Reverse Engineering SPARQL Queries*

Marcelo Arenas
PUC Chile
marenas@ing.puc.cl

Gonzalo I. Diaz
University of Oxford
gonzalo.diaz@cs.ox.ac.uk

Egor V. Kostylev
University of Oxford
egor.kostylev@cs.ox.ac.uk

ABSTRACT

Semantic Web systems provide open interfaces for end-users to access data via a powerful high-level query language, SPARQL. But users unfamiliar with either the details of SPARQL or properties of the target dataset may find it easier to query *by example* — give examples of the information they want (or examples of both what they want and what they do not want) and let the system *reverse engineer* the desired query from the examples. This approach has been heavily used in the setting of relational databases. We provide here an investigation of the reverse engineering problem in the context of SPARQL. We first provide a theoretical study, formalising variants of the reverse engineering problem and giving tight bounds on its complexity. We next explain an implementation of a reverse engineering tool for positive examples. An experimental analysis of the tool shows that it scales well in the data size, number of examples, and in the size of the smallest query that fits the data. We also give evidence that reverse engineering tools can provide benefits on real-life datasets.

1. INTRODUCTION

Semantic Web systems provide open interfaces for end-users to access data within standard formats. The data model views data as collections of RDF triples, a fairly low-level representation, but the Web APIs expose a powerful declarative query language, SPARQL, which allows users to pose queries that combine and filter information. Declarative query languages are powerful, but they are known to have disadvantages in terms of ease of use. An alternative paradigm for querying is "query by example", where users present examples of what they want, and the system generalises them [8, 22, 23]. Querying by example is particularly attractive in an open data setting, since to harness the power of these interfaces users must understand the structure of data as well as the features of SPARQL needed to express their information needs, and this is frequently not the case. Even users familiar with SPARQL and with a given dataset may prefer to explore the data via example and have the system suggest generalisations.

In this paper we give the first study of the problem of querying via examples for SPARQL. We formalise the problem as "reverse engineering SPARQL queries from examples", following a line of research that has been developed for other problems dealing with learning from examples [10], including regular languages [1–3], relational database queries [25], XML queries [9, 20]. and queries in graph databases [7]. A common baseline for these approaches lies in the definability problem [4, 6, 14, 24].

We present several variations of the problem, depending on whether the user presents a dataset with positive examples, a dataset with positive and negative examples, or a dataset with an exact answer. We also vary the subset of SPARQL that the system is permitted to synthesise, starting with simple graph patterns, appending the SPARQL operator for obtaining optional information, and finally considering an extension with SPARQL's feature for filtering results according to some conditions.

Our first contribution is theoretical: we isolate the complexity of all variants of the reverse engineering problem. We provide tight complexity bounds for the reverse engineering problem with positive examples, with positive and negative examples, and with an exact answer, looking at reverse engineering queries in fragments of SPARQL ranging from very expressive (allowing all the operators mentioned above) to very limited (allowing only conjunction).

Having completed the picture of the theory of reverse engineering, we turn to a practical implementation of it. We provide a parameterised algorithm for reverse engineering from positive examples, where the parameters allow tuning several features of the target class. We evaluate our algorithms on real-world and synthetic queries, showing that it scales well both with the input size and the complexity of a target query needed to match a set of examples, that it can reverse engineer complex queries with high accuracy, and that it can be useful as a supplement to an existing SPARQL engine.

2. PRELIMINARIES

2.1 RDF and the query language SPARQL

The RDF (Resource Description Framework) data model is used to represent information about World Wide Web resources, and was released as a W3C (World Wide Web Consortium) recommendation in 2004 [12]. Along with RDF, the W3C defined the query language SPARQL as a recommendation for querying RDF data. We shall present only a simplified version of the full definitions, in line with the formalisation given by [15].

Assume two countably infinite disjoint sets U and L of *IRIs* and *literals*, respectively. An *(RDF) triple* is a tuple $(s, p, o) \in \mathsf{U} \times \mathsf{U} \times$

*The authors would like to thank Michael Benedikt for many fruitful discussions about the results presented in this paper. M. Arenas was funded by Millennium Nucleus Center for Semantic Web Research under Grant NC120004, and G. I. Diaz by Becas Chile of CONICYT Chile.

WWW 2016, April 11–15, 2016, Montréal, Québec, Canada.
ACM 978-1-4503-4143-1/16/04.
http://dx.doi.org/10.1145/2872427.2882989 .

$(\mathsf{U} \cup \mathsf{L})$, and an *(RDF) graph* is a finite set of RDF triples. Define another countably infinite set V of *variables*, disjoint from U and L (variables will be denoted with a question mark).

Next we define the fragment of the SPARQL query language that will be considered in this paper. We start by introducing the notion of (SPARQL) *built-in condition*, defined inductively as follows:

- if $?X, ?Y \in \mathsf{V}$ and $a \in (\mathsf{U} \cup \mathsf{L})$, then $?X = a$, $?X = ?Y$, and $\mathsf{bound}(?X)$ are built-in conditions,
- if R_1 and R_2 are built-in conditions, then $(\neg R_1)$, $(R_1 \vee R_2)$, and $(R_1 \wedge R_2)$ are built-in conditions.

The notion of (SPARQL) *graph pattern* is inductively defined next:

- a triple from $(\mathsf{U} \cup \mathsf{L} \cup \mathsf{V}) \times (\mathsf{U} \cup \mathsf{V}) \times (\mathsf{U} \cup \mathsf{L} \cup \mathsf{V})$ is a graph pattern (called *triple pattern*),
- if P_1 and P_2 are graph patterns, then $(P_1 \,\mathbf{AND}\, P_2)$ and $(P_1 \,\mathbf{OPT}\, P_2)$ are graph patterns,
- if P is a graph pattern and R is a built-in condition, then $(P \,\mathbf{FILTER}\, R)$ is a graph pattern.

EXAMPLE 1. *Assume that* $?X \in \mathsf{V}$, $\{\mathtt{type}, \mathtt{Person}, \mathtt{age}\} \subseteq \mathsf{U}$, *and "32"* $\in \mathsf{L}$. *Then* $P_1 = (?X, \mathtt{type}, \mathtt{Person})$ *and* $P_2 = (?X, \mathtt{type}, \mathtt{Person}) \,\mathbf{AND}\, (?X, \mathtt{age}, \text{"32"})$ *are graph patterns, which intuitively ask for the list of people (elements of type person) and for the list of people whose age is 32. Moreover, the following is also a graph pattern:* $P_3 = [(?X, \mathtt{type}, \mathtt{Person}) \,\mathbf{AND}\, (?X, \mathtt{age}, \text{"32"})] \,\mathbf{OPT}\, (?X, \mathtt{email}, ?Y)$, *where the* **OPT** *operator is used to retrieve the email of each person stored in the variable* $?X$ *if this information is available.*

The SPARQL query language includes also union, projection and some additional built-in predicates; these features of SPARQL are not considered, and are left for future work. To distinguish the entire SPARQL query language from the fragment considered in this paper, the latter is denoted by $\mathrm{SP}[\mathsf{AOF}]$, where A, O and F stand for the operators **AND**, **OPT** and **FILTER**, respectively.

To define the semantics of graph patterns, we define the notion of *mapping*, a partial function from V to $\mathsf{U} \cup \mathsf{L}$ with finite domain, which is denoted by $\mathrm{dom}(\mu)$. Two mappings μ_1 and μ_2 are *compatible*, denoted $\mu_1 \sim \mu_2$, if $\mu_1(?X) = \mu_2(?X)$ for all $?X \in \mathrm{dom}(\mu_1) \cap \mathrm{dom}(\mu_2)$ (i.e. when $\mu_1 \cup \mu_2$ is also a mapping). Given sets Ω_1 and Ω_2 of mappings, define the following operations on them:

$$\Omega_1 \bowtie \Omega_2 = \{\mu_1 \cup \mu_2 \mid \mu_1 \in \Omega_1, \mu_2 \in \Omega_2 \text{ and } \mu_1 \sim \mu_2\},$$
$$\Omega_1 - \Omega_2 = \{\mu \mid \mu \in \Omega_1 \text{ and } \forall \mu_2 \in \Omega_2 : \mu_1 \not\sim \mu_2\}.$$

We start by defining the semantics of built-in conditions. Given a built-in condition R and a mapping μ, we say that μ satisfies R, denoted by $\mu \models R$, if

- R is $?X = a$ for some $?X \in \mathsf{V}$, $a \in \mathsf{U} \cup \mathsf{L}$, and $\mu(?X) = a$,
- R is $?X = ?Y$ for some $?X, ?Y \in \mathsf{V}$ and $\mu(?X) = \mu(?Y)$,
- R is $\mathsf{bound}(?X)$ for some $?X \in \mathsf{V}$ and $?X \in \mathrm{dom}(\mu)$,
- R is $(\neg R_1)$ for a built-in condition R_1 and $\mu \not\models R_1$,
- R is $(R_1 \vee R_2)$ for some R_1, R_2 and $\mu \models R_1$ or $\mu \models R_2$,
- R is $(R_1 \wedge R_2)$ for R_1, R_2, and both $\mu \models R_1$ and $\mu \models R_2$.

We now move to the definition of the semantics of graph patterns. Given a graph pattern P, we define $\mathrm{var}(P) \subseteq \mathsf{V}$ to be the set of variables which occur in P, and likewise for a built-in condition R. Given a triple pattern t and a mapping μ such that $\mathrm{var}(t) \subseteq \mathrm{dom}(\mu)$, we define $\mu(t)$ as the RDF triple obtained from t by replacing every variable $?X$ occurring in t by $\mu(?X)$. Then given a graph pattern P and an RDF graph D, the evaluation of P over D, denoted by $[\![P]\!]_D$, is the set of mappings defined recursively as follows:

- If P is a triple pattern t, then $[\![P]\!]_D = \{\mu \mid \mathrm{var}(t) = \mathrm{dom}(\mu) \text{ and } \mu(t) \in D\}$,
- If $P = (P_1 \,\mathbf{AND}\, P_2)$ for graph patterns P_1 and P_2, then $[\![P]\!]_D = [\![P_1]\!]_D \bowtie [\![P_2]\!]_D$,
- If $P = (P_1 \,\mathbf{OPT}\, P_2)$ for graph patterns P_1 and P_2, then $[\![P]\!]_D = ([\![P_1]\!]_D \bowtie [\![P_2]\!]_D) \cup ([\![P_1]\!]_D - [\![P_2]\!]_D)$,
- If $P = (P_1 \,\mathbf{FILTER}\, R)$ for a graph pattern P_1 and a built-in condition R, then $[\![P]\!]_D = \{\mu \in [\![P_1]\!]_D \mid \mu \models R\}$.

From now on, we use notation $[?X_1 \mapsto a_1, \ldots, ?X_k \mapsto a_k]$ to represent a mapping μ such that $\mathrm{dom}(\mu) = \{?X_1, \ldots, ?X_k\}$ and $\mu(?X_i) = a_i$ for all $i \in [1, k]$.

Note that, as can be shown by simple induction on the structure, for any graph D and pattern P in $\mathrm{SP}[\mathsf{AOF}]$ the semantics is such that any two different mappings in the set of mappings $[\![P]\!]_D$ are incompatible, and the intersection of the domains of mappings in $[\![P]\!]_D$ is non-empty. These two properties play an important role in the paper, and we call sets of mappings satisfying them *consistent*.

EXAMPLE 2. *Continuing with Example 1, consider an RDF graph* D *consisting of following triples:*

$(\mathtt{John}, \mathtt{type}, \mathtt{Person})$, $(\mathtt{John}, \mathtt{age}, \text{"26"})$,
$(\mathtt{Peter}, \mathtt{type}, \mathtt{Person})$, $(\mathtt{Peter}, \mathtt{age}, \text{"32"})$,
$(\mathtt{Susan}, \mathtt{type}, \mathtt{Person})$, $(\mathtt{Susan}, \mathtt{age}, \text{"32"})$,
$(\mathtt{Susan}, \mathtt{email}, \text{"susan@example.org"})$.

Then we have that $[\![P_1]\!]_D$ *consists of the mappings* $[?X \mapsto \mathtt{John}]$, $[?X \mapsto \mathtt{Peter}]$ *and* $[?X \mapsto \mathtt{Susan}]$, *which correspond to the list of people in* D. *Moreover, we have that* $[\![P_2]\!]_D = \{[?X \mapsto \mathtt{Peter}], [?X \mapsto \mathtt{Susan}]\}$, *as the age of* `John` *is "26". Finally, we have that* $[\![P_3]\!]_D$ *consists of the mappings*

$\mu_1 = [?X \mapsto \mathtt{Peter}]$,
$\mu_2 = [?X \mapsto \mathtt{Susan}, ?Y \mapsto \text{"susan@example.org"}]$.

On the one hand, we do not have the email of `Peter` *in* D, *so only variable* $?X$ *is assigned with a value in the mapping* μ_1. *On the other hand, the email of* `Susan` *is in* D, *so we have a value for variable* $?Y$ *in the mapping* μ_2.

Finally, the *size* of a graph pattern P, denoted by $\mathsf{size}(P)$, is the number of triple patterns and atomic filter conditions in P.

2.2 Well-designed patterns

Previous work on SPARQL has identified some anomalies that arise when the **OPT** operator can be used arbitrarily, and [15] showed that one can avoid these anomalies by making a natural restriction on the use of **OPT**. A graph pattern P is said to be *safe* if for every sub-pattern $(P_1 \,\mathbf{FILTER}\, R)$ of P, it holds that $\mathrm{var}(R) \subseteq \mathrm{var}(P_1)$. A graph pattern P is said to be *well-designed* if P is safe and for every sub-pattern P_1 of P of the form $(P_L \,\mathbf{OPT}\, P_R)$, and for every variable $?X \in \mathrm{var}(P_R)$, if $?X$ is mentioned outside of P_1 in P, then $?X \in \mathrm{var}(P_L)$. We denote $\mathrm{SP}[\mathsf{AOFwd}]$ the class of all the well-designed graph patterns.

EXAMPLE 3. *The graph pattern* P_3 *defined in Example 1 is well-designed. On the other hand, if*

$$P_4 = ((?Y, \mathtt{type}, \mathtt{Publication}) \,\mathbf{OPT}\, (?X, \mathtt{email}, ?Z)),$$

then the graph pattern $P_5 = (?X, \mathtt{type}, \mathtt{Person}) \,\mathbf{AND}\, P_4$ *is not well-designed. To see why this is the case, notice that the variable* $?X$ *is mentioned in the right-hand side of* P_4 *and outside* P_4 *in the triple* $(?X, \mathtt{type}, \mathtt{Person})$, *but it is not mentioned in the left-hand side of* P_4. *What is unnatural about the graph pattern* P_5 *is that the triple* $(?X, \mathtt{email}, ?Z)$ *has been placed to give optional information to the triple* $(?Y, \mathtt{type}, \mathtt{Publication})$, *but it is actually giving optional information to the outside triple* $(?X, \mathtt{type}, \mathtt{Person})$.

Empirical studies have shown that well-designed graph patterns are commonly used in practice [16]. Well-designed patterns have many desirable properties. First, the complexity of the query evaluation problem for well-designed graph patterns is lower than for the entire language [15] (coNP versus PSpace), even if only the **OPT** operator is considered [19]. Second, well-designed graph patterns are suitable for query optimisation [11, 15, 17]; in particular, this restriction allows the definition of some simple reordering and optimisation rules [11, 15, 19]. Third, this class of graph patterns captures the intuition behind the **OPT** operator, which is to allow information to be added to an answer whenever it is available, and not to reject such an answer if the information is not present. This intuition, which is formalised in the following paragraph, does not hold for an arbitrary query in $\mathrm{SP}[\mathsf{AOF}]$.

A mapping μ_1 is *subsumed by* a mapping μ_2, denoted by $\mu_1 \sqsubseteq \mu_2$, if $\mathrm{dom}(\mu_1) \subseteq \mathrm{dom}(\mu_2)$ and $\mu_1(?X) = \mu_2(?X)$ for every variable $?X \in \mathrm{dom}(\mu_1)$. Assume that we have two RDF graphs D_1 and D_2 such that $D_1 \subseteq D_2$. If a SPARQL pattern P mentioning only the operators **AND** and **FILTER** is evaluated over D_1 and D_2, then we have that $[\![P]\!]_{D_1} \subseteq [\![P]\!]_{D_1}$. Thus, such a query P is monotone in the sense that if new information is added to an RDF graph then no answer is lost. If we are also allowed to use the **OPT** operator in P, then we would expect a similar behaviour, which is referred to as *weak monotonicity* [5]. More precisely, given that $D_1 \subseteq D_2$, it should be the case that for every mapping $\mu_1 \in [\![P]\!]_{D_1}$, there exists a mapping $\mu_2 \in [\![P]\!]_{D_2}$ such that $\mu_1 \sqsubseteq \mu_2$, as the **OPT** operator was designed only to add information to an answer whenever it is available. However, there exist patterns in $\mathrm{SP}[\mathsf{AOF}]$ that are not weakly monotone [5]; in fact, P_5 in Example 3 is such a query. Well-designed graph patterns come as a solution to this fundamental problem, as shown in the following proposition.

PROPOSITION 1 ([5]). *Every* $\mathrm{SP}[\mathsf{AOFwd}]$ *graph pattern is weakly monotone.*

Last but not least, the semantics of well-designed graph patterns can be characterised in terms of the notion of subsumption of mappings, which is a useful property that will be utilised in this paper. A graph pattern P' is a *reduction* of a graph pattern P if P' can be obtained from P by replacing a sub-pattern $(P_1 \,\mathbf{OPT}\, P_2)$ of P by P_1, that is, if P' is obtained by deleting some optional part of P. The reflexive and transitive closure of the reduction relation is denoted by $\trianglelefteq$. Moreover, $\mathsf{and}(P)$ is the graph pattern obtained from P by replacing every **OPT** operator in P by the **AND** operator. Then the set of partial answers of a graph pattern P over an RDF graph D, denoted by $\mathsf{partials}(P, D)$, is the set of all mappings μ for which there exists $P' \trianglelefteq P$ such that $\mu \in [\![\mathsf{and}(P')]\!]_D$. As shown in the following proposition, partial answers and the notion of subsumption of mappings can be used to characterise the evaluation of a well-designed graph pattern.

PROPOSITION 2 ([15]). *For every RDF graph D, graph pattern P in* $\mathrm{SP}[\mathsf{AOFwd}]$ *and mapping μ, it holds that $\mu \in [\![P]\!]_D$ if and only if μ is a maximal mapping (with respect to $\sqsubseteq$) in* $\mathsf{partials}(P, D)$.

Well-designed patterns also admit **OPT** *normal form*, in which arguments of **AND** and **FILTER** do not use **OPT** [15]; this normalisation can be performed in polynomial time. Moreover, in well-designed patterns the bound operator is moot, as in **OPT** normal form it can be always replaced by True [26]. In this paper we assume normalised well-designed patterns without bound.

Besides the class $\mathrm{SP}[\mathsf{AOFwd}]$ of all well-designed patterns, we also study the reverse engineering problem for the fragment $\mathrm{SP}[\mathsf{AOwd}]$ of well-designed graph patterns formed using only **AND** and **OPT**, and the fragment $\mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]$ of $\mathrm{SP}[\mathsf{AOFwd}]$ obtained by disallowing the use of disjunction in filter expression and restricting the use of negation to only inequalities.

2.3 Complexity classes

In the study of the computational complexity of the reverse engineering problems, we consider the usual complexity classes PTime, NP and coNP, along with the complexity classes Σ_2^p and DP. Recall that if $\mathcal{C}$ is a complexity class, then $\mathsf{NP}^{\mathcal{C}}$ is the class of decision problems that can be solved in polynomial time by a non-deterministic Turing machine with an oracle (or subroutine) for a problem $L \in \mathcal{C}$. Then the second level of the polynomial hierarchy [21] consists of the complexity classes $\Sigma_2^p = \mathsf{NP}^{\mathsf{NP}}$ and $\Pi_2^p = \mathsf{co}\Sigma_2^p$. A prototypical complete problem for Σ_2^p is $\exists\forall 3\textsc{Sat}$, that is, the problem of verifying, given a quantified Boolean formula ϕ of the form $\exists\bar{x}\forall\bar{y}\,\neg\psi$ with ψ a propositional formula (without quantifiers) in conjunctive normal form with each clause using exactly three literals, whether ϕ is valid. DP is the class of problems L for which there exist languages L_1 and L_2 in NP such that $L = L_1 - L_2$ [13] (or, for which there exist L_1 in NP and L_2 in coNP such that $L = L_1 \cap L_2$). The problem $3\textsc{SatUnsat}$ of deciding validity of a quantified formula ϕ of the form $\exists\bar{x}\,\psi_1 \wedge \forall\bar{y}\,\neg\psi_2$, where ψ_1 and ψ_2 are in conjunctive normal form with three literals per clause, is DP-complete. $\mathsf{NP} \subseteq \mathsf{DP}$, $\mathsf{coNP} \subseteq \mathsf{DP}$ and $\mathsf{DP} \subseteq \Sigma_2^p$, and these inclusions are believed to be proper.

3. REVERSE ENGINEERING PROBLEMS

We now formally define the reverse engineering problems considered in this paper. Informally, reverse engineering problems ask whether there exists a query, or *realizer*, which fits the examples. Let $\mathcal{F}$ be any of the SPARQL fragments defined in Section 2.2 (e.g. $\mathrm{SP}[\mathsf{AOFwd}]$). Then the positive examples reverse engineering problem is defined as:

$$\textsc{RevEng}^{+}(\mathcal{F}) = \{(D, \Omega) \mid D \text{ is a nonempty RDF graph, } \Omega \text{ is a set of mappings and } \exists P \in \mathcal{F} : \Omega \subseteq [\![P]\!]_D\}.$$

Moreover, the positive-and-negative examples reverse engineering problem is defined as:

$$\textsc{RevEng}^{\pm}(\mathcal{F}) = \{(D, \Omega, \bar{\Omega}) \mid D \text{ is a non-empty RDF graph, } \Omega, \bar{\Omega} \text{ are sets of mappings and } \exists P \in \mathcal{F} : \Omega \subseteq [\![P]\!]_D \text{ and } \bar{\Omega} \cap [\![P]\!]_D = \varnothing\}.$$

In this case, without loss of generality we always silently assume that $\Omega \cap \bar{\Omega} = \varnothing$, since otherwise the problem is trivial. Finally, the exact-answers reverse engineering problem is defined as:

$$\textsc{RevEng}^{\mathsf{E}}(\mathcal{F}) = \{(D, \Omega) \mid D \text{ is a non-empty RDF graph, } \Omega \text{ is a set of mappings, and } \exists P \in \mathcal{F} : \Omega = [\![P]\!]_D\}.$$

We will call (D, Ω) (and $(D, \Omega, \bar{\Omega})$) an *instance* of the problem and P a *realizer* for the instance.

In order to study the complexity of these reverse engineering problems, we first must understand the corresponding problem of verifying that a given pattern (a.k.a. query) fits some example data. Hence, the positive examples verification problem is defined as:

$$\textsc{Verify}^{+}(\mathcal{F}) = \{(D, P, \Omega) \mid D \text{ is a nonempty RDF graph, } \Omega \text{ is a set of mappings, } P \in \mathcal{F} \text{ and } \Omega \subseteq [\![P]\!]_D\}.$$

The positive-and-negative examples verification problem is:

$$\text{VERIFY}^{\pm}(\mathcal{F}) = \{(D, P, \Omega, \bar{\Omega}) \mid D \text{ is a non-empty RDF graph, } \Omega, \bar{\Omega} \text{ are sets of mappings, } P \in \mathcal{F}, \Omega \subseteq [\![P]\!]_D \text{ and } \bar{\Omega} \cap [\![P]\!]_D = \varnothing\}.$$

Finally, the exact-answers verification problem is defined as:

$$\text{VERIFY}^{\mathsf{E}}(\mathcal{F}) = \{(D, P, \Omega) \mid D \text{ is a non-empty RDF graph, } \Omega \text{ is a set of mappings, } P \in \mathcal{F} \text{ and } \Omega = [\![P]\!]_D\}.$$

Alongside the general versions of the reverse engineering and verification problems, we will consider the case where the number of variables mentioned in Ω (and $\bar{\Omega}$) is bounded by a fixed constant. An upper bound on the complexity of the verification problem when the number of variables is fixed will imply the same upper bound when the pattern is fixed but the data is scaled, i.e. an upper bound in the *data complexity* of verification.

Note that the problems with both positive and negative examples are clearly at least as difficult as their counterparts with only positive examples. However, the exact-answers problems are not immediately reducible to others.

4. COMPUTATIONAL COMPLEXITY OF REVERSE ENGINEERING

Having defined the decision problems to be studied, we now turn to their computational complexity. In this section we will first study the complexity of the auxiliary *verification* problems. While interesting in their own right, the verification problems will be important mainly for their role in obtaining complexity results for the reverse engineering problems. We first give a general overview of the strategy to be used to decide a reverse engineering problem: given a fixed SPARQL fragment $\mathcal{F}$ and a set of example mappings, we may divide the problem into two main challenges:

- Since the fragment $\mathcal{F}$ will usually permit infinitely many queries, we cannot test every query in the fragment as a candidate. Thus, we seek to *bound the set of queries to be considered*. In fact, we will seek to show that if there is any query realising the input, then there is a *canonical realizer*.
- Given a "candidate query" in the fragment, we must decide if it fits the example instance by solving the corresponding verification problem. Then, once we obtain a set of candidate queries it becomes a matter of verifying them one by one.

The algorithmic strategy, then, will be to first construct the canonical query, and then verify that it in fact fits the examples. In what follows we first study the verification problems, and then define the canonical queries in order to study the reverse engineering problems.

4.1 Verification problems

We now examine the complexity of the verification problems, starting with the upper bounds, and progressing towards the results summarised in Table 1. The positive-and-negative examples problem for $\text{SP}[\mathsf{A}]$ admits a straightforward polynomial-time algorithm:

PROPOSITION 3. $\text{VERIFY}^{\pm}(\text{SP}[\mathsf{A}])$ *is in* PTime.

PROOF. For $\text{VERIFY}^{\pm}(\text{SP}[\mathsf{A}])$, given an input $(D, \Omega, \bar{\Omega}, P)$, we must first check that $\Omega \subseteq [\![P]\!]_D$. Thus, for each $\mu \in \Omega$ we first confirm that $\text{dom}(\mu) = \text{var}(P)$ and that $\mu(t) \in D$ for every triple pattern t in P—a simple nested loop. $\bar{\Omega}$ is handled similarly. □

Note that this trivially implies that $\text{VERIFY}^{+}(\text{SP}[\mathsf{A}])$ is in PTime. We defer the case of $\text{VERIFY}^{\mathsf{E}}(\text{SP}[\mathsf{A}])$ for the moment. If we now allow the **OPT** operator, the problem becomes harder. We give an upper bound for $\text{VERIFY}^{+}(\text{SP}[\mathsf{AOFwd}])$; here checking each example mapping μ involves showing that it is maximal, generating the added complexity. In what follows we use the notation $\mu \lneq \nu$ to indicate that mapping μ is properly subsumed by mapping ν, that is $\mu \sqsubseteq \nu$ and $\mu \neq \nu$:

PROPOSITION 4. $\text{VERIFY}^{+}(\text{SP}[\mathsf{AOFwd}])$ *is in* coNP.

PROOF. Consider the following NP algorithm for the complement of $\text{VERIFY}^{+}(\text{SP}[\mathsf{AOFwd}])$. Given input (D, P, Ω) we must decide whether $\Omega \notin [\![P]\!]_D$, for which we use the characterisation provided in Proposition 2. More precisely, for each $\mu \in \Omega$, first check whether $\mu \in \mathsf{partials}(P, D)$; if not return accept (this can be done in polynomial time, as shown in [15]). Otherwise, we now attempt to verify that μ is not a maximal partial solution by guessing a (polynomially-sized) mapping ν, and checking whether $\nu \in \mathsf{partials}(P, D)$ and $\mu \lneq \nu$. If these conditions are all true, then $\mu \notin [\![P]\!]_D$ and we accept. □

Interestingly, if the number of variables in Ω is assumed to be bounded by a fixed constant, then guessing is not necessary and the problem is in PTime. In particular, this implies that the verification problem can be solved in polynomial time in data complexity. Also, note that as a corollary of Proposition 4 we have that $\text{VERIFY}^{+}(\text{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}])$ and $\text{VERIFY}^{+}(\text{SP}[\mathsf{AOwd}])$ are also in coNP, as these two fragments are contained in $\text{SP}[\mathsf{AOwd}]$.

We now show that $\text{VERIFY}^{\mathsf{E}}(\text{SP}[\mathsf{AOFwd}])$ is also in coNP, for which the following characterisation will be useful:

LEMMA 1. *Given an RDF graph D, a set of mappings Ω and a pattern $P \in \text{SP}[\mathsf{AOFwd}]$, it holds that $[\![P]\!]_D \not\subseteq \Omega$ if and only if there exists a mapping $\mu \in \mathsf{partials}(P, D)$ such that (i) $\mu \notin \Omega$ and (ii) for every $\nu \in \Omega$, if $\mu \lneq \nu$ then $\nu \notin \mathsf{partials}(P, D)$.*

PROOF. First assume that there is a mapping $\mu \in \mathsf{partials}(P, D)$ such that both items hold. There are two options for μ:

- Suppose $\mu \in [\![P]\!]_D$. Then, due to the first item, we have that $[\![P]\!]_D \not\subseteq \Omega$.
- Suppose $\mu \notin [\![P]\!]_D$. In this case, since $\mu \in \mathsf{partials}(P, D)$, there exists a mapping $\mu^* \in [\![P]\!]_D$ such that $\mu \lneq \mu^*$. There are two cases to consider for μ^*:
  - if $\mu^* \in \Omega$ then, due to the second item, we have that $\mu^* \notin \mathsf{partials}(P, D)$. Thus, $\mu^* \notin [\![P]\!]_D$, which leads to a contradiction.
  - if $\mu^* \notin \Omega$ then we also conclude that $[\![P]\!]_D \not\subseteq \Omega$.

To prove the other direction of the lemma, assume that $[\![P]\!]_D \not\subseteq \Omega$. Then there exists a mapping $\mu \in [\![P]\!]_D$ such that $\mu \notin \Omega$. By definition we have that $\mu \in \mathsf{partials}(P, D)$ and, thus, we only have to show that the second item of the lemma holds for μ to conclude the proof. Now consider a mapping $\nu \in \Omega$ such that $\mu \lneq \nu$. If $\nu \in \mathsf{partials}(P, D)$, then we obtain a contradiction with the fact that $\mu \in [\![P]\!]_D$, as μ is a maximal partial mapping in $\mathsf{partials}(P, D)$. □

Lemma 1 is the key ingredient to obtaining an upper bound for $\text{VERIFY}^{\mathsf{E}}(\text{SP}[\mathsf{AOFwd}])$:

PROPOSITION 5. $\text{VERIFY}^{\mathsf{E}}$ $(\text{SP}[\mathsf{AOFwd}])$ *is in* coNP.

PROOF. We prove that $\text{VERIFY}^{\mathsf{E}}(\text{SP}[\mathsf{AOFwd}]) \in \text{coNP}$ by showing that the complement of $\text{VERIFY}^{\mathsf{E}}(\text{SP}[\mathsf{AOFwd}])$ is in NP. On input (D, Ω, P), we first check in NP whether $\Omega \notin [\![P]\!]_D$ as

in Proposition 4. If this holds, accept. Otherwise, check whether $[\![P]\!]_D \notin \Omega$. Now by Lemma 1, we must guess a mapping μ such that $\mu \notin \Omega$, $\mathrm{dom}(\mu)$ consists of variables used in Ω, the range of μ consists of the constants used in D and $\mu \in \mathsf{partials}(P, D)$; in particular, notice that μ is of polynomial size in the size of the input. Then, for every $\nu \in \Omega$ such that $\mu \not\equiv \nu$, we verify whether $\nu \notin \mathsf{partials}(P, D)$, which can be done in polynomial time. If this verification succeeds, then we accept. □

As before, if the number of variables in Ω is bounded by a fixed constant, then the resulting problem is again in PTime. Furthermore, as a corollary we have that VERIFY$^{\mathsf{E}}$(SP[AOF$_{\wedge,=,\neq}$wd]), VERIFY$^{\mathsf{E}}$(SP[AOwd]), and VERIFY$^{\mathsf{E}}$(SP[A]) are also in coNP. Our final upper bound is for VERIFY$^{\pm}$(SP[AOFwd]):

PROPOSITION 6. VERIFY$^{\pm}$(SP[AOFwd]) *is in* DP.

PROOF. Let VERIFY^{-}(SP[AOFwd]) be the following decision problem: on input $(D, \bar{\Omega}, P)$, return true if and only if $\bar{\Omega} \cap [\![P]\!]_D = \varnothing$. This problem is easily seen to be in NP: for each mapping $\bar{\mu} \in \bar{\Omega}$, verify that either $\bar{\mu} \notin \mathsf{partials}(D, P)$ (which can be checked in polynomial time) or, if $\bar{\mu} \in \mathsf{partials}(D, P)$, guess a mapping ν and check that $\bar{\mu} \not\equiv \nu$ and $\nu \in \mathsf{partials}(D, P)$, rejecting if this does not hold. Now note that an input $(D, \Omega, \bar{\Omega}, P)$ is in VERIFY$^{\pm}$(SP[AOFwd]) if and only if $(D, \Omega, P) \in$ VERIFY^{+}(SP[AOFwd]) and $(D, \bar{\Omega}, P) \in$ VERIFY^{-}(SP[AOFwd]), from which it is straightforward to conclude that VERIFY$^{\pm}$(SP[AOFwd]) $\in$ DP. □

As a corollary of Proposition 6, we have that VERIFY$^{\pm}$(SP[AOF$_{\wedge,=,\neq}$wd]) and VERIFY$^{\pm}$(SP[AOwd]) are also in DP. Having established complexity upper bounds for all the verification problems, we now turn to the matching lower bounds. We will first show coNP-hardness for VERIFY$^{\mathsf{E}}$(SP[A]), which is also the matching lower bound for VERIFY$^{\mathsf{E}}$(SP[AOwd]), VERIFY$^{\mathsf{E}}$(SP[AOF$_{\wedge,=,\neq}$wd]), and VERIFY$^{\mathsf{E}}$(SP[AOFwd]).

PROPOSITION 7. VERIFY$^{\mathsf{E}}$(SP[A]) *is* coNP*-hard.*

PROOF SKETCH. This can be shown via a polynomial-time many-to-one reduction from the complement of the 3-COLOURABILITY problem, which checks if an undirected graph (not an RDF graph) is 3-colourable. Given an input $G = (V, E)$ to the complement of the 3-colouring problem (where V is a set of nodes and E is a set of edges), we construct an input (D, Ω, P) to the verification problem, with $D = \bigcup_{i,j \in [1,3], i \neq j} \{(c_i, e, c_j)\}$ encoding permitted edges between colours, P encoding the edges of graph G (each edge $(a, b) \in E$ becoming a triple pattern $(?X_a, e, ?Y_b)$ in P), and $\Omega = \varnothing$. The graph is non-3-colourable if and only if there is no assignment of variables from the triple patterns in P to the colours in D, resulting in $[\![P]\!]_D = \varnothing$, whereby $\Omega \subseteq [\![P]\!]_D$ will hold. □

The proposition above completes the picture for the complexity of verification problems for SP[A], as well as the VERIFY$^{\mathsf{E}}$ problem for all the fragments considered. We now continue with the lower bounds on verification problems for SP[AOwd] and SP[AOFwd]:

PROPOSITION 8. VERIFY^{+}(SP[AOwd]) *is* coNP*-hard.*

The proof is a variant of Claim 4.7 in [15], which shows that the *evaluation* problem (given (D, μ, P), check whether $\mu \in [\![P]\!]_D$) for SP[AOwd] is coNP-hard, and is thus omitted here. Again, as a corollary we have that VERIFY^{+}(SP[AOF$_{\wedge,=,\neq}$wd]) and VERIFY^{+}(SP[AOFwd]) are also coNP-hard.

We conclude this section with the matching lower bound for the problem VERIFY$^{\pm}$(SP[AOwd]) (and, hence, for the fragments with filter), followed by Theorem 1 which summarises the results:

	SP[A]	SP[AOwd]	SP[AOF$_{\wedge,=,\neq}$wd]	SP[AOFwd]
VERIFY^{+}	in PTime	coNP-c	coNP-c	coNP-c
VERIFY$^{\pm}$	in PTime	DP-c	DP-c	DP-c
VERIFY$^{\mathsf{E}}$	coNP-c	coNP-c	coNP-c	coNP-c

Table 1: Complexity of verification problems

PROPOSITION 9. VERIFY$^{\pm}$(SP[AOwd]) *is* DP*-hard.*

PROOF SKETCH. By Proposition 8, VERIFY^{+}(SP[AOwd]) is coNP-hard. This implies, in particular, that there exists a polynomial-time many-to-one reduction $\mathcal{R}$ from the complement of 3SAT to VERIFY$^{\pm}$(SP[AOwd]). In fact, we may guarantee that for any input φ to 3SAT, the pattern P in the result $\mathcal{R}(\psi) = (D, \Omega, \bar{\Omega}, P)$ is always of the form $P = P' \,\mathbf{OPT}\, P''$ for $P', P'' \in$ SP[A], where P' does not depend on the input. A reduction is then defined from 3SATUNSAT to VERIFY$^{\pm}$(SP[AOwd]), where given $\phi = \exists \bar{x}_1\, \psi_1 \wedge \forall \bar{x}_2\, \neg\psi_2$ an instance $(D, \Omega, \bar{\Omega}, P)$ is built, making use of $(D_i, \Omega_i, P_i) = \mathcal{R}(\forall \bar{x}_i \neg\psi_i)$, $i = 1, 2$. □

THEOREM 1. *Complexity bounds for the verification problems are as stated in in Table 1.*

4.2 Complexity upper bounds for reverse engineering

We now turn to the reverse engineering decision problems, and progress towards the results summarised in Table 2. In this section we provide complexity upper bounds for the different variants. As mentioned previously, the algorithms which witness these upper bounds follow a common pattern, which consists in first constructing a candidate query (or *realizer*) with the property that if it does not correctly fit the input examples, then there does not exist any query which does, and then verifying that it fits the input examples.

We first consider the SP[A] fragment, and thus the REVENG^{+}(SP[A]), REVENG$^{\pm}$(SP[A]) and REVENG$^{\mathsf{E}}$(SP[A]) decision problems. Intuitively, given an input (D, Ω) to the REVENG$^{\mathsf{E}}$(SP[A]), the candidate query will be the set of all triple patterns which are true in D over all the positive examples in Ω. More precisely, given an RDF graph D and a mapping μ, define the *atomic type of* μ *in* D, denoted by atype(D, μ), as

$$\{t \in (\mathsf{U} \cup \mathsf{V}) \times (\mathsf{U} \cup \mathsf{V}) \times (\mathsf{U} \cup \mathsf{L} \cup \mathsf{V}) \mid \mu(t) \in D\},$$

Essentially, the atomic type is the set of triple patterns that are true in D under μ. We now generalise this notion to a set of mappings Ω, however, for this fragment of SPARQL we restrict to sets of mappings Ω which are *homogeneous*, that is, for every pair of mappings $\mu, \nu \in \Omega$ it is the case that $\mathrm{dom}(\mu) = \mathrm{dom}(\nu)$. For a homogeneous set of mappings Ω, define the *atomic type of* Ω in D, denoted atype(D, Ω), as $\bigcap_{\mu \in \Omega}$ atype(D, μ). Abusing notation, we identify the set atype(D, Ω) and the **AND**-combination of its triple patterns (i.e. an SP[A] pattern). In the case of SP[A], the atomic type is precisely the desired candidate query, as can be seen in the following result:

PROPOSITION 10. *Given an RDF graph D and sets of mappings Ω and $\bar{\Omega}$ the following holds:*
1. *if* $(D, \Omega) \in$ REVENG^{+}(SP[A]) *then* $\Omega \subseteq [\![\mathrm{atype}(D, \Omega)]\!]_D$,
2. *if* $(D, \Omega, \bar{\Omega}) \in$ REVENG$^{\pm}$(SP[A]) *then* $\Omega \subseteq [\![\mathrm{atype}(D, \Omega)]\!]_D$ *and* $\bar{\Omega} \cap [\![\mathrm{atype}(D, \bar{\Omega})]\!]_D = \varnothing$, *and*
3. *if* $(D, \Omega) \in$ REVENG$^{\mathsf{E}}$(SP[A]) *then* $\Omega = [\![\mathrm{atype}(D, \Omega)]\!]_D$.

PROOF SKETCH. For the REVENG^{+}(SP[A]) decision problem (item 1), assume that there exists a query $P \in$ SP[A] such that $\Omega \subseteq [\![P]\!]_D$, and note that P may be interpreted as a set of triple patterns. For every triple pattern t in P and every mapping $\mu \in \Omega$

we have that $\mu(t) \in D$, which implies $t \in \text{atype}(D, \Omega)$, by definition. Therefore, P is a subset of $\text{atype}(D, \Omega)$, and thus $\text{atype}(D, \Omega) \neq \varnothing$ (note also that $\text{var}(P) = \text{var}(\text{atype}(D, \Omega))$). For any mapping $\mu \in \Omega$, $\mu \in [\![\text{atype}(D, \Omega)]\!]_D$ by construction, whereby $\Omega \subseteq [\![\text{atype}(D, \Omega)]\!]_D$. The other cases are similar. □

EXAMPLE 4. *Let* $\Omega = \{\mu_1, \mu_2\}$ *with* $\mu_1 = [?X \mapsto \mathtt{a}]$ *and* $\mu_2 = [?X \mapsto \mathtt{b}]$*, and let* $D = \{(\mathtt{a}, 1, 1), (\mathtt{b}, 1, 1), (\mathtt{a}, 2, 2), (\mathtt{b}, 2, 2)\}$*. Then* $\text{atype}(D, \Omega) = \{(?X, 1, 1), (?X, 2, 2)\}$*, as* $t_1 = (?X, 1, 1)$ *and* $t_2 = (?X, 2, 2)$ *are the only triple patterns* t *for which both* $\mu_1(t) \in D$ *and* $\mu_2(t) \in D$ *hold. Therefore, the corresponding candidate query* $P = (?X, 1, 1)\ \mathbf{AND}\ (?X, 2, 2)$ *defines the input pair. This is not the only possible realizer though; for example, the query* $Q = (?X, 2, 2)$ *also has the property that* $\Omega \subseteq [\![P]\!]_D$.

Proposition 10 leads to the following algorithm template for reverse-engineering in the SP[A] case: first build the atomic type of Ω—which can be done in polynomial time in this case—and then check if it works. Combining this with our results on the verification problem, we obtain the following complexity upper bounds:

COROLLARY 1. REVENG$^+$(SP[A]) *and* REVENG$^\pm$(SP[A]) *are in* PTime*, while* REVENG$^\mathsf{E}$(SP[A]) *is in* coNP.

Next, we generalise the previous process for the SP[AOwd] fragment. To build intuition, first consider, as an example, a query $P = P_1\ \mathbf{OPT}\ P_2$ where both P_1 and P_2 are in SP[A]. For every variable $?X \in \text{var}(P)$ there are two possibilities: if $?X \in \text{var}(P_1)$, then for every mapping $\mu \in [\![P]\!]_D$ it will be the case that $?X \in \text{dom}(\mu)$; if not, and we have that $?X \in \text{var}(P_2)$ and $?X \notin \text{var}(P_1)$, then there may exist mappings $\mu \in [\![P]\!]_D$ such that $?X \notin \text{dom}(\mu)$. Thus, a *hierarchy of variables* exists. In fact, for two variables $?X, ?Y$ such that $?X \in \text{var}(P_1)$ and $?Y \in \text{var}(P_2) \setminus \text{var}(P_1)$ it will be the case that for every mapping $\mu \in [\![P]\!]_D$, if $?Y \in \text{dom}(\mu)$ then $?X \in \text{dom}(\mu)$ (in this example this statement is trivial, as $?X$ is in the domain of every mapping).

The previous reasoning can now be used to outline the problem from a reverse engineering perspective. Consider an RDF graph D and a set of mappings Ω as inputs to REVENG$^+$(SP[AOwd]), where Ω can be divided into two subsets Ω_1 and Ω_2 such that $\Omega = \Omega_1 \cup \Omega_2$, for every mapping $\mu \in \Omega_1$ we have $\text{dom}(\mu) = \{?X, ?Y\}$, and for every mapping $\nu \in \Omega_2$ we have $\text{dom}(\nu) = \{?X\}$. In this case, the form of a candidate query P can be determined by observing the variable hierarchy, concluding that $P = P_1\ \mathbf{OPT}\ P_2$ for some P_1 such that $\text{var}(P_1) = \{?X\}$ and some P_2 such that $\text{var}(P_2) \subseteq \{?X, ?Y\}$. Crucially, it will be determined that no further **OPT** operators are necessary, i.e. $P_1, P_2 \in \text{SP[A]}$. The precise triple patterns in P_1 and P_2 will be determined by a construct analogous to the atomic type.

In the following we formalise the previous intuition, showing that the relation among variables in the set of mappings Ω can be used to determine the form of the candidate query P, and that a generalisation of the atomic type can be used to determine the triple patterns in each subquery of P.

For every variable $?X$ mentioned in Ω define the *coverage* of $?X$ in Ω, denoted $\mathsf{Cov}_\Omega(?X)$ as the set $\{\mu \in \Omega \mid ?X \in \text{dom}(\mu)\}$. Define the *structure* of Ω as the set $\mathcal{C}(\Omega) = \{\mathsf{Cov}_\Omega(?X) \mid ?X \in \text{dom}(\Omega)\}$, containing the coverage of each variable in Ω. The subset relation naturally defines a partial order on the structure $\mathcal{C}(\Omega)$.

EXAMPLE 5. *Consider the set of mappings* Ω *in Figure 1(a). The corresponding structure* $\mathcal{C}(\Omega)$ *is shown in Figure 1(b), where arrows represents the minimal proper superset relation. The topmost node corresponds to the coverage of variable* $?X$*, which is present in all mappings. The two nodes at the second level correspond to the coverages of* $?Y_1$ *and* $?Y_2$*, respectively, while the lone node at the third level represents the coverage of* $?Z$.

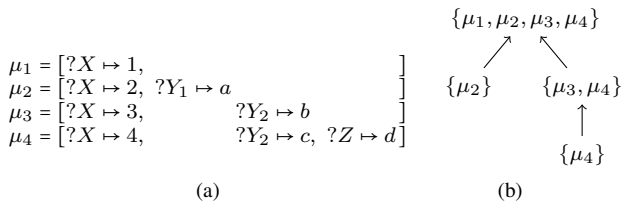

Figure 1: (a) Set of mappings and (b) its structure.

Intuitively, the structure of Ω defined above restricts the possible **OPT** structures of a candidate query. We say that $\mathcal{C}(\Omega)$ (and, by extension, Ω) is *tree-like* if and only if for every set $\Lambda \in \mathcal{C}(\Omega)$ there is at most one minimal proper superset (i.e. parent) of Λ in $\mathcal{C}(\Omega)$. Note that since Ω is consistent, there exists only one set in $\mathcal{C}(\Omega)$ without proper supersets and this set is Ω itself.

EXAMPLE 6. *While the set of mappings in Figure 1 is tree-like, consider replacing mapping* μ_4 *by* $\mu_4' = [?X \mapsto 4, ?Y_1 \mapsto e, ?Y_2 \mapsto c, ?Z \mapsto d]$*. In this case, the coverage of* $?Y_1$ *changes to* $\{\mu_2, \mu_4\}$ *and the node* $\{\mu_4\}$ *becomes a subset of both* $\{\mu_2, \mu_4\}$ *and* $\{\mu_3, \mu_4\}$*. The resulting set of mappings* $\{\mu_1, \mu_2, \mu_3, \mu_4'\}$ *is not tree-like, as* μ_4' *has two minimal proper supersets (i.e. parents):* μ_2 *and* μ_3.

For tree-like sets of mappings we can give a detailed process for constructing a candidate query, and we turn to this now. Given a set of mappings Ω and a set of variables S, let Ω_S consist of all $\mu \in \Omega$ such that $S \subseteq \text{dom}(\mu)$. Finally, for a set Λ in $\mathcal{C}(\Omega)$, let $\mathsf{VarsOf}(\Lambda)$ be the set of variables $?X$ with $\mathsf{Cov}_\Omega(?X) = \Lambda$.

We can now define a canonical query for tree-like sets of mappings. To do this we will recursively define a pattern $P_{\mathsf{can}}(\Lambda, D, \Omega)$ for each Λ in $\mathcal{C}(\Omega)$. If Λ has k maximal proper subsets (i.e. children) $\Lambda_1, \ldots, \Lambda_k$, then $P_{\mathsf{can}}(\Lambda, D, \Omega)$ is the graph pattern

$$(\cdots((\text{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})\ \mathbf{OPT}\ P_{\mathsf{can}}(\Lambda_1, D, \Omega))\ \mathbf{OPT}\ P_{\mathsf{can}}(\Lambda_2, D, \Omega))\cdots\ \mathbf{OPT}\ P_{\mathsf{can}}(\Lambda_k, D, \Omega)). \quad (1)$$

Note that if Λ has no children then $P_{\mathsf{can}}(\Lambda, D, \Omega)$ is just $\text{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$. For Ω tree-like, we define $P_{\mathsf{can}}(D, \Omega)$ to be $P_{\mathsf{can}}(\Omega, D, \Omega)$.

In the construction above the order of the children $\Lambda_1, \ldots, \Lambda_k$ is arbitrary, so the canonical pattern is not unique. However, all of them are equivalent, since $(P_1\ \mathbf{OPT}\ P_2)\ \mathbf{OPT}\ P_3$ is equivalent to $(P_1\ \mathbf{OPT}\ P_3)\ \mathbf{OPT}\ P_2$ for all well-designed queries. This allows us to blur the order of the children and look at canonical queries as *pattern trees* [11], whose tree structure is the same as that of $\mathcal{C}(\Omega)$.

For tree-like sets of mappings we can now formulate a generalisation of Proposition 10, where the subscript tree means that Ω in instances is restricted to tree-like sets.

THEOREM 2. *Given an RDF graph* D*, tree-like set of mappings* Ω *and set of mappings* $\bar{\Omega}$*,*
1. if $(D, \Omega) \in$ REVENG$^+_{\mathsf{tree}}$(SP[AOwd]) *then* $\Omega \subseteq [\![P_{\mathsf{can}}(D, \Omega)]\!]_D$*,*
2. if $(D, \Omega, \bar{\Omega})$ *belongs to* REVENG$^\pm_{\mathsf{tree}}$(SP[AOwd]) *then* $\Omega \subseteq [\![P_{\mathsf{can}}(D, \Omega)]\!]_D$ *and* $\bar{\Omega} \cap [\![P_{\mathsf{can}}(D, \Omega)]\!]_D = \varnothing$*,*
3. if $(D, \Omega) \in$ REVENG$^\mathsf{E}_{\mathsf{tree}}$(SP[AOwd]) *then* $\Omega = [\![P_{\mathsf{can}}(D, \Omega)]\!]_D$.

In fact, the notion of canonical query for tree-like sets of mappings can be straightforwardly adapted to the case of SP[AOF$_{\wedge,=,\neq}$wd]. Here, instead of $\text{atype}(D, \Omega)$ we can consider its generalisation $\text{atype}_{=,\neq}(D, \Omega)$, containing, besides the triple patterns that are true in D under the mappings, all equalities and inequalities on the variables and IRIs that are true in D. When seen as a query, this set is the **AND**-combination of its triple patterns, filtered by the conjunction of all its equalities and inequalities that mention only variables in the triple patterns. The

analog of Theorem 2 for this notion of canonical query holds for $\mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]$.

The above constructions, combined with the results on verification in the previous section, give us the following bounds on the restricted reverse-engineering problems:

COROLLARY 2. $\textsc{RevEng}^{x}_{\mathsf{tree}}(\mathcal{F})$ *with* $x \in \{+, \pm, \mathsf{E}\}$ *and* $\mathcal{F} \in \{\mathrm{SP}[\mathsf{AOwd}], \mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]\}$ *are all in* coNP.

Note that the statements above do not mention our most general class $\mathrm{SP}[\mathsf{AOFwd}]$, which allows for arbitrary filters. We could generalise the atomic type further to obtain the same coNP upper bound for this fragment as well, but it would not be optimal. The class $\mathrm{SP}[\mathsf{AOFwd}]$ is so powerful that a realizer query for tree-like instances *always exists*, as long as the set of mappings satisfies some simple consistency checks:

PROPOSITION 11. *Problems* $\textsc{RevEng}^{x}_{\mathsf{tree}}(\mathrm{SP}[\mathsf{AOFwd}])$ *with* $x \in \{+, \pm, \mathsf{E}\}$ *are all in* PTime.

PROOF SKETCH. We claim that an instance (D, Ω) (or $(D, \Omega, \bar{\Omega})$) has a realizer query if and only if $P_{\mathsf{can}}(D, \Omega)$ mentions all the variables in Ω. Indeed, if the canonical query for $\mathrm{SP}[\mathsf{AOwd}]$ satisfies this, then every $\mu \in \Omega$ is a partial solution of $P_{\mathsf{can}}(D, \Omega)$ over D. We can ensure the maximality of each μ by constructing a custom **FILTER** expression which only admits the desired mappings. □

We now return to the general case of reverse engineering, i.e. where $\mathcal{C}(\Omega)$ is not necessarily tree-like. In this case, there exist many different candidate $\mathrm{SP}[\mathsf{AOwd}]$ queries for an input (D, Ω). Intuitively, these candidates still conform to the structure $\mathcal{C}(\Omega)$, yet in a more relaxed sense: these are queries whose $\mathrm{SP}[\mathsf{A}]$ subqueries can be "merged" to obtain the structure $\mathcal{C}(\Omega)$.

To formalise the previous idea, first recall that a query $P \in \mathrm{SP}[\mathsf{AOwd}]$ is assumed to be in **OPT** normal form, whereby it is possible to consider P to be formed by $\mathrm{SP}[\mathsf{A}]$ queries, combined with the **OPT** operator. Given a query $P \in \mathrm{SP}[\mathsf{AOwd}]$, a subquery $Q \in \mathrm{SP}[\mathsf{A}]$ of P is *left-most* in P if and only if Q is such that there does not exist any subquery $(P_1 \,\mathbf{OPT}\, P_2)$ of P such that Q is a subquery of P_2. Furthermore, given two subqueries $P_1, P_2 \in \mathrm{SP}[\mathsf{A}]$ of P, P_1 is an *ancestor* of P_2 in P if and only if either $P_1 = P_2$ or there exists a subquery $P_1' \,\mathbf{OPT}\, P_2'$ of P such that P_1 is the left-most subquery of P_1' and P_2 is a subquery of P_2'. Finally, given a subquery Q of P, a variable $?X$ is *top-most* in Q if $?X$ is present in the left-most subquery of Q yet absent outside of Q in P. With this, an $\mathrm{SP}[\mathsf{AOwd}]$ query P is a *candidate* for (D, Ω) if there exists a surjective function h from its $\mathrm{SP}[\mathsf{A}]$ subqueries to $\mathcal{C}(\Omega)$ that:
1. each variable $?X \in \mathrm{dom}(\Omega)$ can be associated to an $\mathrm{SP}[\mathsf{A}]$ subquery $Q_{?X}$ of P such that $h(Q_{?X}) = \mathsf{Cov}_{\Omega}(?X)$ and $?X$ is top-most in $Q_{?X}$,
2. preserves the query's tree structure: given $P_1, P_2 \in \mathrm{SP}[\mathsf{A}]$ subqueries of P, if P_1 is an ancestor of P_2 in P, then $h(P_1)$ is a superset (i.e. ancestor) of $h(P_2)$ in $\mathcal{C}(\Omega)$,
3. for each Λ in $\mathcal{C}(\Omega)$ and R in the pre-image $h^{-1}(\Lambda)$, R consists of all triple patterns in $\mathrm{atype}(D, \Omega_S)$, where S is the set containing every variable $?X$ for which $Q_{?X}$ is an ancestor of R in P.

Note that if Ω is tree-like, then there is exactly one candidate query, which is the canonical query.

EXAMPLE 7. *Consider the set of mappings* $\Omega' = \{\mu_1, \mu_2, \mu_3, \mu_4'\}$ *from Example 6. Depending on the graph* D*, a candidate query for the input* (D, Ω') *may follow either of two distinct* **OPT** *structures, both arising by choosing a parent for the* $\{\mu_4'\}$ *element of* $\mathcal{C}(\Omega')$.

We obtain the following generalisation of Theorem 2:

THEOREM 3. *Given an RDF graph* D *and sets* Ω, $\bar{\Omega}$,
1. *if* $(D, \Omega) \in \textsc{RevEng}^{+}(\mathrm{SP}[\mathsf{AOwd}])$ *then there is a candidate query* P *for* (D, Ω) *such that* $\Omega \subseteq [\![P]\!]_D$;
2. *if* $(D, \Omega, \bar{\Omega}) \in \textsc{RevEng}^{\pm}(\mathrm{SP}[\mathsf{AOwd}])$ *then there is a candidate query* P *for* (D, Ω) *with* $\Omega \subseteq [\![P]\!]_D$ *and* $\bar{\Omega} \cap [\![P]\!]_D = \varnothing$;
3. *if* $(D, \Omega) \in \textsc{RevEng}^{\mathsf{E}}(\mathrm{SP}[\mathsf{AOwd}])$ *then there is a candidate query* P *for* (D, Ω) *such that* $\Omega = [\![P]\!]_D$.

If we replacing atype with $\mathrm{atype}_{=,\neq}$ in the definition of candidate queries above, we can prove a similar result to Theorem 3 for $\mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]$. Note also, that candidate patterns are always of polynomial size, so the above result leads to an upper bound for the complexity of the corresponding reverse engineering problems:

COROLLARY 3. $\textsc{RevEng}^{\mathsf{x}}(\mathcal{F})$ *with* $\mathsf{x} \in \{+, \pm, \mathsf{E}\}$ *and* $\mathcal{F} \in \{\mathrm{SP}[\mathsf{AOwd}], \mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]\}$ *are all in* Σ_2^p.

PROOF SKETCH. By Theorem 3 and its adaption to the case of $\mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]$ it suffices to guess a (polynomially-sized) candidate query and *verify* that it realises the input. By Propositions 4 this verification can be done in DP for all cases. As we have provided a non-deterministic, polynomial time algorithm which makes use of an oracle for a problem in DP, we have that the problems (all the mentioned problems) are in Σ_2^p (recall that $\mathsf{NP}^{\mathsf{DP}} = \mathsf{NP}^{\mathsf{NP}} = \Sigma_2^p$). □

We conclude this section with the upper bound for the most general fragment $\mathrm{SP}[\mathsf{AOFwd}]$.

PROPOSITION 12. *The problems* $\textsc{RevEng}^{\mathsf{x}}(\mathrm{SP}[\mathsf{AOFwd}])$ *with* $\mathsf{x} \in \{+, \pm, \mathsf{E}\}$ *are all in* NP.

PROOF SKETCH. Given an instance (D, Ω), it is enough to check that Ω is consistent (in polynomial time) and that there exists a candidate query for (D, Ω) that mentions all variables in Ω. The latter requires guessing of this candidate and checking that all variables are mentioned, which is in NP (recall that ensuring the maximality of the examples may be achieved by constructing custom **FILTER** expressions). □

4.3 Matching lower bounds on reverse engineering problems

In this section we provide complexity lower bounds for the reverse engineering decision problems, thus closing the exact complexities for these problems.

Recall that for our smallest language, $\mathrm{SP}[\mathsf{A}]$, we have PTime upper bounds for the positive and positive-and-negative examples reverse engineering problems, but only a coNP upper bound for the exact variant. Given that the problem of verifying that an $\mathrm{SP}[\mathsf{A}]$ query exactly fits a set of example mappings is coNP-hard, it is not surprising that the corresponding reverse engineering problem is also coNP-hard:

PROPOSITION 13. $\textsc{RevEng}^{\mathsf{E}}(\mathrm{SP}[\mathsf{A}])$ *is* coNP*-hard.*

PROOF SKETCH. This can be shown via a polynomial-time reduction from 3-COLOURABILITY. Given an undirected graph G we construct an instance (D, Ω) such that D consists of a coloured triangle, as in the proof of Proposition 7, along with two disjoint copies of G, while Ω has a mapping to only the two copies, each sending a variable to the IRI representing each vertex of G. To prevent mappings corresponding to automorphisms on G, the two copies have triples representing a strict total order on the vertices, while the triangle allows for any order. We can show that there exists a third mapping from the canonical pattern to the triangle if and only if there is a 3-colouring of G. □

	SP[A]	SP[AOwd]	SP[AOF$_{\wedge,=,\neq}$wd]	SP[AOFwd]
REVENG^{+}	in PTime	Σ_2^p-c	Σ_2^p-c	NP-c
REVENG$^{\pm}$	in PTime	Σ_2^p-c	Σ_2^p-c	NP-c
REVENG$^{\mathsf{E}}$	coNP-c	Σ_2^p-c	Σ_2^p-c	NP-c
REVENG$^{+}_{\mathsf{tree}}$	in PTime	coNP-c	coNP-c	in PTime
REVENG$^{\pm}_{\mathsf{tree}}$	in PTime	coNP-c	coNP-c	in PTime
REVENG$^{\mathsf{E}}_{\mathsf{tree}}$	coNP-c	coNP-c	coNP-c	in PTime

Table 2: Complexity of reverse engineering problems

We move to lower bounds for SP[AOwd] for tree-like cases.

PROPOSITION 14. *The problems* REVENG$^{x}_{\mathsf{tree}}(\mathcal{F})$ *for* $x \in \{+, \pm, \mathsf{E}\}$ *and* $\mathcal{F} \in \{\mathrm{SP}[\mathsf{AOwd}], \mathrm{SP}[\mathsf{AOF}_{\wedge,=,\neq}\mathsf{wd}]\}$ *are* coNP*-hard.*

The proof of this is similar to that of Proposition 13, and has been omitted. Next we consider the general cases of the problems and start with REVENG^{+}(SP[AOwd]).

THEOREM 4. REVENG^{+}(SP[AOwd]) *is* Σ_2^p*-hard.*

PROOF SKETCH. The proof is by reduction of $\exists\forall$3SAT and it is a generalisation of the one for Proposition 13 to the case when Ω in the instance is not tree-like. Given ϕ of the form $\exists\bar{x}\forall\bar{y}\,\neg\psi$ we construct an instance (D, Ω) such that there are $2^{|\bar{x}|}$ different candidate patterns. This corresponds to the $\exists$ part of ϕ. The optional part of each of these candidates can be potentially matched by a part of the RDF graph corresponding to $\bar{y}$, in this way extending a mapping in Ω (in the same way as the new mapping in Proposition 13 could be extended to the triangle); this corresponds to the $\forall$ part of ϕ. This enforces the necessity of going through all the candidates and looking for a possible extension in the worst case. □

Theorem 4 gives a lower bound for REVENG$^{\pm}$(SP[AOwd]). Moreover, a careful inspection of the proof shows that it works for REVENG$^{\mathsf{E}}$(SP[AOwd]) as well, while a small modification of the proof (as in Proposition 14) gives us the same lower bounds for the case of SP[AOF$_{\wedge,=,\neq}$wd].

COROLLARY 4. *The problems* REVENG$^{\mathsf{x}}$(SP[AOF$_{\wedge,=,\neq}$wd]) *with* $x \in \{+, \pm, \mathsf{E}\}$ *are all* Σ_2^p*-hard.*

Finally, we have the NP-hardness for the problems with arbitrary filter, and Theorem 5, which summarises the results of this section:

PROPOSITION 15. *The problems* REVENG$^{\mathsf{x}}$(SP[AOFwd]) *with* $x \in \{+, \pm, \mathsf{E}\}$ *are all* NP*-hard.*

PROOF SKETCH. The proof is again based on the fact that there may be exponentially many candidate patterns, and in this case we need to check whether there exists one that uses all variables or not. This proof can be done by a reduction from the 3SAT problem. □

THEOREM 5. *The complexity results in Table 2 hold.*

5. ALGORITHMS FOR REVERSE ENGINEERING

In this section we discuss the algorithms that can be used to solve the REVENG$^{+}_{\mathsf{tree}}$(SP[AOwd]) and REVENG$^{+}_{\mathsf{tree}}$(SP[AOF$_{\wedge,=,\neq}$wd]) reverse engineering problems. Although the core ideas for these algorithms were presented in the complexity upper bounds results of Section 4.2, we now describe the algorithms and discuss modifications for returning more desirable reverse engineered queries.

Given an input (D, Ω), if Ω is an arbitrary set of mappings (i.e. not necessarily tree-like), then the structure $\mathcal{C}(\Omega)$ (as defined in Section 4.2) does not clearly indicate an **OPT** structure for the reverse engineered query, in which case the algorithm must iterate over all possible **OPT** structures and construct a candidate query for each such structure. In what follows, we focus our attention on the tree-like cases (REVENG$^{+}_{\mathsf{tree}}$(SP[AOwd]) and REVENG$^{+}_{\mathsf{tree}}$(SP[AOF$_{\wedge,=,\neq}$wd])), where the structure $\mathcal{C}(\Omega)$ of Ω immediately gives the **OPT**-structure, from which we construct the unique candidate query (with the additional requirement of deciding which triple patterns to include in the query) in polynomial time. The complexity lower bound results, however, indicate that even once the candidate query P is constructed, we must check that $\Omega \subseteq [\![P]\!]_D$ actually holds (this is due to the fact that the candidate query has the property that $(D, \Omega) \in$ REVENG^{+} if and only if $\Omega \subseteq [\![P]\!]_D$). However, this clean separation between the polynomial time construction of the candidate query and the coNP-complete verification of said query will allow us to use a state-of-the-art SPARQL engine for the final step.

Algorithm 1 outlines a framework for solving the REVENG^{+}(SP[AOwd]) decision problem. Firstly, if Ω is not consistent then there is no query $P \in$ SP[AOF] such that $\Omega \subseteq [\![P]\!]_D$ (see Section 2.1). Function `CheckCanon`(D, Ω) simply verifies that the candidate reverse-engineered query P is in SP[AOwd] (note that if the candidate query is not in SP[AOwd], then we have that the input is not definable). Function `BuildCanon`(D, Ω) uses the structure $\mathcal{C}(\Omega)$ to build the candidate query P, and corresponds to the recursive function $P = P_{\mathsf{can}}(D, \Omega) = P_{\mathsf{can}}(\Omega, D, \Omega)$ which was described in detail in Equation 1 of Section 4.2. Notice that for each leaf element Λ in the structure $\mathcal{C}(\Omega)$ we set $P_{\mathsf{can}}(\Lambda, D, \Omega) = \mathrm{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$, which by definition includes *all existing* triple patterns which satisfy the mappings in $\Omega_{\mathsf{VarsOf}(\Lambda)}$. For this reason, we call this the *maximal algorithm*. Finally, once built, the candidate query P must be checked, returning P if $\Omega \subseteq [\![P]\!]_D$ and **null** otherwise. This final check can be delegated to an external SPARQL engine.

Now we consider a greedy version of the algorithm, which differs from the maximal algorithm only in the construction of the candidate P. Intuitively, for an element $\Lambda \in \mathcal{C}(\Omega)$ it is not necessary for $P_{\mathsf{can}}(\Lambda, D, \Omega)$ to include *all* triple patterns in $\mathrm{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$—merely *enough* triple patterns from $\mathrm{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$ to ensure the the positive examples $\mu \in \Omega$ will in fact be answers $\mu \in [\![P]\!]_D$. More precisely, for each element $\Lambda \in \mathcal{C}(\Omega)$ we define a relaxed $P^{\mathrm{greedy}}_{\mathsf{can}}(\Lambda, D, \Omega)$, which must be a subset of $\mathrm{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$ such that (i) every variable $?X \in \mathsf{VarsOf}(\Lambda)$ is mentioned in at least one triple pattern of $P^{\mathrm{greedy}}_{\mathsf{can}}(\Lambda, D, \Omega)$ (this is a requirement for P to be in SP[AOwd]), and (ii) if Λ' is the parent of Λ in $\mathcal{C}(\Omega)$ (i.e. the unique minimal superset of Λ in $\mathcal{C}(\Omega)$), then for each mapping $\mu \in \Lambda' \setminus \Lambda$, for every ν such that $\mu \not\subseteq \nu$ there must exist a triple pattern $t \in P^{\mathrm{greedy}}_{\mathsf{can}}(\Lambda, D, \Omega)$ such that $\nu(t) \notin D$. The previous condition (ii) effectively ensures that μ is a *maximal* partial answer.

We may now implement `BuildCanon`(D, Ω) to find, for each element $\Lambda \in \mathcal{C}(\Omega)$, all triple patterns $t \in \mathrm{atype}(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$, adding them to $P^{\mathrm{greedy}}_{\mathsf{can}}(\Lambda, D, \Omega)$ until all variables $?X \in \mathsf{VarsOf}(\Lambda)$ have been mentioned at least once, and the maximality of each $\mu \in \Lambda' \setminus \Lambda$ has been assured. We call the resulting modified algorithm the *greedy algorithm*.

The greedy algorithm generates an interesting tradeoff between the quality of the reverse engineered query and complexity. On one hand, as we only add enough triple patterns to the query to justify the positive examples, the resulting query will be relatively small (at the very least, not larger that the query produced by the

maximal algorithm). On the other hand, the process of checking the maximality of a positive example $\mu \in \Omega$ when adding each triple pattern t to $P^{\text{greedy}}_{\mathsf{can}}(\Lambda, D, \Omega)$ takes exponential time, as every possible extension mapping ν such that $\mu \subsetneq \nu$ must be checked (the number of such extensions is exponential in the size of Ω).

Finally, we briefly comment on the modifications required for the REVENG$^+$(SP[AOF$_{\wedge,=,\neq}$wd]) problem. In this case, the construction in Equation 1 can be modified to add a filter expression for each $\Lambda \in \mathcal{C}(\Omega)$. In essence, each $P_{\mathsf{can}}(\Lambda, D, \Omega)$ now consists of a set of triple patterns and a *set of filter comparisons*, which can be of the form $?X = ?Y$, $?X \neq ?Y$, $?X = a$, and $?X \neq a$, for some variables $?X, ?Y \in \mathsf{V}$, and some $a \in \mathsf{U} \cup \mathsf{L}$. In this scenario, the atype$(D, \Omega_{\mathsf{VarsOf}(\Lambda)})$ can be generalised to include all filter comparisons R for which $\mu \models R$ for each $\mu \in \Omega_{\mathsf{VarsOf}(\Lambda)}$.

Algorithm 1: Outline for deciding REVENG$^+_{\mathsf{tree}}$(SP[AOwd]).

Input: RDF graph D, set of mappings Ω.
Output: A query $P \in$ SP[AOwd] such that $\Omega \subseteq [\![P]\!]_D$ if such a query exists and Ω is tree-like; **null** otherwise.

1 **if** Ω *is not consistent and tree-like* **then return null**;
2 $P \leftarrow$ `BuildCanon`(D, Ω);
3 **if** `CheckCanon`$(P, \Omega) =$ ***false*** **then return null**;
4 **if** $\Omega \subseteq [\![P]\!]_D$ **then return** P **else return null** ;

6. EXPERIMENTAL EVALUATION

In this section we describe the experimental settings with which we have studied the implementation of the reverse engineering algorithms. More precisely, we first perform a study on synthetically generated inputs, and then we attempt to reverse engineer SPARQL queries in a real-world setting. For the second scenario, as we do not have access to positive examples provided by users, we indirectly obtain these through the use of query logs. All algorithms have been implemented in Java and run on a machine with a 2.3 GHz Intel Core i7 processor and 16 GB of main memory.

6.1 Reverse engineering random inputs

To test the efficiency and usefulness of our approach for reverse engineering SPARQL queries, we tested Algorithm 1 over synthetically generated inputs. Although generating a random RDF graph D and a random set of mappings Ω is possible, in practice the pair will usually not be definable. For this reason, we opt to first generate a random query $Q \in$ SP[AOwd], and construct the pair (D_Q, Ω_Q) from Q. We now discuss the generation of these inputs.

Random queries are generated as linear pattern trees, i.e. queries of the form $(P_0 \,\mathbf{OPT} (P_1 \,\mathbf{OPT} \cdots (P_{n-1} \,\mathbf{OPT}\, P_n) \cdots))$, where each $P_i \in$ SP[A]. The depth n of each query Q is varied as a parameter. In each node P_i we place a set of triple patterns of the form $t = (s, p, o)$, where $p \in \mathsf{U}$ and $s, o \in \mathsf{U} \cup \mathsf{V}$. We aim to include joins between variables in these queries, and to this end, for each subquery P_i and each variable v which is mentioned in P_i but not in any P_j for $j < i$, we include a triple pattern of the form (u, c, v), where u is a variable mentioned in P_{i-1} and c is a random IRI (notice that u is taken from P_{i-1} to obtain a well-designed graph pattern). In total, ~900 random queries were generated, where ~100 queries are generated with each depth $n \in [0, 8]$.

For each random query Q we next generate two distinct inputs (D_1, Ω) and (D_2, Ω) to be submitted to the learning algorithm. First, we generate an RDF graph D_Q by *freezing* the query Q: for each prefix $Q' \trianglelefteq Q$,[1] we convert the set of triple patterns in Q' into a set of triples by replacing the variables with fresh constants.

[1]Recall that such prefixes are formed by replacing a subquery $(R \;\mathbf{OPT}\; S)$ by R, as defined in Section 2.2.

EXAMPLE 8. *Assume that* $Q = (?X, \texttt{type}, \texttt{Country})\,\mathbf{OPT}\,(?X, \texttt{label}, ?Y)$. *Then for the prefix* $Q_1 = (?X, \texttt{type}, \texttt{Country})$ *of* Q, *we generate the triple* $(\texttt{X}_0, \texttt{type}, \texttt{Country})$ *by replacing variable* $?X$ *by constant* $\texttt{X}_0$. *Moreover, for the prefix* $Q_2 = Q$, *we generate the triples* $(\texttt{X}_1, \texttt{type}, \texttt{Country})$, $(\texttt{X}_1, \texttt{label}, \texttt{Y}_1)$ *by following the same approach, where* $\texttt{X}_1$ *and* $\texttt{Y}_1$ *are fresh constants (different from* $\texttt{X}_0$*). Thus, we have that* $D_Q = \{(\texttt{X}_0, \texttt{type}, \texttt{Country}), (\texttt{X}_1, \texttt{type}, \texttt{Country}), (\texttt{X}_1, \texttt{label}, \texttt{Y}_1)\}$.

The RDF graph D_Q is now used to obtain the full set of answers $\Omega_Q = Q(D_Q)$. For both the inputs (D_1, Ω) and (D_2, Ω) to the algorithm, we extract samples from D_Q and Ω_Q. First, let $\Omega \subseteq \Omega_Q$ be a uniform sample of Ω_Q containing at most 100 mappings. Second, let $D_1 = D_Q$ and $D_2 \subseteq D_Q$ be a uniform sample of D_Q, where each triple in D_Q has a 75% chance of being included in D_2. The first input (D_1, Ω) is the full frozen RDF graph D_Q and a sample of the answers in Ω_Q as positive examples, and it is thus expected to be definable by construction. The second input (D_2, Ω) uses a reduced RDF graph (and the same positive examples), whereby it is possible that this pair will no longer be definable.

For each random query Q, both pairs (D_1, Ω) and (D_2, Ω) are input into a Java implementation of Algorithm 1, for a total of ~1800 distinct runs of the algorithm. Of these, 1064 definable cases and 761 non-definable cases are reported. Figure 2 shows the results of these experiments. The left plot in Figure 2 shows the runtime of the algorithm in milliseconds versus the size of the input (defined to be the sum of the number of triples in the RDF graph and the number of mappings in the set of mappings). As this is a logarithmic plot, the exponential dependency is clear, both for the definable and undefinable cases. On the other hand, the right plot in Figure 2 shows the runtime versus the size of the random query itself, exhibiting a very similar (and exponential) performance behaviour. The average runtime for this set of examples was ~516ms. For the definable examples, the average ratio between the size of the reverse-engineered query P and the original randomly generated query Q was 0.79; this reflects the fact that reverse-engineered queries tend to be relaxed versions of the original query, and thus contain less triple patterns. Similarly, the average difference between the **OPT**-depth of the learned query and the original random query is -1.33, indicating that learned queries tend to use less **OPT** operators. This actually depends on the positive examples, as the **OPT**-depth of the learned query will be exactly the depth of the structure of the set of mappings Ω. Finally, note that merely 32 of the 1064 definable cases learned a query which mentioned a constant that was not originally included in the random query. This eases a concern of the greedy approach, which in principle could add many superfluous triple patterns to the learned query.

6.2 Reverse engineering DBpedia query logs

In order to test our implementation of the reverse engineering algorithm on real-world examples, we turn to the public DBpedia SPARQL endpoint. DBpedia (version 2014) was downloaded and installed locally into the Virtuoso Open Source database manager (version 7.1.0). The DBpedia RDF graph contains over 860 million triples, and the contents is extracted from the Wikipedia and Wikimedia websites. DBpedia provides a public SPARQL endpoint where users may perform SPARQL queries, and the logs for this endpoint are made publicly available as well. To access and study these query logs, we turn to the LSQ project [18], which has extracted and organised the DBpedia query logs.

Although our main goal is to obtain sets of positive examples which can then be input into our reverse engineering algorithm, such sets of positive examples are not available, as no SPARQL reverse engineering or query-by-example systems exists which pub-

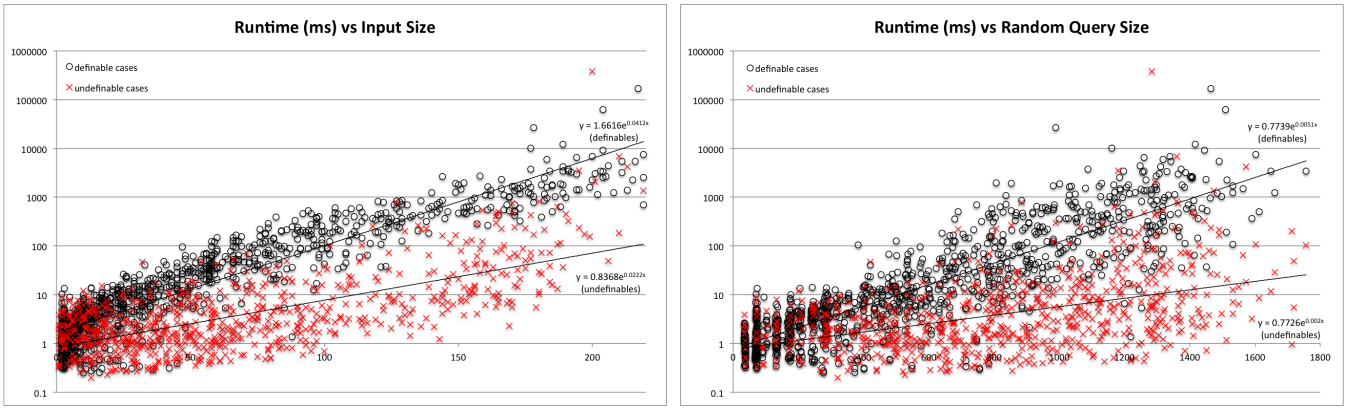

Figure 2: Runtimes for ~2000 randomly generated examples. Left: runtime versus input size; circles (black) represent definable inputs and crosses (red) represent undefinable inputs. Right: runtime versus random query size.

lish these data. Hence, we use the DBpedia query logs to indirectly obtain positive examples. For a query Q from the query logs, we execute the query on the DBpedia RDF graph, which we denote by D_{Dbpedia}, to obtain $\Omega = [\![Q]\!]_{D_{\text{DBpedia}}}$. This set of mappings represents the full set of examples that the user was supposedly interested in when executing query Q, and a random sample $\Omega' \subseteq \Omega$ of Ω can serve as a hypothetical set of positive examples from said user.

Of the ~740,000 queries in the query logs (queries which were executed on the DBpedia SPARQL endpoint), there are ~220,000 queries which make use of the **OPT** operator but not the **FILTER** or **UNION** operators, and ~34,000 which use both the **OPT** and **FILTER** operators but not **UNION**. To understand the sets of mappings which are to be expected, ~124,000 of the 220,000 queries on D_{DBpedia} were selected, and for each such query Q we obtain the full set of results $[\![Q]\!]_{D_{\text{DBpedia}}}$. Of these sets of mappings, none had a structure whose depth was greater than 1; more precisely, 116 of these queries produced an answer set of depth 1, and the rest produced depth 0 (i.e. they were homogeneous). This suggests that a reverse engineering algorithm may be parameterised to produce queries whose **OPT**-depth is limited to 1.

We next selected ~30,000 queries and replicated the procedure from the previous section for randomly generated queries. That is, each query Q was executed on D_{DBpedia} to obtain $\Omega = [\![Q]\!]_{D_{\text{DBpedia}}}$, and a random sample $\Omega' \subseteq \Omega$ was used as the set of positive examples (in many cases the query has exactly one mapping as an answer, in which case we simply use this mapping as the sole positive example). The pair $(D_{\text{DBpedia}}, \Omega')$ then was used as the input. In this experimental setting the average runtime for all queries was 35ms and the average ratio between the learned query size and the original query size was 0.28. These low values can be explained by the fact that many learned queries only have one triple pattern.

To illustrate the behaviour of the algorithm on a slightly more complicated query, we manually prepare the following query Q_2:

```
SELECT * WHERE {
  ?country type Country .
  ?country usesTemplate Infobox_country .
  OPTIONAL { ?country languages ?language }
  OPTIONAL {
    ?country2 type Country .
    ?country wikiLink ?country2 .
    ?country2 usesTemplate Infobox_country .
    ?country2 subject Former_Spanish_colonies } }
```

The answer set $\Omega = [\![Q_2]\!]_{D_{\text{DBpedia}}}$ has ~860 results, and its structure is tree-like of depth 1. A sample $\Omega' \subseteq \Omega$ from Ω is extracted and the pair $(D_{\text{DBpedia}}, \Omega')$ are input into the algorithm. The query learned by the greedy algorithm was similar to Q_2, but without the two triple patterns which mention the IRI `Infobox_country`. The similarity between learned query and original query is an indication that the algorithm in general gives high quality results.

Finally, we showcase the algorithm for the REVENG$^+$ (SP[AOF$_{\wedge,=,\neq}$wd]) decision problem by slightly altering the query Q_2 and adding a single **FILTER** expression, resulting in:

```
SELECT * WHERE { ?country type Country .
  OPTIONAL { ?country languages ?language }
  OPTIONAL {
    ?country2 type Country .
    ?country2 subject Former_Spanish_colonies .
    ?country wikiLink ?country2 .
    FILTER ( ?country != ?country2 ) } }
```

The following is the query learned by the algorithm:

```
SELECT * WHERE { ?country type Country .
  OPTIONAL { ?country languages ?language }
  OPTIONAL {
    ?country2 type Country .
    ?country2 subject Former_Spanish_colonies .
    ?country wikiLink ?country2 .
    ?country2 usesTemplate RefList .
    FILTER ( ?country != ?country2 ) } }
```

Note that the algorithm was able to successfully learn the query, although in this case an extra triple pattern has been added.

7. CONCLUSION

We have shown that the implementation of the greedy reverse engineering algorithm for the positive-examples case is able to correctly learn queries from a set of positive example mappings. An experimental setting with synthetically generated inputs reveals the exponential dependency of the runtime on the input size, which was to be expected given our complexity results. However, an important component of the complexity of the algorithms originates from the final check of the candidate query (see Line 4 in Algorithm 1). Hence, it is in principle possible to produce a candidate query to the user early, while the final check is completed in the background.

The reverse engineering problem has been studied for various fragments of the SPARQL query language. Our study of the complexity of the problem indicates that restricting the examples so that the associate lattice is tree-like has significant benefits to the complexity of reverse engineering. Based on this study we developed a reverse engineering procedure that proceeds by building a single candidate query and checking its correctness via a call to a SPARQL query engine. We have examined the performance of this algorithm both on synthetically generated and real-world query examples, profiling both performance and quality of the results.

8. REFERENCES

[1] D. Angluin. Inductive inference of formal languages from positive data. *Information and Control*, 45(2):117–135, 1980.

[2] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[3] D. Angluin. Queries and concept learning. *Machine Learning*, 2(4):319–342, 1987.

[4] T. Antonopoulos, F. Neven, and F. Servais. Definability problems for graph query languages. In *Joint 2013 EDBT/ICDT Conferences, ICDT '13 Proceedings, Genoa, Italy, March 18-22, 2013*, pages 141–152, 2013.

[5] M. Arenas and J. Pérez. Querying semantic web data with SPARQL. In *PODS*, 2011.

[6] F. Bancilhon. On the completeness of query languages for relational data bases. In *Mathematical Foundations of Computer Science 1978, Proceedings, 7th Symposium, Zakopane, Poland, September 4-8, 1978*, pages 112–123, 1978.

[7] A. Bonifati, R. Ciucanu, and A. Lemay. Learning path queries on graph databases. In *EDBT*, 2015.

[8] A. Bonifati, R. Ciucanu, and S. Staworko. Interactive join query inference with JIM. *PVLDB*, 7(13):1541–1544, 2014.

[9] S. Cohen and Y. Y. Weiss. Learning tree patterns from example graphs. In *ICDT*, 2015.

[10] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[11] A. Letelier, J. Pérez, R. Pichler, and S. Skritek. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.*, 38(4):25, 2013.

[12] F. Manola and E. Miller. RDF Primer, W3C Recommendation 10 February 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210.

[13] C. H. Papadimitriou and M. Yannakakis. The complexity of facets (and some facets of complexity). *J. Comput. Syst. Sci.*, 28(2):244–259, 1984.

[14] J. Paredaens. On the expressive power of the relational algebra. *Inf. Process. Lett.*, 7(2):107–111, 1978.

[15] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.*, 34(3), 2009.

[16] F. Picalausa and S. Vansummeren. What are real SPARQL queries like? In *SWIM*, 2011.

[17] R. Pichler and S. Skritek. Containment and equivalence of well-designed SPARQL. In *PODS*, 2014.

[18] M. Saleem, M. I. Ali, A. Hogan, Q. Mehmood, and A.-C. N. Ngomo. LSQ: The linked SPARQL queries dataset. In *ISWC*, 2015.

[19] M. Schmidt, M. Meier, and G. Lausen. Foundations of SPARQL query optimization. In *ICDT*, 2010.

[20] S. Staworko and P. Wieczorek. Learning twig and path queries. In *ICDT*, 2012.

[21] L. J. Stockmeyer. The polynomial-time hierarchy. *Theor. Comput. Sci.*, 3(1):1–22, 1976.

[22] Q. T. Tran, C. Chan, and S. Parthasarathy. Query by output. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, pages 535–548, 2009.

[23] Q. T. Tran, C. Y. Chan, and S. Parthasarathy. Query reverse engineering. *VLDB J.*, 23(5):721–746, 2014.

[24] R. Willard. Testing expressibility is hard. In *Principles and Practice of Constraint Programming - CP 2010 - 16th International Conference, CP 2010, St. Andrews, Scotland, UK, September 6-10, 2010. Proceedings*, pages 9–23, 2010.

[25] M. Zhang, H. Elmeleegy, C. M. Procopiuc, and D. Srivastava. Reverse engineering complex join queries. In *SIGMOD*, 2013.

[26] X. Zhang and J. V. den Bussche. On the satisfiability problem for SPARQL patterns. *CoRR*, abs/1406.1404, 2014.