the lowest priority if \( T^\pi[\sigma, N] = \emptyset \). When \((\pi, \sigma, N)\) is removed from the top of \( Q \), its corresponding \( T^\pi[\sigma, N] \) has a minimal weight among all the \((\pi, \sigma, N)\)-proper trees; namely, \( T^\pi[\sigma, N] \) has obtained its final value.

By definition, a \((\pi, \sigma, N)\)-proper tree exists only if the labels \( \pi \) and \( \sigma \) can be neighbors according to the given schema \( S \), that is, \( \{ \sigma \} \subseteq \{ \sigma \} \subseteq \{ \sigma \} \subseteq \{ \sigma \} \) and \( \{ \pi \} \subseteq \{ \pi \} \subseteq \{ \pi \} \subseteq \{ \pi \} \). If this condition holds, we write \( \text{S} \models \pi \equiv \sigma \). We can test whether \( \text{S} \models \pi \equiv \sigma \) by applying (twice) the reduction to MINLBEXC that is described in Section 6.3.

The main idea for updating \( T^\pi[\sigma, N] \) is to find \( h \) trees \( t_i \) \( T^\pi[\sigma, M_i] \) (1 \( i \leq h \)) such that the \( M_i \) is a minimal exact cover of \( N \). The triplets for this instance of MINLBEXC are \((M_i, \pi / t_i)\) and the lb-constraint is \( S(\pi, \pi / t_i) \). We can verify that \( S \models \pi \equiv \sigma \) and the leaf constraints hold. If so, we create a new root \( v_\pi \) with the label \( \sigma \), connect \( v_\pi \) to the roots of the \( t_i \) and assign the result to \( T^\pi[\sigma, N] \).

The pseudo code of MinLC (Figure 6) consists of the main procedure and three subroutines. The subroutine Initialize() is called in line 1 of the main procedure. In that subroutine, line 1 initializes \( Q \) to the empty priority queue. The loop of line 2 iterates over all the pairs \( \pi \) and \( \sigma \). Lines 3–5 initialize \( T^\pi[\sigma, N] \) to \( \emptyset \) and insert \((\pi, \sigma, N)\) into \( Q \) for all nonempty subsets \( N \) of \( L \). Finally, lines 6–7 check whether \( \text{S} \models \pi \equiv \sigma \) for all nodes in nLF; if so, \( T^\pi[\sigma, \{ \sigma \}] \) is assigned a new tree \( t \) that consists of a single node \( v_\sigma \), such that \( v_\sigma \) is the root and its label is \( \sigma \). We assume that the procedure singletonRTree(\( \sigma \)) constructs and returns that tree. However, before line 7 sets \( T^\pi[\sigma, \{ \sigma \}] \) to \( t \), line 6 also checks that \( S \models \pi \equiv \sigma \).

After initialization, the main procedure executes the loop of line 2. The top triplet \((\pi, \sigma, N)\) is removed from \( Q \), and then line 5 calls the subroutine Update, which has three arguments: a label \( \pi \), a label \( \sigma \) and an lb-constraint \( B \). This subroutine implements the main idea mentioned earlier. In particular, it tries to improve all the \( T^\pi[\sigma, N] \) under the restriction that the new root \( v_\sigma \) satisfies \( \lambda(\text{nbr}(v_\sigma)) \in B \). In line 5, \( B \) is the lb-constraint \( S(\sigma, \emptyset) \) (rather than \( S(\sigma) \)), because the definition of a \((\pi, \sigma, N)\)-proper tree requires that we can add a neighbor labeled with \( \pi \) to the root \( v_\sigma \).

The third argument \( S(\sigma, \emptyset) \) in line 5 is represented by \( \emptyset \), because \( \sigma \) is already the second argument of Update; similarly, for
the call in line 6 (as explained in Section 6.3, the argument \( B_t \) is actually \( S(l_t) \) for some \( (S(m), l, B_t) \) of labels).

The subroutine \( \text{Update}(\pi, \sigma, B) \) calls \( \text{Assmbl} \) in order to create the new tree \( t \) from \( t_1, \ldots, t_h \), as mentioned earlier. Line 3 of \( \text{Assmbl} \) calls \( \text{singletonRTree}(\sigma) \) to initialize \( t \) to a tree comprising just the root \( \nu \). The loop of line 4 makes each \( t_i \) a subtree of \( t \) by connecting \( \text{root}(t_i) \) to \( \nu \).

The search for \( t_1, \ldots, t_h \) is done by a reduction to the following instance \( I \) of \( \text{MINLBEXC} \). Lines 1–5 of Update construct the set \( C \) that consists of all triples \((M, \tau, w)\), such that \( T^\tau[M, \tau] \neq \perp \) and \( w = W(T^\tau[M, \tau]) \). The bag of labels (of the instance \( I \)) is a nonempty subset \( N \) of \( L \) and the lb-constraint is \( B \). The triples of a legal cover correspond to the entries of \( T^\tau \) that form the trees \( t_1, \ldots, t_h \) mentioned earlier. To simplify the pseudo code, line 6 simultaneously solves the \( \text{MINLBEXC} \) problem for all nonempty subsets \( N \) of \( L \). Hence, \( L \) (rather than some \( N \)) is the first argument of the call to \( \text{ExactMinCovers} \) in line 6. That call returns the array \( \text{MinCovers} \) that has an index \( N \) for every nonempty subset \( N \) of \( L \). \( \text{MinCovers}[N] \) is a minimal exact cover of \( N \) (or \( \perp \) if none exists). Line 7 iterates over all nonempty subsets \( N \) of \( L \) to construct the rooted tree \( t \) that is assigned to \( T^\pi[N, \sigma] \). Here, four cases are considered.

- \( \sigma \notin N \). In this case, \( t \) is constructed from the trees that correspond to the triples of \( \text{MinCovers}[N] \)—the minimal exact cover of \( N \).
- \( \sigma \in N \cap \text{LF} \) and \( |N| > 1 \). We do the same as in the previous case. Note that if \( \sigma \in N \cap \text{LF} \) and \( |N| = 1 \), then \( T^\pi[N, \sigma] \) is assigned its final value in the initialization.
- \( \sigma \in N \setminus \text{LF} \) and \( |N| > 1 \). In this case, \( t \) is constructed from the minimal cover among \( \text{MinCovers}[N] \) and \( \text{MinCovers}[N \setminus \{\sigma\}] \). Note that if \( \sigma \in N \setminus \text{LF} \) and \( |N| = 1 \), then \( T^\pi[N, \sigma] \) is assigned its final value in the initialization.
- \( \sigma \in N \cap \text{nLF} \) and \( |N| > 1 \). We do the same as in the previous case. Note that when \( \sigma \in N \cap \text{nLF} \), the initialization does not create a tree for \( \sigma \). Hence, it is easy to show by induction that the algorithm never creates a tree having a leaf labeled with \( \sigma \).

The first two cases are handled in lines 8–9 of Update, and the last two—in lines 10–11. In lines 9 and 11, there is no need to compare the new value with the previous one, because the weight of \( T^\pi[N, \sigma] \) can only decrease (this is certainly true when it is changed from \( \perp \) to a non-null value, and an easy induction shows that it is always true).

In the main procedure, after the loop of line 2 terminates, we create the output \( t \) by calling Update in line 6. The second and third arguments are the root label \( l_t \) and the lb-constraint \( B_t \); that the root should satisfy. There is no need to require that root \((t)\) can have an additional neighbor labeled with some \( \pi \). Hence, the call in line 6 does not have a test of the form \( S \models \pi \sigma \) (in contrast to line 4). Furthermore, the first argument of Update (in line 6) is the special symbol \( * \) which is not in \( \text{dom}(S) \). Note that \( * \) is only needed to indicate that Update assigns its result to \( T^\pi[l_t, L] \). In addition, in line 6 of Update, we now have to find a minimal exact cover only for \( L \), rather than every nonempty subset \( N \) of \( L \). Similarly, the loop of line 7 also has to consider only \( L \).

### Algorithm \( \text{MinLCP}(\mathcal{S}^W, L, l_t, B_t, LF, nLF) \)

1. \( \text{Initialize}() \)
2. \( \text{while } Q \neq \emptyset \) do
3. \( \langle \pi, \sigma', N \rangle \leftarrow Q.\text{pop}() \)
4. \( \text{for all } \pi \in \text{dom}(S) \text{ where } S \models \pi \sigma \) do
5. \( \text{Update}(\pi, \sigma, S(\sigma); \pi, \sigma, N) \)
6. \( \text{Update}(\pi, l_t, B_t) \)
7. \( \text{return } T^\pi[l_t, L] \)

### Subroutine \( \text{Initialize}() \)

1. \( Q \leftarrow \emptyset \)
2. \( \text{for all } \pi \in \text{dom}(S) \text{ and } M \subseteq L \) do
3. \( t \leftarrow T^\pi[M, M] \)
4. \( \text{if } t \neq \perp \) then
5. \( C \leftarrow C \cup \{(M, \tau, W(t))\} \)
6. \( \text{MinCovers} \leftarrow \text{ExactMinCovers}(L, B, C) \)
7. \( \text{for all nonempty } N \subseteq L \) do
8. \( \text{if } \sigma \notin N \text{ or } (\sigma \in LF \text{ and } |N| > 1) \) then
9. \( T^\pi[N, \sigma] \leftarrow \text{Assmbl}(\sigma, \text{MinCovers}[N]) \)
10. \( \text{if } \sigma \notin N \setminus LF \text{ and } |N| > 1 \) then
11. \( T^\pi[N, \sigma] \leftarrow \min. \text{ of } \text{Assmbl}(\sigma, \text{MinCovers}[N]) \text{ and } \text{Assmbl}(\sigma, \text{MinCovers}[N \setminus \{\sigma\}]) \)

### Subroutine \( \text{Assmbl}(\sigma, C') \)

1. \( C' \leftarrow \perp \)
2. \( \text{for all } \pi \in \text{dom}(S) \text{ and } M \subseteq L \) do
3. \( t \leftarrow T^\pi[M, M] \)
4. \( \text{if } t \neq \perp \) then
5. \( \text{t}_M \leftarrow T^\pi[M, M] \)
6. \( \text{add } \text{t}_M \text{ as a subtree of } t \text{ immediately below root}(t) \)
7. \( \text{return } t \)

### Figure 6: Finding a minimal leaf-constrained \((S, L)\)-pattern

#### 7.1 Correctness and Efficiency

Correctness of the algorithm \( \text{MinLCP} \) is by the next lemma. The proof is similar to showing correctness of the two algorithms: one is \( \text{FindMinPattern} \) [13] and the other finds a minimal Steiner tree [12].

**Lemma 7.1.** Whenever line 3 of \( \text{MinLCP} \) pops \( \langle \pi, \sigma', N \rangle \) from \( Q \), it holds that \( T^\pi[N, \sigma'] \) is either a minimal \((\pi, \sigma', N)\)-proper root, or \( \perp \) if no such tree exists.

The next theorem states the correctness of \( \text{MinLCP} \), which is implied by Lemma 7.1. It also gives the dependency of the running time on the input and on the running time of \( \text{ExactMinCovers} \).
which is a solver for MinLBExC that is discussed in Section 2.8. Recall our assumptions (in Section 6.3) that $L$, $LF$, nLF $\subseteq \Lambda$, and that $B_t = S(l_t)$ is represented by a bag $b$, where $|b| \leq |\Lambda|$. 

Theorem 7.2. As output, MinLCP($S^W$, $L$, $l_t$, $B_t$, LF, nLF) returns a minimal-weight satisfactory $(S, L)$-pattern or $\bot$ if no such pattern exists (hence, MinLCP is a MinLCPattern solver). The number of operations and function calls is polynomial in $2^{(|\Lambda| \cdot |dom(S)|)}$.

Theorem 7.2, together with Lemma 6.1 and Theorem 4.3, imply our main result (Theorem 3.1) when $\Lambda$ is a set.

8. Generalizing the Algorithm

We first discuss how to generalize our main result to the case where $\Lambda$ is a bag. In Section 5, we use $I(t)$ to denote the unique string of the leaves of an $(S, \Lambda)$-pattern $t$, according to the order $\prec$ defined on $\Sigma$. When $\Lambda$ is a bag, $I(t)$ is non-decreasing and no longer unique. Consequently, there may be more than one way to serialize $t$. We let $ser^+(t)$ denote the set of all the different serializations. Observe that for two $(S, \Lambda)$-patterns $t_1$ and $t_2$, we have that $ser^+(t_1)$ and $ser^+(t_2)$ are either the same (if $t_1$ and $t_2$ are isomorphic) or disjoint (otherwise).

Theorem 4.3 enables us to enumerate only strings. So, we are forced to find all the strings of a $ser^+$ before we can find any string that represents an $(S, \Lambda)$-pattern having a larger weight than that of $t$. However, it is easy to show that for all $(S, \Lambda)$-patterns $t$, the cardinality of $ser^+(t)$ is bounded by $\times \Lambda$, which is the product of the multiplicities over the different elements of $\Lambda$. As a result, we are guaranteed to find a top-$k$ set of $(S, \Lambda)$-patterns by producing a top-$k'$ set of strings, where $k' = \times \Lambda \cdot k$.

Some technical details need to be adjusted in Sections 6 and 7. They are mostly straightforward and will be described in the full version. Regarding Section 6, an important observation is that, even when $\Lambda$ is a bag, we can still find each tree $t_i \epsilon [L]$ independently of the other ones. We also need to adjust the definition of the problem MinLCPattern (and consequently the algorithm MinLCP) to accommodate the fact that $L$ is a bag. In MinLCPattern, we treat a constraint $\sigma \in LF$ as “at least one leaf has the label $\sigma$,” while a constraint $\sigma \in nLF$ means “none of the leaves have the label $\sigma$.”

The algorithm MinLCP is mostly unchanged, except for the handling of the constraints in the set LF, which is done as follows. For each $\sigma \in LF$, we define a special new label $\sigma_i$ that replaces $\sigma$ in LF and also replaces one occurrence of $\sigma$ in $L$ (which is now a bag); of course, we also add $\sigma_i$ to $dom(S)$. In the procedure Initialize, we set $\tau^\sigma[\sigma_i \epsilon [\Lambda]]$ to singletonRTree($\sigma$) for each $\sigma_i \in LF$ and $\pi$, such that $S \models \pi - \sigma$.

Finally, we mention the generalization to patterns with directed edges. In [13], it is shown how to define lb-constraints and schemas for the directed case. Theorem 3.1 can be extended to directed schemas. The needed adjustments will be given in the full version.

9. Conclusions

This work continues [13], where we laid the framework for query extraction from a schema with neighborhood constraints, gave an FPT reduction from MinPattern to MinLBC (with the bag $\Lambda$ of labels as the parameter), and devised FPT algorithms for MinLBC under two languages of lb-constraints, namely, regular expressions and circular-arc $\mu$X graphs. The algorithm presented here is an FPT reduction from TopPatterns to MinLBExC. Straightforward adaptations of the algorithms of [13] for MinLBC give FPT algorithms for MinLBExC, and thus an FPT algorithm for TopPatterns, under the above two languages. The main components of our algorithm are an adaptation of Lawler-Murty’s procedure, serialization of a non-redundant $(S, \Lambda)$-pattern, reduction of prefix optimization to the problem MinLCPattern that entails leaf constraints, and finally an algorithm for MinLCPattern. An important future goal is to implement this algorithm and optimize it to the level of practicality needed for dealing with large schemas. For instance, an interesting question is to what extent parallelization techniques for (the ordinary) Lawler-Murty’s procedure [6] can improve efficiency of the overall algorithm.

10. References