Extracting Minimum-Weight Tree Patterns from a Schema with Neighborhood Constraints

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

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

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.

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).

EDBT/ICDT '13, March 18 - 22 2013, Genoa, Italy.

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 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 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.

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[1] 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 `manages` 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 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 $S(\sigma)$.

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 Λ of labels (derived from a user's query), find a minimum-weight (S, Λ)-*pattern*—a tree t such that the bag of labels in t forms a superbag of Λ, 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, Λ)-patterns, rather than just one. In this paper, we consider TOPPATTERNS, which is the problem of generating k minimum-weight (S, Λ)-patterns. Unlike the case of a (single) minimum-weight (S, Λ)-pattern, it is not even clear how to define the meaning of "k minimum-weight (S, Λ)-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 Λ of labels is large, whereas in practice the size of Λ 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 Λ 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[2] or "circular-arc $\overline{\text{mux}}$ 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, Λ)-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, Λ)-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, Λ)-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 subtree that is also an (S, Λ)-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, Λ)-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[3] [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 Λ 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 Λ is a set (rather than a general bag). In Section 5, we devise a *serialization* of a non-redundant (S, Λ)-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, Λ)-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]

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

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

[3] Lawler [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 Λ 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 MINLBEXC and defined in Section 2.8. Our algorithm is FPT if there is an FPT solver of MINLBEXC 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 $\overline{\text{mux}}$ 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 $\uplus$ 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 $[\![B]\!]$, as the set of bags

$$[\![B]\!] \stackrel{\text{def}}{=} \bigcup_{b \in B} \{b' \mid b' \subseteq b\}\,,$$

that is, $[\![B]\!]$ comprises all the bags of B and their subbags.

2.2 Labels and Regular Expressions

We fix an infinite set $\mathbf{\Sigma}$ of *labels*. The set of all the finite bags of labels is denoted by $\mathcal{F}_{\mathbf{\Sigma}}$. We assume that there is a total order $\prec$ on $\mathbf{\Sigma}$. To simplify our complexity analysis, we further assume that $\sigma_1 \prec \sigma_2$ can be tested in constant time (given σ_1 and σ_2).

Regular expressions over $\mathbf{\Sigma}$ are defined by the language

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

where $\sigma \in \mathbf{\Sigma}$ and ϵ is the empty string. A regular expression e defines the language $\mathcal{L}(e)$, which is the set of all the strings over $\mathbf{\Sigma}$ that match the expression e. We define the *bag language* $\mathcal{L}^{\text{b}}(e)$ of e in the standard way (e.g., as in [1]): $\mathcal{L}^{\text{b}}(e)$ is the set of all the bags $b \in \mathcal{F}_{\mathbf{\Sigma}}$, such that the elements of b can be ordered to form a word in $\mathcal{L}(e)$ (in other words, for some string $x \in \mathcal{L}(e)$, 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 variant of the problem we study. For a graph g, the sets of nodes and edges are denoted by $\mathsf{V}(g)$ and $\mathsf{E}(g)$, respectively. Note that an edge of $\mathsf{E}(g)$ is a set $\{u, v\} \subseteq \mathsf{V}(g)$ where $u \neq v$. 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 $\lambda^g(U) \in \mathcal{F}_{\mathbf{\Sigma}}$. 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.

A *tree* is a connected and acyclic graph. A *leaf* is a node v that is incident to at most one edge. A non-leaf node is *internal*. We use $\text{leaves}(t)$ and $\text{internal}(t)$ 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 $\mathcal{F}_{\mathbf{\Sigma}}$. 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 $v \in \mathsf{V}(g)$ it holds that $\lambda(v) \in dom(S)$ and $\lambda(\mathsf{nbr}(v)) \in S(\lambda(v))$. 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.

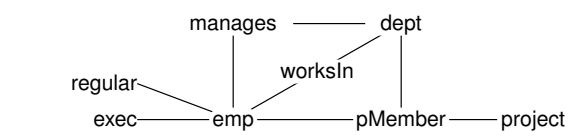
For complexity analysis, we assume that an lb-constraint is represented by an *lb-specification*, which is a finite string in the language of the computational model at hand. In [13], several languages of lb-specifications are discussed. In one of them, a regular expression e specifies the lb-constraint $\mathcal{L}^{\text{b}}(e)$; 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 σ to the lb-constraint $\{\{\!\{\tau_1, \tau_2\}\!\}, \{\!\{\tau_1, \tau_2, \tau_2\}\!\}\}$. Thus, if a graph g conforms to S, then a node labeled with σ must have either two or three neighbors, such that exactly one is labeled with τ_1 and the others—with τ_2. However, a query may refer to only some neighbors; for example, "find edges, such that one endpoint is labeled with σ and the other—with τ_2." We can express this query by the pattern consisting of v_1 and v_2 connected by an edge and labeled with σ and τ_2, respectively. This pattern does not conform to the schema S, because the node labeled with σ 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 $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$ 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 *Λ-pattern* if $\Lambda \subseteq \lambda(\mathsf{V}(t))$, and an *$(S, \Lambda)$-pattern* if t is both an S-pattern and a Λ-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., `exec`, `regular`, etc.). For all $\sigma \in dom(S')$, the lb-constraint $S'(\sigma)$ comprises all the finite bags over the neighbors of σ in the graph s'. For example, $S'(\texttt{dept})$ is the set that consists of every bag that consists of `manages`, `worksIn` and `pMember`, each appearing zero or more times. Let Λ be the bag at the bottom of Figure 1, namely $\{\!\{\texttt{exec}, \texttt{project}, \texttt{regular}\}\!\}$. Every pattern in Figure 2 is an (S', Λ)-pattern (that describes a relationship involving a regular employee, an executive, and a project).



regular, exec : emp
dept : manages worksIn* pMember*
manages, worksIn : dept emp
emp : (regular | (exec manages*)) worksIn* pMember*
pMember : (emp | dept) project
project : pMember*

$$\forall \sigma, \tau \in dom(S):\ W(\sigma) = 0\,,\ W(\sigma, \tau) = 1$$

$$\Lambda = \{\!\{\texttt{exec}, \texttt{project}, \texttt{regular}\}\!\}$$

Figure 1: Schema S (given by regular expressions) and bag Λ

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 $\mathcal{L}^{\mathrm{b}}(e_\sigma)$, where e_σ is the regular expression assigned to σ in the middle of Figure 1. For example, $S(\texttt{emp}) = \mathcal{L}^{\mathrm{b}}(e)$, where e is the expression $(\texttt{regular} \mid (\texttt{exec manages}^*))\ \texttt{worksIn}^*\ \texttt{pMember}^*$. In particular, note that none of the bags of $S(\texttt{emp})$ contain both `regular` and `exec`, and hence, t'_1 is not an S-pattern. Similarly, none of the bags of $S(\texttt{emp})$ contain both `regular` and `manages`, and therefore t'_2 is not an S-pattern as well. On the other hand, the reader can verify that each t_i $(1 \leq i \leq 4)$ is an (S, Λ)-pattern. □

2.4 Non-Redundant Patterns

To define of a *non-redundant* pattern, we fix a label $\star \in \mathbf{\Sigma}$ that is assumed to be never used in the schema S and the bag Λ of labels. 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 $\star$. A Λ-pattern is *basic* if it has no proper subtree that is also a Λ-pattern (i.e., removing any leaf results in a non-Λ-pattern). A Λ-pattern is *non-redundant* if some output designation converts it into a basic Λ-pattern.[4]

EXAMPLE 2.3. Consider again the Λ-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 Λ-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 $\Lambda' = \{\!\{\texttt{exec}, \texttt{regular}\}\!\}$, only t_1 is a non-redundant (and also basic) Λ'-pattern, while each of t_2, t_3 and t_4 is a redundant Λ'-pattern. □

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

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

[4]In [13], basic and non-redundant Λ-patterns are called *non-redundant* and *weakly non-redundant*, respectively.

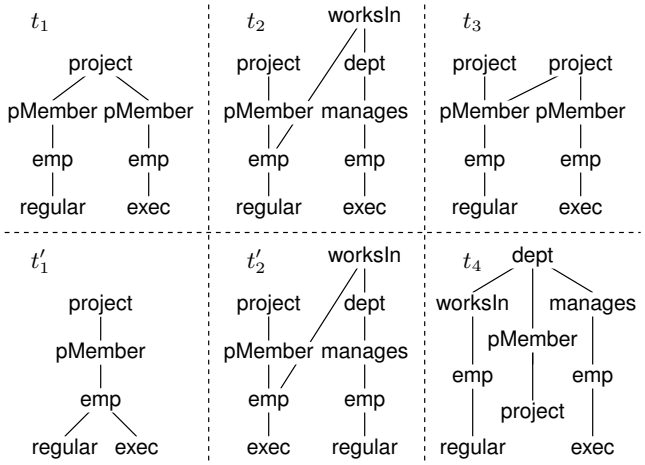


Figure 2: Non-redundant Λ-patterns t_i under S (Figure 1), and t'_i under S' (Example 2.2), for $\Lambda = \{\!\{\texttt{exec}, \texttt{project}, \texttt{regular}\}\!\}$

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 $W : dom(S) \cup (dom(S) \times dom(S)) \to [0, \infty)$ that is symmetric over $dom(S) \times dom(S)$. By *symmetric* we mean 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. 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 $W'(\texttt{project}, \texttt{pMember}) = 2$ and $W'(\texttt{pMember}, \texttt{project}) = 2$. Now, consider the trees t_1, t_2 and t_3 of Figure 2. The reader can easily verify that $W'(t_1) = 4 * 1 + 2 * 2 = 8$, $W'(t_2) = 7 * 1 + 1 * 2 = 9$, and $W'(t_3) = 4 * 1 + 3 * 2 = 10$. □

In [13], we studied the problem MINPATTERN, namely, finding a minimal pattern.

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

2.6 Top-k Patterns

Consider a 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 (S, Λ)-patterns if $|T| = k$ and all of the following hold.

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

If there are only $k' < k$ non-redundant and pairwise non-isomorphic (S, Λ)-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 Λ are those of Figure 1. Now, consider the patterns t_i in Figure 2. The reader can verify that t_1 is the minimal (S, Λ)-pattern. Hence, $\{t_1\}$ is the unique top-1 set. Similarly, the second-minimal (S, Λ)-pattern is t_3, and hence, $\{t_1, t_3\}$ is a top-2 set. Finally, each of t_2 and t_4 is a third-minimal (S, Λ)-pattern, and therefore both $\{t_1, t_3, t_2\}$ 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, Λ)-pattern) to the problem of finding a top-k set of (S, Λ)-patterns.

PROBLEM 2.8. The problem TOPPATTERNS accepts as input a weighted schema S^W, a bag $\Lambda \in \mathcal{F}_{\mathbf{\Sigma}}$ and a natural number k. The goal is to find a top-k set of (S, Λ)-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, Λ)-pattern is already NP-hard [13]. Nevertheless, in practical scenarios the input bag Λ 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), $p(n)$ 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 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 $\{\!\{\Delta_1, \ldots, \Delta_n\}\!\}$, such that each Δ_i is a triple (b_i, τ_i, w_i) in $\mathcal{C}$, $\Gamma \subseteq \biguplus_{i=1}^n b_i$ 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. The problem *minimum labeled-bag exact cover*, which we denote by MINLBEXC, is the same as MINLBC, except that the legal cover $\{\!\{\Delta_1, \ldots, \Delta_n\}\!\}$ is now required to be such that Γ is *equal to* (and not just *a subbag of*) $\biguplus_{i=1}^n b_i$.

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 MINLBEXC. Hence, we have the following theorem.[5]

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

[5]The same result holds for the case where lb-specifications are given as *circular-arc mux graphs* [13].

3. MAIN RESULT

The main result of this paper is that TOPPATTERNS is FPT whenever MINLBEXC is FPT.

THEOREM 3.1. *The following holds for any language of lb-specifications. If* MINLBEXC *(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 Λ 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* MINLBEXC. For a language of lb-specifications, *containment checking* is the problem of testing whether $b \in [\![\mathcal{B}]\!]$, when the input is a bag $b \in \mathcal{F}_{\mathbf{\Sigma}}$ and an lb-constraint $\mathcal{B}$ 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 MINLBEXC. 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 Λ is a *set* rather than a general *bag*, that is, the multiplicity of each label in Λ is 1. In Section 8, we show how to extend our results to a general Λ.

4. ADAPTING LAWLER-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 Γ_x and a weight function $w_x : \Gamma_x^* \to \mathbb{R} \cup \{\infty\}$, where Γ_x^* is the set of all finite strings over Γ_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. $w_x(\mathbf{t}) \leq w_x(\mathbf{s})$ for all $\mathbf{t} \in T$ and $\mathbf{s} \in \Gamma_x^* \setminus T$.

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 Γ_x can be computed in polynomial time in $\|x\|$. Second, the weight $w_x(\mathbf{s})$, where $\mathbf{s}$ is a string in Γ_x^*, can be computed in polynomial time in $\|x\|$ and $|\mathbf{s}|$ (the latter is the length of the string $\mathbf{s}$).

COMMENT 4.1. *We allow ∞ 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, assuming that every string has a weight (even though it could be ∞) results in a simpler abstraction.* □

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

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 $\mathbf{u}$. 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 $\mathbf{u} \in \Gamma_x^*$, find a string $\mathbf{t} \in \Gamma_x^*$ such that:

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

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 $\mathbf{u} \in \Gamma_x^*$, and outputs a string $A(x, \mathbf{u}) \in \Gamma_x^*$. To simplify our complexity analysis, we assume that A returns the weight of $A(x, \mathbf{u})$ in addition to the string $A(x, \mathbf{u})$ itself (hence, there is no need to explicitly account for the cost of computing the weight).

In our adaptation of Lawler-Murty's reduction, a top-strings problem is solved by means of a prefix optimizer. The role of the prefix $\mathbf{u}$ is to encode the constraints that the solution $\mathbf{t} = A(x, \mathbf{u})$ must satisfy. (Note that the size $\mathbf{u}$ 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 $A(x, \mathbf{u})$. We use the monotone functions f and g in order to state upper bounds on the former and the latter, respectively. The size of $A(x, \mathbf{u})$ affects the running time, because $A(x, \mathbf{u})$ 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 $f(\|x\|, |\mathbf{u}|)$ and $|A(x, \mathbf{u})| < g(\|x\|) + |\mathbf{u}|$. Then in $\mathcal{O}\left(k^2 \cdot |\Gamma_x| \cdot g(\|x\|) \cdot f(\|x\|, k \cdot g(\|x\|))\right)$ 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, Λ), where S^W is a weighted schema and Λ is a set of labels. For $x = (S^W, \Lambda)$, we define the alphabet Γ_x to be $dom(S) \cup \{\lhd\}$, where $\lhd$ is a special symbol that is not in $\mathbf{\Sigma}$. Later, we will show how the symbol $\lhd$ will be used for representing the structure of the tree (rather than the labels on its nodes). The strings in Γ_x^* represent the objects to be generated by TOPPATTERNS, namely, non-redundant (S, Λ)-patterns. Therefore, we should define how a non-redundant (S, Λ)-pattern is *serialized*, that is, encoded as a string over Γ_x. Actually, we define a function ser that maps every non-redundant Λ-pattern t to a string over Γ_x, provided that t has only labels of $dom(S)$. If a string $\mathbf{s}$ represents a non-redundant (S, Λ)-pattern t (i.e., $\mathbf{s} = ser(t)$), then its weight $w_x(\mathbf{s})$ is $W(t)$; otherwise, $w_x(\mathbf{s}) = \infty$.

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

Next, we explain the mapping ser. Suppose that t is a non-redundant Λ-pattern with $q + 1$ leaves. By our assumption (in Section 3) that Λ 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 Λ. Recall that $\prec$ is a total order over $\mathbf{\Sigma}$. 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 $\mathbf{l}(t)$ and use it to construct $ser(t)$ 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_1 from v_0 to v_1. Then, we continue from the first unvisited node of p_2 to v_2, 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 ith 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 Λ of Figure 1. In our examples, we assume that $\prec$ is the dictionary order; particularly, `dept` $\prec$ `exec` $\prec$ `project` $\prec$ `regular`. Consider the trees of Figure 3 (where t_1 and t_3 are also in Figure 2). The trees

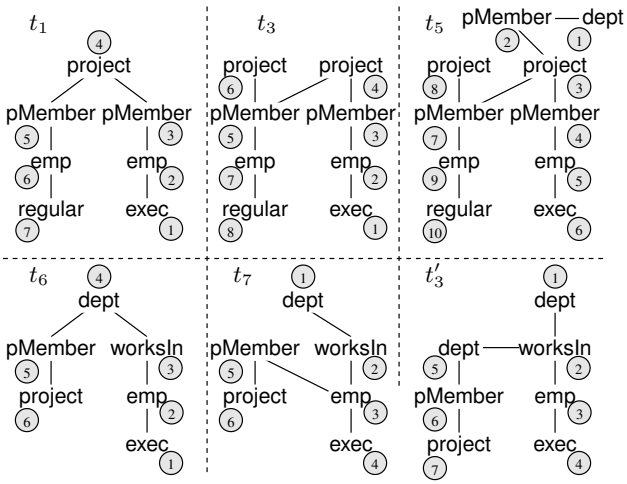


Figure 3: Visit times in ascending-leaf traversal

t_1 and t_3 are non-redundant Λ-patterns, and t_5 is a non-redundant Λ'-pattern for $\Lambda' = \Lambda \cup \{\texttt{dept}\}$. The trees t_6, t_7 and t'_3 are all non-redundant Λ''-patterns for $\Lambda'' = \{\texttt{dept}, \texttt{exec}, \texttt{project}\}$. Next to each node u in Figure 3, $\mathrm{time}(u)$ is written in a grey circle. □

An easy observation is that except for v_0, every node u of t has a unique neighbor u' such that $\mathrm{time}(u') < \mathrm{time}(u)$; we define $\Delta(v_0) = 1$ and $\Delta(u) = \mathrm{time}(u) - \mathrm{time}(u')$. Finally, the string $ser(t)$ is obtained by iterating over the nodes u of t in ascending $\mathrm{time}(u)$, and for each u writing down $\Delta(u) - 1$ occurrences of $\lhd$ followed by the label $\lambda(u)$. (Recall that $\lhd$ is the special symbol that belongs to every Γ_x but not to $\mathbf{\Sigma}$.) As becomes apparent in the next example, if a label in $ser(t)$ is followed by a $\lhd$, then its corresponding node in t is a leaf.

EXAMPLE 5.2. Consider again the trees of Figure 3.

$ser(t_1) = \texttt{exec}, \texttt{emp}, \texttt{pMember}, \texttt{project}, \texttt{pMember}, \texttt{emp}, \texttt{regular}$

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

$$ser(t_3) = \texttt{exec}, \texttt{emp}, \texttt{pMember}, \texttt{project}, \texttt{pMember}, \texttt{project}, \lhd, \texttt{emp}, \texttt{regular}$$

For t_5 we have:

$$ser(t_5) = \texttt{dept}, \texttt{pMember}, \texttt{project}, \texttt{pMember}, \texttt{emp}, \texttt{exec} \lhd, \lhd, \lhd, \texttt{pMember}, \texttt{project}, \lhd, \texttt{emp}, \texttt{regular}$$

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 `dept`. The reason is that $\texttt{dept} \prec \texttt{exec}$ and `dept` is a leaf in t_7 but not in t_6. □

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

PROPOSITION 5.3. *Two non-redundant Λ-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.

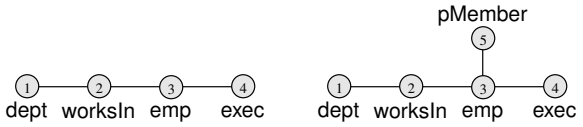


Figure 4: The left part is $\mathrm{tree}(\texttt{dept}, \texttt{worksIn}, \texttt{emp}, \texttt{exec})$, which is also $\mathrm{tree}(\texttt{dept}, \texttt{worksIn}, \texttt{emp}, \texttt{exec}, \lhd)$; the right part is $\mathrm{tree}(\texttt{dept}, \texttt{worksIn}, \texttt{emp}, \texttt{exec}, \lhd, \texttt{pMember})$

PROBLEM 5.4. For instances $x = (S^W, \Lambda)$, prefix optimization is the following problem. Given a weighted schema S^W, a set Λ of labels and a string $\mathbf{u} \in \Gamma_x^*$, find a minimal-weight, non-redundant (S, Λ)-pattern t such that $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t)$, or determine that no such t exists.[6] □

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

Let $\mathbf{u} = \texttt{exec}, \texttt{emp}$. Then t_6 satisfies $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t_6)$. But for t_7 it is not true that $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t_7)$, since $ser(t_7)$ begins with `dept`. It can be shown that t_6 is a minimal (S, Λ)-pattern t satisfying $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t)$; therefore, t_6 is a solution to the prefix optimization for the input S^W, Λ and $\mathbf{u}$.

Let $\mathbf{v} = \texttt{dept}, \texttt{worksIn}, \texttt{emp}, \texttt{exec}, \lhd$. The tree t_7 satisfies $\mathbf{v} \preceq_{\mathrm{prfx}} ser(t_7)$. Now, $\mathbf{v} \preceq_{\mathrm{prfx}} ser(t'_3)$ is also true, but t'_3 is not an S-pattern because the `worksIn` node has two *dept* neighbors.

Next suppose that $\mathbf{w} = \texttt{dept}, \texttt{worksIn}, \texttt{emp}, \texttt{exec}, \lhd, \lhd$. Then $\mathbf{w} \preceq_{\mathrm{prfx}} ser(t'_3)$, but again, t'_3 is not an S-pattern. Actually, it is easy to show that no (S, Λ)-pattern t satisfies $\mathbf{w} \preceq_{\mathrm{prfx}} ser(t)$. □

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, Λ)-pattern under *leaf constraints*. So, for this section, we fix the input S^W, Λ and $\mathbf{u}$. Recall that the goal is to find a minimum-weight,[7] non-redundant[8] (S, Λ)-pattern t such that $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t)$, or determine that no such t exists.

6.1 Consistency

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

Testing consistency can be done by trying to construct a subtree of a non-redundant Λ-pattern from $\mathbf{u}$, as described next. If we succeed, then we denote this subtree by $\mathrm{tree}(\mathbf{u})$.

We say that a label is *ordinary* if it is different from $\lhd$. Let $\mathbf{u}$ have n occurrences of ordinary labels. We partition $\mathbf{u}$ into segments $s_1, \ldots, s_n$. The segment s_i begins at the ith occurrence of

[6] If no such t exists, then formally we return $\mathbf{u}$, 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 nodes 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 $\lhd$. 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] $\mathrm{index}(v_i) = i$ and set $\mathrm{cnt}(v_i)$ to the number of $\lhd$ in s_i. We say that v_i is a *definite leaf* if $i = 1$ or $\mathrm{cnt}(v_i) > 0$. We create an edge $\{v_i, v_j\}$, where $i < j$, if $\mathrm{index}(v_i) = \mathrm{index}(v_j) - \mathrm{cnt}(v_{j-1}) - 1$. 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 $\mathrm{index}(v_j) - \mathrm{cnt}(v_{j-1}) - 1 \leq 0$. Second, it creates a tree t that is inconsistent with $\mathbf{u}$. This happens if there are v_j and v_i, such that $\mathrm{index}(v_i) = \mathrm{index}(v_j) - \mathrm{cnt}(v_{j-1}) - 1$ and v_i is a definite leaf. It also happens if $\mathrm{cnt}(v_n) > 0$, but for every v that is not a definite leaf, $\mathrm{index}(v) > \mathrm{index}(v_n) - \mathrm{cnt}(v_n) - 1$.

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

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

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

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 lb-specifications.

Suppose that $\mathbf{u}$ is a consistent prefix for Λ. The goal is to expand $\mathrm{tree}(\mathbf{u})$ into a minimum-weight (S, Λ)-pattern $t_{\min}$, such that $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t_{\min})$. Expanding $\mathrm{tree}(\mathbf{u})$ in one particular way is done by choosing some nodes of $\mathrm{tree}(\mathbf{u})$ 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 $t_{\min}$. Conceptually, we do it in two stages. Firstly, we create all possible subtrees for each node v of $\mathrm{tree}(\mathbf{u})$ and, secondly, we consider all combinations of those subtrees.

There are several constraints that should be satisfied by each expansion of $\mathrm{tree}(\mathbf{u})$. 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 Λ-pattern. We should choose subtrees so that together with $\mathrm{tree}(\mathbf{u})$ they form an exact cover of Λ. We do it by using a MINLBEXC solver (see Section 2.8).

The input to the MINLBEXC solver is obtained as follows. For each node v of $\mathrm{tree}(\mathbf{u})$, 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, L)-pattern t' that satisfies some constraints (as explained later), and w is the weight of t'. Thus, L is the set of labels that t' can contribute to an exact cover of Λ if we add t' to $\mathrm{tree}(\mathbf{u})$ (by merging its root[10] with node v). Note that the labels l_v are needed only for the MINLBEXC solver, and we assume that they do not appear Γ^*_x, where x is the instance (S^W, Λ).

We first explain intuitively how the triples (L, l_v, w) are created and then give the formal details. For example, if v is a node of $\mathrm{tree}(\mathbf{u})$, such that its label $\lambda(v)$ is in Λ, then we create the triple $(\{\lambda(v)\}, l_v, 0)$. This triple means that v can contribute $\lambda(v)$ to the exact cover just by itself; that is, without adding a subtree at v and, hence, the weight is 0. Observe that if v is a definite leaf of $\mathbf{u}$, then this is the only triple we can create for v, because v cannot have children in any expansion t of $\mathrm{tree}(\mathbf{u})$, such that $\mathbf{u} \preceq_{\mathrm{prfx}} ser(t)$.

Let v' and v'' be two consecutive leaves in an ascending-leaf traversal of some tree t. The fact that v'' is the next leaf after v' follows just from the labels of the leaves; it is not affected at all by the relative positions of v' and v'' in t. Thus, if we add some new subtrees to t, we can continue an existing ascending-leaf traversal of t provided that two conditions are satisfied. First, the new subtrees are not rooted at existing leaves. Second, the labels of all new leaves are greater (under $\prec$) than those of all existing leaves.

However, in our case, the situation is a bit more complicated, because we want to continue a partial traversal, which is given by the prefix $\mathbf{u}$. So, for example, if $k_{\mathrm{last}} > 0$ (i.e., $\mathbf{u}$ ends with one or more $\lhd$), then a new node that is added at the end of $\mathbf{u}$ can be connected to an existing non-leaf node v only if $\mathrm{index}(v) < \mathrm{index}(v_{\mathrm{last}}) - k_{\mathrm{last}}$. This means the following. Suppose that we add some subtrees to existing non-leaf nodes. Then the subtree that has the next leaf (in the traversal) can be connected only to some of the existing non-leaf nodes, but the other subtrees can be connected to any one of them.

Suppose that the last label of $\mathbf{u}$ is that of v_{last} (i.e., $k_{\mathrm{last}} = 0$). The first label l added at the end of $\mathbf{u}$ makes v_{last} a leaf (if $l = \lhd$) or continues the path to the next leaf in the traversal (if l is ordinary). Either way (similarly to the case of $k_{\mathrm{last}} > 0$), the initial expansion of $\mathbf{u}$ determines the next leaf (after the last definite leaf of $\mathbf{u}$) or is on the path that leads to the next leaf, which we denote by v_{next}. After reaching v_{next}, we can continue the traversal to the other new leaves provided that their labels are greater than $\lambda(v_{\mathrm{next}})$.

It thus follows that the label of the first definite leaf v_{next} after those already in $\mathbf{u}$, determines which triples (L, l_v, w) should be added to the input for the MINLBEXC solver. (That label also implies some constraints that will be described shortly.) Therefore, we should consider all the possible choices for the label ξ of $\lambda(v_{\mathrm{next}})$; for each one of them, we should create the appropriate input for the MINLBEXC solver, and then find a minimum-weight (S, Λ)-pattern $t^{\xi}_{\min}$. The desired (S, Λ)-pattern $t_{\min}$ is a minimum pattern among all the $t^{\xi}_{\min}$. As choices for ξ, we should consider every label in $dom(S)$ that is greater than all the labels of the definite leaves of $\mathrm{tree}(\mathbf{u})$.

Next, we fully describe how to create the triples (L, l_v, w) once we choose a particular ξ as the label of v_{next}. In most triples, the weight is initially unknown and we denote it by $\perp$. Each triple $(L, l_v, \perp)$ corresponds to a minimum-weight L-pattern[11] $t^{\xi}_v[L]$ that satisfies some constraints, as explained in the next section. We find $t^{\xi}_v[L]$ using the algorithm of Section 7, and its weight w updates the triple to (L, l_v, w).

Let D be the set of labels that appear in the definite leaves. Next are the four rules for creating triples for a given ξ.

Rule 1. For each node $v \neq v_{\mathrm{last}}$, such that $\lambda(v) \in \Lambda$ and $\lambda(v) \neq \xi$, we create the triple $(\{\lambda(v)\}, l_v, 0)$. We apply this rule also to v_{last} if it is a definite leaf (in this case $\lambda(v_{\mathrm{last}}) \prec \xi$).

As explained earlier, a triple $(\{\lambda(v)\}, l_v, 0)$ means that v can contribute $\lambda(v)$ to the exact cover just by itself; that is, without adding a subtree at v and, hence, the weight is 0. This is the only

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

[10] In this section, we use the term "root" informally. We fully define it in the next section.

[11] A triple does not mention the ξ for which it was created, to avoid cumbersome notation. However, ξ appears in the corresponding pattern $t^{\xi}_v[L]$, because ξ imposes some constraints on $t^{\xi}_v[L]$, as explained later.

rule that applies to definite leaves. If v_{last} is not a definite leaf, then only the third rule applies to it. Applying this rule requires that $\lambda(v) \neq \xi$, because ξ 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 v_{last} nor a definite leaf, we create the triples $(L, l_v, \perp)$ for all nonempty subsets L of $\Lambda - D$ such that $\xi \notin L$.

This rule says that nodes of $\text{tree}(\mathbf{u})$ 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 ξ nor the label of any definite leaf.

Rule 3. If v_{last} is not a definite leaf, we create $(L, l_{v_{\text{last}}}, \perp)$ for every L such that $\xi \in L \subseteq \Lambda - D$. If $\lambda(v_{\text{last}}) = \xi$, then we also create the triple $(\{\xi\}, l_{v_{\text{last}}}, 0)$, because v_{last} can contribute ξ just by itself.

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

Rule 4. If v_{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 $\text{index}(v) < \text{index}(v_{\text{last}}) - k_{\text{last}}$. 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 v_{next}. As explained earlier, these patterns can be merged only with some of the non-leaf nodes of $\text{tree}(\mathbf{u})$. Similarly to the previous rule, the leaves of $t_v^\xi[L]$ cannot have any label of D, but one of them must be labeled with ξ.

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 $t_v^\xi[L]$ (see Section 7). The triple $(L, l_v, \perp)$ is updated to (L, l_v, w), where $w = W(t_v^\xi[L]) - W(\lambda(v))$. We subtract $W(\lambda(v))$ from the weight of $t_v^\xi[L]$, because v and the root of $t_v^\xi[L]$ are going to be merged and, hence, the weight of these two nodes should be counted only once. Let $\mathcal{C}^\xi$ be the resulting set of triples.

The input to the MINLBEXC solver comprises $\mathcal{C}^\xi$, Λ (i.e., the set to be covered) and the lb-constraint $\mathcal{B}$ 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 $\mathcal{B}$ by the regular expression $l_{v_1}?l_{v_2}?\ldots l_{v_n}?$, where $v_1, \ldots, v_n$ are the nodes of $\text{tree}(\mathbf{u})$. The four rules guarantee that only the definite leaves can contribute their labels to the exact cover; moreover, ξ will be the label of the next leaf (after the last definite leaf of $\text{tree}(\mathbf{u})$) in the ascending-leaf traversal of the expansion $t_{\min}^\xi$ of $\text{tree}(\mathbf{u})$.

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 $\text{tree}(\mathbf{u})$ are unchanged. The loop of line 3 iterates over all the labels ξ of P. For each ξ, 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 $t_v^\xi[L]$ that corresponds to $(L, l_v, \perp)$. Line 7 tests if $t_v^\xi[L]$ 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 $t_v^\xi[L]$ that corresponds to a triple in the exact cover, line 13 merges the root of $t_v^\xi[L]$ with node v of $\text{tree}(\mathbf{u})$. The resulting tree is assigned to $t_{\min}^\xi$. Line 14 returns a minimum-weight pattern among all the $t_{\min}^\xi$ that were found.

Algorithm PrefixToLeaf$(S^W, \Lambda, \mathbf{u})$

1: $P \leftarrow \{\xi \in \Lambda \mid$ for all labels l of definite leaves, $l \prec \xi\}$
2: Create a unique label l_v for each $v \in \mathsf{V}(\text{tree}(\mathbf{u}))$ {These labels are needed for the exact cover}
3: **for all** labels $\xi \in P$ **do**
4: Construct the set of triples $\mathcal{C}^\xi$ (rules in the text)
5: **for all** $(L, l_v, \perp) \in \mathcal{C}^\xi$ **do**
6: compute the corresponding $t_v^\xi[L]$ (alg. in Sec. 7)
7: **if** $t_v^\xi[L]$ exists **then**
8: replace $\perp$ with the weight of $t_v^\xi[L]$
9: **else**
10: delete $(L, l_v, \perp)$ from $\mathcal{C}^\xi$
11: $M^\xi \leftarrow$ a minimal exact cover of Λ from $\mathcal{C}^\xi$, such that each l_v appears at most once
12: **if** M^ξ exists **then**
13: $t_{\min}^\xi \leftarrow$ the expansion of $\text{tree}(\mathbf{u})$ with the patterns corresponding to the triples of M^ξ
14: **return** a $t_{\min}^\xi$ with the minimum weight

Figure 5: General reduction of prefix optimization to minimum-weight pattern under leaf constraints

In the next section, we precisely define the trees $t_v^\xi[L]$.

6.3 Constrained Minimal Patterns

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

First, we need to verify that $\text{tree}(\mathbf{u})$ is an S-pattern; if not, it cannot be expanded into an (S, Λ)-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 $\text{tree}(\mathbf{u})$ is an S-pattern by reducing this problem to MINLBEXC. Thus, for each node v of $\text{tree}(\mathbf{u})$, we construct the following instance of MINLBEXC. There is a triple $(\{\sigma\}, \sigma, 1)$ for each occurrence of a label σ in the bag $\lambda(\mathsf{nbr}(v))$. The bag to be covered is $\lambda(\mathsf{nbr}(v))$ and the lb-constraint is $S(\lambda(v))$. Clearly, there is an exact cover if and only if $\lambda(\mathsf{nbr}(v)) \in [\![S(\lambda(v))]\!]$.

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

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

$$\mathcal{B}_{-b} \stackrel{\text{def}}{=} \{b' \mid b \uplus b' \in \mathcal{B}\}. \tag{2}$$

Finally, $\mathcal{B}_r = S(l_r)_{-b}$.

Recall that in the third and fourth rules of the previous section, ξ is required to be the label of a leaf in $t_v^\xi[L]$. Moreover, by the definition of ξ, all labels $l \in \Lambda$, such that $l \prec \xi$, can appear only in non-leaf nodes of $t_v^\xi[L]$. Thus, we use two sets $\mathsf{LF}, \mathsf{nLF} \subseteq L$

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 $t_v^{\xi}[L]$ should satisfy $\mathsf{LF} \subseteq \lambda(\text{leaves}(t_v^{\xi}[L])) \subseteq L \setminus \mathsf{nLF}$. In summary, $\mathsf{nLF} = \{l \mid l \in \Lambda \wedge l \prec \xi\}$ and $\mathsf{LF} = \{\xi \mid \xi \in L\}$ (i.e., LF is empty if ξ 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 l_r, an lb-constraint $\mathcal{B}_r$ for the root, and two sets $\mathsf{LF}, \mathsf{nLF} \subseteq L$ of leaf and non-leaf constraints, respectively. An (S, L)-pattern t with a root v_r is *satisfactory* if the following conditions hold.

1. $\lambda(v_r) = l_r$ and $\lambda(\mathsf{nbr}(v_r)) \in [\![\mathcal{B}_r]\!]$.
2. $\mathsf{LF} \subseteq \lambda(\text{leaves}(t)) \subseteq L \setminus \mathsf{nLF}$.

In Condition 2, the meaning of $\subseteq$ is bag containment; in particular, $\lambda(\text{leaves}(t)) \subseteq L \setminus \mathsf{nLF}$ implies that no two leaves of t have the same label. Note that Condition 2 implies non-redundancy with respect to Λ. In MINLCPATTERN, the goal is to find a satisfactory (S, L)-pattern of a minimum weight.

A MINLCPATTERN *solver* is an algorithm for MINLCPATTERN. The input for the solver consists of six arguments: S^W, L, l_r, $\mathcal{B}_r$, LF and nLF. Each of L, LF and nLF is a subset of Λ. Recall that $\mathcal{B}_r = S(l_r)_{-b}$, where b is the bag of labels of the neighbors of a node in $\text{tree}(\mathbf{u})$. We assume that $\mathcal{B}_r$ is represented by b, and we check $\hat{b} \in [\![\mathcal{B}_{-b}]\!]$ by testing $\hat{b} \uplus b \in [\![\mathcal{B}]\!]$. We further assume that $|b| \leq |\Lambda|$, because the degree of any node in a non-redundant (S, Λ)-pattern is at most $|\Lambda|$. Therefore, we only need to bound the running time of the solver by a function of S^W and Λ. The next lemma gives the formal result of this section.

LEMMA 6.1. *Consider a language for lb-specifications. If a* MINLCPATTERN *solver runs in time bounded by $h(\|S^W\|, |\Lambda|)$, then there is a prefix optimizer A for* TOPPATTERNS*, such that the following hold for a given input comprising $x = (S^W, \Lambda)$ and $\mathbf{u}$.*

1. *A runs in time bounded by a function $f(\|S^W\|, |\Lambda|, |\mathbf{u}|)$ that is a polynomial in $h(\|S^W\|, |\Lambda|)$, $2^{|\Lambda|}$ and $|\mathbf{u}|$.*
2. *$|A(x, \mathbf{u})| \leq g(\|x\|) + |\mathbf{u}|$ where $g(\|x\|) = 2|dom(S)|^2|\Lambda|$.*

Part 1 of the lemma uses Theorem 2.9, because PrefixToLeaf (Figure 5) calls a MINLBEXC solver with a regular expression as the lb-constraint. Note that the function $f(\|S^W\|, |\Lambda|, |\mathbf{u}|)$ in Lemma 6.1 has the same role as the function $f(\|x\|, |\mathbf{u}|)$ in Theorem 4.3, except that in the former x is broken into its two components (S^W and Λ) to describe different dependencies of f on their sizes. Part 2 requires the observation that no matter which MINLCPATTERN solver is used, we can always post-process its output so that the total number of nodes that our prefix optimizer (i.e., the algorithm of Figure 5) adds to $\text{tree}(\mathbf{u})$ is at most $2|dom(S)|^2|\Lambda|$.

7. HANDLING LEAF CONSTRAINTS

Next, we describe the algorithm MinLCP of Figure 6 for solving MINLCPATTERN. This algorithm extends FindMinPattern of [13] by also handling leaf constraints.[12]

The input consists of a weighted schema S^W, a set L of labels, a root label l_r, an lb-constraint $\mathcal{B}_r$, and leaf constraints $\mathsf{LF}, \mathsf{nLF} \subseteq \Lambda$. The goal is to find a satisfactory (S, L)-pattern of a minimal weight. We assume that $\mathsf{LF} \cap \mathsf{nLF} = \emptyset$ or else there is no such an (S, L)-pattern.

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

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

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

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

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

The main idea for updating $\mathcal{T}^{\pi}[\sigma, N]$ is to find h trees $t_i = \mathcal{T}^{\sigma}[\tau_i, M_i]$ $(1 \leq i \leq h)$, such that the M_i are a minimal exact cover of N. The triples for this instance of MINLBEXC are $(M_i, \tau_i, W(t_i))$ and the lb-constraint is $S(\sigma)_{-\{\!\{\pi\}\!\}}$. We should verify that $S \models \pi\text{–}\sigma$ and the leaf constraints hold. If so, we create a new root v_r with the label σ, connect v_r to the roots of the t_i and assign the result to $\mathcal{T}^{\pi}[\sigma, N]$.

The pseudo code of MinLCP (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 $\mathcal{Q}$ to the empty priority queue. The loop of line 2 iterates over all the pairs π and σ. Lines 3–5 initialize $\mathcal{T}^{\pi}[\sigma, N]$ to $\perp$ and insert $\langle \pi, \sigma, N \rangle$ into $\mathcal{Q}$ for all nonempty subsets N of L. Finally, lines 6–7 check whether σ is in L but not in nLF; if so, $\mathcal{T}^{\pi}[\sigma, \{\sigma\}]$ is assigned a new tree t that consists of a single node v_r, such that v_r is the root and its label is σ. We assume that the procedure singletonRTree(σ) constructs and returns that tree. However, before line 7 sets $\mathcal{T}^{\pi}[\sigma, \{\sigma\}]$ to t, line 6 also checks that $S \models \pi\text{–}\sigma$.

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

The third argument $S(\sigma)_{-\{\!\{\pi\}\!\}}$ in line 5 is represented by $\{\!\{\pi\}\!\}$, because σ is already the second argument of Update; similarly, for

[12] The notation we use in this section is similar, but not identical, to that of [13].

the call in line 6 (as explained in Section 6.3, the argument $\mathcal{B}_r$ is actually $S(l_r)_{-b}$ for some (small) bag b of labels).

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

The search for $t_1, \ldots, t_h$ is done by a reduction to the following instance I of MINLBEXC. Lines 1–5 of Update construct the set $\mathcal{C}$ that consists of all triples (M, τ, w), such that $\mathcal{T}^\sigma[\tau, M] \neq \perp$ and $w = W(\mathcal{T}^\sigma[\tau, M])$. The bag of labels (of the instance I) is a nonempty subset N of L and the lb-constraint is $\mathcal{B}$. The triples of a legal cover correspond to the entries of $\mathcal{T}^\sigma$ that form the trees $t_1, \ldots, t_h$ mentioned earlier. To simplify the pseudo code, line 6 simultaneously solves the MINLBEXC problem for all nonempty subsets N of L. Hence, L (rather than some N) is the first argument of the call to ExactMinCovers in line 6. That call returns the array $MinCovers$ that has an index N for every nonempty subset N of L. $MinCovers[N]$ 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 $\mathcal{T}^\pi[\sigma, N]$. Here, four cases are considered.

- $\sigma \notin N$. In this case, t is constructed from the trees that correspond to the triples of $MinCovers[N]$—the minimal exact cover of N.

- $\sigma \in N \cap \mathsf{LF}$ and $|N| > 1$. We do the same as in the previous case. Note that if $\sigma \in N \cap \mathsf{LF}$ and $|N| = 1$, then $\mathcal{T}^\pi[\sigma, N]$ is assigned its final value in the initialization. Furthermore, since we create the tree only from $MinCovers[N]$ (and never from $MinCovers[N \setminus \{\sigma\}]$), an induction shows that σ must be the label of a leaf.

- $\sigma \in N \setminus (\mathsf{LF} \cup \mathsf{nLF})$ and $|N| > 1$. In this case, t is constructed from the minimal cover among $MinCovers[N]$ and $MinCovers[N \setminus \{\sigma\}]$. Note that if $\sigma \in N \setminus (\mathsf{LF} \cup \mathsf{nLF})$ and $|N| = 1$, then $\mathcal{T}^\pi[\sigma, N]$ is assigned its final value in the initialization.

- $\sigma \in N \cap \mathsf{nLF}$ and $|N| > 1$. We do the same as in the previous case. Note that when $\sigma \in N \cap \mathsf{nLF}$, the initialization does not create a tree for σ. Hence, it is easy to show by induction that the algorithm never creates a tree having a leaf labeled with σ.

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 $\mathcal{T}^\pi[\sigma, N]$ 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_r and the lb-constraint $\mathcal{B}_r$ that the root should satisfy. There is no need to require that $\mathrm{root}(t)$ can have an additional neighbor labeled with some π. Hence, the call in line 6 does not have a test of the form $S \models \pi\text{–}\sigma$ (in contrast to line 4). Furthermore, the first argument of Update (in line 6) is the special symbol $\star$ which is not in $dom(S)$. Note that $\star$ is only needed to indicate that Update assigns its result to $\mathcal{T}^\star[l_r, L]$. 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 MinLCP$(S^W, L, l_r, \mathcal{B}_r, \mathsf{LF}, \mathsf{nLF})$

```
1: Initialize()
2: while Q ≠ ∅ do
3:    ⟨σ, σ', N⟩ ← Q.pop()
4:    for all π ∈ dom(S) where S ⊨ π–σ do
5:       Update(π, σ, S(σ)_{−{{π}}})
6: Update(⋆, l_r, B_r)
7: return T^⋆[l_r, L]
```

Subroutine Initialize()

```
1: Q ← empty priority queue
2: for all π, σ ∈ dom(S) do
3:    for all nonempty N ⊆ L do
4:       T^π[σ, N] ← ⊥
5:       Q.insert(⟨π, σ, N⟩)
6:    if σ ∈ L \ nLF and S ⊨ π–σ then
7:       T^π[σ, {σ}] ← singletonRTree(σ)
```

Subroutine Update$(\pi, \sigma, \mathcal{B})$

```
1: C ← ∅
2: for all τ ∈ dom(S) and M ⊆ L do
3:    t ← T^σ[τ, M]
4:    if t ≠ ⊥ then
5:       C ← C ∪ {(M, τ, W(t))}
6: MinCovers ← ExactMinCovers(L, B, C)
7: for all nonempty N ⊆ L do
8:    if σ ∉ N or (σ ∈ LF and |N| > 1) then
9:       T^π[σ, N] ← Assmbl(σ, MinCovers[N])
10:   if σ ∈ N \ LF and |N| > 1 then
11:      T^π[σ, N] ← min. of Assmbl(σ, MinCovers[N]) and
         Assmbl(σ, MinCovers[N \ {σ}])
```

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

```
1: if C' = ⊥ then
2:    return ⊥
3: t ← singletonRTree(σ)
4: for all (M, τ, w) ∈ C' do
5:    t_M ← T^σ[τ, M]
6:    add t_M as a subtree of t immediately below root(t)
7: return t
```

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

7.1 Correctness and Efficiency

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

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

The next theorem states the correctness of 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 ExactMinCovers,

which is a solver for MINLBEXC that is discussed in Section 2.8. Recall our assumptions (in Section 6.3) that $L, \mathsf{LF}, \mathsf{nLF} \subseteq \Lambda$, and that $\mathcal{B}_r = S(l_r)_{-b}$ is represented by a bag b, where $|b| \leq |\Lambda|$.

THEOREM 7.2. *As output,* $\mathsf{MinLCP}(S^W, L, l_r, \mathcal{B}_r, \mathsf{LF}, \mathsf{nLF})$ *returns a minimal-weight satisfactory* (S, L)*-pattern or* $\perp$ *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 Λ is a set.

8. GENERALIZING THE ALGORITHM

We first discuss how to generalize our main result to the case where Λ is a bag. In Section 5, we use $\mathbf{l}(t)$ to denote the unique string of the leaves of an (S, Λ)-pattern t, according to the order $\prec$ defined on $\mathbf{\Sigma}$. When Λ is a bag, $\mathbf{l}(t)$ 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, Λ)-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 $ser^*(t)$ before we can find any string that represents an (S, Λ)-pattern having a larger weight than that of t. However, it is easy to show that for all (S, Λ)-patterns t, the cardinality of $ser^*(t)$ is bounded by $\times\Lambda$, which is the product of the multiplicities over the different elements of Λ. As a result, we are guaranteed to find a top-k set of (S, Λ)-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 Λ is a bag, we can still find each tree $t_v^{\xi}[L]$ 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 $\sigma \in \mathsf{LF}$ as "*at least one* leaf has the label σ," while a constraint $\sigma \in \mathsf{nLF}$ means "*none* of the leaves have the label σ." The algorithm MinLCP is mostly unchanged, except for the handling of the constraints in the set LF, which is done as follows. For each $\sigma \in \mathsf{LF}$, we define a special new label σ_l that replaces σ in LF and also replaces one occurrence of σ in L (which is now a bag); of course, we also add σ_l to $dom(S)$. In the procedure Initialize, we set $\mathcal{T}^{\pi}[\sigma, \{\!\{\sigma_l\}\!\}]$ to $\mathsf{singletonRTree}(\sigma)$ for each $\sigma_l \in \mathsf{LF}$ and π, such that $S \models \pi\text{–}\sigma$.

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 Λ of labels as the parameter), and devised FPT algorithms for MINLBC under two languages of lb-constraints, namely, regular expressions and circular-arc $\overline{\text{mux}}$ 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, Λ)-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

[1] C. Beeri and T. Milo. Schemas for integration and translation of structured and semi-structured data. In *ICDT*, pages 296–313. Springer, 1999.

[2] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *ICDE*, pages 431–440. IEEE, 2002.

[3] S. Cohen, Y. Kanza, B. Kimelfeld, and Y. Sagiv. Interconnection semantics for keyword search in XML. In *CIKM*, pages 389–396. ACM, 2005.

[4] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

[5] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940. ACM, 2008.

[6] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Optimizing and parallelizing ranked enumeration. *PVLDB*, 4(11):1028–1039, 2011.

[7] M. Grohe and J. Flum. *Parameterized Complexity Theory*. Theoretical Computer Science. Springer, 2006.

[8] H. Hamacher and M. Queyranne. K-best solutions to combinatorial optimization problems. *Annals of Operations Research*, 4:123–143, 1985/6.

[9] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *VLDB*, pages 670–681. Morgan Kaufmann, 2002.

[10] A. Kemper, D. Kossmann, and B. Zeller. Performance tuning for SAP R/3. *IEEE Data Eng. Bull.*, 22(2):32–39, 1999.

[11] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182. ACM, 2006.

[12] 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, 2006.

[13] B. Kimelfeld and Y. Sagiv. Finding a minimal tree pattern under neighborhood constraints. In *PODS*, pages 235–246. ACM, 2011.

[14] B. Kimelfeld, Y. Sagiv, and G. Weber. ExQueX: exploring and querying XML documents. In *SIGMOD Conference*, pages 1103–1106. ACM, 2009.

[15] E. L. Lawler. A procedure for computing the k best solutions to discrete optimization problems and its application to the shortest path problem. *Management Science*, 18(7):401–405, 1972.

[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] K. G. Murty. An algorithm for ranking all the assignments in order of increasing cost. *Operations Research*, 16(3):682–687, 1968.

[20] L. Qin, J. X. Yu, and L. Chang. Keyword search in databases: the power of RDBMS. In *SIGMOD Conference*, pages 681–694. ACM, 2009.

[21] 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.

[22] 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.

[23] J. Y. Yen. Finding the k shortest loopless paths in a network. *Management Science*, 17:712–716, 1971.

[24] 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.