

TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computing Science

Master's Thesis

**Querying Resource Description
Framework Graphs**

M.L.A. Ansems

Supervisors:

Prof. dr. E.H.L. Aarts (Technische Universiteit Eindhoven)
Dr. ir. J.H.M. Korst (Philips Research Eindhoven)

Eindhoven, January 2005

Abstract

Resource Description Framework (RDF) is a standard representation language that describes how to define metadata for documents on the World Wide Web. The web documents extended with this metadata form the Semantic Web and attribute to machine understanding of web facilities. Retrieving specific information from the Semantic Web amounts to querying the RDF metadata. In this document, an RDF data set is represented as a directed, labeled graph. Furthermore, an algorithm for querying these graphs is presented and implemented.

Acknowledgements

This master's thesis completes my studies at the Department of Mathematics and Computing Science of the Technische Universiteit Eindhoven. This project is carried out in the Media Interaction group at the Philips Research Laboratories in Eindhoven during the period from September 2003 until December 2004.

I want to thank all colleagues in the group who made my stay at Philips an enjoyable one. A special word of appreciation goes to Herman ter Horst. Frequently, he explained RDF concepts down to the minutest detail with much patience and clarity.

I would also like to thank my supervisors Jan Korst and Emile Aarts for their useful suggestions and critical comments on my work. I appreciate their patience and belief in a good ending of my project, when it was interrupted for five months as I had to recover from an accident.

I am grateful to my mother for taking care of me during this difficult period.

Finally, I would like to thank my boyfriend Thijs for all his love and support.

Monique Ansems
Eindhoven, December 2004

Contents

Abstract	iii
Acknowledgements	iv
1 Introduction	2
1.1 Semantic Web	2
1.2 Resource Description Framework	3
1.3 Querying RDF Data	4
1.4 Overview	4
2 Resource Description Framework	5
2.1 RDF Concepts	5
2.2 RDF Schema	6
2.3 RDF Levels	7
2.4 RDFS Closure	8
2.5 Entailment	10
2.5.1 Simple Entailment	10
2.5.2 RDFS Entailment	11
2.5.3 Complexity	11
2.6 Conclusion	11
3 RDF Query Languages And Systems	12
3.1 Querying RDF Levels	12
3.2 RDF Query Languages	12
3.3 RDF Query Systems	13
3.4 Conclusion	14
4 RDF Subgraph Problem	15
4.1 RDF Graph Definitions	15
4.2 RDF Subgraph Problem Definition	19
4.2.1 Special Cases	22
4.3 Conclusion	22
5 RDF Subgraph Algorithm	23
5.1 Subgraph Isomorphism Algorithms	23
5.1.1 Conclusion	24
5.2 Ullmann’s Algorithm	25

5.3	RDF Subgraph Algorithm	26
5.3.1	Implementation	28
5.4	Possible Improvements	29
5.4.1	Label Preprocessing	29
5.4.2	Vertex Fingerprint Preprocessing	29
5.4.3	Decomposition Preprocessing	31
5.5	Conclusion	31
6	Experiments	32
6.1	Other RDF Systems	32
6.2	Real-life Application	33
6.2.1	Museum Graph	33
6.2.2	Museum Repository	35
6.2.3	Conclusion	35
6.3	Semi-random RDF Semantic Graphs	36
6.4	Conclusion	40
7	Conclusion	41
7.1	Suggestions For Further Research	42
7.2	References	43
A	Supplementary RDF Issues	45
A.1	Simple Entailment Rules	45
A.2	RDFS Closure Computation	46
A.3	XML Clash	47
A.4	Finite Container Construction	47
B	Ullmann's Algorithm For Subgraph Isomorphism	48
B.1	Given Algorithm	48
B.2	Original Experiments	49
B.3	New Experiments	49
B.4	Possible Improvements	50
B.4.1	First Fingerprint	50
B.4.2	Second Fingerprint	51
C	RDF Subgraph Algorithm Implementation	53
C.1	Implementation	53
C.2	Experiments	54
D	Museum Graph	59

Chapter 1

Introduction

The World Wide Web [12] is a huge information repository with the aim to be useful for human-human communication as well as to contribute to machine understandability of this information. Until recently, however, most information on the Web has been designed for human consumption and was semi-structured. This made machine interpretation difficult. Resource Description Framework (RDF) is a standard representation language describing how to define extra data for documents on the World Wide Web [7]. The web documents extended with this extra data form the so-called Semantic Web [9] and contribute to machine understanding of and machine reasoning with information on the Web. Searching in the Semantic Web to retrieve specific information amounts to querying the RDF data and will be elaborated on in this document.

1.1 Semantic Web

The Semantic Web [9] extends the current World Wide Web with structured data that is intended for machine interpretation. This data has a well-defined meaning, provided by the Resource Description Framework language, which contributes to a better and faster machine understanding of web documents. For example, it is easy to see for humans that a web document about planets and a web document about famous actors have a different meaning. However, a machine concludes that both web documents contain the word “stars” and are therefore related to each other. When metadata “heavenly body, comet, sun” is added to the planet document and metadata “Hollywood, celebrity, famous name” is added to the other document, the differences between these sites also become clear to the machine.

Besides interpreting information, new information can be formed by machine reasoning. Both can be done by an RDF system. To mention an example of such a system, the MusicBrainz system [29] provides a large database of music metadata and uses RDF to exchange metadata of digital audio and video tracks.

1.2 Resource Description Framework

Resource Description Framework is a standard representation language describing how to define extra data for web documents. Usually, this extra data is additional descriptive information about the structure and content of a web document, called metadata. However, this extra data can also be an overview of the document content itself.

The syntax of RDF is directly based on XML [2], just as other web languages. Therefore, RDF data can be included in web documents; see also Figure 1.1. To obtain this XML data, a Web browser is needed to retrieve web documents and an XML/RDF parser is used to extract the RDF data from a web document. RDF data can also be stored in databases. For example, this is done in the MusicBrainz system [29]. An RDF system can interpret RDF data and is able to derive new information from this data by reasoning. Furthermore, RDF systems can query RDF data.

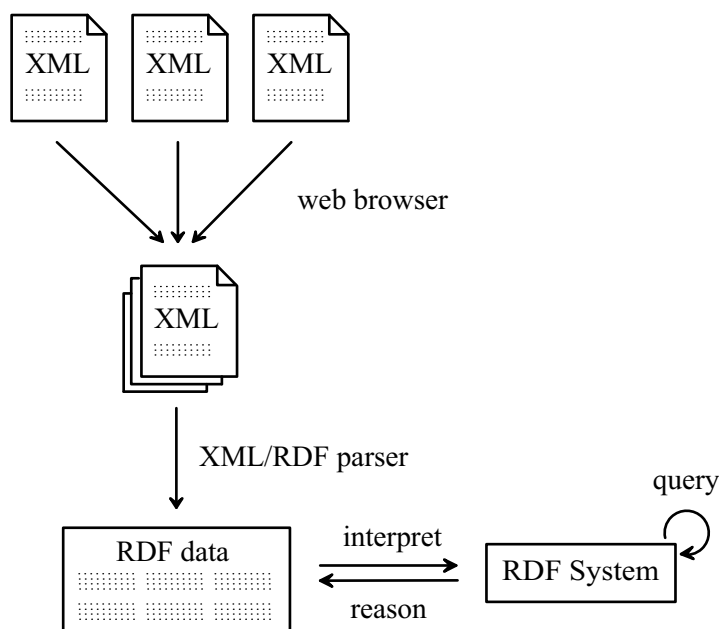


Figure 1.1: RDF Overview

The strength of RDF is that RDF data can be approached at different, but equivalent levels [16]. The syntactic level represents RDF data by the XML syntax which can therefore be directly inserted in web documents. This level can be converted into a structural level where RDF data is considered as an unordered set of piecewise structured data, which is well-suited for database storage. At the semantic level, this set is converted into a coherent unity which makes relations in the data more clear. In this document, such a coherent unity is represented as a directed, labeled graph.

1.3 Querying RDF Data

Recently, several RDF query languages and systems have been introduced. Note that RDF data can be queried at the syntactic level as well as the structural level and the semantic level.

Querying RDF data at the syntactic level (XML syntax) can be done by existing XML query languages, such as XQuery [13]. Querying sets of structured data is done by query languages such as RDQL [6] and query systems such as Jena [3]. Querying RDF data at the semantic level is a recent field of research. Currently, only few systems are based on this query method, for example the Sesame system [1]. Note that considering RDF data as graphs enables the possibility of using existing graph definitions and algorithms.

This document deals with querying at the semantic level and consequently querying directed, labeled graphs. This can be considered as finding a query subgraph in a directed, labeled graph, which is closely related to the existing Subgraph Isomorphism problem [21].

1.4 Overview

Querying RDF data at the semantic level is elaborated on in the following chapters. The remainder of this document is organized as follows. In Chapter 2, some important RDF concepts are discussed such as RDF graph, RDF schema, RDFS closure and entailment. Some leading RDF query languages and systems are briefly presented in Chapter 3. The given RDF concepts are elaborated on in graph theoretical terms in Chapter 4. Also, the RDF Subgraph problem is defined, i.e., querying directed, labeled graphs. Some Subgraph Isomorphism algorithms are given in Chapter 5. One of these algorithms is used as the basis for the RDF Subgraph algorithm and is explained in detail. Furthermore, the RDF Subgraph Algorithm is presented. Chapter 6 contains experimental results of the RDF Subgraph Algorithm. Finally, in Chapter 7, some conclusions and suggestions for further research are given.

Chapter 2

Resource Description Framework

Resource Description Framework (RDF) is a standard representation language describing how to define extra data for web documents. Usually, this extra data is additional descriptive information about the structure and content of a web document.

This chapter is organized as follows. The basic RDF concepts are explained in Section 2.1. Note that the specifications of RDF [7] only contain definitions of RDF concepts at the structural level. RDF schema is a semantic extension of RDF. It provides mechanisms for describing groups of related resources and the relationships between these groups. Furthermore, RDF schemas contribute to adding new information. RDF schema is described in Section 2.2. The strength of RDF is that RDF data can be approached at different, but equivalent levels, which is explained in Section 2.3. As mentioned, reasoning can be applied to derive new information. This results in an extended graph, called RDFS closure, and is explained in Section 2.4. In Section 2.5, the term entailment is explained. In brief, entailment is a term used to verify whether RDF data sets have the same semantics. Note that Appendix D contains an example of an RDF data set regarding museum information.

2.1 RDF Concepts

The basic idea of RDF is to provide data in which elements are connected to each other. Those elements are called *resources* and the connections are named *properties*. Two resources connected by a property form a *statement*. For example, painter Rembrandt (a resource) has painted (property) painting Abraham (a resource). This statement can simple be written as follows.

(Rembrandt, Paints, Abraham)

A set of statements is called an RDF graph. The following definition of an RDF graph is given by the specifications of RDF [7]. Note that in terms of RDF, a triple is equivalent to a statement and so these terms are used interchangeably. The terms used in the following definition are explained below.

Definition (RDF (Triple) Graph). An *RDF graph* is a set of RDF triples. An RDF triple (s, p, o) consists of three components, namely

- a subject s which is a URI reference or a blank node,
- a predicate p which is a URI reference, and
- an object o , which is a URI reference, a blank node or a literal. □

URI stands for Uniform Resource Identifier. *URI references* are resources and basically represent ‘something that exists and can be identified’. URI references of web pages correspond to the URLs of the pages, for example a web page about Rembrandt is a URI reference resource, viz. “http://www.european-history.com/rembrandt.html”. A URI reference, however, can also identify an object that is not directly accessible via the Web, such as a painting in a museum, say Abraham. The URI reference for Abraham then is “http://www.louvre.fr/#Abraham”. In order to abbreviate these URI references, the XML namespace facility can be used. A namespace is basically a set of URI references. By linking a resource to a namespace, confusion between independent and possibly conflicting definitions of the same resource is avoided. An example of abbreviating a URI reference is as follows. Once “xmlns: euro = http://www.european-history.com/” is defined, all prefixes “http://www.european-history.com/” can be replaced by prefix “euro:”.

Blank nodes are also resources and represent ‘something that exists’. The difference with URI references is that blank nodes have no URI, although every blank node is a unique resource. In order to distinguish blank nodes from each other, local blank node identifiers are used. Blank nodes can only be used for a subject or object in a statement. A blank node can be allocated to another resource which means that the blank node is a copy of that resource, except for its name. This term is mainly used in the entailment rules, see Section 2.5. An example of a statement with a blank node is saying that Rembrandt has painted something.

(Rembrandt, Paints, BlankNodeIdentifier)

A *property* of a triple is also called a predicate. In terms of RDF, predicate and property are equivalent and will be used interchangeably. A property defines a binary relation between resources and is itself also a resource. This means that properties can also be used as subjects or objects of triples, as is shown in the following statement.

(Paints, UsesAttribute, Paintbrush)

Literals identify words and values, such as numbers and dates, and are not seen as resources. They can only be the object of a statement. Literals are either plain or typed: plain literals are strings combined with an optional language tag and typed literals are strings combined with a datatype. In this document, literals are considered as strings.

2.2 RDF Schema

The term RDF schema (RDFS) is used to define groups of related resources and the relationships between these resources. An RDF schema is an RDF graph which uses specific RDFS terms, such as Class, SubClassOf, SubPropertyOf, Domain and Range. These terms

are explained next. Note that RDF graphs can use properties and classes of RDF schema graphs, and so implicitly use all the characteristics of these properties/classes defined in the schema. These characteristics become explicit when reasoning is applied to an RDF schema.

Resources may be divided into groups called *Classes* and the members of a class are known as instances. Classes are also resources. Subclasses are denoted by the term *SubClassOf*. If a class is a subclass of another class, then all resources which are an instance of the first class are also an instance of the second class, as is shown in the next example. Class Painter is a subclass of class Artist. If it is defined that Rembrandt is an instance of Painter, it can be concluded by reasoning that Rembrandt is also an instance of Artist.

$$\frac{(Painter, SubClassOf, Artist) \quad (Rembrandt, IsInstanceOf, Painter)}{(Rembrandt, IsInstanceOf, Artist)}$$

As mentioned, a property defines a relation between two resources. The term *SubPropertyOf* is used to state that one property is a subproperty of another meaning that all pairs of resources related by the first property are also related by the second property. This is explained by the following example. Property Paints is a subproperty of property Creates. If it is defined that Rembrandt has painted Abraham, it can be concluded by reasoning that Rembrandt has also created Abraham.

$$\frac{(Paints, SubPropertyOf, Creates) \quad (Rembrandt, Paints, Abraham)}{(Rembrandt, Creates, Abraham)}$$

Domains and *Ranges* can be defined for properties. In the following example, the first two statements define that property Paints connects instances of class Painter to instances of class Painting. Given that Rembrandt has painted Abraham, it can be concluded by reasoning that Rembrandt is an instance of class Painter and Abraham is an instance of class Painting.

$$\frac{(Paints, Domain, Painter) \quad (Paints, Range, Painting) \quad (Rembrandt, Paints, Abraham)}{(Rembrandt, IsInstanceOf, Painter) \quad (Abraham, IsInstanceOf, Painting)}$$

2.3 RDF Levels

As is explained in detail in [16], the strength of RDF is that RDF data can be approached at three different, but equivalent levels. Those levels are explained next.

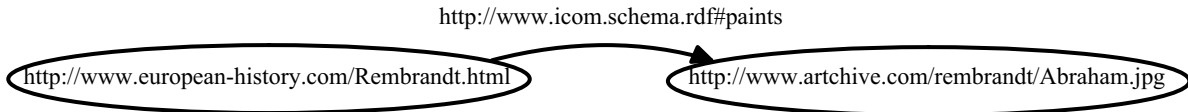
The *syntactic level* considers RDF as data contained in web documents expressed in the XML syntax. Using this syntax gives rise to extra XML-specific information. Note that an RDF graph is an unordered set whereas XML data is a linearly ordered tree. Therefore, an RDF graph can not be represented in a unique XML tree, but can be written into differently ordered XML trees. Nevertheless, these trees are all semantically equivalent. An example of an RDF triple in the XML syntax is as follows.

```
< rdf : Description rdf : about = "http://www.european-history.com/rembrandt.html" >
  < http://www.icom.com/schema.rdf#paints
    rdf : resource = "http://www.artchive.com/rembrandt/abraham.jpg" / >
< /rdf : Description >
```

The *structural level* considers an RDF graph as an unordered set of piecewise structured triples, as is explained in Section 2.1. This representation is usually chosen when RDF data is stored in databases. An example of an RDF triple is given next.

(*http://www.european-history.com/rembrandt.html*,
http://www.icom.com/schema.rdf#paints,
http://www.artchive.com/rembrandt/abraham.jpg)

At the *semantic level*, the unordered set of triples is combined to a coherent unity which rapidly makes relations in the data clear. In this document, such a unity is represented as a directed, labeled graph and is called an RDF semantic graph. An example of such a graph is as follows.



2.4 RDFS Closure

As mentioned, reasoning can be applied to an RDF graph G to derive new information from this graph. This results in a new, extended RDF graph called an RDFS closure graph.

Lots of rules to reason with RDF data are given in the specifications of RDF [7], and are called the RDF and RDFS entailment rules. However, the way and order to apply these rules is not dictated. Therefore, many RDFS closure computations are possible. Note that all computations starting with a given graph result in semantically equivalent RDFS closure graphs.

A possible computation for the RDFS closure is given in the specifications of RDF [7] and is briefly presented in Appendix A. This computation is based on adding axiomatic triples first and then applying rules until the graph remains unchanged. Because outputs of rules will trigger other rules, this computation is, however, iterative and redundant. Furthermore, these RDFS closure graphs are infinite, due to the infinite axiomatic triple set. Fortunately, this infinite set can be converted into an equivalent finite set by the Finite Container Construction presented in Appendix A. This results in a semantically equivalent, finite closure graph, also called a *partial RDFS closure graph* [24]. An advantage of this partial closure, besides that it is finite, is that it can be computed in polynomial time. This is proven next, after the introduction of some terms used in this proof.

Rules lg, rdf1 and rdfs2 are reasoning rules and are explained in Appendix A. The following formulation of RDF graph G is used, where U is an infinite set of URI references, B an infinite set of blank nodes and L an infinite set of literals.

$$G \subset U \cup B \times U \times U \cup B \cup L$$

Furthermore, partial RDFS closure graph $G_{s,K}$ is computed following the Finite Container Construction and the computation steps presented in Appendix A. Note that this proof is

based on a proof given in [24].

Lemma. *Given a finite RDF graph G with K being the finite¹ set of container member indices in G . The partial RDFS closure graph $G_{s,K}$ of G is finite and computable in polynomial time.*

Proof. Given a finite RDF graph $G \subset U \cup B \times U \times U \cup B \cup L$, the set of axiomatic triples A is finite² so the first step of the RDFS closure computation, adding the axiomatic triples, will give a finite graph $G_K \cup A$. Given that only rule lg adds new resources³, applying rule lg results in $G_K \cup A \xrightarrow{lg} U_0, B_0, L_0$ where U_0, B_0, L_0 are finite sets. The next step of the RDFS closure computation amounts to applying (RDF and RDFS) entailment rules, which results in $G_{s,K} \subset U_0 \cup B_0 \times U_0 \times U_0 \cup B_0 \cup L_0$. Because of the finiteness of the sets U_0, B_0 and L_0 , $G_{s,K}$ is finite.

Next, it has to be proven that $G_{s,K}$ is computable in polynomial time. Let $g = |G|$ the number of RDF triples and $k = |K|$ the number of container members. The first step of the closure computation process adds⁴ $A = 48 + 3k$ axiomatic triples and the next step, rule lg, adds at most g triples⁵. It follows that $G_K \cup A$ has at most $2g + 3k + 48$ triples, so that the sets U_0, B_0, L_0 can be chosen to satisfy⁶ $|U_0 \cup B_0 \cup L_0| \leq 3(2g + 3k + 48)$. The maximum number of triples for the closure then is $|G_{s,K}| \leq 27(2g + 3k + 48)^3$ as no new resources are added. Therefore, in the remaining steps at most $27(2g + 3k + 48)^3$ rule applications can add a new triple to the partial closure graph under construction. For each of the rules used, it can be determined whether a successful rule application exists in at most linear or quadratic time as a function of the size of the partial closure graph under construction. It follows that $G_{s,K}$ can be computed in time polynomial in g , i.e., the number of triples of G . \square

As mentioned, the closure computation given by the specifications of RDF [7] is iterative and redundant. It is possible to compute an RDFS closure in a more efficient way by advanced systems which use, for example, forward-chaining rule-based techniques. Unfortunately, the computation of the closure graph does not solely depend on the number of data and schema statements, but merely on the content of these statements. This field of research is explained in [17]. In brief, it can be concluded that the size of an RDFS closure and the most efficient RDFS closure computation differ for every given RDF graph.

The fact that the size of an RDFS closure differs for every graph is explained in the following example. WordNet [11] is an online lexical reference system and is represented as an RDF data set of 473,626 statements. The RDFS closure of WordNet consists of 573,457 statements, which is 20% bigger. However, if WordNet had defined some extra transitive SubPropertyOf predicates, the closure would become 250% bigger, so about 1.3 million statements.

¹As graph G is finite, it contains a finite number of container members, and so the set of container member indices K is finite.

²The set of RDF axiomatic triples is made finite by set K and the finite container construction given in Appendix A.

³Rule lg adds allocated blank nodes for literals.

⁴The finite container construction given in Appendix A adds three new triples for each container member.

⁵If every triple has a literal as an object, then allocated blank nodes are added for all these literals, and so g triples are added.

⁶ $|U_0 \cup B_0 \cup L_0| = 3(2g + 3k + 48)$ when all the resources and literals are pairwise unequal.

2.5 Entailment

The term *entailment* denotes a semantic relationship between RDF graphs which holds when the graphs have the same semantics. Note that entailment is not the same as equivalence, as entailment verifies whether RDF graphs have the same semantics whereas equivalence determines whether RDF graphs also have the same structure.

Four types of entailment are distinguished, namely simple entailment, RDF entailment, RDFS entailment, and D entailment. Simple entailment amounts to verifying whether RDF graphs have the same semantics. RDF entailment also compares the semantics of RDF graphs, but applies some basic reasoning first. RDFS entailment covers RDF entailment, and applies more extensive reasoning, e.g. by using schema information. D entailment is an optional entailment layer covering the previous entailments, and verifies the correctness of given XML datatypes. Since the rule set for D entailment is not complete yet, D entailment is outside the scope of this research. As RDF entailment is included in RDFS entailment, only the terms simple entailment and RDFS entailment are explained next.

2.5.1 Simple Entailment

Simple entailment verifies whether two given, finite RDF graphs have the same semantics. The definition of simple entailment [5] uses the term ‘instance of’. A graph A is an instance of graph B if resources or literals in A correspond to blank nodes in B . In other words, if B is a generalization of A . Next, simple entailment and instance are defined and depicted in Figure 2.1.

Definition (Simple Entailment). RDF graph G *simply entails* RDF graph Q if and only if a subgraph G' of G is an instance of Q . \square

Definition (Instance). Suppose M is a mapping from a set of blank nodes to some set of literals, blank nodes and URI references. Then any RDF graph G' obtained from an RDF graph Q by replacing some or all of the blank nodes in Q by M is an *instance* of Q . \square

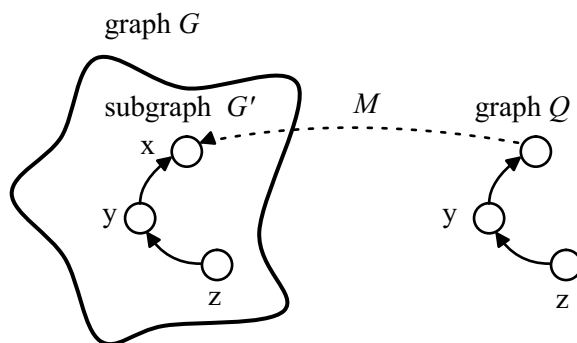


Figure 2.1: Simple Entailment

Note that simple entailment can be implemented in two ways, namely the instance solving method and the equivalence solving method. Both are explained next.

Instance Solving Method An obvious method to check whether G simply entails Q is to actually find a subgraph G' of G and demonstrate that G' is an instance of Q , i.e., that G' is obtained from Q by replacing some or all blank nodes in Q by literals, URI references or blank nodes. This amounts to finding a subgraph and a vertex mapping.

Equivalence Solving Method Another method to check whether G simply entails Q is using the simple entailment rules given by the specifications of RDF [7]; see also Appendix A. First, those rules are applied to G and generate a generalization of G by adding allocated blank nodes. Second, a subgraph G' of G has to be found to which Q is equivalent. A disadvantage of the simple entailment rules is that blank nodes can proliferate so applying these rules naively would not result in an efficient search process, since the rules will not terminate and can produce arbitrarily many redundant derivations of equivalent triples.

2.5.2 RDFS Entailment

Just as simple entailment, RDFS entailment verifies whether two given RDF graphs have the same semantics. However, extensive reasoning is applied to one of the graphs first. This reasoning results in a partial RDFS closure graph which contains all extended data and schema information; see also Section 2.4. Because of this closure graph, it is obvious that RDFS entailment leads to more meaningful conclusions than simple entailment.

Note that the definition of RDFS entailment given in [5] contains the possibility of an XML-clash. An XML-clash is a rarely occurring event during the RDFS closure computation of a graph. As a detection and correction procedure for an XML-clash is presented in Appendix A, this term is left out in the following definition of RDFS entailment.

Definition (RDFS Entailment). RDF graph G *rdfs-entails* RDF graph Q if and only if partial RDFS closure graph $G_{s,K}$ simply entails Q . \square

2.5.3 Complexity

For two given finite RDF graphs G and Q , simple entailment is proven to be NP-complete [25]. The partial RDFS closure graph $G_{s,K}$ can be computed in polynomial time and so RDFS entailment is NP-complete as well. Simple entailment can be computed in polynomial time when Q is assumed to have no blank nodes [25]. This is elaborated on in Chapter 4.

2.6 Conclusion

The basic concept of RDF is an RDF graph consisting of triples of the form (*subject, predicate, object*). Extensive reasoning can be applied to an RDF graph which results in a finite partial RDFS closure graph. This graph is computed for RDFS entailment. Simple entailment verifies whether two graphs have the same semantics and is NP-complete.

Chapter 3

RDF Query Languages And Systems

As mentioned, RDF data can be queried at the syntactic level as well as the structural level and the semantic level. These query levels are discussed in Section 3.1. Recently, several RDF query languages have been developed although no standard language has yet been determined. This is discussed in Section 3.2. Finally, two leading RDF systems are presented in Section 3.3.

3.1 Querying RDF Levels

This section deals with querying RDF data at the three different representation levels.

Considering an RDF graph in the XML representation, XML query languages such as XQuery can be used. However, these languages are designed to query XML trees and disregard the special semantics of RDF data.

A number of query languages have been proposed and implemented for querying RDF graphs represented as sets of triples. As this data is piecewise structured, this representation is suitable for database storage. This brings about lots of database optimizations and query possibilities.

Querying RDF data represented as directed, labeled graphs is a recent field of research. Representing RDF data as these graphs makes relations in the data more clear and enables the possibility of using existing graph definitions and algorithms. In this document, querying a directed, labeled graph is considered as finding a query subgraph in this graph, which is closely related to the Subgraph Isomorphism problem [21].

3.2 RDF Query Languages

Several languages for querying RDF documents have been proposed, some in the tradition of database query languages such as SQL and OQL, other more closely related to rule languages. Nonetheless, no standard for RDF query languages has yet been determined. In [23], a comparison of six RDF query languages is given, based on five languages properties,

such as expressiveness and safety. It can be concluded that the same queries formulated in different query languages can return different answers. This field of research is also discussed in [14, 22]. Next, two commonly used query languages, RDQL and RQL, are explained briefly.

RDQL [6] stands for RDF Data Query Language and is an implementation of an SQL-like query language. RDQL is based on the triple set representation of an RDF graph, and makes no distinction between schema and data information. An example of an RDQL query is as follows.

```
select ?painter where (?painter, < paints >, < Abraham >)
```

RQL [8] is a declarative RDF query language based on the graph representation of an RDF graph supporting RDF schema information. Furthermore, this typed language follows a functional approach and supports generalized path expressions featuring variables for vertices (classes) and edges (properties). RQL follows an OQL-like syntax, as is shown in the following query example.

```
select Painter from {Painter} paints {Abraham}
```

3.3 RDF Query Systems

In [22], an evaluation of several RDF systems is presented. Jena and Sesame are two leading RDF query systems. Jena is an open-source Java class library based on query language RDQL. Sesame can, among other languages, handle RQL and allows storage and querying of RDFS. Both systems are explained next.

Jena [3] considers RDF data as a set of triples, and makes no distinction between data and schema information. Jena is based on the RDQL query language [6] and has extensive RDF-based querying facilities, comparable in syntax to SQL queries for traditional relational databases. The storage of RDF data for Jena is always done in databases, and so database optimizations are used. Jena used to store literals and resources in common tables that were referenced by statements. Recently, however, Jena has added a database layout using a de-normalized schema in which literals and resources are stored directly in statement tables. This uses more storage space but enables faster data insertion and retrieval. A Jena reasoner subsystem is proposed as a first step towards trying to deal with schema information and it includes a generic rule based inference engine together with configured rule sets for RDFS.

Sesame [1] is an open source system for managing and querying RDF data and supports RDF schemas and reasoning. An advantage of Sesame is its architecture of layers. The actual storage of RDF is separated from functional modules offering RDF operations, and separated from the communication with these functional modules from outside the Sesame system. So, unlike Jena, Sesame does not depend on a specific storage device. An important layer is RDF SAIL (Storage And Inference Layer) which is used to abstract from the actual storage device. This layer has methods for storing, removing and querying RDF in/from a repository. It is possible to build a SAIL layer on top of any kind of storage, for example a database or a file or a peer-to-peer network. However, current implementations are based on relational databases.

SeRQL is a query language similar to RQL developed by Sesame designers to offer specific functionality. Sesame supports inferencing and user-defined entailment rules to allow custom,

domain-specific inferencing.

The RQL query Module [1] of Sesame handles a query by building it into a query tree model. Each object of this model now evaluates itself, fetching data from SAIL where needed. An advantage of this approach is the streaming fashion in which the results can be returned. Also, schema data can be stored separately in the SAIL which optimizes the semantic aim of RDF.

3.4 Conclusion

Existing RDF query systems can interpret and reason with RDF data. They mostly work with databases and available database optimizations, based on the RDF triple set representation. Note that this document only deals with querying RDF data, and it is assumed that given RDF data is already interpreted and reasoned.

Chapter 4

RDF Subgraph Problem

RDF data is represented as a directed, labeled graph, called an RDF semantic graph. This chapter describes how RDF is related to graph-theoretical terms and gives a definition for semantic graphs. Furthermore, the equivalence of an RDF (triple) graph and an RDF semantic graph is proven. Querying RDF data is formulated as the RDF Subgraph problem, using the semantic graph definition.

The following definitions from [21] are used in this chapter. Subgraph Isomorphism is defined on two graphs, and verifies whether one graph has a subgraph isomorphic to the other graph. Given an integer K , Clique verifies whether a graph contains a clique of size K or more.

Definition (Subgraph Isomorphism). Given two graphs $G = (V_1, E_1)$ and $H = (V_2, E_2)$, does there exist a subset $V' \subseteq V_1$ and a subset $E' \subseteq E_2$ such that $|V'| = |V_2|$, $|E'| = |E_2|$ and there exists a one-to-one function $f : V_2 \rightarrow V'$ satisfying $\{u, v\} \in E_2$ if and only if $\{f(u), f(v)\} \in E'$? \square

Definition (Clique). Given a graph $G = (V, E)$ and a positive integer $K \leq |V|$, does G contain a clique of size K or more, i.e., a subset $V' \subseteq V$ with $|V'| \geq K$ such that every two vertices in V' are joined by an edge in E ? \square

4.1 RDF Graph Definitions

In this section, a definition for an RDF semantic graph is given. The definition of an RDF (triple) graph is already presented in Chapter 2, but is given here again for convenience.

Definition (RDF (Triple) Graph). An *RDF graph* is a set of RDF triples. An RDF triple (s, p, o) consists of three components, namely

- a subject s which is a URI reference or a blank node,
- a predicate p which is a URI reference, and
- an object o , which is a URI reference, a blank node or a literal. \square

This definition can be rewritten in the following equivalent formulation. Let U be an infinite set of URI references, B an infinite set of blank nodes and L an infinite set of literals. An

RDF graph G is a set of triples, denoted by $G = \{t_1, \dots, t_n\}$, with triple $t_i = (s_i, p_i, o_i)$ where $s_i \in U \cup B$, $p_i \in U$ and $o_i \in U \cup B \cup L$. Next, a conversion procedure is presented to transform an RDF (triple) graph into an RDF semantic graph.

Conversion procedure for RDF (triple) graph to RDF semantic graph. Triple (s, p, o) denotes that s has a relation to o called p . Converting this triple into graph theoretical terms can be done by the following construction, see also Figure 4.1.

subject s \rightarrow vertex v_i with label s
 predicate p \rightarrow arc e with label p , connecting v_i to v_j
 object o \rightarrow vertex v_j with label o

A set of triples can now be converted into a semantic graph by applying the construction above for every triple of the set. However, if a subject or object already occurs as a vertex label in the graph, the same vertex must be used.

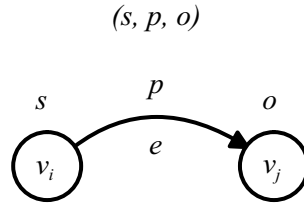


Figure 4.1: Conversion of RDF triple into Semantic Graph

The definition of a semantic graph can now be given as follows.

Definition (RDF Semantic Graph). Given that U is a set of URI references, B a set of blank nodes and L a set of literals, an *RDF semantic graph* G is a 5-tuple (V, E, f, l_v, l_e) consisting of

- $V = \{v_1, \dots, v_n\}$ the set of vertices,
- $E = \{e_1, \dots, e_m\}$ the set of arcs,
- subject-object function $f : E \rightarrow V \times V$
 such that every vertex is connected, i.e.,
 $\forall v \in V : \exists e \in E : f(e) = (v_i, v_j) \wedge v \in \{v_i, v_j\}$, (1)
- labeling function $l_v : V \rightarrow U \cup B \cup L$
 such that every vertex label is unique, i.e.,
 $\forall v_i, v_j \in V : l_v(v_i) = l_v(v_j) \Rightarrow v_i = v_j$ (2)
 and every literal labeled vertex has no outgoing arcs, i.e.,
 $l_v(v) \in L \Rightarrow outdegree(v) = 0$, and (3)
- labeling function $l_e : E \rightarrow U$
 such that arc labels between the same vertices are pairwise disjoint, i.e.,
 $\forall_{i \neq j} e_i, e_j \in E : f(e_i) = f(e_j) \Rightarrow l_e(e_i) \neq l_e(e_j)$. (4) \square

This definition is based on the pseudograph definition [4], where multiple edges and loops are allowed. The set of arcs is not defined as an ordered pair of vertices, i.e., $E \rightarrow V \times V$, but as a set of arcs $E = e_1, \dots, e_m$, because multiple arcs between the same pair of vertices are possible and every arc must have a unique label. Function $f : E \rightarrow V \times V$ is chosen for the subject-object allocation of an arc. Since a triple always defines a connection between two resources, it is not possible to have an unconnected vertex in the semantic graph (Condition 1). Reusing an existing vertex in the conversion construction is crucial to obtain a consistent semantic graph representation. So, considering U , B and L as infinite sets, every vertex label occurs at most once in an RDF graph (Condition 2). According to the specifications of RDF [7], subjects are not allowed to be a literal, so only vertices corresponding to an object can have a literal label. According to Condition 3 the labeling function can only label a vertex with a literal when the outdegree of the vertex is equal to 0. Condition 4 is also necessary to obtain a consistent semantic graph. An RDF graph is an unordered set of triples, so equivalent triples do not occur. Therefore, two arcs with the same label between two vertices are not allowed in a semantic graph as they correspond to the same triple.

Lemma 1. *In an RDF semantic graph, every vertex label is unique.*

Proof. This lemma follows directly from the construction of an RDF semantic graph. \square

Lemma 2. *For an RDF semantic graph with n vertices and m arcs, it holds that $n \leq 2m$.*

Proof. If every vertex is connected to exactly one arc, it holds that $n = 2m$. As vertices must be connected, $n < 2m$ holds in all other cases. Therefore, it holds that $n \leq 2m$, i.e., the number of vertices is bounded by the number of arcs. \square

Lemma 3. *In an RDF semantic graph with n vertices, every arc label occurs at most n^2 times.*

Proof. According to Condition 4, every arc label can occur at most once between two vertices. Because loops are allowed, every arc label occurs at most $n \times n = n^2$ times in a graph. \square

Lemma 4. *An RDF graph represented as a set of triples is equivalent to an RDF graph represented as a semantic graph.*

Proof. semantic graph \Rightarrow set of triples. A label of a vertex is an element of $U \cup B \cup L$, and so corresponds to a subject or an object. A label of an arc is an element of U and so corresponds to a predicate. All vertices and arcs have exactly one label because of the labeling functions l_v and l_e . Every arc connects two vertices, i.e., function f is defined for all elements of E , so a 3-tuple (vertex label, arc label, vertex label) corresponds to an RDF triple (*subject, predicate, object*). Condition 1 excludes the occurrence of unconnected vertices, in accordance with the triple construction where it is also not possible to have ‘empty elements’, for instance (*subject, predicate,*). Condition 3 corresponds exactly to one of the basic RDF issues stating that only objects can be a literal. Conditions 2 and 4 contribute to a consistent conversion.

set of triples \Rightarrow semantic graph. A subject is an element of $U \cup B$, and so corresponds to a label of a vertex. A predicate is an element of U , and so corresponds to an arc label. An object is an element of $U \cup B \cup L$, and so corresponds to a vertex label. The triple (s, p, o) amounts to a relation p from s to o . This can also be represented as an arc with label p from a vertex with label s to a vertex with label o . So an RDF triple corresponds to two labeled vertices connected with a labeled arc. It is assumed that the given triples satisfy the basic

RDF issues, and therefore, Conditions 1 to 4 hold by definition. \square

Next, a method to verify whether an arbitrary graph is an RDF semantic graph is given. Let G be a given directed, labeled graph. First, G is written into a 5-tuple (V, E, f, l_v, l_e) , according to the following steps. The number of vertices of G is called n and V is the set of vertices with $|V| = n$. The number of arcs is called m and E is the set of arcs with $|E| = m$. Function f defines for every arc the vertex it comes from and the vertex it goes to. Finally, labeling functions l_v and l_e are constructed for the mappings from the vertices and arcs to their labels. If G cannot be written into this 5-tuple, G is not an RDF semantic graph. Secondly, the following conditions have to be verified for $G = (V, E, f, l_v, l_e)$, which correspond to Conditions 1 to 4 given in the RDF semantic graph definition.

- Condition 1. Every vertex is connected, i.e., every vertex is contained at least once in f . This can be computed efficiently by walking through f and marking item i in an array $a[1..n]$ at the occurrence of vertex v_i in f . G does not contain an unconnected vertex if for every vertex v_i it holds that $a[i] > 0$ at the end of the matching procedure.
- Condition 2. There are no two vertices with the same URI reference or literal label. This can be verified by sorting the vertex labels alphabetically in an array $b[1..n]$ and checking for every element $b[i]$ whether it differs from its successor.
- Condition 3. A literal label is only attributed to an object vertex, i.e., a vertex with a literal label must have outdegree = 0. An array $c[1..n]$ can store the number of times a vertex occurs in the first position of function f . For every vertex v_i it holds that $c[i]$ is the outdegree of v_i at the end of the matching procedure.
- Condition 4. Labels of arcs with the same subject and object vertex have to be pairwise disjoint. This can be done by sorting f alphabetically and checking whether arcs between the same vertices have different labels.

Lemma 5. *It can be verified in $O(m \log m)$ time whether a given directed, labeled graph is an RDF semantic graph.*

Proof. Writing G into a 5-tuple of the form (V, E, f, l_v, l_e) amounts to defining sets and functions for the vertices and arcs of G . Because n is bounded by m (lemma 2), this costs $O(m)$ time. Condition 1 and 3 walk through function f , array a and array c . This costs $2m + n + n = O(m)$ time. Condition 2 and 4 sort an array of length n and length m and walk through these arrays. This costs $n \log n + m \log m + n + m = O(m \log m)$ time. Consequently, the verification whether a given graph is an RDF semantic graph can be carried out in $O(m + m + m \log m) = O(m \log m)$ time. \square

Concluding, an RDF semantic graph has the following graph properties.

- The graph is directed and the vertices and arcs are labeled.
- The graph may have multiple but differently labeled arcs between two vertices.
- The graph may contain cycles.
- The graph does not have to be connected.

4.2 RDF Subgraph Problem Definition

As discussed in Section 2.5, RDFS entailment results in more meaningful conclusions than simple entailment because a partial RDFS closure graph is computed and used. Therefore, RDFS entailment is used as the basis for the RDF Subgraph problem. Although a partial closure graph can be computed in polynomial time, the most efficient computation is specific for every graph; see Section 2.4. Therefore, it is assumed that a partial RDFS closure graph is already computed and given. What remains is verifying whether graphs have the same semantics. Consequently, the RDF Subgraph problem amounts to simple entailment with a given partial RDFS closure graph.

Section 2.5.1 describes two possible implementations of simple entailment. As explained, the simple entailment rules for the equivalence solving method can give rise to infinite blank node reproductions and require specific application methods. Therefore, the instance solving method is chosen, which basically amounts to finding a subgraph and a vertex mapping.

Assuming that G is a partial RDFS closure graph, the RDF Subgraph problem amounts to the question whether RDF semantic graph G simply entails RDF semantic graph Q . In other words, given two RDF semantic graphs G and Q , does G contain a subgraph which is an instance of Q ? This is the main issue of this document. The problem is formulated as a decision problem, because the theory of NP-completeness can only be applied to decision problems. Such problems have only two possible solutions, the answer ‘yes’ or ‘no’. Next, a formal definition of the RDF Subgraph problem is given, which is depicted in Figure 4.2.

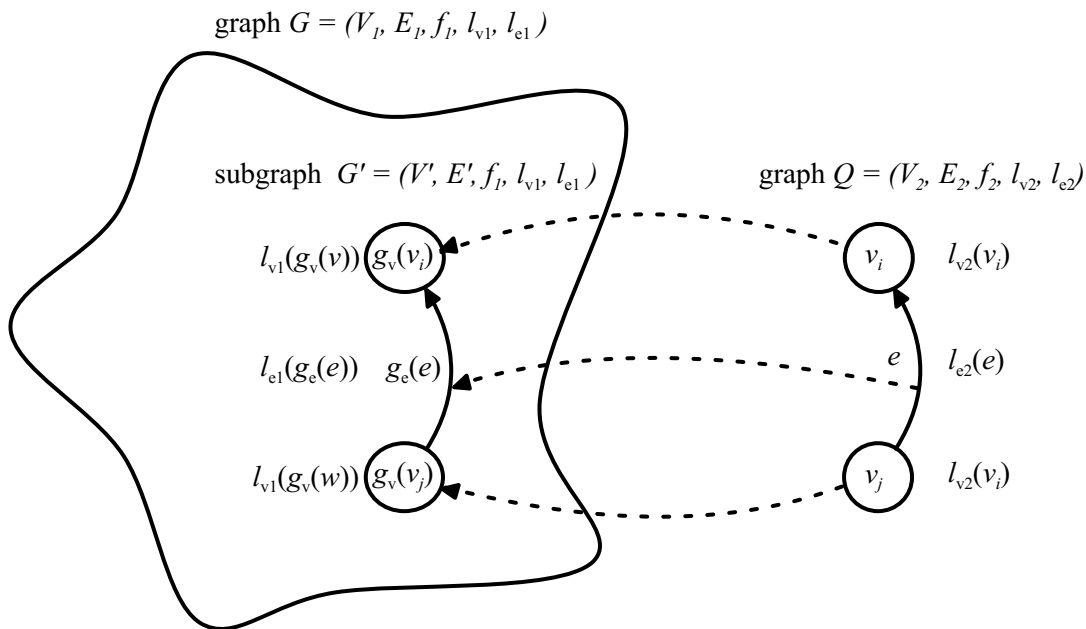


Figure 4.2: RDF Subgraph Problem

Definition (RDF Subgraph Problem). Given two RDF semantic graphs $G = (V_1, E_1, f_1, l_{v_1}, l_{e_1})$ and $Q = (V_2, E_2, f_2, l_{v_2}, l_{e_2})$, is there a $G' = (V', E', f_1, l_{v_1}, l_{e_1})$ with $V' \subseteq V_1$ and $E' \subseteq E_1$ such that

- $|V_2| \geq |V'|$,
- $|E_2| \geq |E'|$,
- there exists a surjection $g_v : V_2 \rightarrow V'$,
- there exists a surjection $g_e : E_2 \rightarrow E'$,
- subjects and objects are preserved, i.e.,
 $f_2(e) = (v_i, v_j)$ if and only if $f_1(g_e(e)) = (g_v(v_i), g_v(v_j))$, and
- URI reference and literal labels in Q map on identical labels in G' , i.e.,
 $\forall v \in V_2 : (l_{v_2}(v) \in U \cup L) \Rightarrow (l_{v_2}(v) = l_{v_1}(g_v(v)))$
 $\forall e \in E_2 : l_{e_2}(u) = l_{e_1}(g_e(u))$. □

The simple entailment concept states that the semantics of subgraph G' has to be the same as the semantics of graph Q , which is fundamentally different than stating that G' has to be isomorphic to Q . Several vertices of Q can be mapped on the same vertex of G' , so $|V_2| \geq |V'|$ has to hold and function g_v has to be a surjection, idem for arcs. Note that therefore the RDF Subgraph problem is not the same as the Subgraph Isomorphism problem. Only when $|V_2| = |V'|$ and $|E_2| = |E'|$, the mappings are 1 : 1 and the graphs are also isomorphic.

In [25], simple entailment is proven to be NP-complete by reduction to Clique. A proof that the RDF Subgraph problem is NP-complete is given next and is based on the proof given in [25]. Note that the definition of Clique, which is known to be NP-complete, is given in the beginning of this chapter.

Lemma 6. *The RDF Subgraph problem is NP-complete.*

Proof. It is clear that the RDF Subgraph Problem is in NP: guess a vertex mapping function h from Q to G and check whether Q with the vertices mapped is a subgraph of G . What remains is a reduction of Clique to RDF Subgraph. An instance of the clique problem consists of a finite undirected graph $G = (V, E)$ and a positive integer $k \leq |V|$. E can be assumed to contain no loops $\{v, v\}$. An instance $G = (V, E), k$ of the clique problem is transformed into an instance for the RDF Subgraph problem, say RDF graphs Q' and G' . RDF graph Q' has exactly k vertices, all with a blank node label, and contains every possible arc with label p between those vertices, except for loops. RDF graph G' is formed by all vertices $v \in V$ which have a blank node label, and by converting each edge $\{v, w\} \in E$ into two arcs, one from vertex v to vertex w and one from w to v , both with label p . This transformation can be done in polynomial time. It is now sufficient to prove that the finite undirected graph without loops G has a clique of size $\geq k$ if and only if there is a function h mapping blank node labeled vertices of Q' to vertices of G' , such that $h(Q')$ is a subgraph of G' . If there is such a function h , than h has to be injective. Note that otherwise two vertices of Q' can be mapped on one vertex in G' , and so the arcs in Q' between those vertices form a loop in G' , contradicting the assumption that G does not contain loops. So, since h is injective, G has a clique of size k . □

Lemma 7. *If Q contains no blank node labeled vertices, the RDF Subgraph problem is polynomially solvable.*

Proof. The RDF Subgraph problem verifies whether graph G contains a subgraph G' that is an instance of graph Q . Let h be a mapping from a set of blank node labeled vertices of Q to a set of vertices of G . Then G' has to be obtained from Q by replacing blank node labeled vertices of Q by mapping h . However, it is assumed that Q does not contain blank node labeled vertices, so mapping h is empty. Therefore, G' must have the exact same sets of vertex labels and arc labels as Q . The vertex label set of Q consists only of URI reference and literal labels, each corresponding to unique vertices in Q . This set also corresponds to a unique set of vertices in G . Consequently, verifying whether G contains such a subgraph G' amounts to checking for every vertex v_i of Q whether the label of v_i occurs as a vertex label in G . Secondly, it has to be verified whether the vertices found in G are connected by the same arc labels as the vertices in Q . This verification can be done in a time polynomial in the number of vertices and arcs of graph G . \square

The RDF Subgraph Problem is defined as a decision problem for two given graphs. In order to query G , graph Q has to contain some variable labels for vertices and arcs. The RDF Subgraph problem then amounts to finding all possible values for these variables. First, an RDF semantic query graph Q is defined, in which blank node labels are also considered as variable labels. This is because simple entailment allows a mapping h from blank nodes of Q to arbitrary resources of G ; see Section 2.5.1.

Definition (RDF Semantic Query Graph). Given that U is a set of URI references, L a set of literals and K a set of variables, an RDF semantic query graph Q is a 5-tuple (V, E, f, l_v, l_e) consisting of

- $V = \{v_1, \dots, v_n\}$ the set of vertices,
- $E = \{e_1, \dots, e_m\}$ the set of arcs,
- subject-object function $s : E \rightarrow V \times V$
such that every vertex is connected, i.e.,
 $\forall v \in V : \exists e \in E : f(e) = (v_i, v_j) \wedge v \in \{v_i, v_j\}$,
- labeling function $l_v : V \rightarrow U \cup L \cup K$
such that every vertex label is unique, i.e.,
 $\forall v_i, v_j \in V : l_v(v_i) = l_v(v_j) \Rightarrow v_i = v_j$
and every literal labeled vertex has no outgoing arcs, i.e.,
 $l_v(v) \in L \Rightarrow \text{outdegree}(v) = 0$, and
- labeling function $l_e : E \rightarrow U \cup K$
such that every variable arc label occurs at most once, i.e.,
 $\forall e_i \in E : l_e(e) \in K \Rightarrow \neg(\exists e_j \in E \text{ and } i \neq j : l_e(e_j) = l_e(e_i))$
and such that arc labels between the same vertices are pairwise disjoint, i.e.,
 $\forall_{i \neq j} e_i, e_j \in E : f(e_i) = f(e_j) \Rightarrow l_e(e_i) \neq l_e(e_j)$. \square

Note that the semantic query graph definition differs from the semantic graph definition in the labeling of the vertices and arcs, i.e., the set of variables K is added and all variable labels have to be unique. This design decision excludes the possibility of using the same variable

label more than once in a query. In order to obtain the same values for variables, the values found for the variables can be checked for equality afterwards, since all possible values for variables are returned.

The RDF Subgraph problem now amounts to finding all possible values for the variable labels K of graph Q . Note that the mapping of the variables is not explicitly mentioned in the definition given next, but follows implicitly.

Definition (RDF Subgraph Problem). Given an RDF semantic graph $G = (V_1, E_1, f_1, l_{v_1}, l_{e_1})$ and an RDF semantic query graph $Q = (V_2, E_2, f_2, l_{v_2}, l_{e_2})$, find all $G' = (V', E', f_1, l_{v_1}, l_{e_1})$ with $V' \subseteq V_1$ and $E' \subseteq E_1$ such that

- $|V_2| \geq |V'|$,
- $|E_2| \geq |E'|$,
- there exists a surjection $g_v : V_2 \rightarrow V'$,
- there exists a surjection $g_e : E_2 \rightarrow E'$,
- subjects and objects are preserved, i.e.,
 $f_2(e) = (v_i, v_j)$ if and only if $f_1(g_e(e)) = (g_v(v_i), g_v(v_j))$, and
- URI reference and literal labels in Q map on identical labels in G' , i.e.,
 $\forall v \in V_2 : (l_{v_2}(v) \in U \cup L) \Rightarrow (l_{v_2}(v) = l_{v_1}(g_v(v)))$
 $\forall e \in E_2 : (l_{e_2}(e) \in U) \Rightarrow (l_{e_2}(e) = l_{e_1}(g_e(e)))$. □

Since this document is about querying RDF data, only the second RDF Subgraph problem is elaborated on, i.e., the problem with a semantic (RDFS closure) graph and a semantic query graph as input. An algorithm for the RDF Subgraph problem is given in Chapter 5.

4.2.1 Special Cases

Some special cases of the RDF Subgraph Problem can be distinguished. When the set of variables K is an empty set or when K only contains arc variables, Q contains no vertex variables and so no blank node labeled vertices. According to Lemma 7, the RDF Subgraph problem can then be solved in polynomial time.

4.3 Conclusion

The RDF semantic graph and the RDF semantic query graph have been defined. The RDF Subgraph problem amounts to finding all possible values in graph G for variables of graph Q . The RDF Subgraph problem is proven to be polynomially solvable when Q contains no vertex variables. Otherwise, the RDF Subgraph problem is NP-complete.

Chapter 5

RDF Subgraph Algorithm

The RDF Subgraph problem amounts to finding a query subgraph in a semantic graph. This problem is closely related to the Subgraph Isomorphism problem [21]. Therefore, several existing Subgraph Isomorphism algorithms are presented in Section 5.1. The algorithm chosen as the basis for the RDF Subgraph Algorithm is elaborated on in Section 5.2. In Section 5.3, the RDF Subgraph Algorithm is formulated. Some possible improvements for this algorithm are given in Section 5.4.

5.1 Subgraph Isomorphism Algorithms

Subgraph Isomorphism (SI) is defined on two graphs, and checks whether one graph has a subgraph isomorphic to the other graph. This problem is NP complete and a definition of SI is given in Chapter 4. Because SI has many applications, it has been studied thoroughly over the last three decades.

Several SI algorithms use specific graph and problem representations. In [15], directed graphs are represented by linear formulas using a prefix expression. Those graphs can therefore easily be compared with trees. This tree-search backtrack algorithm constructs successively larger subformulas which match in the directed graphs. In [19], directed graphs are represented as Boolean functions and the problem is reduced to a relational formulation. The algorithm uses Binary Decision Diagrams (BDDs) to compute and represent all isomorphisms of a small graph into a large graph. Some SI algorithms are used in specific research fields. For example in [20], a SI algorithm is given for matching chemical graphs, based on finding the maximal common substructure between two graphs by identifying a maximal clique in a compatibility graph. A widely known algorithm for SI is given by J.Ullmann in [30] and is, because of its generality and effectiveness, commonly used as a basis for new algorithms. Such an algorithm is presented in [27] and uses a compact representation of model graphs preprocessed in an off-line step. An interesting algorithm dealing only with the graph isomorphism problem is Nauty [26] which is claimed to be the fastest graph matching algorithm. A more detailed description of the last three algorithms is given next.

The algorithm presented in [30] is a depth-first tree-search algorithm. Given two graphs, this algorithm creates a vertex mapping tree where each node of the tree represents a possible mapping from a vertex of one graph to a vertex of the other graph. A set of nodes of this tree

represents a partial vertex mapping. The algorithm checks the subgraph isomorphism conditions for such a partial mapping, and if it represents an infeasible mapping, the algorithm backtracks and continues on the next subtree. The algorithm also applies a forward-checking procedure to a partial mapping, which tests for each vertex mapping ($v_i \rightarrow v_j$) of the partial mapping whether the neighbors of v_i can map on the neighbors of v_j . If this is not possible, the algorithm backtracks as well. This forward-checking procedure, also called the refinement procedure, attributes to a considerably reduction of the number of partial mappings generated during the search.

In [27], an algorithm for error-tolerant subgraph isomorphism is given which allows that some errors are made during the mapping process. This algorithm works on a set of small model graphs and an unknown large input graph, all directed and labeled. It is based on a compact representation of the model graphs, created in an off-line preprocessing step. In this preprocessing step, the model graphs are decomposed into smaller subgraphs and represented in terms of these subgraphs. So, subgraphs occurring multiple times in the model graphs are represented only once. The algorithm first maps the smallest subgraphs to the input graph, and then combines these subgraphs to larger subgraphs, until an original model graph is formed and so a subgraph of the input graph is found isomorphic to one of the model graphs. During this mapping, small errors in the input graph, for example missing edges, have to be overcome to make the mapping work. Each error-correction has a certain cost and the optimal error correcting subgraph isomorphism is defined by the one with the least costs. To reduce the solutions with high costs in an early stage, only subgraphs with minimum costs are combined. One of the difficulties of this algorithm is to determine the error-correction costs.

Nauty [26] is a powerful graph isomorphism algorithm and is based on an ordered partitioning of the vertices. A leaf partition is a partition with only singleton sets and so defines a relabeling of the graph. Nauty performs two major operations, namely refining a partition and generating the children of a partition. The refining part uses vertex invariants to compute, step by step, labels for all vertices who eventually result in one canonical label for the whole graph. This graph label amounts to the concatenation of the adjacency matrix of the ‘smallest’ automorphism. Creating a child of a partition is done by splitting a set into two sets: one containing a single vertex, the other containing the rest of the set. Consequently, Nauty amounts to a depth first search of the space of partitions, with the added optimization that each partition is refined before expanding its children. This algorithm results in the same canonical label for isomorphic graphs.

5.1.1 Conclusion

There is no SI algorithm available for semantic graphs, i.e., for directed, labeled pseudographs. The algorithm described in [27] tolerates small syntactic differences between the given graphs. This is comparable to the entailment definition given in Section 2.5 stating that graphs must have the same semantics, not the same structure. However, this algorithm requires an expensive preprocessing step and exponential memory space. Nauty [26] is a powerful, fast algorithm but is, unfortunately, only developed for graph isomorphism. The algorithm given in [30], from now on called Ullmann’s Algorithm, is an exact, well-studied algorithm which is still used as a basis for new algorithms. Therefore, the RDF Subgraph Algorithm is based

on Ullmann's Algorithm. Some concepts from other algorithms are considered in Section 5.4 as possible improvements of the RDF Subgraph Algorithm.

5.2 Ullmann's Algorithm

As already mentioned, Ullmann's Algorithm [30] is a depth-first tree-search backtrack algorithm. To explain this algorithm in more detail, a small graph $G_\alpha = (V_\alpha, E_\alpha)$ and a large graph $G_\beta = (V_\beta, E_\beta)$ are defined with their corresponding adjacency matrices $A = [a_{ij}]$ and $B = [b_{ij}]$.

First, a matrix M^0 of size $|V_\alpha| \times |V_\beta|$, called the initial mapping matrix, is constructed by the following conditions.

$$\begin{aligned} m_{ij}^0 &= 1 && \text{if degree of vertex } v_i \text{ of } G_\alpha \text{ is smaller than or equal to degree of vertex } v_j \text{ of } G_\beta \\ m_{ij}^0 &= 0 && \text{otherwise} \end{aligned}$$

So, matrix M^0 represents all possible vertex mappings from vertices of G_α to vertices of G_β . Infeasible mappings are filtered out based on the degree of the vertices. Starting from this matrix, all possible matrices M' are generated where every row of M' contains exactly one 1 and every column contains at most one 1. A completed matrix M' then contains a vertex mapping representing a subgraph isomorphism. During the construction of a M' , a refinement procedure is applied, which checks whether a subgraph isomorphism is still possible in the remainder of M' . If this is not the case, the algorithm backtracks immediately to construct a new M' .

The refinement procedure is a forward-check and determines for each vertex of G_α whether it can be mapped onto at least one vertex of G_β such that the subgraph isomorphism conditions are locally true. Refining a matrix M amounts to checking the following condition for every $m_{ij} = 1$ in M .

$$\forall_{1 \leq x \leq p_\alpha} (a_{ix} = 1) \Rightarrow \exists_{1 \leq y \leq p_\beta} (m_{xy} \cdot b_{jy} = 1)$$

$m_{ij} = 1$ is left unchanged if this condition is true and is set to 0 otherwise. This change may affect the condition for other 1's in M , and so, the refinement procedure applies this condition to each $m_{ij} = 1$ in M over and over again until there is an iteration in which none of the 1's is changed. This procedure terminates in a finite number of steps because the number of 1's in M is finite and a 0 is never changed into a 1.

When the construction of M' is completed and M' remains unchanged by the refinement procedure, the refinement condition holds for every $m_{ij} = 1$ in M' . This means that M' specifies a bijective mapping from V_α to V_β such that if there is an edge between two vertices in G_α , there is also an edge between the corresponding vertices in G_β . So M' represents a subgraph isomorphism from G_α to G_β .

In order to carry out new experiments, this algorithm is implemented. The recursive procedure given in Figure 5.1 describes the implemented algorithm. This implementation is based on an implementation of Ullmann's Algorithm given in [18]. Note that the refinement procedure is included in the step that verifies whether vertex mapping $v_i \rightarrow v_j$ is a feasible mapping.

```

input: graphs  $G_\alpha = (V_\alpha, E_\alpha)$  and  $G_\beta = (V_\beta, E_\beta)$ 
output: all subgraph isomorphisms from  $G_\alpha$  to  $G_\beta$ 

 $M^0 = \text{MakeInitialMapping}(G_\alpha, G_\beta)$ ;
refine  $M^0$ ;
Match ( $v_0, M^0$ );

procedure Match( $v_i, M$ )
  input: row number  $v_i$  and mapping matrix  $M$ 
  output: the mappings between the two graphs

  if all vertices of  $G_\alpha$  are mapped
  then output matrix  $M$ 
  else compute all mapping candidates  $v_j$  of  $G_\beta$  for vertex  $v_i$  of  $G_\alpha$ 
    for each mapping candidate  $v_j$ 
      if  $v_i \rightarrow v_j$  is a feasible mapping
        then new matrix  $newM$  is matrix  $M$  with mapping  $v_i \rightarrow v_j$  applied;
          Match( $v_{i+1}, newM$ )
        end if
      end for each
    end if
  end procedure

```

Figure 5.1: Pseudo-code for Ullmann's Algorithm

Results of experiments carried out with this implementation of Ullmann's Algorithm are given in Appendix B. It can be concluded that these results are comparable to results of other implementations of Ullmann's Algorithm [19].

5.3 RDF Subgraph Algorithm

Ullmann's Algorithm is used as the basis for the RDF Subgraph Algorithm. In this section, the adaptations to Ullmann's Algorithm are explained to conclude with the actual RDF Subgraph Algorithm.

The input for the RDF Subgraph Algorithm consists of a semantic query graph $Q = (V_2, E_2, f_2, l_{v_2}, l_{e_2})$ and a (partial RDFS closure) semantic graph $G = (V_1, E_1, f_1, l_{v_1}, l_{e_1})$, as defined in Chapter 4.

First, the initial mapping matrix M^0 is constructed. As a semantic graph contains vertex and arc labels, this information can be used for the construction of M^0 , instead of only the degrees of the vertices. For every vertex v_i of graph Q and vertex v_j of graph G , this construction starts with comparing their labels, which results in a large reduction of infeasible mappings, as every non-variable vertex label occurs at most once in a graph; see Lemma 1 in Section 4.1.

Furthermore, the arc information of these vertices can be used, so v_i can be mapped on v_j if the arcs connected to v_i can correspond to arcs connected to v_j . This means that the set of incoming arc labels of v_i , denoted by $\bigcup_{in} v_i$, must be a subset of the set of incoming arc labels of v_j , idem for the outgoing arc labels, denoted by $\bigcup_{out} v_i$. Note that the set of arc labels of v_j is combined with K , i.e., the set of variable labels of Q , because graph Q can contain variables. Consequently, the initial mapping matrix M^0 is constructed as follows.

$$m_{ij}^0 = \begin{cases} 1 & \text{if } l_{v_2}(v_i) \in K \quad \text{and } (\bigcup_{in} v_i \subseteq \bigcup_{in} v_j \cup K) \wedge (\bigcup_{out} v_i \subseteq \bigcup_{out} v_j \cup K) \\ 1 & \text{if } l_{v_2}(v_i) = l_{v_1}(v_j) \text{ and } (\bigcup_{in} v_i \subseteq \bigcup_{in} v_j \cup K) \wedge (\bigcup_{out} v_i \subseteq \bigcup_{out} v_j \cup K) \\ 0 & \text{otherwise.} \end{cases}$$

Like Ullmann's Algorithm, M^0 contains all possible mappings from vertices of Q to vertices of G .

The refinement procedure checks for every $m_{ij} = 1$ in M whether the neighbors of vertex v_i can still be mapped on neighbors of vertex v_j . For a $m_{ij} = 1$, this amounts to checking the following.

- For every non-variable outgoing arc of v_i there exists an outgoing arc of v_j with the same label. Furthermore, both arcs must go to vertices which can be mapped on each other according to M .
- For every variable outgoing arc of v_i there exists an outgoing arc of v_j . Furthermore, both arcs must go to vertices which can be mapped on each other according to M .
- For every non-variable incoming arc of v_i there exists an incoming arc of v_j with the same label. Furthermore, both arcs must come from vertices which can be mapped on each other according to M .
- For every variable incoming arc of v_i there exists an incoming arc of v_j . Furthermore, both arcs must come from vertices which can be mapped on each other according to M .

These refinement conditions are formulated as follows.

$$\forall_{v_x \in V_2} (f_2(e_p) = (v_i, v_x) \Rightarrow \exists_{v_y \in V'} (m_{xy} = 1 \text{ and } f_1(e_q) = (v_j, v_y)) \text{ and } l_{e_2}(e_p) \subseteq l_{e_1}(e_q) \cup K)$$

and

$$\forall_{v_x \in V_2} (f_2(e_p) = (v_x, v_i) \Rightarrow \exists_{v_y \in V'} (m_{xy} = 1 \text{ and } f_1(e_q) = (v_y, v_j)) \text{ and } l_{e_2}(e_p) \subseteq l_{e_1}(e_q) \cup K)$$

A recursive description of the RDF Subgraph algorithm is given in Figure 5.2. The refinement procedure is included in the step that verifies whether $v_i \rightarrow v_j$ is a feasible mapping. Note that $v_i \rightarrow v_j$ is a feasible mapping when matrix M with this mapping applied satisfies the refinement conditions.

When the construction of M' is completed and M' remains unchanged by the refinement procedure, the refinement conditions hold for every $m_{ij} = 1$ in M' . This means that M' specifies a vertex mapping from Q to G such that if there is an arc between two vertices in Q , there is also a corresponding arc between the corresponding vertices in G . Note that this mapping is no longer a bijection but a surjection because more than one vertex of Q can be

input: graphs $Q = (V_2, E_2, f_2, l_{v_2}, l_{e_2})$ and $G = (V_1, E_1, f_1, l_{v_1}, l_{e_1})$
output: all RDF Subgraph matches from Q to G

$M^0 = \text{MakeInitialMapping}(Q, G);$

refine $M^0;$

$\text{RDFMatch}(v_0, M^0);$

procedure $\text{RDFMatch}(v_i, M)$

input: row number v_i and mapping matrix M

output: the mappings between the two graphs

if all vertices of Q are mapped

then output matrix M

else compute all mapping candidates v_j of G for vertex v_i of Q

for each mapping candidate v_j

if $v_i \rightarrow v_j$ is a feasible mapping

then new matrix $newM$ is matrix M with mapping $v_i \rightarrow v_j$ applied;

$\text{Match}(v_{i+1}, newM)$

end if

end for each

end if

end procedure

Figure 5.2: Pseudo-code for RDF Subgraph Algorithm

mapped on the same vertex of G . This also holds for the arc mapping. A mapping defined in M no longer represents a subgraph isomorphism, but a so-called RDF Subgraph match, according to the definitions given in Section 4.2.

5.3.1 Implementation

The implementation of an RDF semantic graph is explained next. The implementation of the RDF Subgraph algorithm is presented in Appendix C. Results of experiments carried out using this implementation are given in Chapter 6.

As pseudographs are commonly represented by adjacency matrices, the arc labels of a semantic graph are stored in an adjacency matrix whereas the vertex labels are contained in a separate array. So, an RDF semantic graph with n vertices and m edges is implemented by

- an array of size n containing the (unique) vertex labels and
- an adjacency matrix of size $n \times n$ containing the sets of arc labels between vertices.

An example of an implemented RDF semantic graph is given in Figure 5.3. As shown, vertex labels A and B are stored in an array of length 2. The arc labels C, D and E are stored in a matrix of size 2×2 .



Figure 5.3: Example of Implemented RDF Semantic Graph

5.4 Possible Improvements

Some concepts of the SI algorithms presented in Section 5.1 can be used to improve the RDF Subgraph Algorithm. The following possible improvements are elaborated on next:

- handle labels in a preprocessing step,
- compute vertex fingerprints in a preprocessing step, and
- decompose the graph in a preprocessing step.

Note that in practise, graph G is a stable, large graph for which the preprocessing can be done offline. Preprocessing query Q usually has to be done in run-time.

5.4.1 Label Preprocessing

As mentioned, an RDF semantic graph is implemented as an array of vertex labels and an adjacency matrix of arc labels. These arc labels are long strings and each label can occur in the matrix a number of times; see Lemma 3 in Chapter 4. Therefore, it is more efficient to store the arc labels in a separate table and create unique (short) identifiers for these labels. These identifiers are then stored in the adjacency matrix which saves memory. Moreover, it also saves computation time when the same index table is used for both graphs, since only the unique identifiers have to be compared instead of the complete labels.

A second label preprocessing is placing the sets of arc labels in alphabetical order. This can either be done in the adjacency matrices themselves, or in possible vertex fingerprints; see Section 5.4.2. Sorting a set of length n costs $O(n \log n)$ time, which is done in a preprocessing step. Finding an element in this sorted set can be done by a binary search ($O(\log n)$) instead of a linear search ($O(n)$).

5.4.2 Vertex Fingerprint Preprocessing

In Appendix B, a vertex fingerprint is presented for general graphs to improve the initial mapping matrix of Ullmann's Algorithm. This fingerprint is based on numbers of vertices and arcs and shows that such fingerprints do not lead to better results for a given small, sparse graph and a given large, dense graph. As fingerprints of the small graph are always smaller than fingerprints of the large graph, comparing the fingerprints of v_i and v_j always

results in a possible mapping from v_i to v_j . This does not lead to an improvement of the initial mapping matrix. In order to improve this matrix for the RDF Subgraph algorithm, vertex fingerprints have to contain extra information, e.g. vertex or arc labels.

For convenience, the construction of initial mapping matrix M^0 for the RDF Subgraph algorithm is given again.

$$m_{ij}^0 = \begin{cases} 1 & \text{if } l_{v_2}(v_i) \in K \quad \text{and } (\bigcup_{in} v_i \subseteq \bigcup_{in} v_j \cup K) \wedge (\bigcup_{out} v_i \subseteq \bigcup_{out} v_j \cup K) \\ 1 & \text{if } l_{v_2}(v_i) = l_{v_1}(v_j) \text{ and } (\bigcup_{in} v_i \subseteq \bigcup_{in} v_j \cup K) \wedge (\bigcup_{out} v_i \subseteq \bigcup_{out} v_j \cup K) \\ 0 & \text{otherwise.} \end{cases}$$

If v_i has no variable label, this construction starts with verifying whether the vertex labels of v_i and v_j are equal, which already results in a large reduction of infeasible mappings. Second, it is verified whether the arcs connected to v_i can correspond to arcs connected to v_j .

Arc Label Fingerprint

As the arcs are stored in the adjacency matrices, it costs relatively much time to check whether arcs of Q can correspond to arcs of G . Therefore, the incoming and outgoing arc labels of a vertex are combined in an arc label fingerprint, which can be computed in a preprocessing step. This makes searching in the adjacency matrices superfluous. Furthermore, sorting the labels in the fingerprint alphabetically reduces the search in fingerprints from linear search to binary search. Note that the storage space of these fingerprints is minimized when arc label tables and unique identifiers are used, as is explained in Section 5.4.1. An arc label fingerprint C consists of a set of incoming and a set of outgoing arc labels, which is formulated as follows.

$$C = \{\{label, \dots, label\}, \{label, \dots, label\}\}$$

Consequently, the construction of initial mapping matrix M^0 is as follows.

$$m_{ij}^0 = \begin{cases} 1 & \text{if } l_{v_2}(v_i) \in K \quad \text{and } C_i \subseteq C_j \cup K \\ 1 & \text{if } l_{v_2}(v_i) = l_{v_1}(v_j) \text{ and } C_i \subseteq C_j \cup K \\ 0 & \text{otherwise.} \end{cases}$$

Note that $C_i \subseteq C_j$ amounts to checking whether the incoming label set of C_i is a subset of the incoming label set of C_j , and whether the outgoing label set of C_i is a subset of the outgoing label set of C_j . Experiments using the arc label fingerprint are carried out in Chapter 6.

Neighbor Fingerprint

Neighbor fingerprint N is used as an extra condition in the construction of initial mapping matrix M^0 . N denotes the set of neighbor names of a vertex, i.e., the labels of adjacent vertices, and is computed in a preprocessing step. If neighbor fingerprint N_i of vertex v_i is a subset of neighbor fingerprint N_j of vertex v_j , then $v_i \rightarrow v_j$ is a possible mapping. This fingerprint requires multiple storage of vertex labels, so creating vertex label tables and unique identifiers for vertices is useful, as is explained in Section 5.4.1. Neighbor fingerprint N and the construction of M^0 are formulated as follows.

$$N = \{label, \dots, label\}$$

$$m_{ij}^0 = \begin{cases} 1 & \text{if } l_{v_2}(v_i) \in K \quad \text{and} \quad N_i \subseteq N_j \cup K \text{ and} \\ & (\bigcup_{in} v_i \subseteq \bigcup_{in} v_j \cup K) \wedge (\bigcup_{out} v_i \subseteq \bigcup_{out} v_j \cup K) \\ 1 & \text{if } l_{v_2}(v_i) = l_{v_1}(v_j) \text{ and} \quad N_i \subseteq N_j \cup K \text{ and} \\ & (\bigcup_{in} v_i \subseteq \bigcup_{in} v_j \cup K) \wedge (\bigcup_{out} v_i \subseteq \bigcup_{out} v_j \cup K) \\ 0 & \text{otherwise.} \end{cases}$$

Experiments using the neighbor fingerprint are carried out in Chapter 6.

5.4.3 Decomposition Preprocessing

The structure of a graph depends on the semantics of this graph. For example, a graph containing one class and lots of instances of that class has a star-shaped structure. A graph denoting a set of highly related resources has a dense structure. Two decompositions are presented next.

A possible graph decomposition is to consider schema data as a separate component. However, after applying the closure computation, it is not clear which part of the closure graph is solely schema information. Another possible decomposition of a graph is to consider each class of the graph and its instances as a separate component. This can also be done for domains and ranges. Note that there is no requirement that a resource has to be an instance of at least one class. Furthermore, resources can belong to several classes, which is usually the case after the closure computation.

As decompositions depend on the semantics of a graph, it is hard to come up with a general decomposition for semantic graphs. Note that this is an interesting field of further research.

5.5 Conclusion

Because the RDF Subgraph problem has similarities to Subgraph Isomorphism, some Subgraph Isomorphism algorithms are studied. Ullmann's Algorithm is chosen as the basis for the RDF Subgraph Algorithm as it is an exact and well-studied algorithm. The RDF Subgraph Algorithm is formulated and implemented. Some improvements are presented.

Chapter 6

Experiments

Until recently, no query results of RDF systems have been published. The main reason for this is that few large RDF data sets are available. Moreover, the performance of any given system varies depending on the structure of the schemas and data used to evaluate it. In [22], four RDF systems are compared, using exactly the same data sets and queries. This is discussed in Section 6.1. In Section 6.2, the RDF Subgraph Algorithm is tested on a small real-life RDF application, called the Museum Graph. Finally, the RDF Subgraph Algorithm is tested on large semi-random semantic graphs in Section 6.3.

6.1 Other RDF Systems

[22] is a recent article that presents an evaluation of RDF systems. It must be noted, however, that this evaluation only uses one RDF data set, called the Lehigh University Benchmark, and conducts experiments with 14 well-chosen queries. In [22], it is pointed out that the performance of any given system varies, depending on the structure of the schemas and data used to evaluate it. Furthermore, results also depend on the computer and storage environment. For example, different database management systems, such as MySQL and PostgreSQL, result in different computation times. Moreover, different query languages can also result in different answers, and so different computation times; see Section 3.2. Consequently, the results of querying the Lehigh University Benchmark can not be seen as a conclusive judgement about the RDF systems.

Some results of querying the Lehigh University Benchmark are given in Table 6.1. The first system is Sesame using a database persistent storage; see Section 3.3. The second system is DLDB [28], a repository for processing and querying large amounts of RDF data. DLDB also uses database storage. Note that Jena was also preliminarily tested in [22], but turned out to be much slower than Sesame in answering nearly all the queries. Therefore no results for Jena were included in [22]. The four queries in Table 6.1 are expressed in number of statements (q), number of vertex variables (v_v) and number of arc variables (v_a). The Lehigh University Benchmark is expressed in the number of statements (m). Query 1 and 4 result in the same number of matches but have different computation times. For Query 3, Sesame has a much longer computation time and returns only half the number of matches found compared to the DLDB system. Query 4 shows that adding a variable arc increases the number of matches. In [22], these results are explained in more detail.

Queries		Lehigh University Benchmark	Sesame DB		DLDB	
number	q v_v v_a	m	answers	time (sec)	answers	time (sec)
1	2 1 0	103,074	4	0.046	4	0.059
		645,649	4	0.043	4	0.226
		1,316,322	4	0.040	4	0.412
2	6 3 0	103,074	0	51.878	0	0.181
		645,649	9	368.423	9	2.320
		1,316,322	28	711.678	28	14.556
3	6 3 0	103,074	103	34.034	208	0.634
		645,649	600	256.770	1245	7.751
		1,316,322	1233	460.267	2540	19.971
4	1 1 1	103,074	5916	0.218	5916	0.187
		645,649	36,682	1.398	36,682	2.937
		1,316,322	75,547	14.021	75,547	7.870

Table 6.1: Query Test Results of Sesame and DLDB

As [22] is only recently published and for lack of time, the RDF Subgraph Algorithm could not be tested on the Lehigh University Benchmark.

6.2 Real-life Application

In this section, the RDF Subgraph Algorithm is tested on a small real-life RDF application, called the Museum Graph. The Museum Graph is part of the Museum Repository [10] given by Sesame [1] as a test demo. Therefore, the queries carried out on the Museum Graph by the RDF Subgraph Algorithm can also be carried out on the Museum Repository by Sesame. Note that the Museum Repository is twice as large and contains more data about painters and paintings. Although the RDF Subgraph Algorithm and Sesame are tested with the same queries and (almost) the same data sets, these query results should not be pairwise compared, as explained in Section 6.1.

6.2.1 Museum Graph

The Museum Graph is an RDF semantic graph elaborated on in Appendix D. It is about painter Rembrandt and two of his paintings using two schemas. The closure graph of the Museum Graph consists of 78 vertices and 242 arcs and is denoted as graph G . Next, two queries are carried out on G using the RDF Subgraph Algorithm presented in Section 5.3.

Query $Q1$ is formulated as follows. *Give all paintings of Rembrandt.* This leads to the following RDF semantic query graph $Q1$.

vertex labels = [http://www.european-history.com/rembrandt.html, k0,
http://www.icom.com/schema.rdf#Painting]

adjacency matrix =

{}	{http://www.icom.com/schema.rdf#creates}	{}
{}	{}	{http://www.w3.org/1999/02/22-rdf-syntax-ns#type}
{}	{}	{}

Given graphs $Q1$ and G , the RDF Subgraph Algorithm returns the following answer in 0.047 sec.

variable vertex k0 maps on vertex http://www.artchive.com/rembrandt/abraham.jpg
variable vertex k0 maps on vertex http://www.artchive.com/rembrandt/artist-at-easel.jpg

A second query $Q2$ is formulated as follows. *Give all resources and properties connected to http://www.artchive.com/rembrandt/artist-at-easel.jpg.* The corresponding query graph $Q2$ is given next.

vertex labels = [k3, http://www.artchive.com/rembrandt/artist-at-easel.jpg, k1]

adjacency matrix =

{}	{k2}	{}
{}	{}	{k0}
{}	{}	{}

Given graphs $Q2$ and G , the RDF Subgraph Algorithm returns the following answer in 0.062 sec.

variable vertex k3 maps on vertex http://european-history.com/rembrandt.html
variable vertex k1 maps on vertex http://www.icom.com/schema.rdf#Painting
variable arc k2 has value(s) http://www.icom.com/schema.rdf#creates,
http://www.icom.com/schema.rdf#paints
variable arc k0 has value(s) http://www.w3.org/1999/02/22-rdf-syntax-ns#type

variable vertex k3 maps on vertex http://european-history.com/rembrandt.html
variable vertex k1 maps on vertex http://www.w3c.org/2000/01/rdf-schema#Resource
variable arc k2 has value(s) http://www.icom.com/schema.rdf#creates,
http://www.icom.com/schema.rdf#paints
variable arc k0 has value(s) http://www.w3.org/1999/02/22-rdf-syntax-ns#type

variable vertex k3 maps on vertex http://european-history.com/rembrandt.html
variable vertex k1 maps on vertex http://www.icom.com/schema.rdf#Artifact
variable arc k2 has value(s) http://www.icom.com/schema.rdf#creates,
http://www.icom.com/schema.rdf#paints
variable arc k0 has value(s) http://www.w3.org/1999/02/22-rdf-syntax-ns#type

variable vertex k3 maps on vertex http://european-history.com/rembrandt.html
variable vertex k1 maps on vertex "oil on canvas"@en
variable arc k2 has value(s) http://www.icom.com/schema.rdf#creates,
http://www.icom.com/schema.rdf#paints
variable arc k0 has value(s) http://www.icom.com/schema.rdf#technique

6.2.2 Museum Repository

The Museum Repository [10] is an RDF triple graph given by Sesame and consists of 406 statements. Queries *Q1* and *Q2* of Section 6.2.1 are also carried out on the Museum Repository using the Sesame system, as is shown next.

Query *Q1*, i.e., *give all paintings of Rembrandt*, is written as the following RQL query.

```
select Y
from X icom:creates Y : icom:Painting
where X like "http://www.european-history.com/rembrandt.html"
using namespace icom = http://www.icom.com/schema.rdf#
```

Sesame returns the following answer in 0.094 sec.

```
Y
http://www.artchive.com/rembrandt/abraham.jpg
http://www.artchive.com/rembrandt/artist-at-easel.jpg
```

Query *Q2*, i.e., *give all resources and properties connected to http://www.artchive.com/rembrandt/artist-at-easel.jpg*, is written in RQL as follows.

```
select @W, X, @Y, Z
from U @W X, Z @Y U
where U like "http://www.artchive.com/rembrandt/artist-at-easel.jpg"
```

Sesame returns the following answer in 0.702 sec.

```
@W                                X
http://www.w3.org/...rdf-syntax-ns#type  http://www.icom.com/schema.rdf#Painting
http://www.icom.com/schema.rdf#exhibited  http://www.louvre.fr/
http://www.icom.com/schema.rdf#technique  "oil on canvas"@en
@Y                                Z
http://www.icom.com/schema.rdf#paints     http://www.european-history.com/rembrandt.html
http://www.icom.com/schema.rdf#paints     http://www.european-history.com/rembrandt.html
http://www.icom.com/schema.rdf#paints     http://www.european-history.com/rembrandt.html
```

6.2.3 Conclusion

The computation times of the queries carried out in the previous sections are encapsulated in Table 6.2. Note that the computation time of Sesame includes time to handle the query and retrieve data. Consequently, it can be concluded that the RDF Subgraph Algorithm and Sesame have comparable total times.

	RDF Subgraph Algorithm (sec)	Sesame (sec)
Q1	0.047	0.094
Q2	0.062	0.702

Table 6.2: Computation Times for RDF Subgraph Algorithm and Sesame for Real-life Application

Both the RDF Subgraph Algorithm and Sesame return the same answers for query $Q1$. Query $Q2$, however, leads to different results. The RDF Subgraph algorithm returns all possible combinations for the variables, including the schema information such as ‘creates’ and ‘Resource’. Besides some more information of this painting which is included in the Museum Repository, Sesame, on the other hand, does not return this schema information. These differences are briefly mentioned in Chapter 3.

6.3 Semi-random RDF Semantic Graphs

As mentioned, few large RDF data sets are currently available. Moreover, these data sets are stored in databases as RDF triples, or in web documents using the XML notation. To be able to use these data sets for the RDF Subgraph Algorithm, the data sets must be converted into RDF semantic graphs, using the conversion procedure presented in Section 4.1. An alternative is to generate large random RDF semantic graphs, and carry out random queries on these graphs. This is elaborated on in this section. Note that the large graphs are constructed by extending the query graph such that at least one answer can be found, and are therefore called semi-random graphs.

As discussed in [22], the performance of any given system varies, depending on the structure of the data used to evaluate it. Consequently, the structure of the large semi-random graphs is of big importance. A graph representing a set of highly related resources consists of relatively few vertices and lots of arcs. Preliminary tests show that query answers are found quickly. This is because much (arc) information is available and only few vertices must be observed. Another possible graph structure represents a more sparse graph and so the number of vertices is more in proportion to the number of arcs. This graph structure is elaborated on next to generate semi-random semantic graphs.

The RDF Subgraph Algorithm needs two graphs as input, namely a small semantic query graph Q and a large semantic graph G . It is assumed that G is a partial RDFS closure graph, however, this is not considered in the construction of G . As mentioned, graph G is constructed by extending graph Q in order to obtain at least one query answer. For the construction of Q , the following parameters are defined.

- p is the number of vertices.
- a is the number of outgoing arcs to generate for each vertex of Q .
- v_v is the number of vertex variables.
- v_a is the number of arc variables.

Note that the total number of arcs in Q is $q = p \times a$. The vertices and arcs of graph Q are used as the basis for the construction of graph G . Furthermore, the following parameters for G are given.

- n is the number of vertices.
- b is the number of outgoing arcs for each vertex of G .

The total number of arcs in G is $m = n \times a$. The labels for these graphs are generated randomly, using prefixes ‘u’ (URI references), ‘b’ (blank nodes), ‘l’ (literals) and ‘k’ (variables) followed by a random number.

An example of a randomly generated query graph with $p = 3$, $a = 1$, $v_v = 1$ and $v_a = 1$ is given in Figure 6.1. Note that literal labeled vertices are not allowed to have outgoing arcs, and so a represents the number of incoming arcs to generate for these vertices.

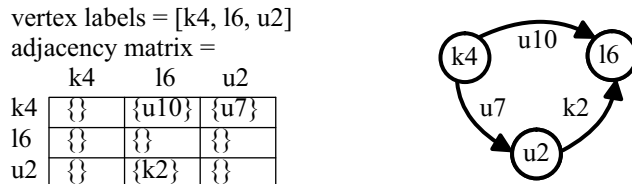


Figure 6.1: Example of Random Query Graph

These semi-random semantic graphs are used to conduct experiments on the RDF Subgraph Algorithm. The query parameters for these experiments are derived from the query benchmark [22] for the Lehigh University Benchmark data set; see Section 6.1. Experimental results are given in Table 6.3. The time to generate random graphs is not included in the computation times. Each row gives average results obtained with 25 randomly generated graphs. Note that s.d. denotes the standard deviation for these results.

For the first experiments, v_v and v_a are both chosen 0, so Q contains no variables. For this case, the RDF Subgraph problem is polynomially solvable, as proven in Section 4.2. Exactly one match is found, which agrees with the constructions of G and Q and with the fact that Q does not contain any variables. In the second set of experiments, Q contains only arc variables. This is also a special case of the RDF Subgraph problem which can be solved in polynomial time as well. When Q contains vertex variables, i.e., $v_v > 0$, the RDF Subgraph problem is NP-complete. As can be seen in the last experiments, the computation time increases with the number of vertex variables v_v and the number of statements m . More experiment results are given in Appendix C.

Additionally, the arc label fingerprint and the neighbor fingerprint presented in Section 5.4 are implemented and tested. Experimental results are given in Table 6.3. For the arc label fingerprint, the computation times are reduced by 25% to 75%. This is due to the sorted arrays and binary searches. It can be concluded that this fingerprint is an improvement of the RDF Subgraph Algorithm. The neighbor fingerprint usually results in computation times that are between 3% and 10% faster. However, this fingerprint sometimes results in computation times that are upto 6% longer when the query graph is sparse and contains several vertex variables. This can be explained as follows. A fingerprint N_i of the query consists of few labels which are possibly variable. The fingerprints N_j of the large graph contain several labels. Comparing these fingerprints generally results in $N_i \subseteq N_j \cup K$ being true. This does not lead to a reduction of infeasible mappings, but does lead to longer computation times. It can be concluded that this fingerprint is a slight improvement of the RDF Subgraph Algorithm.

Note that in practise, the fingerprint preprocessing can only be done for graph G , as this is considered to be a stable, known graph. The fingerprints of queries Q have to be computed in run-time before the RDF Subgraph Algorithm.

Semi-random graphs contain labels of the form ‘u53’ which can be considered as unique identifiers. So the label preprocessing step presented in Section 5.4.1, i.e., storing the labels in a separate table and create unique identifiers for them, is not implemented.

Table 6.3: Results of Experiments for RDF Subgraph Algorithm using Semi-random Graphs

query graph Q				graph G		RDF Subgraph Algorithm				Arc Label Fingerprint				Neighbor Fingerprint			
p	q	v_v	v_a	n	m	RDF matches		time (sec)		RDF matches		time (sec)		RDF matches		time (sec)	
						av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.
2	2	0	0	200	2000	1.0	0.0	0.0029	0.0092	1.0	0.0	0.0018	0.0049	1.0	0.0	0.0031	0.0076
				1000	50,000	1.0	0.0	0.015	0.012	1.0	0.0	0.0018	0.0066	1.0	0.0	0.010	0.009
				5000	250,000	1.0	0.0	0.211	0.466	1.0	0.0	0.0062	0.014	1.0	0.0	0.42	0.014
6	6	0	0	200	2000	1.0	0.0	0.0043	0.008	1.0	0.0	0.0031	0.0076	1.0	0.0	0.005	0.008
				1000	50,000	1.0	0.0	0.0425	0.011	1.0	0.0	0.0050	0.0086	1.0	0.0	0.023	0.01
				5000	250,000	1.0	0.0	1.44	5.76	1.0	0.0	0.022	0.024	1.0	0.0	1.21	0.24
6	6	0	3	200	2000	1.4	0.69	0.0044	0.010	1.16	0.37	0.0025	0.0058	1.28	0.53	0.0032	0.008
				1000	50,000	1.48	0.75	0.041	0.018	1.5	0.92	0.0079	0.019	1.5	0.92	0.027	0.01
				5000	250,000	1.0	0.0	0.189	0.080	1.0	0.0	0.016	0.017	1.0	0.0	0.119	0.016
3	6	0	6	200	2000	2.92	2.09	0.0025	0.0073	2.84	1.64	0.0019	0.0067	3.0	1.47	0.0075	0.021
				1000	50,000	2.24	2.93	0.023	0.013	3.4	2.06	0.0006	0.0031	3.8	2.01	0.012	0.009
				5000	250,000	3.0	1.41	0.163	0.397	2.5	1.57	0.0094	0.012	3.0	1.55	0.122	0.014
2	2	1	0	200	2000	1.2	0.4	0.045	0.020	1.04	0.2	0.019	0.019	1.08	0.27	0.040	0.012
				1000	50,000	1.0	0.0	3.442	0.176	1.2	0.4	1.23	1.50	1.04	0.2	3.19	0.12
				5000	250,000	1.0	0.0	86.67	7.40	1.0	0.0	32.28	39.35	1.0	0.0	83.9	6.25
2	2	2	0	200	2000	1.32	0.68	0.077	0.014	1.12	0.59	0.059	0.025	1.16	0.37	0.082	0.020
				1000	50,000	1.44	0.75	6.791	0.177	1.4	0.50	5.02	1.28	1.2	0.49	6.90	0.166
				5000	250,000	1.08	0.27	172.4	14.57	1.2	0.4	133.1	35.26	1.0	0.0	174.05	14.75
3	6	1	0	200	2000	1.0	0.0	0.041	0.018	1.0	0.0	0.028	0.028	1.0	0.0	0.041	0.015
				1000	50,000	1.0	0.0	3.429	0.15	1.0	0.0	1.52	1.47	1.0	0.0	3.09	0.021
				5000	250,000	1.0	0.0	87.49	7.01	1.0	0.0	47.98	39.04	1.0	0.0	85.14	7.58
3	6	2	1	200	2000	1.12	0.32	0.08	0.019	1.16	0.04	0.052	0.025	1.04	0.2	0.077	0.012
				1000	50,000	1.08	0.27	6.516	0.16	1.04	0.2	4.53	1.57	1.1	0.3	6.20	0.07
				5000	250,000	1.0	0.0	174.92	16.06	1.0	0.0	125.5	41.1	1.0	0.0	168.32	14.8
6	6	2	2	200	2000	41.6	60.4	0.106	0.038	39.48	70.12	0.041	0.062	36.52	55.30	0.101	0.052
				1000	50,000	584.08	904.12	7.657	2.478	785.8	994.9	1.06	1.22	1251.2	1454.7	7.827	1.81
				5000	250,000	1161.5	990.83	197.58	60.16	902.8	1516.0	6.44	7.73	718.3	1085.6	213.7	95.4

6.4 Conclusion

In [22] RDF systems are evaluated based on a large RDF data set benchmark. It can be concluded that the performance of any given system varies, depending on the structure of the schemas and data used to evaluate it. Few large RDF data sets are available and, therefore, the RDF Subgraph Algorithm is tested on a small real-life application and on large semi-randomly generated graphs. Some additional vertex fingerprints are implemented and tested. The arc label fingerprint leads to significant faster computation times.

Note that the test environment for these experiments is as follows: Windows XP Professional version 2002, Pentium 4 CPU 2.4 GHz, 512 MB RAM.

Chapter 7

Conclusion

Resource Description Framework is a standard representation language that describes how to define metadata for Web documents. The basic concept of RDF is an RDF graph consisting of *(subject, predicate, object)* triples. Most existing RDF query systems are based on this triple set representation. Representing an RDF data set as a directed, labeled graph is a recent field of research which enables the possibility of using existing graph definitions and algorithms. In this document, such a directed, labeled graph is called an RDF semantic graph.

Simple entailment is defined by the specifications of RDF and verifies whether RDF graphs have the same semantics. When one of these graphs is turned into a query graph, the simple entailment concept can be used to find query answers. This problem is called the RDF Subgraph problem, and is defined for semantic graphs. The RDF Subgraph problem is proven to be NP-complete.

As the RDF Subgraph problem is similar to Subgraph Isomorphism, some Subgraph Isomorphism algorithms are studied. Ullmann's Algorithm is chosen as the basis for the RDF Subgraph Algorithm because it is an exact and well-studied algorithm. The RDF Subgraph Algorithm is implemented and tested on a small real-life application and on semi-randomly generated graphs. The arc label fingerprint is presented as an improvement of the RDF Subgraph Algorithm. Finally, experimental results of other RDF systems are given.

7.1 Suggestions For Further Research

According to the specifications of RDF, a (partial) RDFS closure graph has to be computed before querying. A possible improvement is to keep schema and data information separate, and retrieve and reason only specific information for a given query. Some advantages and disadvantages for these two methods are given next.

A (partial) RDFS closure graph contains all data and schema information and is therefore easier to query. A closure graph is several times larger than the schema and data graph separately. It also takes some time to compute a closure graph, but for stable graphs this has to be done only once in an offline preprocessing step. Updating the closure graph for small changes can be done locally, without recalculating the whole closure graph.

By keeping the schema data separate, the graphs keep their original size which saves memory. It is also more easy to make changes in either the schema data or RDF data. Unfortunately, for this method, the querying becomes more complicated because specific schema information has to be retrieved and reasoned with during run-time.

7.2 References

- [1] About sesame. <http://www.openrdf.org/about.jsp>.
- [2] Extensible markup language. <http://www.w3.org/XML/>.
- [3] Jena: A semantic web framework for java. <http://jena.sourceforge.net/>.
- [4] Pseudograph definition. <http://www.mathworld/pseudograph>.
- [5] RDF semantics. <http://www.w3c.org/TR/rdf-mt/>.
- [6] RDQL: A query language for RDF. <http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/>.
- [7] Resource description framework. <http://www.w3.org/RDF/>.
- [8] RQL: The RDF query language. <http://139.91.183.30:9090/RDF/RQL/>.
- [9] Semantic web. <http://www.w3.org/2001/sw/>.
- [10] Sesame museum demonstration. <http://www.openrdf.org/sesame/actionFrameset.jsp?repository=museum>.
- [11] Wordnet library. <http://www.semanticweb.org/library>.
- [12] World wide web consortium. <http://www.w3c.org>.
- [13] Xquery: An XML query language. <http://www.w3c.org/TR/xquery/>.
- [14] R. Angles and C , Gutierrez. RDF query languages need support for graph properties. Technical report, Department of Computer Science, Universidad de Chile, Chile, 2004.
- [15] A.T. Berztiss. A backtrack procedure for isomorphism of directed graphs. *Journal of the Association for Computing Machinery*, 20(3):pp. 365–377, 1973.
- [16] J. Broekstra. A generic architecture for storing and querying RDF and RDF schema. *ISWC*, LNCS 2342:pp. 54–68, 2002.
- [17] J. Broekstra and A. Kampman. Inferencing and truth maintenance in RDF schema. <http://www.cs.vu.nl/~jbroeks/papers/inferencing.pdf>.
- [18] L.P. Cordella and P. Foggia. An improved algorithm for matching large graphs. Technical report, Università degli Studi di Napoli “Federico I”, Napoly, Italy, <http://amalfi.dis.unina.it/graph/db/papers/vf-algorithm.pdf>.
- [19] J. Cortadella and G. Valiente. A relational view of subgraph isomorphism. Technical report, Department of Software, Technical University of Catalonia, Barcelona, Spain, <http://www.lsi.upc.es/~valiente/abs-relmics-2000.pdf>.
- [20] P.J. Durand and J.W. Baker. An efficient algorithm for similarity analysis of molecules. *Internet Journal of Chemistry*, 2(17), 1999.

-
- [21] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W.H. Freeman and Company, New York, 1979.
- [22] Y. Guo, Z. Pan, and J. Heflin. An evaluation of knowledge base systems for large OWL datasets. *ISWC*, LNCS 3298:pp. 274–288, 2004.
- [23] P. Haase and J. Broekstra. A comparison of RDF query languages. *ISWC*, LNCS 3298:pp. 502–517, 2004.
- [24] H.J. ter Horst. Extending the RDFS entailment lemma. *ISWC*, LNCS 3298:pp. 77–91, 2004.
- [25] H.J. ter Horst. Semantic web ontologies and entailment: Some complexity results. Technical report, Philips Research, Eindhoven, The Netherlands, 2004.
- [26] M. McKay. Nauty user’s guide (version 1.5). TRCS 9002, Computer Science Department, Australian National University, Australia, 1990.
- [27] B.T. Messmer and H. Bunke. A new algorithm for error-tolerant subgraph isomorphism detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(5), 1998.
- [28] J. Pan, Z. Pan, and J. Heflin. DLDB: Extending relational databases to support semantic web queries. *ISWC*, 2003.
- [29] H.J. ter Horst and W. ten Kate. The semantic web: Applications, languages, semantics. Technical report, Philips Research, Eindhoven, The Netherlands.
- [30] J.R. Ullmann. An algorithm for subgraph isomorphism. *Journal of the Association for Computing Machinery*, 23(1):pp. 31–42, 1976.

Appendix A

Supplementary RDF Issues

This appendix discussed some RDF issues in more detail. In this appendix, the following notation is used: ‘u’ for URI references, ‘b’ for blank node identifiers and ‘l’ for literals. When a resource is allowed to be a URI reference or a blank node, the notation ‘ub’ is used.

A.1 Simple Entailment Rules

The Equivalence Solving Method for simple entailment uses the entailment rules given in Table A.1. These rules are applied to an RDF graph to generate a generalization of this graph by adding lots of allocated blank nodes. However, these rules allow blank nodes to proliferate and applying them naively would not result in an efficient search process, since the rules will not terminate and can produce arbitrarily many redundant derivations of equivalent triples.

rule name	if graph contains	then add
se1	(ub, u, ubl)	(ub, u, b_{ubl}) where b_{ubl} identifies a blank node allocated to ubl by rule se1 or se2.
se2	(ub, u, ubl)	(b_{ub} , u, ubl) where b_{ub} identifies a blank node allocated to ub by rule se1 or se2.

Table A.1: Simple Entailment Rules

A.2 RDFS Closure Computation

A possible computation for the RDFS closure $G_{s,K}$ of an RDF graph G given in the specifications of RDF [7] is formulated as follows.

1. Add to G all the RDF and RDFS axiomatic triples.
2. Apply rule lg to any triple containing a literal until the graph is unchanged by the rule.
3. Apply rules rdf2 and rdfs1 until the graph is unchanged.
4. Apply rule rdf1, rule gl and the remaining RDFS entailment rules until the graph is unchanged.

In the next tables, some of the triples and rules used for the RDFS closure computation are given. An overview of all the triples and rules is given in the RDF Semantics document [5].

(rdf:type, rdf:type, rdf:Property)
(rdf:subject, rdf:type, rdf:Property)
(rdf:predicate, rdf:type, rdf:Property)
(rdf:object, rdf:type, rdf:Property)
(rdf:first, rdf:type, rdf:Property)
(rdf:rest, rdf:type, rdf:Property)
(rdf:value, rdf:type, rdf:Property)
(rdf:_1, rdf:type, rdf:Property)
...
(rdf:nil, rdf:type, rdf:List)

Table A.2: RDF axiomatic triples

rule name	if graph contains rule	then add rule
lg	(ub, u, l)	(ub, u, b _l) where b _l identifies a blank node allocated to the literal l by this rule.
gl	(ub, u, b) where b identifies a blank node allocated to the literal l by rule lg.	(ub, u, l)
rdf1	(ub, u, ubl)	(u, rdf:type, rdf:Property)
rdf2	(ub, u, l) where l is a well-typed XML literal	(b _l , rdf:type, rdf:XMLLiteral) where b _l identifies a blank node allocated to l by rule lg

Table A.3: Literal Rules and RDF Entailment Rules

A.3 XML Clash

The original definition of RDFS entailment given in [5] contains the possibility of an XML clash. In case of an XML clash, an ill-typed XML literal is claimed to be well-formed. An XML clash can occur during the RDFS closure computation when the following two triples $(a, p, \text{"s"} \wedge \wedge \text{XMLLiteral})$ and $(p, \text{rdfs:range}, \text{rdfs:Literal})$ are defined for an (ill-typed) XML literal "s". Applying rule lg will add triple (a, p, b_l) where b_l is a blank node allocated to "s". Rule rdfs3 will now add triple $(b_l, \text{rdf:type}, \text{rdfs:Literal})$, claiming that the literal b_l is allocated to, i.e., "s", is well-formed. This is in contrast to the given that "s" is ill-typed. Strictly speaking, this contradiction can lead to more 'wrong' conclusions, and the RDFS closure is said to be of no use. However, in practice, the occurrence of an XML clash will lead to a warning message.

In order to detect an XML clash, the following steps can be taken:

- Before RDFS closure computation: Determine for all given XML literals whether they are well-formed or ill-typed, using an XML checker.
- After RDFS closure computation: Check for every ill-typed XML literal "s" whether statement $(b_l, \text{rdf:type}, \text{rdfs:Literal})$ occurs and b_l is allocated to "s".
If such a statement occurs, return a warning.

Given this procedure to detect and finalize an XML clash, the occurrence of an XML clash is disregarded from now on.

A.4 Finite Container Construction

The RDF axiomatic triple set given in the RDF Semantics document [5] is infinite because of the container elements. A container can have an infinite number of elements and the RDF axiomatic triple set provides rules for all of these elements. However, in practice, every set is finite, and so only adding the axiomatic triples for existing container elements will lead to the same semantic meaning. So, instead of adding the infinite triples $(\text{rdf:}_i, \text{rdf:type}, \text{rdf:Property})$ for $1 \leq i \leq \infty$, the following finite container construction can be applied.

For all rdf:_i occurring in the graph:

- Add triple $(\text{rdf:}_i, \text{rdfs:domain}, \text{rdfs:Resource})$
- Add triple $(\text{rdf:}_i, \text{rdfs:range}, \text{rdfs:Resource})$
- Add triple $(\text{rdf:}_i, \text{rdf:type}, \text{rdfs:ContainerMembershipProperty})$

This finite container construction leads to a finite, equivalent RDF axiomatic triple set.

Appendix B

Ullmann's Algorithm For Subgraph Isomorphism

Ullmann [30] describes a depth-first tree-search algorithm that finds all the subgraph isomorphisms between two given graphs. This algorithm works on two graphs at the same time and after each vertex mapping in the tree search a refinement procedure is applied which generally results in a reduction in the number of successor vertices that must be searched in. If a mapping is not feasible, the algorithm backtracks to an other mapping.

B.1 Given Algorithm

The implementation of Ullmann's Algorithm given in [30] is formulated as follows.

- Step 1 $M := M^0$, $d := 1$, $H_1 := 0$,
for all $i := 1, \dots, p_\beta$ set $F_i := 0$;
refine M , if exit FAIL then terminate algorithm;
- Step 2 If there is no value of j such that $m_{dj} = 1$ and $F_j = 0$ then go to step 7,
 $M_d := M$;
if $d = 1$ then $k = H_1$ else $k := 0$;
- Step 3 $k := k + 1$,
If $m_{dk} = 0$ or $F_k = 1$ then go to step 3;
for all $j \neq k$ set $m_{dj} = 0$;
refine M ; if exit FAIL then go to step 5;
- Step 4 If $d < p_\alpha$ then go to step 6
else give output to indicate that an isomorphism has been found;
- Step 5 $M := M_d$;
If there is no $j > k$ such that $m_{dj} = 1$ and $F_j = 0$ then go to step 7;
go to step 3;
- Step 6 $H_d := k$, $F_k := 1$; $d := d + 1$;
go to step 2,
- Step 7 If $d = 1$ then terminate algorithm,
 $d := d - 1$; $M := M_d$, $k := H_d$; $F_k := 0$;
go to step 5;

The initial mapping matrix M^0 is constructed as follows.

$$\begin{aligned} m_{ij}^0 &= 1 && \text{if degree of vertex } v_i \text{ in } G_\alpha \text{ is smaller than or equal to degree of vertex } v_j \text{ in } G_\beta \\ m_{ij}^0 &= 0 && \text{otherwise} \end{aligned}$$

The refinement procedure amounts to checking the following condition for every $m_{ij} = 1$ in M .

$$\forall_{1 \leq x \leq p_\alpha} (a_{ix} = 1) \Rightarrow \exists_{1 \leq y \leq p_\beta} (m_{xy} \cdot b_{jy} = 1)$$

B.2 Original Experiments

The original results [30] of experiments carried out using Ullmann's Algorithm are given in Table B.1, where p_α is the number of vertices of G_α and p_β and q_β are respectively the number of vertices and edges of G_β . The experiments are carried out 50 times and s.d. represents the standard deviation.

p_α	p_β	q_β		nr of isomorphisms		time in sec	
		average	s.d.	average	s.d.	average	s.d.
6	12	21.1	3.1	960.8	140.4	14.5	13.11
8	12	23.5	2.9	1223.0	142.8	44.5	55.4
10	12	26.0	3.4	949.1	121.2	124.0	90.4
7	14	28.3	4.0	4769.9	88.9	97.6	118.7

Table B.1: Original Results of Experiments with Ullmann's Algorithm

Note that, given p_α and p_β , graphs G_α and G_β are generated randomly using a random number generator such that its adjacency matrix is filled with 0's and 1's with a probability of 0.25 for the off-diagonal elements. As explained in [30], the experiments only use connected graphs so a generated adjacency matrix is checked for connectedness and rejected if it is not connected. Furthermore graph G_β is made more dense by applying the following rule after the random generation:

$$\forall_{1 \leq i, j \leq p_\alpha} (b_{i,j} = b_{i,j} \vee a_{i,j}) \tag{B.1}$$

B.3 New Experiments

In order to carry out new experiments, Ullmann's Algorithm is implemented, as is explained in Section 5.2. The results of the new experiments are given in Table B.2. Some results for larger graphs are added based on values of p_α and p_β of experiments of other Ullmann's Algorithm implementations [19]. Furthermore, results are added for pure randomly generated graphs, which obviously shows that less isomorphisms are found for a pure randomly generated G_β . For these pure randomly generated graphs, q_β is equal to $\frac{p_\beta * (p_\beta - 1)}{2} * 0.25$, and is therefore left out in Table B.2. Note that *SRG* stands for semi-random graphs and *PRG* stands for pure random graphs.

	p_α	p_β	q_β		nr of isomorphisms		time in sec	
			average	s.d.	average	s.d.	average	s.d.
SRG	6	12	21.9	2.7	1076.0	1313.1	0.018	0.021
PRG	6	12			308.2	460.3	0.005	0.010
SRG	8	12	24.0	2.8	3020.3	5455.4	0.087	0.109
PRG	8	12			648.7	2394.6	0.022	0.093
SRG	10	12	27.2	3.6	2636.8	3970.5	0.206	0.240
PRG	10	12			232.7	768.1	0.013	0.038
SRG	7	14	28.7	2.5	5914.4	8616.3	0.095	0.119
PRG	7	14			1763.3	3973.1	0.028	0.054
SRG	3	64	504.9	18.9	14,322.7	3731.1	0.133	0.036
PRG	3	64			14,086.2	4298.9	0.123	0.029
SRG	3	128	2018.3	39.5	120,778.0	27,987.1	1.717	0.271
PRG	3	128			113,477.3	36,047.9	1.568	0.334
SRG	3	256	8149.3	62.1	950,977.4	213,651.6	23.823	4.083
PRG	3	256			961,703.8	236,196.2	23.157	4.073
SRG	4	128	2042.8	42.7	3,468,605.4	1,953,024.4	62.875	21.219
PRG	4	128			3,072,648.6	1,447,829.3	53.727	23.604

Table B.2: Experimental Results of New Implementation of Ullmann's Algorithm

B.4 Possible Improvements

Because all possible mappings are based on the initial mapping matrix M^0 , improving the construction of M^0 can lead to a reduction of infeasible mappings in M^0 , and so a reduction of the total computation time. In order to improve the construction of M^0 , graph G_β is preprocessed to add vertex fingerprints. These fingerprints attribute to a faster and more precise check for the vertex mappings. Next, two possible fingerprints are introduced.

B.4.1 First Fingerprint

A fingerprint for vertex v is an array containing the number of edges connected to neighbors of v , in order of increasing distance, formulated as follows.

$$(d_{out}(N_0), d_{out}(N_1), \dots, d_{out}(N_k))$$

For a given vertex v : $N_0(v) = \{v\}$
 $N_i(v) = \{u \mid d(v, u) = i\}$

where $d(v, u)$ is the minimum distance between v and u . If v and u are not connected, then by definition $d(v, u) = \infty$. $d_{out}(S)$ gives the number of edges that each run between a vertex in S and a vertex outside S .

The original mapping matrix M^0 is constructed using the degrees of vertices itself, i.e., with fingerprint $(d_{out}(N_0))$. The new construction of M^0 is as follows.

$$\begin{aligned}
 m_{ij}^0 &= 1 && \text{if fingerprint of vertex } v_j \text{ in } G_\beta \text{ is not pairwise smaller than} \\
 &&& \text{fingerprint of vertex } v_i \text{ in } G_\alpha \\
 m_{ij}^0 &= 0 && \text{otherwise}
 \end{aligned}$$

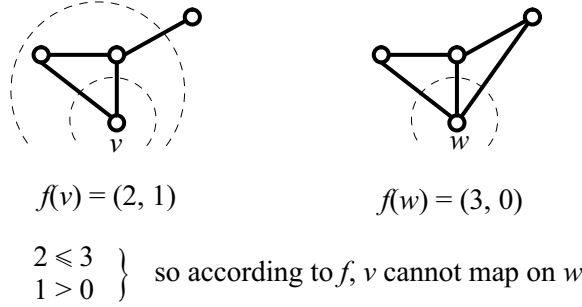


Figure B.1: Counter Example for First Fingerprint

Although this fingerprint looks like a good improvement, a counter example in Figure B.1 shows that this fingerprint is incorrect. Two vertices in graph G_α can have path length 2, whereas those vertices can have path length 1 in G_β . So the concept of the corresponding constructed subgraphs is not correct.

B.4.2 Second Fingerprint

The second fingerprint is based on the first fingerprint, however, $N_i(v)$ now denotes the number of vertices that can be visited in i steps or less from vertex v , so $N_i(v) = \sum_{j=1}^i N_j(v)$. This fingerprint is formulated as follows.

$$(N_1, N_2, \dots, N_k)$$

The new construction of initial mapping matrix M^0 is given next.

$$\begin{aligned}
 m_{ij}^0 &= 1 && \text{if fingerprint of vertex } v_i \text{ in } G_\alpha \text{ is piecewise smaller than} \\
 &&& \text{or equal to fingerprint of vertex } w_j \text{ in } G_\beta \\
 m_{ij}^0 &= 0 && \text{otherwise}
 \end{aligned}$$

The correctness of this fingerprint is proven next.

Lemma. *For given vertex v_i of graph G_α and vertex v_j of graph G_β , if fingerprint of v_i is piecewise smaller than or equal to fingerprint of v_j , then v_i can map on v_j .*

Proof. Given fingerprints $(N_1(v), N_2(v), \dots, N_k(v))$ and $(N_1(w), N_2(w), \dots, N_k(w))$ of respectively vertex v_i and v_j . For $k = 1$, $N_1(v)$ is the number of neighbors of vertex v_i plus v_i itself. So if $N_1(v) \leq N_1(w)$, the degree of v_i is smaller than or equal to the degree of v_j and it is trivial that v_i can map on v_j . For all other elements of the fingerprints: assume $N_i(v) \leq N_i(w)$ for $1, \dots, i$ and $i < k - 1$. The vertices of $N_i(v)$ then form a connected subgraph in G_α with center v_i , say $sub_{\alpha,i}(v)$. The same holds for $N_i(w)$ in G_β , so $sub_{\beta,i}(w)$. Now, $N_{i+1}(v) > N_{i+1}(w)$ means subgraph $sub_{\alpha,i+1}(v)$ has more vertices than subgraph $sub_{\beta,i+1}(w)$. According to the mapping definition, $sub_{\alpha,i+1}(v)$ cannot map on $sub_{\beta,i+1}(w)$, and so vertex v_i cannot map on vertex v_j . So only if $(N_i(v) \leq N_i(w))$ holds for all $i \leq k$, vertex v_i can map on vertex v_j . \square

The length of this fingerprint is not fixed, however, it is not profitable to compute large fingerprints. Because of the high connectivity of dense graphs, the size of N_i decreases rapidly by increasing the fingerprint length, and so (N_i) becomes really small. For sparse graphs, the size of N_i is small from the beginning of the fingerprint construction, but $d_{out}(N_i)$ becomes less significant.

Applying this fingerprint to randomly generated graphs does not have spectacular results. Namely, N_i is constructed in a cumulative way and graph G_β is always (a lot) bigger than G_α , so, independent of the length of the fingerprints, the fingerprints of vertices in G_β are always bigger than the fingerprints of vertices in G_α . Comparing these fingerprints will always result in the answer that the vertices of G_α can be mapped on the vertices in G_β , which does not lead to an improvement of the initial mapping matrix.

Coming up with other, more complicated fingerprints does not lead to better results. First of all because those graphs are unlabelled and undirected, so using the number of vertices or number of edges is the only possibility. Furthermore, G_α is small and sparse and G_β is big and dense, so the fingerprints for G_β will always be bigger than those of G_α .

Appendix C

RDF Subgraph Algorithm Implementation

This appendix describes the implementation of the RDF Subgraph Algorithm given in Section 5.3, and contains experimental results of the RDF Subgraph Algorithm carried out with semi-randomly generated graphs.

C.1 Implementation

The input for the RDF Subgraph Algorithm consists of a (RDFS partial closure) semantic graph G and an RDF query semantic graph Q . Basically, those graphs are labeled, directed pseudographs which are commonly represented by an adjacency matrix. The vertex labels are stored in an array. So RDF graph $G = (V, E, s, l_v, l_e)$ with $|V| = n$ and $|E| = m$ can be represented as follows.

- An array L of size n containing unique vertex labels.
- An adjacency matrix A of size $n \times n$ containing for every $A[i, j]$ the set of arc labels from v_i to v_j .

Note that the local identifier of a blank node can be considered as the label of that blank node. URI references, blank node identifiers and literals all have different underlying meanings, but are syntactically just strings, and so are treated as strings.

The input for the RDF Subgraph Algorithm are the following two semantic graphs.

- RDF semantic query graph Q with p vertices and q arcs, so array L of size p and adjacency matrix $A = [a_{i,j}]$ of size $p \times p$.
- RDF semantic graph G with n vertices and m arcs, so array F of size n and adjacency matrix $B = [b_{i,j}]$ of size $n \times n$.

Initial mapping matrix M^0 of size $p \times n$ is now constructed in accordance with

$$m_{ij}^0 = \begin{cases} 1 & \text{if } L[i] \in K \quad \text{and } (\bigcup_{in} a_i \subseteq \bigcup_{in} b_j \cup K) \text{ and } (\bigcup_{out} a_i \subseteq \bigcup_{out} b_j \cup K) \\ 1 & \text{if } L[i] = F[j] \quad \text{and } (\bigcup_{in} a_i \subseteq \bigcup_{in} b_j \cup K) \text{ and } (\bigcup_{out} a_i \subseteq \bigcup_{out} b_j \cup K) \\ 0 & \text{otherwise.} \end{cases}$$

where $\bigcup_{in} a_i$ is the set of incoming arc labels of vertex i and $\bigcup_{out} a_i$ is the set of outgoing arc labels of vertex i . The arc labels of b_i are united with the set of variable labels (K) because graph Q can contain variables.

The condition for the refinement procedure is as follows.

$$\forall_{1 \leq x \leq p} (|\ a_{ix} \ | \geq 1) \Rightarrow \exists_{1 \leq y \leq n} (m_{xy} = 1 \text{ and } a_{ix} \subseteq b_{jy} \cup K)$$

and

$$\forall_{1 \leq x \leq p} (|\ a_{xi} \ | \geq 1) \Rightarrow \exists_{1 \leq y \leq n} (m_{xy} = 1 \text{ and } a_{xi} \subseteq b_{yj} \cup K)$$

The RDFmain class and the RDFmatch class form the most important parts of the implementation and are given in Figure C.1.

C.2 Experiments

All results of experiments carried out using the RDF Subgraph Algorithm and semi-randomly generated graphs are given in Table C.1. Furthermore, this table contains experimental results of using the arc label and neighbor fingerprints. The results of these experiments are discussed in Section 6.3. Note that the following notation is used next.

- p denotes the number of vertices of Q
- q denotes the number of arcs of Q
- v_v denotes the number of vertex variables of Q
- v_a denotes the number of arc variables of Q
- n denotes the number of vertices of G
- m denotes the number of arcs of G
- av. denotes the average
- s.d. denotes the standard deviation for the results
- ** denotes that this experiment is not carried out

```
public class RDFmain {

    public static void main(String args[]) {
        Graph smallGraph = new Graph('graphQ.txt');
        Graph bigGraph = new Graph('graphG.txt');

        RDFmatch Effy = new RDFmatch();
        double totalMatchesFound = Effy.RDFmatch(smallGraph, bigGraph);
    }
}

public class RDFmatch {
    Matrix M0;
    double matchesFound;

    public double RDFmatch(Graph smallGraph, Graph bigGraph) {
        matchesFound = 0;
        M0 = smallGraph.makeMappingWith(bigGraph);
        if (M0.refine(smallGraph, bigGraph)) {
            RDFmatch(0, M0, out);
        }
        return matchesFound;
    }

    public void RDFmatch(int d, Matrix Mat) {
        if (d == Mat.rows) {
            matchesFound = matchesFound + 1;
        } else {
            for (int j = 0; j < Mat.columns; j++) {
                if (Mat.getElement(d, j) == 1) {
                    Matrix nextM = Mat;
                    nextM.clearRow(d, j);
                    if (nextM.refine(graph1, graph2)) {
                        RDFmatch(d + 1, nextM, out);
                    }
                }
            }
        }
    }
}
}
```

Figure C.1: Code for RDF Subgraph Algorithm

query graph Q				graph G		RDF Subgraph Algorithm				Arc Label Fingerprint				Neighbor Fingerprint			
p	q	v_v	v_a	n	m	RDF matches		time (sec)		RDF matches		time (sec)		RDF matches		time (sec)	
						av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.
3	6	0	0	200	2000	1.0	0.0	0.0037	0.008	1.0	0.0	0.0018	0.0066	1.0	0.0	0.0019	0.005
				1000	50,000	1.0	0.0	0.023	0.011	1.0	0.0	0.0024	0.0071	1.0	0.0	0.016	0.0012
				5000	250,000	1.0	0.0	0.112	0.089	1.0	0.0	0.0093	0.014	1.0	0.0	0.057	0.014
3	6	1	0	200	2000	1.0	0.0	0.041	0.018	1.0	0.0	0.028	0.028	1.0	0.0	0.041	0.015
				1000	50,000	1.0	0.0	3.429	0.15	1.0	0.0	1.52	1.47	1.0	0.0	3.09	0.021
				5000	250,000	1.0	0.0	87.49	7.01	1.0	0.0	47.98	39.04	1.0	0.0	85.14	7.58
3	6	2	1	200	2000	1.12	0.32	0.08	0.019	1.16	0.04	0.052	0.025	1.04	0.2	0.077	0.012
				1000	50,000	1.08	0.27	6.516	0.16	1.04	0.2	4.53	1.57	1.1	0.3	6.20	0.07
				5000	250,000	1.0	0.0	174.92	16.06	1.0	0.0	125.5	41.1	1.0	0.0	168.32	14.8
3	6	0	6	200	2000	2.92	2.09	0.0025	0.0073	2.84	1.64	0.0019	0.0067	3.0	1.47	0.0075	0.021
				1000	50,000	2.24	2.93	0.023	0.013	3.4	2.06	0.0006	0.0031	3.8	2.01	0.012	0.009
				5000	250,000	3.0	1.41	0.163	0.397	2.5	1.57	0.0094	0.012	3.0	1.55	0.122	0.014
3	6	3	6	200	2000	65.64	62.56	0.33	0.38	53.9	68.9	0.203	0.19	66.1	99.7	0.301	0.348
				1000	50,000	327.24	346.56	55.29	102.43	526.9	1081.8	48.5	80.8	245.8	191.6	53.4	38.8
				5000	250,000	45.4	52.9	421.8	454.7	61.4	101.6	373.2	391.4	**	**	**	**

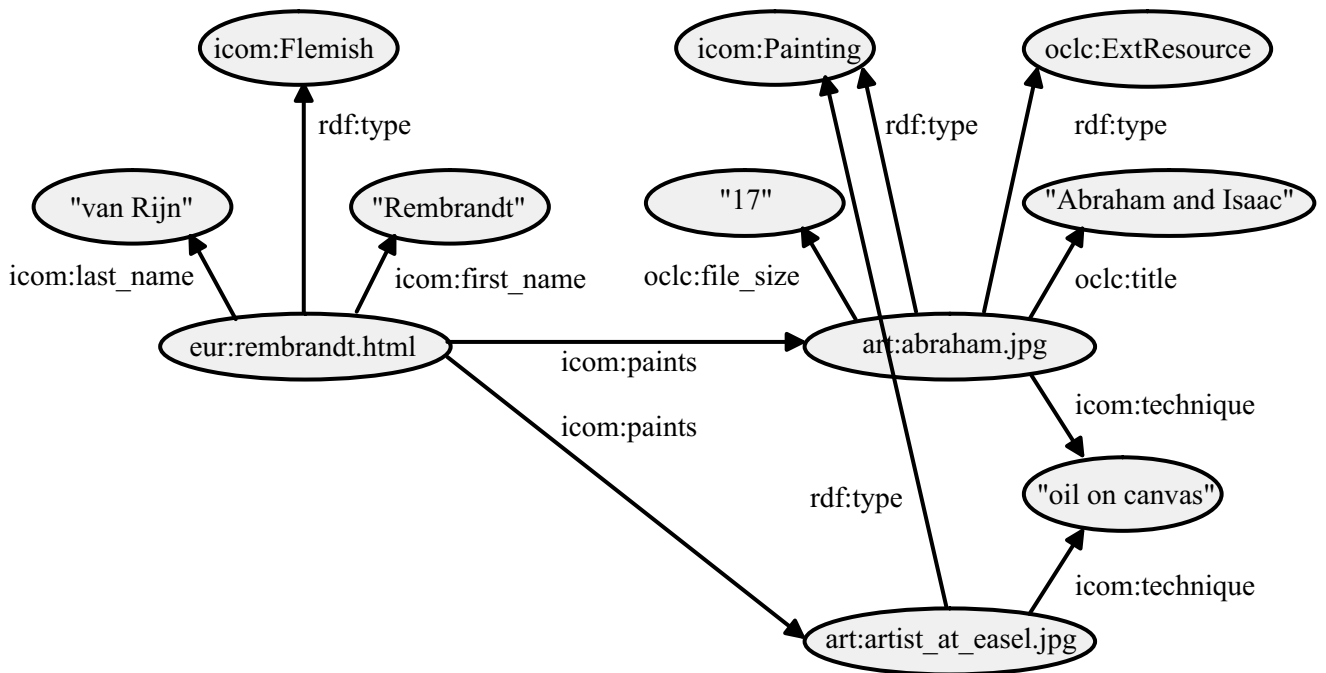
query graph Q				graph G		RDF Subgraph Algorithm				Arc Label Fingerprint				Neighbor Fingerprint			
p	q	v_v	v_a	n	m	RDF matches		time (sec)		RDF matches		time (sec)		RDF matches		time (sec)	
						av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.	av.	s.d.
6	6	0	0	200	2000	1.0	0.0	0.0043	0.008	1.0	0.0	0.0031	0.0076	1.0	0.0	0.005	0.008
				1000	50,000	1.0	0.0	0.0425	0.011	1.0	0.0	0.0050	0.0086	1.0	0.0	0.023	0.01
				5000	250,000	1.0	0.0	1.44	5.76	1.0	0.0	0.022	0.024	1.0	0.0	1.21	0.24
6	6	3	0	200	2000	1.28	0.45	0.119	0.026	1.08	0.27	0.056	0.033	1.36	0.56	0.118	0.031
				1000	50,000	1.12	0.32	9.709	0.29	1.1	0.3	4.42	2.78	1.2	0.6	9.30	0.09
				5000	250,000	1.04	0.20	260.2	22.7	1.04	0.2	113.4	51.7	1.0	0.0	260.3	20.13
6	6	0	3	200	2000	1.4	0.69	0.0044	0.010	1.16	0.37	0.0025	0.0058	1.28	0.53	0.0032	0.008
				1000	50,000	1.48	0.75	0.041	0.018	1.5	0.92	0.0079	0.019	1.5	0.92	0.027	0.01
				5000	250,000	1.0	0.0	0.189	0.080	1.0	0.0	0.016	0.017	1.0	0.0	0.119	0.016
6	6	2	2	200	2000	41.6	60.4	0.106	0.038	39.48	70.12	0.041	0.062	36.52	55.30	0.101	0.052
				1000	50,000	584.08	904.12	7.657	2.478	785.8	994.9	1.06	1.22	1251.2	1454.7	7.827	1.81
				5000	250,000	1161.5	990.83	197.58	60.16	902.8	1516.0	6.44	7.73	718.3	1085.6	213.7	95.4
6	6	6	6	200	2000	1,365,726	1,309,324	707.67	835.52	**	**	**	**	**	**	**	**

Table C.1: Results of RDF Subgraph Algorithm, Arc Label and Neighbor Fingerprint

Appendix D

Museum Graph

The Museum Graph is part of the Museum Repository [10] given by Sesame as a test demo, and is about painter Rembrandt and two of his paintings, 'Abraham and Isaac' and 'Artist at his easel'. The relations between Rembrandt and these paintings are given in Figure D.1 as an RDF semantic graph. This graph can be converted into a set of triples as is shown in Table D.1. The XML/RDF notation of the Museum Graph is given in Table D.2. The relations defined in this graph are derived from the *icom* and *oclc* schemas. An overview of the schema data and Rembrandt data are given in Figure D.2. The partial RDFS closure graph is shown in the triple notation in Table D.3.



art = <http://www.artchive.com/rembrandt/>
eur = <http://www.european-history.com/>
rdf = <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

Figure D.1: Museum Graph (RDF Semantic Graph)

Table D.1: Museum Graph (RDF Triple Graph)

subject	predicate	object
http://www.european-history.com/rembrandt.html	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.icom.com/schema.rdf#Flemish
http://www.european-history.com/rembrandt.html	http://www.icom.com/schema.rdf#paints	http://www.artchive.com/rembrandt/abraham.jpg
http://www.european-history.com/rembrandt.html	http://www.icom.com/schema.rdf#paints	http://www.artchive.com/rembrandt/artist-at-easel.jpg
http://www.european-history.com/rembrandt.html	http://www.icom.com/schema.rdf#last-name	“van Rijn”@en
http://www.european-history.com/rembrandt.html	http://www.icom.com/schema.rdf#first-name	“Rembrandt”@en
http://www.artchive.com/rembrandt/abraham.jpg	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.icom.com/schema.rdf#Painting
http://www.artchive.com/rembrandt/abraham.jpg	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.oclc.org/schema.rdf#ExtResource
http://www.artchive.com/rembrandt/abraham.jpg	http://www.icom.com/schema.rdf#technique	“oil on canvas”@en
http://www.artchive.com/rembrandt/abraham.jpg	http://www.oclc.org/schema.rdf#file-size	“17”@en
http://www.artchive.com/rembrandt/abraham.jpg	http://www.oclc.org/schema.rdf#title	“Abraham and Isaac”@en
http://www.artchive.com/rembrandt/artist-at-easel.jpg	http://www.w3.org/1999/02/22-rdf-syntax-ns#type	http://www.icom.com/schema.rdf#Painting
http://www.artchive.com/rembrandt/artist-at-easel.jpg	http://www.icom.com/schema.rdf#technique	“oil on canvas”@en

```

<?xml version="1.0" encoding="UTF-8"?>
<rdf:RDF
  xmlns:icom="http://www.icom.com/schema.rdf#"
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:oclc="http://www.oclc.org/schema.rdf#" >
<!-- Data statements -->

<rdf:Description rdf:about="http://www.artchive.com/rembrandt/artist-at-easel.jpg">
  <rdf:type rdf:resource="http://www.icom.com/schema.rdf#Painting" />
  <icom:exhibited rdf:resource="http://www.louvre.fr" />
  <icom:technique xml:lang="en">oil on canvas</icom:technique>
</rdf:Description>

<rdf:Description rdf:about="http://www.european-history.com/rembrandt.html">
  <rdf:type rdf:resource="http://www.icom.com/schema.rdf#Flemish" />
  <icom:paints rdf:resource="http://www.artchive.com/rembrandt/abraham.jpg" />
  <icom:paints rdf:resource="http://www.artchive.com/rembrandt/artist-at-easel.jpg" />
  <icom:last-name xml:lang="en">van Rijn</icom:last-name>
  <icom:first-name xml:lang="en">Rembrandt</icom:first-name>
</rdf:Description>

<rdf:Description rdf:about="http://www.artchive.com/rembrandt/abraham.jpg">
  <rdf:type rdf:resource="http://www.icom.com/schema.rdf#Painting" />
  <rdf:type rdf:resource="http://www.oclc.org/schema.rdf#ExtResource" />
  <icom:technique xml:lang="en">oil on canvas</icom:technique>
  <oclc:title xml:lang="en">Abraham and Isaac</oclc:title>
  <oclc:file-size xml:lang="en"> 17 </oclc:file-size>
</rdf:Description>

</rdf:RDF>

```

Table D.2: Museum Graph (XML notation)

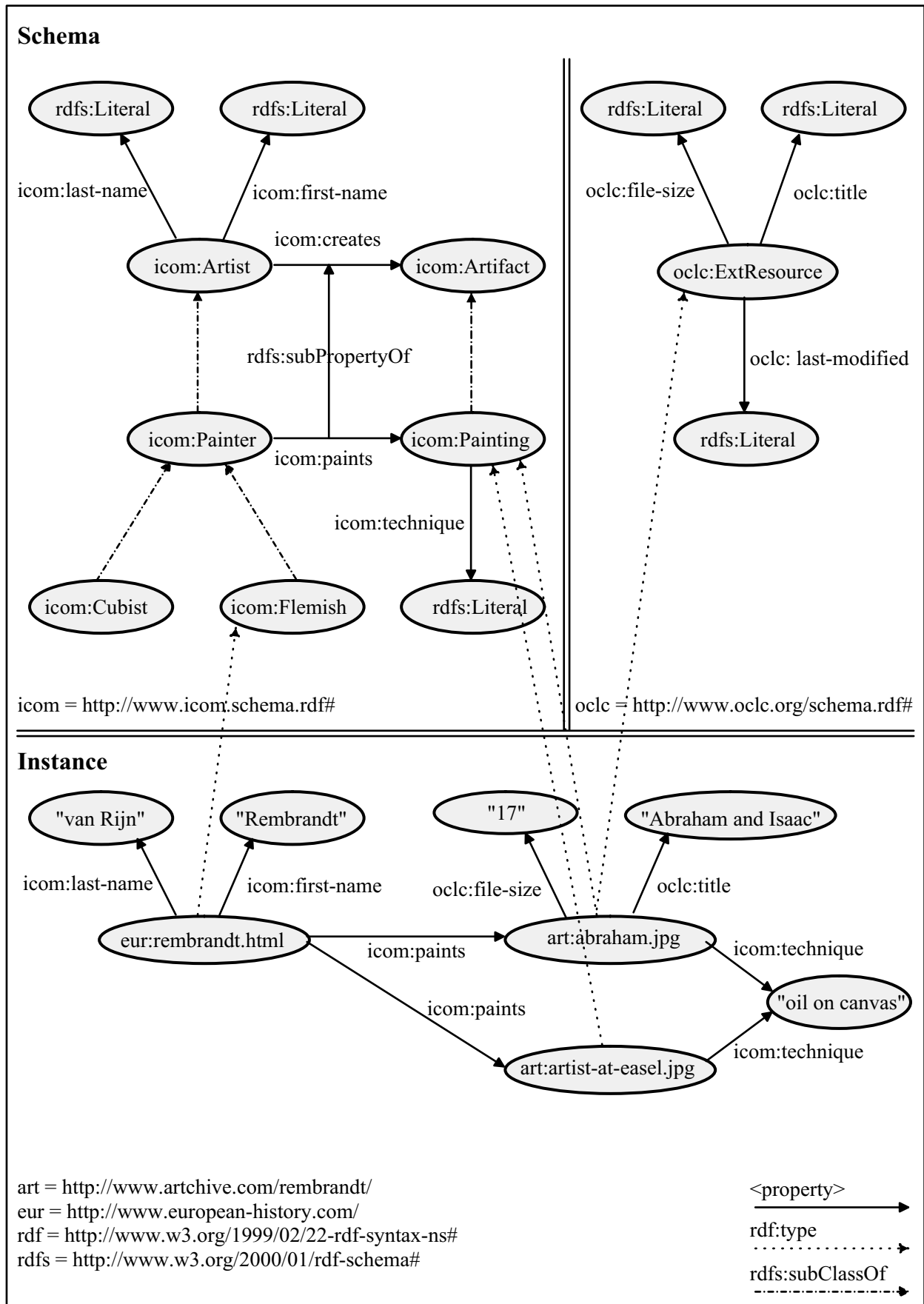


Figure D.2: Museum Graph and Schemas

