SEMANTICS AND COMPLEXITY OF SPARQL 1.1 PROPERTY PATHS

SEBASTIÁN CONCA

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Advisor:
MARCELO ARENAS

Santiago de Chile, March 2012

© MMXI, SEBASTIÁN CONCA
SEMANTICS AND COMPLEXITY OF SPARQL 1.1 PROPERTY PATHS

SEBASTIÁN CONCA

Members of the Committee:
MARCELO ARENAS
JORGE BAIER
JORGE PEREZ
MIGUEL RÍOS

Thesis submitted to the Office of Research and Graduate Studies in partial fulfillment of the requirements for the degree of Master of Science in Engineering

Santiago de Chile, March 2012

© MMXI, SEBASTIÁN CONCA
Gratefully to my family
ACKNOWLEDGEMENTS

I want to thank my advisor Marcelo Arenas for his immense support and close collaboration.
# TABLE OF CONTENTS

ACKNOWLEDGEMENTS .............................................. iv

LIST OF FIGURES ................................................... vii

LIST OF TABLES ..................................................... viii

ABSTRACT ............................................................ ix

RESUMEN ............................................................ x

1. INTRODUCTION .................................................. 1

2. EXPERIMENTS ................................................... 7
   2.1. Experimental setting ........................................ 7
   2.2. The clique experiment ...................................... 8
   2.3. The foaf experiment ....................................... 10

3. FORMALIZING SPARQL ............................................ 12

4. PROPERTY PATHS ................................................. 15

5. INTRACTABILITY OF SPARQL 1.1 IN THE PRESENCE OF PROPERTY PATHS 18
   5.1. Auxiliary terminology ...................................... 18
   5.2. Property path evaluation .................................. 21
   5.3. The complexity of the entire language .................... 36

6. INTRACTABILITY FOR ALTERNATIVE SEMANTICS THAT COUNT PATHS .. 37
   6.1. Classical paths over acyclic RDF graphs .................. 37
   6.2. Simple paths ............................................... 48

7. AN EXISTENTIAL SEMANTICS TO THE RESCUE .................... 51
   7.1. Discarding duplicates from the standard and simple-paths semantics .... 52
   7.2. Existential semantics and SPARQL 1.1 ...................... 55
   7.3. Experiments for the existential semantics .................. 60
8. CONCLUSIONS: OUR RESULTS IN PERSPECTIVE 65
REFERENCES 67
LIST OF FIGURES

1.1 Time in seconds for processing Cliq-1 w.r.t. the clique size $n$ (time axis in log-scale) ........................... 2
1.2 RDF graph representing a clique with 8 nodes .......................................................... 2

4.1 Procedures used in the evaluation of property-path triples of the form $(u, path^*, v)$ .................. 17

5.1 Example RFD Graph $G$ .................................................................................. 19
5.2 A tree of recursive calls generated by invoking procedure ALP ........................................ 20
5.3 Simplified view of the tree shown in Figure 5.2 ...................................................... 21
5.4 Alternative representation of the tree shown in Figure 5.2 ........................................... 22
5.5 Tree of recursive calls generated by invoking $ALP(a_1, p, Result, Visited, clique(5))$, with $Result$ an empty list and $Visited$ an empty set. Element $a_5$ is represented as a grey node. The depth in the tree is shown in the left-hand side, while the number of repetitions of each node at each depth is shown in the right-hand side. .......................... 32

6.1 NFA $M_0$ used as example in the proof of Theorem 6.1 ........................................... 42
6.2 The building block of $G'$ for the NFA $M_0$ .................................................................. 42
6.3 RDF graph $G'$ for NFA $M_0$ and a string $x$ such that $|x| = 2$ .............................. 43
6.4 Construction of the set $\Gamma$ of property path for NFA $M_0$ .................................... 44
6.5 RDF graph $G$ for NFA $M_0$ and a string $x$ such that $|x| = 2$ .............................. 45


**LIST OF TABLES**

1.1 Time in seconds for processing query Foaf-1 ........................................... 3
1.2 Time in seconds for processing Foaf-1D .................................................. 6

2.1 Time in seconds and number of solutions for query Cliq-1 (CliqF-1 for RDF::Query) . 9
2.2 Time in seconds and number of solutions for queries Cliq-2 (left) and Cliq-3 (right) . 9
2.3 Description of the files used in the foaf experiment. ................................. 11
2.4 Time in seconds, number of solutions, and output size for query Foaf-1 ............. 11

5.1 Number of occurrences of the mapping with empty domain in the answer to property-
    path triple \((a_1, p_s, a_n)\) over the RDF graph \(\text{clique}(n)\), according to the semantics for
    property paths proposed by the W3C. .................................................. 34

7.1 Time in seconds for queries Cliq-1D, Cliq-2D, and Cliq-3D (symbol “–” means one-
    hour timeout). ..................................................................................... 64
ABSTRACT

SPARQL –the standard query language for querying RDF– provides only limited navigational functionalities, although these features are of fundamental importance for graph data formats such as RDF. This has led the W3C to include the property path feature in the upcoming version of the standard, SPARQL 1.1.

We tested several implementations of SPARQL 1.1 handling property path queries, and we observed that their evaluation methods for this class of queries have a poor performance even in some very simple scenarios. To formally explain this fact, we conduct a theoretical study of the computational complexity of property paths evaluation. Our results imply that the poor performance of the tested implementations is not a problem of these particular systems, but of the specification itself. In fact, we show that any implementation that adheres to the SPARQL 1.1 specification (as of November 2011) is doomed to show the same behavior, being the key issue the need for counting solutions imposed by the current specification. We provide several intractability results, that together with our empirical results, provide strong evidence against the current semantics of SPARQL 1.1 property paths. Finally, we put our results in perspective, and propose a natural alternative semantics with tractable evaluation, that we think may lead to a wide adoption of the language by practitioners, developers and theoreticians.

Keywords: SPARQL, Property Paths, Computational Complexity.
RESUMEN

SPARQL – el lenguaje de consultas standar para bases de datos RDF – proporciona sólo funcionalidades limitadas de navegación, sin embargo, estas características son de fundamental importancia en los modelos de datos basadas en grafos, como es el caso de RDF. Esto ha llevado a la W3C ha incluir funcionalidades como property path en la próxima versión del standar, SPARQL 1.1.

Se han puesto a prueba distintas implementaciones de SPARQL 1.1 que manejan consultas con property paths, observando un bajo rendimiento en sus métodos de evaluación para este tipo de consultas, incluso, para escenarios simples. En búsqueda de una explicación formal a este comportamiento, se ha realizado un estudio de la complejidad computacional de la evaluación de consultas con property paths. Los resultados muestran que el bajo rendimiento de las implementaciones probadas, no corresponde a un problema de estos sistemas en particular, si no más bien, tiene que ver con la especificación. De hecho, se muestra que cualquier implementación que siga la especificación de SPARQL 1.1 (a Noviembre del 2011), está destinada a mantener el mismo comportamiento, siendo el mayor problema, la necesidad de contar soluciones impuesta en la propuesta actual. En esta tesis se incluyen diversos resultados teóricos que demuestran la imposibilidad de computar consultas de property paths en tiempos razonables, que junto a los resultados empíricos, entregan una evidencia contundente en contra de la actual propuesta de semántica para property paths en SPARQL 1.1. Finalmente, se porpone una semántica natural alternativa que resuelve los problemas de desempeño, permitiendo así la adopción del standar por parte de usuarios, desarrolladores y teóricos.

Palabras Claves: SPARQL, Property Paths, Complejidad Computacional.
1. INTRODUCTION

It has been noted that, although RDF is a graph data format, its standard query language, SPARQL, provides only limited navigational functionalities. This has led the W3C to include the property-path feature in the upcoming version of the standard, SPARQL 1.1. Property paths are essentially regular expressions that retrieve pairs of nodes of an RDF graph that are connected by paths conforming to those expressions. In this paper, we study the semantics of property paths and the complexity of evaluating them. We perform this study both from a theoretical and a practical point of view, and provide strong arguments against the current semantics of SPARQL 1.1 property paths.

We began our study by testing several SPARQL 1.1 implementations, and we were faced with an intriguing empirical observation: all these implementations of SPARQL 1.1 fail to give an answer in a reasonable time (one hour) even for small input graphs and very simple property path expressions. We conduct two sets of experiments, the clique experiments and the foaf experiments, The former considers synthetic data, and the latter real RDF data extracted from the Web (see Sections 2.2 and 2.3 for a detailed exposition).

We test four implementations: ARQ (ZZ1, 2011), RDF::Query (ZZ3, 2011), KGRAM-Corese (ZZ2, 2011), and Sesame (ZZ4, 2011). In both experiments our empirical results were really surprising. Even for very simple queries and small input graphs, the implementations exceeded a timeout of 60 minutes.

For the first experiment, we consider RDF graphs representing cliques (complete graphs) of different sizes. For example, Figure 1.2 shows a clique with 8 nodes in N3 notation. In this scenario, we tested the performance of the implementations by using a very simple query:

\[ \text{Cliq-1: } \text{SELECT * WHERE :a0 (:p)* :a1} \]

that essentially asks for paths of arbitrary length between two fixed nodes. The experimental behavior for this query was quiet surprising: no implementation was able to handle a clique with 13 nodes. That is, all implementations fail to give an answer after one hour for an input RDF graph with only 156 triples and 970 bytes of size on disk. In particular, Sesame fails for a clique with 10 nodes, KGRAM and RDF::Query for 12 nodes, and ARQ for 13 nodes.
Our experiments show that for all implementations, the time needed to process Cliq-1 seems to grow doubly-exponentially w.r.t. the input data file (see the graph in Figure 1.1, which is in logarithmic scale). We also tested queries with nested stars, showing that nesting has an unexpected impact in query evaluation. In particular, we tested the query:

\[
\text{Cliq-2: } \text{SELECT } * \text{ WHERE } \{ :a0 \ ( (:p)* )* :a1 \},
\]

for which no implementation was able to handle even a clique with 8 nodes. That is, the implementations fail for the input graph shown in Figure 1.2 (which occupied only 378 bytes on disk).

To show that this behavior also appears with real data, we devised an experiment with data crawled from the Web. We constructed RDF graphs from foaf documents crawled by following
foaf:knows links starting from Axel Polleres’ foaf document. We considered several test cases of increasing size, from 9.2 KB (38 nodes and 119 triples) to 25.8 KB (76 nodes and 360 triples), and we tested the following simple query asking for the network of friends of Axel Polleres:

\[
\text{Foaf-1: SELECT * WHERE \{ axel:me (foaf:knows)* ?x \}.}
\]

As in the previous case, the results are striking. For query Foaf-1 all the implementations exceeded the timeout for an input RDF graph of 14.8 KB (with only 54 nodes and 201 triples). Table 1.1 shows the behavior of the different implementations for different input sizes. The “–” symbol in the table means timeout (one hour).

As our experiments show, for the tested implementations, property path evaluation is essentially infeasible in practice. But, what is the reason for this behavior? Is this only a problem of the particular implementations that we tested? Or is there a fundamental problem in the SPARQL 1.1 specification? Our theoretical results show that this last question is the key to understand this issue. In fact, we formally prove that, essentially, any implementation that follows the SPARQL 1.1 specification (as of November 2011) (Harris & Seaborne, 2011) will be doomed to show the same behavior.

We begin our theoretical study by formalizing the semantics of property paths. SPARQL 1.1 defines a bag (or multiset) semantics for these expressions. That is, when evaluating property-path expressions one can obtain several duplicates for the same solution.

---

1axel: prefix is <http://www.polleres.net/foaf.rdf#>. 
one duplicate for every different path in the graph satisfying the expression. For example, every solution of query Cliq-1 is an empty tuple that can have several duplicates in the output. ARQ for instance, represents this empty tuple as \(|\) \(\mid\), and for query Cliq-1 it returns several copies of \(|\) \(\mid\). Since RDF graphs containing cycles may lead to an infinite number of paths, satisfying a particular expression, the official specification defines the semantics by means of a particular counting procedure, which handles cycles in a way that ensures that the final count is finite. We formalize this procedure, and some other alternative semantics, and prove theoretical bounds on the computational complexity of the evaluation problem, showing that the bag semantics for property paths is the main reason for the infeasibility of the evaluation of property paths in SPARQL 1.1.

Our theoretical study allowed us to formally prove some extremely large lower bounds for property-path evaluation: for query Cliq-2 and the RDF graph in Figure 1.2, we show that every implementation that strictly adheres to the SPARQL 1.1 specification should provide as output a file of size more than 79 Yottabytes! It should be noticed that some studies estimate that the cumulative capacity of all the digital stores in the world in 2011 is less than 1 Yottabyte (Gantz, 2008)\(^2\).

We study the computational complexity of the setting for property paths proposed by the W3C, as well as for several alternative settings. Given the bag semantics of property paths, we measure the complexity in terms of counting complexity classes. The most studied and used intractable counting class is \(\#P\) (Valiant, 1979a), which is, intuitively, the counting class associated to the NP problems: while the prototypical NP-complete problem is checking if a propositional formula is satisfiable (SAT), the prototypical \(\#P\)-complete problem is counting how many truth assignments satisfy a propositional formula (COUNTSAT). We also make a distinction between data complexity and combined complexity. Data complexity is the complexity of evaluating a query on a database instance assuming that the query is fixed, that is, the complexity is measured only in terms of the size of the database. Combined complexity considers both the query and the database instance as input of the problem (Vardi, 1982). For instance, it has been proved that for the graph pattern fragment of SPARQL 1.0, data complexity

\(^2\)1 Yottabyte (YB) = 1 trillion Terabytes.
is polynomial, while combined complexity is PSPACE-complete (Pérez, Arenas, & Gutierrez, 2009). Since in practice queries tend to be several orders of magnitude smaller than databases, it has been argued that data complexity is the realistic way of analyzing the complexity of query languages (Vardi, 1982).

We prove several complexity results. In particular, one of our main results states that property-path evaluation according to the W3C semantics is \#P-complete in data complexity. Moreover, we prove that the combined complexity of this problem is not even inside \#P. It has been argued that a possibility to deal with the problem of counting paths in the presence of cycles is to consider only simple paths (a simple path is a path with no repeated nodes). We prove that for this alternative semantics the problem is still \#P-complete for data complexity, and remains in \#P for combined complexity. Thus, although the evaluation problems for these two semantics are intractable, the one based on simple paths has lower combined complexity. Notice that all these lower bounds can also be used to probe lower bounds for the whole SPARQL 1.1 language, and thus, we conclude that query evaluation in SPARQL 1.1 is just infeasible.

All our results indicate that evaluating property-path queries according to the official SPARQL 1.1 semantics is essentially infeasible. But not all are bad news. A possible solution to this problem is to not use a semantics that considers duplicates, but instead a more traditional existential semantics for path queries, as it has been done for years in graph databases (Mendelzon & Wood, 1995; Calvanese, Giacomo, Lenzerini, & Vardi, 1999; Barceló, Hurtado, Libkin, & Wood, 2010), in XML (Marx, 2005; Gottlob, Koch, & Pichler, 2005), and even on RDF (Alkhaateeb, Baget, & Euzenat, 2009; Pérez, Arenas, & Gutierrez, 2010) previous to SPARQL 1.1. It is well-known that for this semantics the evaluation problem is tractable, and even linear in data complexity.

As our final theoretical result, we prove that the existential semantics for property paths is equivalent to the SPARQL 1.1 semantics when duplicates are eliminated. Notice that the language has a special feature for this: `SELECT DISTINCT`. Thus, since the existential semantics can be efficiently evaluated, one would expect implementations to take advantage
of the `SELECT DISTINCT` feature. Unfortunately, as our final experiments show, no significant improvement in performance can be observed in the tested implementations when the `SELECT DISTINCT` feature is used. For example, consider the following query:

**Foaf-1D:** `SELECT DISTINCT * WHERE { axel:me (foaf:knows)* ?x`.  

Although the tested implementations spent less time processing some inputs, none of them was able to process an input file of 14.8 KB (Table 1.2). As a comparison, we also tested two implementations of existential paths in SPARQL 1.0: Psparql (Alkhateeb et al., 2009) and Gleen (ZZ6, 2011), which return the same answers as the other tested implementations for the queries using the `SELECT DISTINCT` feature. The numbers speak for themselves (Table 1.2).

**Organization of the paper:** In Chapter 2, we present our experiments with a focus on repeatability. Chapters 3 and 4 present the formalization of SPARQL 1.1 and property paths. Chapter 5 presents our main complexity results. In Chapter 6, we study alternative semantics for counting paths. Chapter 7 introduces the existential semantics and provide some experimental and theoretical results. Finally, we outline in Chapter 8 a proposal for a semantics with tractable query evaluation.

---

**TABLE 1.2. Time in seconds for processing Foaf-1D**

<table>
<thead>
<tr>
<th>Input</th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Kgram</th>
<th>Sesame</th>
<th>Psparql</th>
<th>Gleen</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.2KB</td>
<td>2.24</td>
<td>47.31</td>
<td>2.37</td>
<td>–</td>
<td>0.29</td>
<td>1.39</td>
</tr>
<tr>
<td>10.9KB</td>
<td>2.60</td>
<td>204.95</td>
<td>6.43</td>
<td>–</td>
<td>0.30</td>
<td>1.32</td>
</tr>
<tr>
<td>11.4KB</td>
<td>6.88</td>
<td>3222.47</td>
<td>80.73</td>
<td>–</td>
<td>0.30</td>
<td>1.34</td>
</tr>
<tr>
<td>13.2KB</td>
<td>24.42</td>
<td>–</td>
<td>394.61</td>
<td>–</td>
<td>0.31</td>
<td>1.38</td>
</tr>
<tr>
<td>14.8KB</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.33</td>
<td>1.38</td>
</tr>
<tr>
<td>17.2KB</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.35</td>
<td>1.42</td>
</tr>
<tr>
<td>20.5KB</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.44</td>
<td>1.50</td>
</tr>
<tr>
<td>25.8KB</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.45</td>
<td>1.52</td>
</tr>
</tbody>
</table>
2. EXPERIMENTS

In this chapter, we describe our experimental setting and present more details about the data and results described in the introduction. We assume some familiarity with RDF and the most simple SPARQL features, in particular, with the SELECT and FILTER keywords (Harris & Seaborne, 2011), and we only treat property paths at an intuitive level (we formalize the language in Chapters 3 and 4).

Recall that RDF is a graph data format, composed of triples of the form :s :p :o, with :s and :o representing nodes in the graph, and :p representing a property (edge) relating these two nodes.

As described in the current SPARQL 1.1 specification, “a property path is a possible route through a graph between two graph nodes [...] (and) query evaluation determines all matches of a path expression [...]” (Harris & Seaborne, 2011). More specifically, property-path expressions are regular expressions over properties (edge labels) in the graph.

The most interesting feature that property paths add to the language is the possibility of asking for paths of arbitrary length in a graph. For example, if :p is a property, then (:p) * is a property-path expression that matches pairs of nodes that are connected by a sequence of zero or more :p properties in the graph. The star operator (*) and its derivatives (like the one or more construct) are the only operators that add expressiveness to the language. The official semantics of the other property-path constructors are defined in terms of SPARQL 1.0 operators (Harris & Seaborne, 2011) (and thus, can be simulated in the previous version of the SPARQL standard). Hence, our tests focus on the star operator, and, in particular, on the most simple expressions that can be generated by using this construct.

2.1. Experimental setting

In our tests, we consider the following SPARQL 1.1 implementations:

**ARQ** – version 2.8.8, 21 April 2011 (ZZ1, 2011): ARQ is a java implementation of SPARQL for Jena (Carroll et al., 2004). When testing ARQ, we use the command-line tool `sparql` provided in the standard distribution.
**RDF::Query** – version 2.907, 1 June 2011 (ZZ3, 2011): RDF::Query is a perl module implementing SPARQL 1.1, and we test it with the executable tool query.pl provided with the standard distribution.

**KGRAM** – version 3.0, September 2011 (ZZ2, 2011): KGRAM (Corby & Faron-Zucker, 2010) provides a set of java libraries that implements SPARQL. To test this engine, we implemented a command-line tool kgsparql.java.

**Sesame** – version 2.5.1, 23 September 2011 (ZZ4, 2011): Sesame provides a set of java libraries to execute SPARQL 1.1 queries. To test Sesame, we implemented a command-line tool sesame.java.

We run all our tests in a dedicated machine with the following configuration: Debian 6.0.2 Operating System, Kernel 2.6.32, CPU Intel Xeon X3220 Quadcore with 2.40GHz, and 4GB PC2-5300 RAM. Whenever we run a java program, we set the java virtual machine to be able to use all the available RAM (4 GB). All tests were run considering main memory storage. This should not be considered as a problem since the maximum size of the input RDF graphs that we used was only 25.8 KB. We considered a timeout of 60 minutes. For each test, the number reported is the average of the results obtained by executing the test (at least) 4 times. No experiment showed a significant standard deviation.

### 2.2. The clique experiment

In our first experiment, we considered cliques (complete graphs) of different sizes, from a clique with 2 nodes (containing 2 triples) to a clique with 13 nodes (156 triples). Query Cliq-1 described in the introduction was the first query to be tested. Since this query has no variables, the solution is an *empty tuple*, which, for example, in ARQ is represented by the string `| |`, and in Sesame by the string `[]` (when the query solution is printed to the standard output). RDF::Query does not support queries without variables, thus for this implementation we tested the following query:

**CliqF-1:** `SELECT * WHERE :a0 (:p)* ?x FILTER (?x = :a1)`.
Table 2.1. Time in seconds and number of solutions for query Cliq-1 (CliqF-1 for RDF::Query)

<table>
<thead>
<tr>
<th>n</th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Kgram</th>
<th>Sesame</th>
<th>Solutions</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1.18</td>
<td>0.90</td>
<td>0.57</td>
<td>0.76</td>
<td>16</td>
</tr>
<tr>
<td>6</td>
<td>1.19</td>
<td>1.44</td>
<td>0.60</td>
<td>1.24</td>
<td>65</td>
</tr>
<tr>
<td>7</td>
<td>1.37</td>
<td>5.09</td>
<td>0.95</td>
<td>2.36</td>
<td>326</td>
</tr>
<tr>
<td>8</td>
<td>1.73</td>
<td>34.01</td>
<td>1.38</td>
<td>9.09</td>
<td>1,957</td>
</tr>
<tr>
<td>9</td>
<td>2.31</td>
<td>295.88</td>
<td>5.38</td>
<td>165.28</td>
<td>13,700</td>
</tr>
<tr>
<td>10</td>
<td>4.15</td>
<td>2899.41</td>
<td>228.68</td>
<td>–</td>
<td>109,601</td>
</tr>
<tr>
<td>11</td>
<td>31.21</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>986,410</td>
</tr>
<tr>
<td>12</td>
<td>1422.30</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>9,864,101</td>
</tr>
<tr>
<td>13</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 2.2. Time in seconds and number of solutions for queries Cliq-2 (left) and Cliq-3 (right)

<table>
<thead>
<tr>
<th>n</th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Sol.</th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Sol.</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.40</td>
<td>0.76</td>
<td>1</td>
<td>2.00</td>
<td>0.77</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1.19</td>
<td>0.84</td>
<td>6</td>
<td>1.42</td>
<td>6.85</td>
<td>42</td>
</tr>
<tr>
<td>4</td>
<td>1.65</td>
<td>19.38</td>
<td>305</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>5</td>
<td>97.06</td>
<td>–</td>
<td>418,576</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

Table 2.1 shows the result obtained for this experiment in terms of the time (in seconds) and the number of solutions produced as output, when the input is a clique with $n$ nodes. The symbol “–” in the table means timeout of one hour. Notice that this table contains a tabular representation of the numbers shown in Figure 1.1.

We also tested the impact of using nested stars. In particular, we consider query Cliq-2 described in the introduction and query

\[
\text{Cliq-3: \ SELECT * WHERE \ { :a0 (((:p)*)*) :a1 } }
\]

For these expressions containing nested stars, Sesame produces a run-time error (we have reported this bug in the Sesame’s mailing list), and KGRAM does not produce the expected output according to the official SPARQL 1.1 specification (Harris & Seaborne, 2011). Thus, for these cases it is only meaningful to test ARQ and RDF::Query (we use FILTER for RDF::Query, as we did for the case of query CliqF-1). The results are shown in Table 2.2.

As described in the introduction, our results show the infeasibility of evaluating property paths including the star operator in the the four tested implementations. We emphasize only
here the unexpected impact of nesting stars: for query Cliq-3 both implementations that we tested fail for an RDF graph representing a clique with only 4 nodes, which contains only 12 triples and has a size of 126 bytes in N3 notation. Although in this example the nesting of the star operator does not seem to be natural, it is well known that nesting is indeed necessary to represent some regular languages (Eggan, 1963). It is also notable how the number of solutions increase w.r.t. the input size. For instance, for query Cliq-1, ARQ returns more than 9 million solutions for a clique with 12 nodes (ARQ’s output in this case has more than 9 million lines containing the string | |).

2.3. The foaf experiment

For our second experiment, we use real data crawled from the Web. We decided to consider the foaf:knows property, as it has been used as a paradigmatic property for examples regarding path queries (notice that it is in all the examples used to describe property paths in the official SPARQL 1.1 specification (Harris & Seaborne, 2011)).

To construct our datasets we use the SemWeb Client Library (ZZ7, 2011), which provides a command-line tool semwebquery that can be used to query the Web of Linked Data. The tool receives as input a SPARQL query Q, an integer value k and a URI u. When executed, it first retrieves the data from u, evaluates Q over this data, and follows the URIs mentioned in it to obtain more data. This process is repeated k times (see (Hartig, Bizer, & Freytag, 2009) for a description of this query approach). We use a CONSTRUCT query to retrieve URIs linked by foaf:knows properties with Axel Polleres’ foaf document as the starting URI. We set the parameter k as 3, which already gave us a file of 1.5MB containing more than 33,000 triples. To obtain a file of reasonable size, we first filtered the data by removing all triples that mention URIs from large Social Networks sites (in particular, we remove URIs from MyOpera.com and SemanticTweet.com), and then we extracted the strongly connected component to which Axel Polleres’ URI belongs, obtaining a file of 25.8 KB. From this file, we constructed several test cases by deleting subsets of nodes and then recomputing the strongly connected component. With this process we constructed 8 different test cases from 9.2 KB to 25.8 KB. The description of these files is shown in Table 2.3. Just as an example of the construction process, file D is...
Table 2.3. Description of the files (name, number of nodes, number of RDF triples, and size in disk) used in the foaf experiment.

<table>
<thead>
<tr>
<th>File</th>
<th>#nodes</th>
<th>#triples</th>
<th>size (N3 format)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>38</td>
<td>119</td>
<td>9.2KB</td>
</tr>
<tr>
<td>B</td>
<td>43</td>
<td>143</td>
<td>10.9KB</td>
</tr>
<tr>
<td>C</td>
<td>47</td>
<td>150</td>
<td>11.4KB</td>
</tr>
<tr>
<td>D</td>
<td>52</td>
<td>176</td>
<td>13.2KB</td>
</tr>
<tr>
<td>E</td>
<td>54</td>
<td>201</td>
<td>14.8KB</td>
</tr>
<tr>
<td>F</td>
<td>57</td>
<td>237</td>
<td>17.2KB</td>
</tr>
<tr>
<td>G</td>
<td>68</td>
<td>281</td>
<td>20.5KB</td>
</tr>
<tr>
<td>H</td>
<td>76</td>
<td>360</td>
<td>25.8KB</td>
</tr>
</tbody>
</table>

Table 2.4. Time in seconds, number of solutions, and output size for query Foaf-1

<table>
<thead>
<tr>
<th>File</th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Kgram</th>
<th>Sesame</th>
<th>Solutions</th>
<th>Size (ARQ)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>5.13</td>
<td>75.70</td>
<td>313.37</td>
<td>–</td>
<td>29,817</td>
<td>2MB</td>
</tr>
<tr>
<td>B</td>
<td>8.20</td>
<td>325.83</td>
<td>–</td>
<td>–</td>
<td>122,631</td>
<td>8.4MB</td>
</tr>
<tr>
<td>C</td>
<td>65.87</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1,739,331</td>
<td>120MB</td>
</tr>
<tr>
<td>D</td>
<td>292.43</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>8,511,943</td>
<td>587MB</td>
</tr>
<tr>
<td>E</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>

constructed from file E by deleting the node corresponding to Richard Cyganiack’s URI, and then computing the strongly connected component to which Axel’s URI belong.

We tested query Foaf-1 described in the introduction, which asks for the network of friends of Axel Polleres. Since the graphs in our test cases are strongly connected, this query retrieves all the nodes in the graph (possibly with duplicates). The time to process the query, the number of solutions produced, and the size of the output produced by ARQ are shown in Table 2.4 (file E is the last file shown in the table, as all implementations exceed the timeout limit for the larger files). As for the case of the clique experiment, one of the most notable phenomenon is the large increase in the output size.

In the following chapters, we provide theoretical results that explain the behavior showed by our tests. We begin by formalizing the SPARQL language and the official semantics of property paths.
3. FORMALIZING SPARQL

In the following chapters, we formalize the semantics of property paths proposed by the W3C (Harris & Seaborne, 2011), and then study the complexity of evaluating property paths under such semantics. To this end, we present in this chapter an algebraic formalization of the core operators in SPARQL 1.1, which follows the approach given in (Pérez, Arenas, & Gutierrez, 2006; Pérez et al., 2009). Start by assuming there are pairwise disjoint infinite sets $I$ (IRIs), $B$ (blank nodes) and $L$ (literals). A tuple $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$ is called an RDF triple, where $s$ is the subject, $p$ is the predicate and $o$ is the object. A finite set of RDF triples is called an RDF graph. Moreover, assume the existence of an infinite set $V$ of variables disjoint from the above sets, and assume that every element in $V$ starts with the symbol ?.

A SPARQL 1.1 graph pattern expression is defined recursively as follows: (1) A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a triple pattern); (2) if $P_1$ and $P_2$ are graph patterns, then $(P_1 \text{ AND } P_2)$, $(P_1 \text{ OPT } P_2)$, $(P_1 \text{ UNION } P_2)$ and $(P_1 \text{ MINUS } P_2)$ are graph patterns; and (3) if $P$ is a graph pattern and $R$ is a SPARQL 1.1 built-in condition, then the expression $(P \text{ FILTER } R)$ is a graph pattern. In turn, a SPARQL 1.1 built-in condition is constructed using elements of the set $(I \cup V)$, equality, logical connectives and some built-in predicates (Prud’hommeaux & Seaborne, 2008). In this paper, we restrict to the fragment where a built-in condition is a Boolean combination of terms constructed by using $=$ and predicate bound, that is, (1) if $?X, ?Y \in V$ and $c \in I$, then $\text{bound}(?X)$, $?X = c$ and $?X \neq ?Y$ are built-in conditions; and (2) if $R_1$ and $R_2$ are built-in conditions, then $\neg R_1$, $(R_1 \lor R_2)$ and $(R_1 \land R_2)$ are built-in conditions. Finally, if $P$ is a graph pattern and $W$ is a set of variables, then $(\text{SELECT } W \ P)$, $(\text{SELECT DISTINCT } W \ P)$, $(\text{SELECT } * \ P)$ and $(\text{SELECT DISTINCT } * \ P)$, are queries in SPARQL 1.1.

To define the semantics of SPARQL 1.1 queries, we borrow some terminology from (Prud’hommeaux & Seaborne, 2008; Pérez et al., 2009). A mapping $\mu$ is a partial function $\mu : V \rightarrow (I \cup L)$. Abusing notation, given $a \in (I \cup L)$ and a mapping $\mu$, we assume that $\mu(a) = a$, and for a triple pattern $t = (s, p, o)$, we assume that $\mu(t) = (\mu(s), \mu(p), \mu(o))$. 
The domain of $\mu$, denoted by $\text{dom}(\mu)$, is the subset of $V$ where $\mu$ is defined. Two mappings $\mu_1$ and $\mu_2$ are compatible, denoted by $\mu_1 \sim \mu_2$, when for all $x \in \text{dom}(\mu_1) \cap \text{dom}(\mu_2)$, it is the case that $\mu_1(x) = \mu_2(x)$, i.e. when $\mu_1 \cup \mu_2$ is also a mapping. The mapping with empty domain is denoted by $\mu_\emptyset$ (notice that this mapping is compatible with any other mapping). Finally, given a mapping $\mu$ and a set $W$ of variables, the restriction of $\mu$ to $W$, denoted by $\mu|_W$, is a mapping such that $\text{dom}(\mu|_W) = \text{dom}(\mu) \cap W$ and $\mu|_W(\cdot X) = \mu(\cdot X)$ for every $\cdot X \in \text{dom}(\mu) \cap W$. Notice that if $W = \emptyset$, then $\mu|_W = \mu_\emptyset$.

The semantics of a SPARQL 1.1 query is defined as a bag (or multiset) of mappings (Harris & Seaborne, 2011), which is a set of mappings in which every element $\mu$ is annotated with a positive integer that represents the cardinality of $\mu$ in the bag. Formally, we represent a bag of mappings as a pair $(\Omega, \text{card}_\Omega)$, where $\Omega$ is a set of mappings and $\text{card}_\Omega$ is a function such that $\text{card}_\Omega(\mu)$ is the cardinality of $\mu$ in $\Omega$ (we assume that $\text{card}_\Omega(\mu) > 0$ for every $\mu \in \Omega$, and $\text{card}_\Omega(\mu') = 0$ for every $\mu' \not\in \Omega$). With this notion, we have the necessary ingredients to define the semantics of SPARQL 1.1 queries. As in (Pérez et al., 2006; Pérez et al., 2009), this semantics is defined as a function $\llbracket \cdot \rrbracket_G$ that takes a query and returns a bag of mappings. More precisely, the evaluation of a graph pattern $P$ over an RDF graph $G$, denoted by $\llbracket P \rrbracket_G$, is defined recursively as follows (for the sake of readability, the semantics of filter expressions is presented separately).

- If $P$ is a triple pattern $t$, then $\llbracket P \rrbracket_G = \{ \mu \mid \text{dom}(\mu) = \text{var}(t) \text{ and } \mu(t) \in G \}$, where $\text{var}(t)$ is the set of variables mentioned in $t$. Moreover, for every $\mu \in \llbracket P \rrbracket_G$: $\text{card}_{\llbracket P \rrbracket_G}(\mu) = 1$.
- If $P$ is $(P_1 \text{ AND } P_2)$, then $\llbracket P \rrbracket_G = \{ \mu_1 \cup \mu_2 \mid \mu_1 \in \llbracket P_1 \rrbracket_G, \mu_2 \in \llbracket P_2 \rrbracket_G \text{ and } \mu_1 \sim \mu_2 \}$. Moreover, for every $\mu \in \llbracket P \rrbracket_G$ we have that $\text{card}_{\llbracket P \rrbracket_G}(\mu)$ is given by the expression:

$$\sum_{\mu_1 \in \llbracket P_1 \rrbracket_G} \left[ \sum_{\mu_2 \in \llbracket P_2 \rrbracket_G} \left( \text{card}_{\llbracket P_1 \rrbracket_G}(\mu_1) \cdot \text{card}_{\llbracket P_2 \rrbracket_G}(\mu_2) \right) \right].$$

- If $P$ is $(P_1 \text{ OPT } P_2)$, then $\llbracket P \rrbracket_G = \llbracket (P_1 \text{ AND } P_2) \rrbracket_G \cup \{ \mu \in \llbracket P_1 \rrbracket_G \mid \forall \mu' \in \llbracket P_2 \rrbracket_G : \mu \not\sim \mu' \}$. Moreover, for every $\mu \in \llbracket P \rrbracket_G$, if $\mu \in \llbracket (P_1 \text{ AND } P_2) \rrbracket_G$, then $\text{card}_{\llbracket P \rrbracket_G}(\mu) = \text{card}_{\llbracket (P_1 \text{ AND } P_2) \rrbracket_G}(\mu)$, and if $\mu \not\in \llbracket (P_1 \text{ AND } P_2) \rrbracket_G$, then $\text{card}_{\llbracket P \rrbracket_G}(\mu) = \text{card}_{\llbracket P_1 \rrbracket_G}(\mu)$. 

13
• If \( P \) is \( (P_1 \text{ UNION } P_2) \), then \( \llbracket P \rrbracket_G = \{ \mu \mid \mu \in \llbracket P_1 \rrbracket_G \text{ or } \mu \in \llbracket P_2 \rrbracket_G \} \). Moreover, for every \( \mu \in \llbracket P \rrbracket_G \):

\[
\text{card}_{\llbracket P \rrbracket_G}(\mu) = \text{card}_{\llbracket P_1 \rrbracket_G}(\mu) + \text{card}_{\llbracket P_2 \rrbracket_G}(\mu).
\]

• If \( P \) is \( (P_1 \text{ MINUS } P_2) \), then \( \llbracket P \rrbracket_G = \{ \mu \in \llbracket P_1 \rrbracket_G \mid \forall \mu' \in \llbracket P_2 \rrbracket_G : \mu \not\sim \mu' \text{ or } \text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset \} \). Moreover, for every \( \mu \in \llbracket P \rrbracket_G \), it holds that \( \text{card}_{\llbracket P \rrbracket_G}(\mu) = \text{card}_{\llbracket P_1 \rrbracket_G}(\mu) \).

The evaluation of a SPARQL 1.1 query \( Q \) over an RDF graph \( G \), denoted by \( \llbracket Q \rrbracket_G \), is defined as follows. If \( Q \) is SPARQL 1.1 query (SELECT \( W \) \( P \)), then \( \llbracket Q \rrbracket_G = \{ \mu|_W \mid \mu \in \llbracket P \rrbracket_G \} \) and for every \( \mu \in \llbracket Q \rrbracket_G \):

\[
\text{card}_{\llbracket Q \rrbracket_G}(\mu) = \sum_{\mu' \in \llbracket P \rrbracket_G : \mu = \mu'|_W} \text{card}_{\llbracket P \rrbracket_G}(\mu').
\]

If \( Q \) is SPARQL 1.1 query (SELECT \( * \) \( P \)), then \( \llbracket Q \rrbracket_G = \llbracket P \rrbracket_G \) and \( \text{card}_{\llbracket Q \rrbracket_G}(\mu) = \text{card}_{\llbracket P \rrbracket_G}(\mu) \) for every \( \mu \in \llbracket Q \rrbracket_G \). If \( Q \) is SPARQL 1.1 query (SELECT DISTINCT \( W \) \( P \)), then \( \llbracket Q \rrbracket_G = \{ \mu|_W \mid \mu \in \llbracket P \rrbracket_G \} \) and for every \( \mu \in \llbracket Q \rrbracket_G \), we have that \( \text{card}_{\llbracket Q \rrbracket_G}(\mu) = 1 \).

Finally, if \( Q \) is SPARQL 1.1 query (SELECT DISTINCT \( * \) \( P \)), then \( \llbracket Q \rrbracket_G = \llbracket P \rrbracket_G \) and for every \( \mu \in \llbracket Q \rrbracket_G \), we have that \( \text{card}_{\llbracket Q \rrbracket_G}(\mu) = 1 \).

To conclude the definition of the semantics of SPARQL 1.1, we need to define the semantics of filter expressions. Given a mapping \( \mu \) and a built-in condition \( R \), we say that \( \mu \) satisfies \( R \), denoted by \( \mu \models R \), if (omitting the usual rules for Boolean connectives): (1) \( R \) is bound(?X) and ?X \( \in \text{dom}(\mu) \); (2) \( R \) is ?X = c, ?X \( \in \text{dom}(\mu) \) and \( \mu(?X) = c \); (3) \( R \) is ?X =?Y, ?X \( \in \text{dom}(\mu) \), ?Y \( \in \text{dom}(\mu) \) and \( \mu(?X) = \mu(?Y) \). Then given an RDF graph \( G \) and a graph pattern expression \( P = (P_1 \text{ FILTER } R) \), we have that \( \llbracket P \rrbracket_G = \{ \mu \in \llbracket P_1 \rrbracket_G \mid \mu \models R \} \), and for every \( \mu \in \llbracket P \rrbracket_G \), we have that \( \text{card}_{\llbracket P \rrbracket_G}(\mu) = \text{card}_{\llbracket P_1 \rrbracket_G}(\mu) \).
4. PROPERTY PATHS

In this chapter, we use the framework presented in the previous chapter to formalize the semantics of property paths in SPARQL 1.1. We start by presenting the syntax of property paths. More specifically, according to (Harris & Seaborne, 2011), a property path is recursively defined as follows: (1) if \( a \in I \), then \( a \) is a property path, and (2) if \( p_1 \) and \( p_2 \) are property paths, then \( p_1\mid p_2 \), \( p_1/p_2 \) and \( p_1^* \) are property paths. Thus, from a syntactical point of view, property paths are regular expressions over the vocabulary \( I \), being \( \mid \) disjunction, \( / \) concatenation and \( ( )^* \) the Kleene star. It should be noticed that the definition of property paths in (Harris & Seaborne, 2011) includes some additional features that are common in regular expressions, such as \( p? \) (zero or one occurrences of \( p \)) and \( p^+ \) (one or more occurrences of \( p \)). In this paper, we focus on the core operators \( \mid \), \( / \) and \( ( )^* \), as they suffice to prove the infeasibility of the evaluation of property paths in SPARQL 1.1.

A property-path triple is a tuple \( t \) of the form \( (u, p, v) \), where \( u, v \in (I \cup V) \) and \( p \) is a property path. SPARQL 1.1 includes as atomic formulas triple patterns and property-path triples. Thus, to complete the definition of the semantics of SPARQL 1.1, we need to specify how property-path triples are evaluated over RDF graphs, that is, we need to extend the definition of the function \( \llbracket \cdot \rrbracket_G \) to include property-path triples.

To define the semantics of property-path triples we follow closely the standard specification (Harris & Seaborne, 2011). Assume that \( u, v \in (I \cup V) \), \( W = (\{u, v\} \cap V) \) and \( p \) is a property path. Notice that if \( u, v \in I \), then \( W = \emptyset \). Then the evaluation of property-path triple \( t = (u, p, v) \) over an RDF graph \( G \), denoted by \( \llbracket t \rrbracket_G \), is defined recursively as follows. If \( p = a \), where \( a \in I \), then \( (u, p, v) \) is a triple pattern and \( \llbracket t \rrbracket_G \) is defined as in Chapter 3. Otherwise, we have that either \( p = p_1\mid p_2 \) or \( p = p_1/p_2 \) or \( p = p_1^* \), where \( p_1, p_2 \) are property paths, and \( \llbracket t \rrbracket_G \) is defined as follows. First, if \( p = p_1\mid p_2 \), then \( \llbracket t \rrbracket_G \) is defined in (Harris & Seaborne, 2011) as the result of evaluating the pattern \( ((u, p_1, v) \cup (u, p_2, v)) \) over \( G \). Thus, we have that:

\[
\llbracket t \rrbracket_G = \{ \mu \mid \mu \in \llbracket (u, p_1, v) \rrbracket_G \text{ or } \mu \in \llbracket (u, p_2, v) \rrbracket_G \},
\]
and for every $\mu \in \llbracket t \rrbracket_G$, we have that:

$$\text{card}(\llbracket t \rrbracket_G(\mu)) = \text{card}(\llbracket (u,p_1,v) \rrbracket_G(\mu)) + \text{card}(\llbracket (u,p_2,v) \rrbracket_G(\mu)).$$

Second, if $p = p_1/p_2$, then assuming that $?X$ is a variable such that $?X \not\in W$, we have that $\llbracket t \rrbracket_G$ is defined in (Harris & Seaborne, 2011) as the result of first evaluating the pattern $((u, p_1, ?X) \text{ AND } (?X, p_2, v))$ over $G$, and then projecting over the variables of property-path triple $t$ (and, thus, projecting out the variable $?X$). Thus, we have that:

$$\llbracket t \rrbracket_G \ = \ \{(\mu_1 \cup \mu_2)_{|W} \mid \mu_1 \in \llbracket (u,p_1,?X) \rrbracket_G, \mu_2 \in \llbracket (?X,p_2,v) \rrbracket_G \text{ and } \mu_1 \sim \mu_2\},$$

and for every $\mu \in \llbracket t \rrbracket_G$, we have that:

$$\text{card}(\llbracket t \rrbracket_G(\mu)) = \sum_{\mu_1 \in \llbracket (u,p_1,?X) \rrbracket_G} \left( \sum_{\mu_2 \in \llbracket (?X,p_2,v) \rrbracket_G} \text{card}(\llbracket (u,p_1,?X) \rrbracket_G(\mu_1)) \cdot \text{card}(\llbracket (?X,p_2,v) \rrbracket_G(\mu_2)) \right).$$

Finally, if $p = p^*_1$, then $\llbracket t \rrbracket_G$ is defined in (Harris & Seaborne, 2011) in terms of the procedures COUNT and ALP shown in Figure 4.1. More precisely,

$$\llbracket t \rrbracket_G \ = \ \{\mu \mid \text{dom}(\mu) = W \text{ and COUNT}(\mu(u), p_1, \mu(v), G) > 0\}.$$ 

Moreover, for every $\mu \in \llbracket t \rrbracket_G$, it holds that

$$\text{card}(\llbracket t \rrbracket_G(\mu)) \ = \ \text{COUNT}(\mu(u), p_1, \mu(v), G).$$

Procedure ALP in Figure 4.1 is taken from (Harris & Seaborne, 2011). It is important to notice that lines 5 and 6 in ALP formalize, in our terminology, the use of a procedure call eval in the definition of ALP in (Harris & Seaborne, 2011). According to (Harris & Seaborne, 2011), procedure ALP has to be used as follows to compute $\text{card}(\llbracket t \rrbracket_G(\mu))$, where $t = (u, p^*_1, v)$. Assuming that Result is the empty list and Visited is the empty set, first one has to invoke $\text{ALP}(\mu(u), p, \text{Result}, \text{Visited}, G)$, then one has to check whether $\mu(v)$ appears in the resulting list Result, and if this is the case then $\text{card}(\llbracket t \rrbracket_G(\mu))$ is set as the number of occurrences of $\mu(v)$ in the list Result. For the sake of readability, we have encapsulated in the auxiliary procedure COUNT these steps to compute $\text{card}(\llbracket t \rrbracket_G(\mu))$ from procedure ALP, and we have defined $\llbracket t \rrbracket_G$ by
**Function** \( \text{COUNT}(a, \text{path}, b, G) \)

**Input:** \( a, b \in I \), \( \text{path} \) is a property path and \( G \) is an RDF graph.

1: \( \text{Result} := \) empty list
2: \( \text{Visited} := \) empty set
3: \( \text{ALP}(a, \text{path}, \text{Result}, \text{Visited}, G) \)
4: \( n := \) number of occurrences of \( b \) in \( \text{Result} \)
5: return \( n \)

**Procedure** \( \text{ALP}(a, \text{path}, \text{Result}, \text{Visited}, G) \)

**Input:** \( a \in I \), \( \text{path} \) is a property path, \( \text{Result} \) is a list of elements from \( I \), \( \text{Visited} \) is a set of elements from \( I \) and \( G \) is an RDF graph.

1: \( \text{if} \ a \in \text{Visited} \text{ then} \)
2: \( \text{return} \)
3: \( \text{end if} \)
4: add \( a \) to \( \text{Visited} \), and add \( a \) to \( \text{Result} \)
5: \( \Omega := \llbracket (a, \text{path}, \?X) \rrbracket_G \)
6: let \( \text{Next} \) be the list of elements \( b = \mu(\?X) \) for \( \mu \in \Omega \), such that the number of occurrences of \( b \) in \( \text{Next} \) is \( \text{card}_{\Omega}(\mu) \)
7: for each \( c \in \text{Next} \) do
8: \( \text{ALP}(c, \text{path}, \text{Result}, \text{Visited}, G) \)
9: end for
10: remove \( a \) from \( \text{Visited} \)

**Figure 4.1.** Procedures used in the evaluation of property-path triples of the form \((u, \text{path}^*, v)\).

using \( \text{COUNT} \), thus formalizing the semantics proposed by the W3C in (Harris & Seaborne, 2011).

The idea behind algorithm \( \text{ALP} \) is to incrementally construct paths that conform to a property path of the form \( p_1^* \), that is, to construct sequences of nodes \( a_1, a_2, \ldots, a_n \) from an RDF graph \( G \) such that each node \( a_{i+1} \) is reachable from \( a_i \) in \( G \) by following the path \( p_1 \), but with the important feature (implemented through the use of the set \( \text{Visited} \)) that each node \( a_i \) is distinct from all the previous nodes \( a_j \) selected in the sequence (thus avoiding cycles in the sequence \( a_1, a_2, \ldots, a_n \)).
5. INTRACTABILITY OF SPARQL 1.1 IN THE PRESENCE OF PROPERTY PATHS

In this chapter, we study the complexity of evaluating property paths according to the semantics proposed by the W3C. Specifically, we study the complexity of computing $\text{card}_{[[t]_G]}(\cdot)$, as this computation embodies the main task needed to evaluate a property-path triple. For the sake of readability, we focus here on computing such functions for property-path triples of the form $(a, p, b)$ where $a, b \in I$. It is important to notice that this is not a restriction, as for every property path triple $t$ and every mapping $\mu$ whose domain is equal to the set of variable mentioned in $t$, it holds that $\text{card}_{[[t]_G]}(\mu) = \text{card}_{[[\mu(t)]_G]}(\mu_\emptyset)$ (recall that $\mu_\emptyset$ is the mapping with empty domain). Thus, we study the counting problem $\text{COUNTW3C}$, whose input is an RDF graph $G$, elements $a, b \in I$ and a property path $p$, and whose output is the value $\text{card}_{[[a,p,b]_G]}(\mu_\emptyset)$.

It is important to notice that property paths are part of the input of the previous problem and, thus, we are formalizing the combined complexity of the evaluation problem (Vardi, 1982). As it has been observed in many scenarios, and, in particular, in the context of evaluating SPARQL (Pérez et al., 2009), when computing a function like $\text{card}_{[[a,p,b]_G]}(\cdot)$, it is natural to assume that the size of $p$ is considerably smaller than the size of $G$. This assumption is very common when studying the complexity of a query language. In fact, it is named data complexity in the database literature (Vardi, 1982), and it is defined in our context as the complexity of computing $\text{card}_{[[a,p,b]_G]}(\cdot)$ for a fixed property-path $p$. More precisely, assume given a fixed property path $p$. Then $\text{COUNTW3C}(p)$ is defined as the problem of computing, given an RDF graph $G$ and elements $a, b \in I$, the value $\text{card}_{[[a,p,b]_G]}(\mu_\emptyset)$.

5.1. Auxiliary terminology

The main technique used in the proofs of the results of Chapter 5 is the analysis of the tree of recursive calls generated by invoking procedure $\text{COUNT}$. In this section, we explain by means of an example the terminology used when analyzing this tree. Let $G$ be the following RDF graph:

Assume that we want to compute $\text{card}_{[[a,(t|r),c]_G]}(\mu_\emptyset)$. According to the semantics defined in Chapter 4, this value is equal to the result of the call $\text{COUNT}(a, (t|r), c, G)$, which in turn
is computed by invoking $\text{ALP}(a, (t|r), \text{Result, Visited}, G)$, where $\text{Result}$ is an empty list and $\text{Visited}$ is an empty set, and then returning the number of occurrences of $c$ in $\text{Result}$. The tree of recursive calls generated by invoking $\text{ALP}(a, (t|r), \text{Result, Visited}, G)$ is shown in Figure 5.2.

As shown in Figure 5.2, the initial call $\text{ALP}(a, (t|r), \text{Result, Visited}, G)$ first adds $a$ to the set $\text{Visited}$ and to the list $\text{Result}$ (line 4 of procedure $\text{ALP}$) to indicate that $a$ has been visited, then computes in line 6 a list $\text{Next}$ containing all the elements $d$ for which there exists a mapping $\mu \in \llbracket (a, (t|r), ?X) \rrbracket_G$ such that $\mu(?X) = d$, with the additional condition that the number of occurrences of $d$ in $\text{Next}$ is equal to $\text{card}_{\llbracket (a, (t|r), ?X) \rrbracket_G}(\mu)$, and then invokes $\text{ALP}(d, (t|r), \text{Result, Visited}, G)$ in line 8 for each such element $d$. In this case, we have that $\text{Next} = [c, b, b]$ (the order of the list is not relevant for the final result), so procedure $\text{ALP}$ is invoked three times. The first such call is $\text{ALP}(c, (t|r), \text{Result, Visited}, G)$, which is marked with the number 2 in Figure 5.2. This call adds $c$ to the set $\text{Visited}$ and to the list $\text{Result}$ (line 4) to indicate that $c$ has been visited, then computes in line 6 a list $\text{Next}$ containing all the elements $d$ for which there exists a mapping $\mu \in \llbracket (c, (t|r), ?X) \rrbracket_G$ such that $\mu(?X) = d$, with the additional condition that the number of occurrences of $d$ in $\text{Next}$ is equal to $\text{card}_{\llbracket (c, (t|r), ?X) \rrbracket_G}(\mu)$, and then invokes $\text{ALP}(d, (t|r), \text{Result, Visited}, G)$ in line 8 for each such element $d$. Given that $\text{Next} = []$ in this case, procedure $\text{ALP}$ is not invoked and $c$ is removed from the set $\text{Visited}$ (line 10).

The third call in the tree is $\text{ALP}(b, (t|r), \text{Result, Visited}, G)$, where $\text{Visited} = \{a\}$ and $\text{Result} = [a, c]$, since $c$ was included in $\text{Result}$ in the call with number 2.

As in the previous two cases, $\text{ALP}(b, (t|r), \text{Result, Visited}, G)$ includes $b$ in the set $\text{Visited}$ and in the list $\text{Result}$, and then computes the list $\text{Next}$, which is equal to $[a, c, c]$ in this case.
Thus, the fourth call in the tree of recursive calls is $\text{ALP}(a,(t|r),\text{Result, Visited, }G)$, where
Figure 5.3. Simplified view of the tree shown in Figure 5.2

Visit\(ed = \{a, c\}\) and Result \(= [a, c, b]\). But given that \(a \in Visit\(ed\) in this case, this call immediately returns modifying neither Visit\(ed\) nor Result, as the condition in line 1 is satisfied (this is mark with a symbol \(\times\) in Figure 5.2). The remaining calls in the tree of recursive calls are shown in Figure 5.2.

In the proofs of Theorem 5.1 and Lemma 5.3, we analyze the tree of recursive calls generated by invoking procedure Count, which in turn invokes procedure ALP. In our analysis, we use a simplified view of the tree of recursive calls shown in Figure 5.2, where we just use as labels of the nodes in the tree the URIs that are used in the calls to the procedure ALP. For example, the following is the simplified view of the tree shown in Figure 5.2:

Notice that in this figure two nodes are marked with the symbol \(\times\), which correspond with the recursive calls marked with the same symbol in Figure 5.2. These nodes have the same label as the root node and, thus, when they are reached, their label is already in the set Visit\(ed\), which implies that nothing will be added to the set Visit\(ed\) and to the list Result in these calls.

It is important to notice that in some of the following proofs, we also use a modified version of the simplified view just presented, where we use numbers to indicate the number of occurrences of some nodes with the same label at a particular depth in the tree. For example, the following is an alternative representation of the tree shown in Figure 5.2:

5.2. Property path evaluation

To pinpoint the complexity of \textsc{CountW3C} and \textsc{CountW3C}(p), where \(p\) is a property path, we need to consider the complexity class \#P mentioned in the introduction (we refer the reader
to (Valiant, 1979a) for its formal definition). A function $f$ is said to be in $\#P$ if there exists a non-deterministic Turing Machine $M$ that works in polynomial time such that for every string $w$, the value of $f$ on $w$ is equal to the number of accepting runs of $M$ with input $w$. As mentioned in the introduction, a prototypical $\#P$-complete problem is the problem of computing, given a propositional formula $\varphi$, the number of truth assignments satisfying $\varphi$. Clearly $\#P$ is a class of intractable computation problems (Valiant, 1979a).

The used notion of reduction for the class $\#P$ is based on oracles (Valiant, 1979a). More precisely, the class of functions that can be computed in polynomial time is denoted by $\text{FP}$, and $\text{FP}^L$ is used to denote the class of functions that can be computed in polynomial time with access to an oracle (or sub-routine) for the counting problem $L$, assuming that $L$ can be computed in unit time. A function $f$ is $\#P$-complete iff $\#P \subseteq \text{FP}^f$ and $f \in \#P$. Notice that this notion of reduction implies that if $\#P$ can be computed in polynomial time then $y$ can be computed in polynomial time too. The $\#P$-complete problems showed in (Valiant, 1979b), follow that notion of reduction; we take some of these problems as a base for the $\#P$-completness proofs included in this chapter.

Our first result shows that property path evaluation is intractable.

**Theorem 5.1.** \textsc{CountW3C}(p) is in \#P for every property path $p$. Besides, \textsc{CountW3C}(c*) is \#P-complete, where $c \in I$.

**Proof. Membership:** Let $p$ be a fixed property path. To prove that \textsc{CountW3C}(p) $\in \#P$, we give a non-deterministic polynomial-time algorithm $C_{Sp}$ that receives as input an RDF graph
$G$ and elements $a, b \in I$, and such that the number of valid executions of $CS_p$ with this input corresponds to $\text{card}[_{(a,p,b)}]_G(\mu_\emptyset)$.

To define algorithm $CS_p$, we define for every property path $r$, a non-deterministic algorithm $\text{GUESS}_r$ that, given an RDF graph $G$ and an element $a \in I$, guesses an element $b \in I$ such that:

$$b \in \{c \in I \mid \exists \mu \in \llbracket (a, r, ?X) \rrbracket_G : \mu(?X) = c\}.$$

Let $\text{coin}()$ be a non-deterministic function that returns $\text{true}$ or $\text{false}$, and $\text{stop}()$ be a procedure that terminates the execution of $\text{GUESS}_r$ in a non-accepting state. Then $\text{GUESS}_r$ is defined by considering the structure of $r$: (1) $r \in I$, (2) the main operator of $r$ is union, (3) the main operator of $r$ is concatenation, or (4) the main operator of $r$ is the Kleene star. In what follows, we present separately each of these cases, analyzing in each one of them the time complexity of the algorithm $\text{GUESS}_r$ (which is denoted by $t_r$). It is important to notice that given a property path $r$, finding the main operator of $r$ can be done in polynomial time in the size of $r$ (in fact, this step can be done in constant time for the procedure $\text{GUESS}_r$ as $r$ is fixed).

(1) Assume that $c \in I$. Procedure $\text{GUESS}_c$ is defined as follows:

\[
\text{GUESS}_c(a, G) \\
1: \text{for each } (a, c, b) \in G \text{ do} \\
2: \quad \text{if } \text{coin}() \text{ then} \\
3: \quad \quad \text{return } b \\
4: \quad \text{end if} \\
5: \quad \text{end for} \\
6: \text{stop}()
\]

Notice that if no node is selected by the previous procedure, then the execution of the algorithm is not valid and, thus, this execution is terminated by invoking the procedure $\text{stop}()$ in line 6. The time complexity of the algorithm in this case is linear in the size of $G$, as the for loop (line 1) has to iterate over all the triples in $G$ until a node is selected. Thus, we have that:

$$t_c(|G|) \leq \alpha_1 \cdot |G| + \alpha_2,$$
where $\alpha_1$, $\alpha_2$ are constants.

(2) Assume that $r_1$ and $r_2$ are property paths. Procedure $\text{GUESS}_{r_1|r_2}$ is defined as follows:

$\text{GUESS}_{r_1|r_2}(a, G)$

1: if $\text{coin}()$ then 2: return $\text{GUESS}_{r_1}(a, G)$ 3: else 4: return $\text{GUESS}_{r_2}(a, G)$ 5: end if

The main operator in the property path $r_1|r_2$ is union, so the algorithm guesses which one of the two property paths $r_1$ or $r_2$ to consider. The time complexity of this procedure depends on the time complexity of the calls $\text{GUESS}_{r_1}$ and $\text{GUESS}_{r_2}$:

$$t_{r_1|r_2}(|G|) \leq \max\{t_{r_1}(|G|), t_{r_2}(|G|)\} + \alpha_3$$

$$\leq t_{r_1}(|G|) + t_{r_2}(|G|) + \alpha_3,$$

where $\alpha_3$ is a constant.

(3) Assume that $r_1$ and $r_2$ are property paths. Procedure $\text{GUESS}_{r_1/r_2}$ is defined as follows:

$\text{GUESS}_{r_1/r_2}(a, G)$

1: $a_{\text{middle}} := \text{GUESS}_{r_1}(a, G)$ 2: return $\text{GUESS}_{r_2}(a_{\text{middle}}, G)$

The main operator in the property path $r_1/r_2$ is concatenation, so the algorithm first has to execute procedure $\text{GUESS}_{r_1}$ and then procedure $\text{GUESS}_{r_2}$. The time complexity of this procedure depends on the time complexity of the calls $\text{GUESS}_{r_1}$ and $\text{GUESS}_{r_2}$:

$$t_{r_1/r_2}(|G|) \leq t_{r_1}(|G|) + t_{r_2}(|G|) + \alpha_4,$$

where $\alpha_4$ is a constant.

(4) Assume that $r_1$ is a property path. Procedure $\text{GUESS}_{r_1^*}$ is defined as follows:

$\text{GUESS}_{r_1^*}(a, G)$
1: \textit{used} := ∅
2: \(a_{next} = a\)
3: \textbf{while} coin() \textbf{do}
4: \(a_{new} := \text{GUESS}_{r_1}(a_{next}, G)\)
5: \textbf{if} \(a_{new} \in \text{used}\) \textbf{then}
6: \textit{stop}()
7: \textbf{end if}
8: \textit{used} := \textit{used} \cup \{a_{new}\}
9: \(a_{next} := a_{new}\)
10: \textbf{end while}
11: \textbf{return} \(a_{next}\)

The main operator in the property path \(r_1^*\) is the Kleene star. The semantics of property paths proposed in (Harris & Seaborne, 2011) does not allow node repetitions when constructing a path that conforms to an expression of the form \(r_1^*\). Procedure \text{GUESS}_{r_1} uses the set \text{used} to check for possible repetitions, invoking the procedure \textit{stop}() to terminates an invalid execution of \text{GUESS}_{r_1} where such a repetition happens. Checking whether a node is in the set \text{used} (line 5) takes time proportional to the number of nodes of \(G\). As a node cannot be used twice, the maximum number of times the loop in line 3 can be executed is bounded by the number of nodes of \(G\). As this number is bounded by the size of \(G\), we conclude that:

\[ t_{r_1^*}(|G|) \leq |G| \cdot t_{r_1}(|G|) + \alpha_5 \cdot |G|^2 + \alpha_6, \]

where \(\alpha_5\) y \(\alpha_6\) are constants.

Recall that \(\text{sh}(r)\) is the star height of \(r\) (see Section 5.2 for the definition of \(\text{sh}\)). The following Lemma shows that the time complexity of procedure \text{GUESS}_r depends on the way the star symbol is used in \(r\).
Lemma 5.1. For every property path \( r \), there exist constants \( \alpha_r \) and \( \beta_r \) such that for every RDF graph \( G \):

\[
t_r(|G|) \leq \alpha_r \cdot |G|^{sh(r)+1} + \beta_r.
\]

We prove this lemma by induction on the structure of \( r \).

- **Basis:** Assume that \( r = c \), where \( c \in I \). Then we have that \( sh(r) = 0 \), and that:

\[
t_r(|G|) \leq \alpha_1 \cdot |G| + \alpha_2
\]

\[
= \alpha_1 \cdot |G|^{sh(r)+1} + \alpha_2
\]

- **Inductive step:** Assume that the property holds for property paths \( r_1 \) and \( r_2 \), that is:

\[
t_{r_1}(|G|) \leq \alpha_{r_1} \cdot |G|^{sh(r_1)+1} + \beta_{r_1},
\]

\[
t_{r_2}(|G|) \leq \alpha_{r_2} \cdot |G|^{sh(r_2)+1} + \beta_{r_2}.
\]

To conclude the proof, we consider three cases.

- **Union:** Assume that \( r = r_1 \mid r_2 \).

Then we have that \( sh(r) = \max\{sh(r_1), sh(r_2)\} \), and that:

\[
t_r(|G|) \leq t_{r_1}(|G|) + t_{r_2}(|G|) + \alpha_3
\]

\[
\leq (\alpha_{r_1} \cdot |G|^{sh(r_1)+1} + \beta_{r_1}) + (\alpha_{r_2} \cdot |G|^{sh(r_2)+1} + \beta_{r_2}) + \alpha_3
\]

\[
\leq \left(\alpha_{r_1} + \alpha_{r_2}\right) \cdot |G|^\max\{sh(r_1),sh(r_2)\} + (\beta_{r_1} + \beta_{r_2} + \alpha_3)
\]

\[
= \left(\alpha_{r_1} + \alpha_{r_2}\right) \cdot |G|^{sh(r)+1} + (\beta_{r_1} + \beta_{r_2} + \alpha_3)
\]
• **Concatenation:** Assume that \( r = r_1 / r_2 \).

Then we have that \( \text{sh}(r) = \max\{\text{sh}(r_1), \text{sh}(r_2)\} \), and that:

\[
\begin{align*}
t_r(|G|) & \leq t_{r_1}(|G|) + t_{r_2}(|G|) + \alpha_4 \\
& \leq (\alpha_{r_1} \cdot |G|^{\text{sh}(r_1)+1} + \beta_{r_1}) + (\alpha_{r_2} \cdot |G|^{\text{sh}(r_2)+1} + \beta_{r_2}) + \alpha_4 \\
& \leq (\alpha_{r_1} + \alpha_{r_2}) \cdot |G|^{\max\{\text{sh}(r_1),\text{sh}(r_2)\}+1} + (\beta_{r_1} + \beta_{r_2} + \alpha_4) \\
& = (\alpha_{r_1} + \alpha_{r_2}) \cdot |G|^{\text{sh}(r)+1} + (\beta_{r_1} + \beta_{r_2} + \alpha_4)
\end{align*}
\]

• **Kleene star:** Assume that \( r = r_1^* \).

Then we have that \( \text{sh}(r) = \text{sh}(r_1) + 1 \), and that:

\[
\begin{align*}
t_r(|G|) & \leq |G| \cdot t_{r_1}(|G|) + \alpha_5 \cdot |G|^2 + \alpha_6 \\
& \leq |G| \cdot (\alpha_{r_1} \cdot |G|^{\text{sh}(r_1)+1} + \beta_{r_1}) + \alpha_5 \cdot |G|^2 + \alpha_6 \\
& = \alpha_{r_1} \cdot |G|^{|\text{sh}(r_1)|+2} + \beta_{r_1} \cdot |G| + \alpha_5 \cdot |G|^2 + \alpha_6 \\
& = \alpha_{r_1} \cdot |G|^{\text{sh}(r)+1} + \beta_{r_1} \cdot |G| + \alpha_5 \cdot |G|^2 + \alpha_6 \\
& \leq (\alpha_{r_1} + \beta_{r_1} + \alpha_5) \cdot |G|^{\text{sh}(r)+1} + \alpha_6
\end{align*}
\]

We now define the algorithm \( \text{CS}_p \) by considering the procedure \( \text{GUESS}_p \).

\[
\text{CS}_p(a, b, G)
\]

1: \( a_f := \text{GUESS}_p(a, G) \)
2: if \( a_f = b \) then
3: \hspace{1em} accept
4: else
5: \hspace{1em} stop()
6: end if

It is not difficult to see that the structure of calls made by \( \text{CS}_p(a, b, G) \) corresponds to the structure of calls made by \( \text{COUNT}(a, p, b, G) \). Thus, we have that the number of accepting executions of \( \text{CS}_p(a, b, G) \) corresponds to \( \text{card}_{\mu}[(a,b)]_G(\mu_\emptyset) \). Therefore, to prove that \( \text{COUNTW3C}(p) \in \#P \), it only remains to show that \( \text{CS}_p \) runs in polynomial time. By definition
of CS\(_p\), we have that the execution time of CS\(_p(a, b, G)\) is bounded by \(t_p(|G|) + \alpha\), where \(\alpha\) is a constant. Thus, from Lemma 5.1 we conclude that algorithm CS\(_p\) runs in time \(O(|G|^{\text{sh}(p)+1})\), which is polynomial as \(p\) is a fixed property path.

**Hardness:** We now move to the proof that COUNT\(_W3C(c^*)\) is \#P-hard, where \(c\) is an arbitrary element from \(I\). We know from Theorem 6.3 that COUNT\(_SIMPLEPATH(c^*)\) is \#P-hard and, hence, \#P-hardness of COUNT\(_W3C(c^*)\) follows from the following lemma.

**Lemma 5.2.** For every RDF graph \(G\) and pair of elements \(a, b \in I\), it holds that:

\[
\text{card}_{[(a,c^*,b)]_G}(\mu_0) = \text{card}_{[(a,c^*,b)]_G}^{\text{s-path}}(\mu_0).
\]

For the proof of the lemma, we use the terminology presented in Section 5.1 for the analysis of the tree of recursive calls generated by invoking procedure ALP. Besides, we use here the term u-path to refer to a usual path in a tree, which is not to be confused with the notion of path for RDF graphs introduced in Section 6.

First, we prove that \(\text{card}_{[(a,c^*,b)]_G}(\mu_0) \leq \text{card}_{[(a,c^*,b)]_G}^{\text{s-path}}(\mu_0)\). Consider the tree of calls generated by invoking COUNT\((a, c, b, G)\) (recall that \(\text{card}_{[(a,c^*,b)]_G}(\mu_0) = \text{COUNT}(a, c, b, G)\)). Initially, this procedure call to ALP\((a, c, \text{Result}, \text{Visited}, G)\), where Result is an empty list and Visited is an empty set, and then returns the number of occurrences of \(b\) in Result. In the call ALP\((a, c, \text{Result}, \text{Visited}, G)\), algorithm ALP first adds \(a\) to the set Visited (line 4) to indicate that \(a\) has been visited, then computes in line 6 a set Next containing all the elements \(d\) such that \((a, c, d) \in G\), and then invokes ALP\((d, c, \text{Result}, \text{Visited}, G)\) in line 8 for each such element \(d\).

In general, in a call of the form ALP\((d, c, \text{Result}, \text{Visited}, G)\), if \(d\) is not in the set Visited, then algorithm ALP first adds \(d\) to the set Visited (line 4) to indicate that \(d\) has been visited, then computes in line 6 a set Next containing all the elements \(e\) such that \((d, c, e) \in G\), and then invokes ALP\((e, c, \text{Result}, \text{Visited}, G)\) in line 8 for each such element \(e\). Thus, the number of occurrences of \(b\) in Result corresponds to the number of u-paths from \(a\) to \(b\) in the tree of recursive calls generated from the initial call ALP\((a, c, \text{Result}, \text{Visited}, G)\). By the previous discussion, we have that each of these u-paths corresponds to a path of the form \(a_1, c, a_2, c, a_3, \ldots, a_{n-1}, c, a_n\), where \(n \geq 1\), \(a_1 = a\), \(a_n = b\), \((a_i, c, a_{i+1}) \in G\) for every \(i \in \{1, \ldots, n - 1\}\), and \(a_i \neq a_j\) for every
where \( i, j \in \{1, \ldots, n\} \) such that \( i \neq j \). That is, each such u-path corresponds to a simple path from \( a \) to \( b \) in \( G \) that conforms to \( c^* \). Therefore, given that \( \text{card}_{[(a,c^*,b)]_G}(\mu_\emptyset) = \text{COUNT}(a, c, b, G) \), we conclude that \( \text{card}_{[(a,c^*,b)]_G}(\mu_\emptyset) \leq \text{card}_{[(a,c^*,b)]_G}^{\text{path}}(\mu_\emptyset) \).

Second, we prove that \( \text{card}_{[(a,c^*,b)]_G}^{\text{path}}(\mu_\emptyset) \leq \text{card}_{[(a,c^*,b)]_G}(\mu_\emptyset) \). Let \( \pi \) be a simple path from \( a \) to \( b \) in \( G \) that conforms to \( c^* \). That is, assume that \( \pi \) is a path

\[
a_1, a_2, c, a_3, \ldots, a_{n-1}, c, a_n,
\]

where \( n \geq 1, a_1 = a, a_n = b, (a_i, c, a_{i+1}) \in G \) for every \( i \in \{1, \ldots, n-1\} \), and \( a_i \neq a_j \) for every \( i, j \in \{1, \ldots, n\} \) such that \( i \neq j \). Given that every call \( \text{ALP}(d, c, \text{Result, Visited}, G) \) removes element \( d \) from the set \( \text{Visited} \) in its last step (line 10), we have that the same element can be used in different u-paths of the tree of recursive calls generated from the initial call \( \text{ALP}(a, c, \text{Result, Visited}, G) \). Thus, there exists a u-path in this tree that corresponds to \( \pi \), as the elements \( a_1, \ldots, a_n \) in \( \pi \) are pairwise distinct. Therefore, given that the value returned by \( \text{COUNT}(a, c, b, G) \) corresponds to the number of u-paths from \( a \) to \( b \) in the tree of recursive calls generated from the initial call \( \text{ALP}(a, c, \text{Result, Visited}, G) \) (as shown in the previous paragraph), we conclude that \( \text{card}_{[(a,c^*,b)]_G}^{\text{path}}(\mu_\emptyset) \leq \text{card}_{[(a,c^*,b)]_G}(\mu_\emptyset) = \text{COUNT}(a, c, b, G) \). \( \Box \)

Theorem 5.1 shows that the problem of evaluating property paths under the semantics proposed by the W3C is intractable in data complexity. In fact, it shows that one will not be able to find efficient algorithms to evaluate even simple property paths such as \( c^* \), where \( c \) is an arbitrary element of \( \mathbf{I} \).

The proof of Theorem 5.1 reveals that the complexity of the problem \( \text{COUNTW3C}(p) \) depends essentially on the way the star symbol is used in \( p \). More precisely, the star height of a property path \( p \), denoted by \( \text{sh}(p) \), is the maximum depth of nesting of the star symbols appearing in \( p \) (Eggan, 1963), that is: (1) \( \text{sh}(p) = 0 \) if \( p \in \mathbf{I} \), (2) \( \text{sh}(p) = \max\{\text{sh}(p_1), \text{sh}(p_2)\} \) if \( p = p_1|p_2 \) or \( p = p_1/p_2 \), and (3) \( \text{sh}(p) = \text{sh}(p_1) + 1 \) if \( p = p_1^* \). Then for every positive integer \( k \), define \( SH_k \) as the class of property paths \( p \) such that \( \text{sh}(p) \leq k \), and define \( \text{COUNTW3C}(SH_k) \)
as the problem of computing, given an RDF graph $G$, elements $a, b \in I$ and a property path $p \in SH_k$, the value $\text{card}_{[(a,p,b)]_G}(\mu_\emptyset)$. Then we can generalize Theorem 5.1 as follows:

**Theorem 5.2.** $\text{COUNTW3C}(SH_k)$ is $\#P$-complete for each $k \geq 1$.

We now move to the study of the combined complexity of the problem $\text{COUNTW3C}$. In what follows, we formalize the clique experiment presented in Section 2.2, and then provide lower bounds in this scenario for the number of occurrences of a mapping in the result of the procedure (ALP) used by the W3C to define the semantics of property paths (Harris & Seaborne, 2011). Interestingly, these lower bounds show that the poor behavior detected in the experiments is not a problem with the tested implementations, but instead a characteristic of the semantics of property paths proposed in (Harris & Seaborne, 2011). These lower bounds provide strong evidence that evaluating property paths under the semantics proposed by the W3C is completely infeasible, as they show that $\text{COUNTW3C}$ is not even in $\#P$.

Fix an element $c \in I$ and an infinite sequence $\{a_i\}_{i \geq 1}$ of pairwise distinct elements from $I$, which are all different from $c$. Then for every $n \geq 2$, let $\text{clique}(n)$ be an RDF graph forming a clique with nodes $a_1, \ldots, a_n$ and edge label $c$, that is, $\text{clique}(n) = \{(a_i, c, a_j) \mid i, j \in \{1, \ldots, n\} \text{ and } i \neq j\}$. Moreover, for every property path $p$, define $\text{COUNTCLIQUE}(p, n)$ as $\text{card}_{[(a_1,p,a_n)]_{\text{clique}(n)}}(\mu_\emptyset)$. Then we have that:

**Lemma 5.3.** For every property path $p$ and $n \geq 2$:

$$\text{COUNTCLIQUE}(p^*, n) = \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \text{COUNTCLIQUE}(p, n)^k}{(n-k-1)!}$$

**Proof.** To obtain the value of $\text{COUNTCLIQUE}(p^*, n)$, we analyze the tree of recursive calls generated by invoking $\text{COUNT}(a_1, p, a_n, \text{clique}(n))$ (recall that $\text{card}_{[(a_1,p,a_n)]_{\text{clique}(n)}}(\mu_\emptyset) = \text{COUNT}(a_1, p, a_n, \text{clique}(n))$). In particular, we use the terminology presented in Section 5.1 for the analysis of the tree of recursive calls generated by invoking procedure ALP.

Assume first that $p = c$, where $c$ is the element from $I$ used in the predicate position of every triple in $\text{clique}(n)$. Initially, procedure $\text{COUNT}(a_1, p, a_n, \text{clique}(n))$ calls to $\text{ALP}(a_1, p, \text{Result}, \text{Visited}, \text{clique}(n))$, where $\text{Result}$ is an empty list and $\text{Visited}$ is an empty
set, and then returns the number of occurrences of \( a_n \) in Result. From a node \( a_i \) at depth \( k \) in this tree, \( (n - k) \) nodes can be visited (expanded and added to Result), because the path from \( a_1 \) to \( a_i \) in this tree contains \( k \) nodes, which cannot be visited again given the test in line 1 of algorithm ALP. For example, Figure 5.5 shows the tree of recursive calls generated for \( \text{clique}(5) \).

In this tree, element \( a_1 \) is at depth 1, from which \( 4 = (5 - 1) \) nodes can be visited: \( a_2, a_3, a_4 \) and \( a_5 \). In turn, from the node \( a_3 \) at depth 2, it holds that \( 3 = (5 - 2) \) nodes can be visited: \( a_2, a_4 \) and \( a_5 \), as \( a_1 \) and \( a_3 \) have already been visited. In the tree shown in Figure 5.5, element \( a_5 \) is represented as a grey node. Notice that the subtrees whose root is \( a_5 \) have been pruned, as \( a_5 \) cannot appear again in these subtrees and, thus, they cannot contribute to increase the number of occurrences of \( a_5 \) in Result. In general, in our analysis of the tree of recursive calls generated from the initial call \( \text{ALP}(a_1, p, \text{Result}, \text{Visited}, \text{clique}(n)) \), we do not need to consider the subtrees of this tree whose root is the node \( a_n \). Thus, we call a node \( a_i \) expandable if \( i \neq n \).

Let \( \text{expandable}(k) \) be the number of expandable nodes at depth \( k \). From the discussion in the previous paragraph, we have that from each expandable node at depth \( k \), one can visit \( (n - k) \) distinct nodes at depth \( (k + 1) \). One of these \( (n - k) \) nodes is \( a_n \) and, thus, \( (n - k - 1) \) nodes are expandable. We use this property in the following claim to obtain a formula for the value of \( \text{expandable}(k) \).

**Claim 5.1.** For every \( k \geq 1 \):

\[
\text{expandable}(k) = \frac{(n - 2)!}{(n - k - 1)!}
\]

**Proof.** This result can be proved by induction on \( k \).

- **Base case:** We have that \( \text{expandable}(1) = 1 \) since \( a_1 \) is the only node at depth 1, which is expandable as \( n \geq 2 \). Thus, we have that:

\[
\text{expandable}(1) = 1 = \frac{(n - 2)!}{(n - 1 - 1)!}
\]

- **Inductive Step:** Assume that the property holds for \( k \):

\[
\text{expandable}(k) = \frac{(n - 2)!}{(n - k - 1)!}
\]
As pointed out above, from each expandable node at depth \( k \), one can visit \((n - k - 1)\) nodes that are expandable. Thus, we have that:

\[
\text{expandable}(k + 1) = \text{expandable}(k) \cdot (n - k - 1) \\
= \frac{(n - 2)!}{(n - k - 1)!} \cdot (n - k - 1) \\
= \frac{(n - 2)!}{(n - k - 2)!} \\
= \frac{(n - 2)!}{(n - (k + 1) - 1)!}
\]

As stated above, one of the nodes visited from each expandable node at depth \( k \) is \( a_n \). Thus, the number of occurrences of \( a_n \) at depth \( k \) is equals to the number of expandable nodes at depth \( k - 1 \).

Up to this point, we have assumed that \( p = c \). Assume now that \( p \) is an arbitrary property path. Then to count the number of occurrences of \( a_n \) at depth \( k \), we need to consider the fact that the same node can appear several times at the same depth. More precisely, according to lines 5 and 6 of algorithm ALP, a node \( a_i \) at depth 2 appears as many times as \( \text{card}_{[(a_1, p, ?X)]_G} (\mu) \), where \( \mu \) is a mapping in \( [(a_1, p, ?X)]_G \) such that \( \mu(?X) = a_i \). Thus, given that for every pair
of distinct elements $\ell, m \in \{1, \ldots, n\}$:

$$\text{card}_{[(a_\ell, p, a_m)]_{\text{clique}(n)}}(\mu_\emptyset) = \text{card}_{[(a_1, p, a_n)]_{\text{clique}(n)}}(\mu_\emptyset),$$

we conclude that the number of occurrences of a node $a_i$ at depth 2 is $\text{COUNTCLIQUE}(p, n)$.

Moreover, we obtain by continuing this expansion that the number of occurrences of a node $a_i$ at depth $k$ is $\text{COUNTCLIQUE}(p, n)^{k-1}$.

From the discussion in the previous two paragraphs, we conclude that the number of occurrences of $a_n$ at depth $k$ is equal to the number of expandable nodes at depth $k - 1$ multiplied by the number of repetitions of element $a_n$ at depth $k$. That is, the number of occurrences of $a_n$ at depth $k$ is equal to $\text{expandable}(k - 1) \cdot \text{COUNTCLIQUE}(p, n)^{k-1}$. Thus, given that $\text{COUNTCLIQUE}(p^*, n)$ is equal to the total number of occurrences of $a_n$ in the tree of recursive calls generating by invoking $\text{ALP}(a_1, p, \text{Result}, \text{Visited}, \text{clique}(n))$, we conclude by Claim 5.1 and the fact that $a_n$ does not occur at depth 1 (since $n \geq 2$) that:

$$\text{COUNTCLIQUE}(p^*, n) = \sum_{k=2}^{n} \text{expandable}(k - 1) \cdot \text{COUNTCLIQUE}(p, n)^{k-1}$$

$$= \sum_{k=1}^{n-1} \text{expandable}(k) \cdot \text{COUNTCLIQUE}(p, n)^{k}$$

$$= \sum_{k=1}^{n-1} \frac{(n - 2)! \cdot \text{COUNTCLIQUE}(p, n)^{k}}{(n - k - 1)!}$$

□

Let $p_0 = c$ and $p_{s+1} = p_s^*$, for every $s \geq 0$. For example, $p_1 = c^*$ and $p_3 = ((c^*)^*)^*$. From Lemma 5.3, we obtain that:

$$\text{COUNTCLIQUE}(p_{s+1}, n) = \sum_{k=1}^{n-1} \frac{(n - 2)! \cdot \text{COUNTCLIQUE}(p_s, n)^{k}}{(n - k - 1)!}, \quad (5.1)$$

for every $s \geq 0$. This formula can be used to obtain the number of occurrences of the mapping with empty domain in the answer to the property-path triple $(a_1, p_s, a_n)$ over the RDF graph $\text{clique}(n)$.

For instance, the formula states that if a system implements the semantics proposed by the W3C in (Harris & Seaborne, 2011), then with input $\text{clique}(8)$ and $(a_1, (c^*)^*, a_8)$, the
TABLE 5.1. Number of occurrences of the mapping with empty domain in the answer to property-path triple \((a_1, p_s, a_n)\) over the RDF graph clique\(n\), according to the semantics for property paths proposed by the W3C.

<table>
<thead>
<tr>
<th>(s)</th>
<th>(n)</th>
<th>(\text{COUNTCLIQUE}(p_s, n))</th>
<th>(s)</th>
<th>(n)</th>
<th>(\text{COUNTCLIQUE}(p_s, n))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>2</td>
<td>1</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>5</td>
<td>418576</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>42</td>
<td>3</td>
<td>5</td>
<td>(&gt; 10^{23})</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1806</td>
<td>4</td>
<td>5</td>
<td>(&gt; 10^{93})</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>5</td>
<td>1</td>
<td>6</td>
<td>65</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>305</td>
<td>2</td>
<td>6</td>
<td>28278702465</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>56931605</td>
<td>3</td>
<td>6</td>
<td>(&gt; 10^{53})</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>(&gt; 10^{23})</td>
<td>4</td>
<td>6</td>
<td>(&gt; 10^{269})</td>
</tr>
</tbody>
</table>

empty mapping would have to appear more than \(79 \cdot 10^{24}\) times in the output. Thus, even if a single byte is used to store the empty mapping\(^1\), then the output would be of more than 79 Yottabytes in size! Table 5.1 shows more lower bounds obtained with formula (5.1). Notice that these numbers coincide with the results obtained in our experiments (Tables 2.1 and 2.2). Also notice that, for example, for \(n = 6\) and \(s = 2\) the lower bound is of more than 28 billions, and for \(n = 4\) and \(s = 3\) is of more than 56 millions, which explain why the tested implementations exceeded the timeout for queries Cliq-2 and Cliq-3 (Table 2.2). Most notably, Table 5.1 allows us to provide a cosmological lower bound for evaluating property paths: if one proton is used to store the mapping with empty domain, with input \(\text{clique}(6)\) (which contains only 30 triples) and \((a_1, (((c^*)^*)^*)^*, a_6)\), every system implementing the semantics proposed by the W3C (Harris & Seaborne, 2011) would have to return a file that would not fit in the observable universe!

From Lemma 5.3, we obtain the following double-exponential lower bound for \(\text{COUNTCLIQUE}(p_s, n)\).

**Lemma 5.4.** For every \(n \geq 2\) and \(s \geq 1\):

\[
\text{COUNTCLIQUE}(p_s, n) \geq (n-2)!^{(n-1)^{s-1}}
\]

\(^1\)Recall that the empty mapping \(\mu_\emptyset\) is represented as the four-bytes string | | in ARQ, and as the two-bytes string [] in Sesame.
Proof. From Lemma 5.3, we know that:

\[
\text{COUNTCLIQUE}(p_{s+1}, n) = \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \text{COUNTCLIQUE}(p_s, n)^k}{(n-k-1)!}
\]

We use this fact in the inductive proof of the lemma.

- **Base case:** Given that \(\text{COUNTCLIQUE}(p_0, n) = \text{COUNTCLIQUE}(b, n) = 1\), we have that:

\[
\text{COUNTCLIQUE}(p_1, n) = \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \text{COUNTCLIQUE}(p_0, n)^k}{(n-k-1)!}
= \sum_{k=1}^{n-1} \frac{(n-2)! \cdot 1^k}{(n-k-1)!}
\geq \frac{(n-2)!}{(n-(n-1)-1)!}
= (n-2)!
= (n-2)!(n-1)^0
\]

- **Inductive step:** Assume that the property holds for \(s\), that is:

\[
\text{COUNTCLIQUE}(p_s, n) \geq (n-2)!(n-1)^{s-1}
\]

Then we have that:

\[
\text{COUNTCLIQUE}(p_{s+1}, n) = \sum_{k=1}^{n-1} \frac{(n-2)! \cdot \text{COUNTCLIQUE}(p_s, n)^k}{(n-k-1)!}
\geq \text{COUNTCLIQUE}(p_s, n)^n
\geq \left((n-2)!(n-1)^{s-1}\right)^n
= (n-2)!(n-1)^s
\]

From this bound, we obtain that COUNTW3C is not in \(#P\). Besides, from the proof of Theorem 5.1, we obtain that COUNTW3C is in the complexity class \(#\text{EXP}\), which is defined as \(#P\) but considering non-deterministic Turing Machines that work in exponential time.
Theorem 5.3. \textsc{countw3c} is in \#EXP and not in \#P.

It is open whether \textsc{countw3c} is \#EXP-complete.

5.3. The complexity of the entire language

We consider now the data complexity of the evaluation problem for the entire language. More precisely, we use the results proved in the previous section to show the major impact of using property paths on the complexity of evaluating SPARQL 1.1 queries. The evaluation problem is formalized as follows. Given a fixed SPARQL 1.1 query $Q$, define $\text{evalw3c}(Q)$ as the problem of computing, given an RDF graph $G$ and a mapping $\mu$, the value $\text{card}_{[Q]_G}(\mu)$.

It is easy to see that the data complexity of SPARQL 1.1 without property paths is polynomial. However, from Theorem 5.1, we obtain the following corollary that shows that the data complexity is considerably higher if property paths are included, for the case of the semantics proposed by the W3C (Harris & Seaborne, 2011). In this corollary, we show that $\text{evalw3c}(Q)$ is in the complexity class $\text{fp}^{\#p}$, which is the class of functions that can be computed in polynomial time if one has access to an efficient subroutine for a \#P-complete problem (or, more formally, one has an oracle for a \#P-complete problem).

Corollary 5.1. $\text{evalw3c}(Q)$ is in $\text{fp}^{\#p}$, for every SPARQL 1.1 query $Q$. Moreover, there exists a SPARQL 1.1 query $Q_0$ such that $\text{evalw3c}(Q_0)$ is \#P-hard.
6. INTRACTABILITY FOR ALTERNATIVE SEMANTICS THAT COUNT PATHS

The usual graph theoretical notion of path has been extensively and successfully used when defining the semantics of queries including regular expressions (Mendelzon & Wood, 1995; Calvanese et al., 1999; Alkhateeb et al., 2009; Pérez et al., 2010; Barceló et al., 2010). Nevertheless, given that the W3C SPARQL 1.1 Working Group is interested in counting paths, the classical notion of path in a graph cannot be naively used to define a semantics for property-path queries, given that cycles in an RDF graph may lead to an infinite number of different paths. In this section, we consider two alternatives to deal with this problem. We consider a semantics for property paths based on classical paths that is only defined for acyclic RDF graphs, and we consider a general semantics that is based on simple paths (which are paths in a graph with no repeated nodes). In both cases, we show that query evaluation based on counting is intractable. Next we formalize these two alternative semantics and present our complexity results.

A path $\pi$ in an RDF graph $G$ is a sequence $a_1, c_1, a_2, c_2, \ldots, a_n, c_n, a_{n+1}$ such that $n \geq 0$ and $(a_i, c_i, a_{i+1}) \in G$ for every $i \in \{1, \ldots, n\}$. Path $\pi$ is said to be from $a$ to $b$ in $G$ if $a_1 = a$ and $a_{n+1} = b$, it is said to be nonempty if $n \geq 1$, and it is said to be a simple path, or just s-path, if $a_i \neq a_j$ for every distinct pair $i, j$ of elements from $\{1, \ldots, n + 1\}$. Finally, given a property path $p$, path $\pi$ is said to conform to $p$ if $c_1 c_2 \cdots c_n$ is a string in the regular language defined by $p$.

6.1. Classical paths over acyclic RDF graphs

We first define the semantics of a property-path triple considering classical paths, that we denote by $\llbracket t \rrbracket^\text{path}\!_G$. Notice that we have to take into consideration the fact that the number of paths in an RDF graph may be infinite, and thus we define this semantics only for acyclic graphs. More precisely, an RDF graph $G$ is said to be cyclic if there exists an element $a$ mentioned in $G$ and a nonempty path $\pi$ in $G$ from $a$ to $a$, and otherwise it is said to be acyclic. Then assuming that $G$ is acyclic, the evaluation of a property-path triple $t$ over $G$ in terms of classical paths, denoted
by $[t]_G^\text{path}$, is defined as follows. Let $t = (u, p, v)$ and $W = \{u, v\} \cap V$, then

$$[t]_G^\text{path} = \{\mu \mid \text{dom}(\mu) = W \text{ and there exists a path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p\},$$

and for every $\mu \in [t]_G^\text{path}$, the value $\text{card}_{[t]_G^\text{path}}(\mu)$ is defined as the number of paths from $\mu(u)$ to $\mu(v)$ in $G$ that conform to $p$.

Similarly as we defined the problem COUNTW3C in Section 5, we define the problem COUNTPATH as the problem of computing $\text{card}_{[(a,p,b)]_G^\text{path}}(\mu_\emptyset)$ given as input an acyclic RDF graph $G$, elements $a, b \in I$, and property path $p$. We also define, given a fixed property path $p$, the problem COUNTPATH($p$) as the the problem of computing, given an acyclic RDF graph $G$ and elements $a, b \in I$, the value $\text{card}_{[(a,p,b)]_G^\text{path}}(\mu_\emptyset)$.

To pinpoint the exact complexity of the problems COUNTPATH and COUNTPATH($p$), we need to consider two counting complexity classes: #L and SPANL. We introduce these classes here, and we refer the reader to (Álvarez & Jenner, 1993) for their formal definitions. #L is the counting class associated with the problems that can be solved in logarithmic space in a non-deterministic Turing Machine (NTM). In fact, a function $f$ is said to be in this class if there exists an NTM $M$ that works in logarithmic space such that for every string $w$, the value of $f$ on $w$ is equal to the number of accepting runs of $M$ with input $w$. A prototypical #L-complete problem is the problem of computing, given a deterministic finite automaton $A$ and a string $w$, the number of strings that are accepted by $A$ and whose length is smaller than the length of $w$ (Álvarez & Jenner, 1993). SPANL is defined in a similar way to #L, but considering logarithmic-space NTMs with output. More precisely, a function $f$ is said to be in this class if there exists such TM $M$ such that for every string $w$, the value of $f$ on $w$ is equal to the number of different outputs of $M$ with input $w$. A prototypical SPANL-complete problem is the problem of computing, given a non-deterministic finite automaton $A$ and a string $w$, the number of strings that are accepted by $A$ and whose length is smaller than the length of $w$ (Álvarez & Jenner, 1993). Although classes #L and SPANL look alike, they are quite different in terms of complexity: #L is known to be included in FP, the class of functions that can be computed in
polynomial time, while it is known that \( \text{SPANL} \) is a class of intractable computation problems, if \( \text{SPANL} \subseteq \text{FP} \), then \( P = \text{NP} \). Our first result shows that even for the simple case considered in this section, the problem of evaluating property paths is intractable. In this case we use the notion of parsimonious reductions, that is the strongest notion of reduction for counting problems. More precisely, let \( f, g \) be functions in \( \#L \). A function \( h \) is a parsimonious reductions from \( f \) to \( g \) iff for each input \( w \), \( f(w) = g(h(w)) \). Notice that the function \( h \) is a valid reduction iff \( h \) can be computed using logarithmic space. A function \( f \) is \( \#L \)-complete iff \( \#L \subseteq f \) and \( f \in \#L \). Given a function \( f \) that is known to be \( \#L \)-complete, a function \( g \) is \( \#L \)-complete iff exists a parsimonious reductions from \( f \) to \( g \). In the same way, we can define \( \text{SPANL} \)-completeness and parsimonious reductions for \( \text{SPANL} \) functions.

**Theorem 6.1.** \( \text{COUNTPATH} \) is \( \text{SPANL} \)-complete.

**Proof. Membership:** To prove that \( \text{COUNTPATH} \in \text{SPANL} \), we give a non-deterministic logarithmic-space algorithm \( \text{CACYCLIC} \) that receives as input an RDF graph \( G \), elements \( a, b \in I \) and a property path \( p \), and such that the number of valid outputs of \( \text{CACYCLIC} \) with this input corresponds to \( \text{card}_{[\llbracket a,p,b \rrbracket_G]}(\mu_b) \). To define algorithm \( \text{CACYCLIC} \), we first define algorithm \( \text{GUESSSTATE} \) that, given an \( \epsilon \)-NFA \( M = (Q, \Sigma, q_0, \delta, F) \) (a non-deterministic finite automaton with \( \epsilon \)-transitions), a state \( q \in Q \) and a symbol \( \alpha \in \Sigma \), guesses a state that is reachable from \( q \) (according to \( \delta \)) when reading \( \alpha \).

**Algorithm** \( \text{GUESSSTATE}(M, q, \alpha) \)

1. \( q_{\text{now}} := q \)
2. \( \alpha_{\text{now}} := \epsilon \)
3. **while** \( \alpha_{\text{now}} = \epsilon \) **do**
   4. guess a transition \( (q_j, \beta, q_k) \in \delta \) such that \( q_j = q_{\text{now}} \) and \( \beta \in \{\alpha, \epsilon\} \)
   5. \( q_{\text{now}} := q_k \)
   6. \( \alpha_{\text{now}} := \beta \)
4. **end while**
5. **return** \( q_{\text{now}} \)
As the only variables that are stored in \textsc{GuessState} are \(q_{\text{now}}\) and \(\alpha_{\text{now}}\), it is clear that \textsc{GuessState} runs in logarithmic space. Notice that as \(M\) is an \(\epsilon\)-NFA, it could be the case that there is no transition \((q_j, \beta, q_k) \in \delta\) such that \(q_j = q_{\text{now}}\) and \(\beta \in \{\alpha, \epsilon\}\). If this is the case, line 4 in the algorithm \textsc{GuessState} fails, and this algorithm terminates in a non-accepting state.

As a second ingredient for the definition of algorithm \textsc{Cyclic}, we define algorithm \textsc{CyclicNfa} that receives as input an RDF graph \(G\), elements \(a, b \in I\) and an \(\epsilon\)-NFA \(M = (Q, \Sigma, q_0, \delta, F)\), and gives as output (by using an instruction \textbf{print}) a path \(\pi = a_1, p_1, a_2, \ldots, p_{n-1}, a_n\) in \(G\), such that \(a_1 = a, a_n = b\) and \(p_1p_2\cdots p_{n-1}\) is a string in the regular language accepted by \(M\).

\begin{algorithm}
\begin{algorithmic}[1]
\State \textbf{print} \(a\)
\State \(q_{\text{now}} := q_0\)
\State \(a_{\text{now}} := a\)
\While {\(a_{\text{now}} \neq b\)}
\State guess a triple \((a_j, r, a_k) \in G\) such that \(a_j = a_{\text{now}}\)
\State \(q_{\text{now}} := \textsc{GuessState}(M, q_{\text{now}}, r)\)
\State \(a_{\text{now}} := a_k\)
\State \textbf{print} \(r\)
\State \textbf{print} \(a_{\text{now}}\)
\EndWhile
\If {\(q_{\text{now}} \in F\)}
\State \textbf{accept}
\EndIf
\end{algorithmic}
\end{algorithm}

\textsc{CyclicNfa} is a non-deterministic procedure that guesses triple by triple a path \(\pi = a_1, p_1, a_2, \ldots, p_{n-1}, a_n\) in \(G\) such that \(a_1 = a, a_n = b\) and \(p_1p_2\cdots p_{n-1}\) is a string in the regular language accepted by \(M\) (this condition is checked in line 11). In each step, \textsc{CyclicNfa} guesses the next state of the \(\epsilon\)-NFA \(M\) by using algorithm \textsc{GuessState}. As for the case of algorithm \textsc{GuessState}, if there is no triple \((a_j, r, a_k) \in G\) such that \(a_j = a_{\text{now}}\), then
line 5 of procedure CACYCLICNFA fails, and the algorithm terminates in a non-accepting state. Moreover, given that the only variables that are stored in CACYCLICNFA are \( q_{\text{now}} \), \( a_{\text{now}} \) and GUESSSTATE works in logarithmic space, we have that CACYCLICNFA also works in logarithmic space.

It is important to notice that if two different executions of procedure CACYCLICNFA guess the same path \( \pi \), then the output of these executions is the same (as this output is \( \pi \)). Thus, we conclude from the discussion in the previous paragraph that CACYCLICNFA is a non-deterministic procedure that works in logarithmic space such that the valid outputs of CACYCLICNFA are in a one-to-one correspondence with the paths

\[
\pi = a_1, p_1, a_2, ..., p_{n-1}, a_n \text{ in } G
\]

such that \( a_1 = a \), \( a_n = b \) and \( p_1 p_2 \cdots p_{n-1} \) is a string in the regular language accepted by \( M \).

We now have the necessary ingredients to prove that COUNTPATH \( \in \text{SPANL} \). More specifically, this property is a simple corollary of the following facts: (1) given a property path \( p \), one can construct in logarithmic space an \( \epsilon \)-NFA \( M_p \) that accepts the regular language defined by \( p \) (e.g. see (Álvarez & Jenner, 1993) and (Jiang & Ravikumar, 1991)); (2) if \( f_1, f_2 \) are functions that can be computed by non-deterministic Turing Machines that work in logarithmic space, then the composition of \( f_1 \) with \( f_2 \) can also be computed by a non-deterministic Turing Machine that works in logarithmic space; and (3) given an RDF graph \( G \), elements \( a, b \in I \) and an \( \epsilon \)-NFA \( M \), the number of valid outputs of the call CACYCLICNFA\((G, a, b, M)\) is equal to the number of paths \( \pi = a_1, p_1, a_2, ..., p_{n-1}, a_n \) in \( G \) such that \( a_1 = a \), \( a_n = b \) and \( p_1 p_2 \cdots p_{n-1} \) is a string in the regular language accepted by \( M \).

**Hardness:** We now prove that COUNTPATH is \( \text{SPANL} \)-hard. In this proof, we reduce from the following counting problem: Given an NFA \( M = (Q, \Sigma = \{0, 1\}, q_1, \delta, F) \) and a string \( x \in \{0, 1\}^* \), count the number strings of length at most \( |x| \) accepted by \( M \). This problem is known to be \( \text{SPANL} \)-complete (Álvarez & Jenner, 1993). More specifically, given an NFA \( M \) and a string \( x \), we show how to construct an RDF graph \( G \) with elements \( S, E \in I \) and a property path \( p \) such that the number of paths between \( S \) and \( E \) in \( G \) that conform to \( p \) is
equals to the number of strings of length at most $|x|$ accepted by $M$. Notice that as $M$ is a non-deterministic automaton, different executions of $M$ may accept the same string. To deal with this issue, we define $G$ in such a way that for each possible string there is only one path between $S$ and $E$, no matter the number of executions of $M$ that accept this string. For the sake of readability, we first consider the problem of counting the number of strings accepted by $M$ of length exactly $|x|$ and construct an RDF graph $G'$ to solve this problem, and then we show how to extend $G'$ to generate $G$.

Assume that $N$ is the number of states in $Q$ and $t$ is an arbitrary element from $I$. The building block in the definition of $G'$ is an RDF graph forming a path of $(N+1)$ nodes connected by edges with label $t$. For example, assume that $M_0$ is the NFA shown in Figure 6.1. Then the building block of the RDF graph constructed in the reduction from $M_0$ is the RDF graph shown in Figure 6.2. Recall that this graph has 4 nodes as $M_0$ has 3 states.

![Figure 6.1. NFA $M_0$ used as example in the proof of Theorem 6.1.](image1)

![Figure 6.2. The building block of $G'$ for the NFA $M_0$.](image2)

RDF Graph $G'$ is constructed by putting together $(|x|+1)$ of the building blocks defined above. More precisely, assume that $S, E, s, e, 0, 1$ are pairwise distinct elements from $I$, which are all distinct from $t$. Then to construct $G'$, first we add a triple with subject $S$ and property $s$, and having as object the first element of one the $(|x|+1)$ building blocks. Second, we form a sequence with the $(|x|+1)$ building block, connecting the last element of each of these blocks with the first element of the following one in the sequence (if there exists such block) by an edge with label 0 and an edge with label 1. Finally, we add a triple with property $e$ and object $E$, and
Figure 6.3. RDF graph $G'$ for NFA $M_0$ and a string $x$ such that $|x| = 2$.

having as subject the last element of the last building block in the aforementioned sequence. For example, Figure 6.3 shows the RDF graph $G'$ generated from NFA $M_0$ shown in Figure 6.1 and a string $x$ such that $|x| = 2$.

The idea behind the definition of $G'$ is that each string in $\{0, 1\}^*$ of length $|x|$ should be represented by exactly one path between $S$ and $E$ in $G'$. For instance, in the graph shown in Figure 6.3, there are 4 paths between $S$ and $E$ representing all the possible words of length 2 over the alphabet $\{0, 1\}$: 00, 01, 10, 11. But not only that, the edges with label $t$ in $G'$ are used to represent the different ways in which NFA $M$ could accept a string $x$ of length $|x|$, but with the important property that each of these different ways is encoded by the same path in $G'$.

We now show how to encode the transition function $\delta$ of $M$ in a property path $p$ in a way that ensures that the previous condition is satisfied.

Assume that the states of $M$ are $Q = \{q_1, \ldots, q_N\}$ (recall that $N$ is the number of states of $M$). Moreover, for every $r \in I$, define $r^i$ as: $r^0 = \varepsilon$ and $r^{i+1} = r/r^i$ ($i \geq 0$). For example, $r^2 = r/r$ and $r^5 = r/r/r/r/r/r$ (notice that we omit $\varepsilon$ if possible). Then define a set $\Gamma$ of property paths as follows:

1. $(s/t)$ belongs to $\Gamma$.
2. For every $i \in \{1, \ldots, N\}$ such that $q_i \in F$, property path $(t^{N-i}/e)$ belongs to $\Gamma$.
3. For each $(q_i, \alpha, q_j) \in \delta$, where $i, j \in \{1, \ldots, N\}$ and $\alpha \in \{0, 1\}$, property path $(t^{N-i}/\alpha/t^j)$ belongs to $\Gamma$.

Assume that $\Gamma = \{p_1, p_2, \ldots, p_\ell\}$. Then property path $p$ is defined as $(p_1 \mid p_2 \mid \cdots \mid p_\ell)^*$. For example, Figure 6.4 shows the set $\Gamma$ of property paths constructed for the NFA $M_0$ shown in Figure 6.1. In this example, we have that:

\[ p = (s/t \mid e \mid t/t/0/t/t \mid t/t/0/t/t \mid t/1/t/t/t \mid 1/t/t/t)^*. \]
We show the intuition behind the definition of property path $p$ by considering NFA $M_0$ shown in Figure 6.1, RDF graph $G'$ shown in Figure 6.3 and property path $p$ in equation (1). Assume that $\pi$ is a path from $S$ to $E$ in $G'$ conforming to $p$. Then we have that $\pi = S, s, \ldots$. Thus, given that $(s/t)$ is the only property path in $p$ mentioning $s$, we conclude that $\pi$ is of the form $S, s, a_1, t, a_2, \ldots$ since $\pi$ conforms to $p$, where $a_1, a_2$ are elements from $I$. Now, the only way to continue navigating graph $G'$ is by traversing two consecutive edges with label $t$ before reaching an edge with label either 0 or 1. The only property paths in $p$ containing two consecutive symbols $t$ before the symbols 0, 1 are $t/t/0/t/t$ and $t/t/0/t/t/t$, thus the next part of $\pi$ has to conform to one of these property paths. Assume that $t/t/0/t/t$ is the property path used in this part, so we conclude that $\pi$ is of the form $S, s, a_1, t, a_2, t, a_3, t, a_4, 0, a_5, t, a_6, t, a_7, \ldots$. Notice that $t/t/0/t/t$ is included in $\Gamma$ because $(q_1, 0, q_2)$ is a transition in $\delta$, so the part of $\pi$ just mentioned represents the fact that in the NFA $M_0$, one reaches the state $q_2$ by reading the symbol 0 at the initial state $q_1$. Now, we have that the only way to continue navigating graph $G'$ is by traversing a single edge with label $t$ before reaching an edge with label 0 or 1. The only property path in $p$ containing a single symbol $t$ before the symbols 0, 1 is $t/1/t/t/t$, so we conclude that $\pi$ is of the form $S, s, a_1, t, a_2, t, a_3, t, a_4, 0, a_5, t, a_6, t, a_7, t, a_8, 1, a_9, t, a_{10}, t, a_{11}, t, a_{12}, \ldots$. Notice that $t/1/t/t/t$ is included in $\Gamma$ because $(q_2, 1, q_3)$ is a transition in $\delta$, so the part of $\pi$ just added represents the fact that in the NFA $M_0$, one reaches the state $q_3$ by reading the symbol 1 at the state $q_2$. Thus, we have that $\pi$ represents the fact that $(q_1, 0, q_2), (q_2, 1, q_3)$ is a valid execution of NFA $M_0$ for string 01 from the initial state $q_1$. Notice that we ensure that the same state $q_2$ is used in the consecutive configurations $(q_1, 0, q_2), (q_2, 1, q_3)$ by representing $(q_1, 0, q_2)$ as $t/t/0/t/t$ and $(q_2, 1, q_3)$ as $t/1/t/t/t$ in our reduction (in general, we represent $(q_i, \alpha, q_j)$ as $t^{N-i}/\alpha/t^j$ and $(q_j, \beta, q_k)$ as $t^{N-j}/\beta/t^k$, so that by concatenating these property paths we obtain

<table>
<thead>
<tr>
<th>Automaton $M_0$ ($N = 3$)</th>
<th>Rule applied</th>
<th>Element of $\Gamma$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_3 \in F$</td>
<td>(1)</td>
<td>s/t</td>
</tr>
<tr>
<td>$(q_1, 0, q_2) \in \delta$</td>
<td>(2)</td>
<td>e</td>
</tr>
<tr>
<td>$(q_1, 0, q_3) \in \delta$</td>
<td>(3)</td>
<td>$t/t/0/t/t$</td>
</tr>
<tr>
<td>$(q_2, 1, q_3) \in \delta$</td>
<td>(3)</td>
<td>$t/1/t/t/t$</td>
</tr>
<tr>
<td>$(q_3, 1, q_3) \in \delta$</td>
<td>(3)</td>
<td>$1/t/t/t$</td>
</tr>
</tbody>
</table>

**Figure 6.4.** Construction of the set $\Gamma$ of property path for NFA $M_0$. 


property path $t^{N_i}/\alpha/t^N/\beta/t^k$ having exactly $N$ elements $t$ between the symbols $\alpha$ and $\beta$).

Finally, given that $\pi$ is a path from $S$ to $E$ in $G$ that conforms to $p$, we have that property path $e$ from $p$ has to be considered to complete the definition of $\pi$:

$$\pi = S, s, a_1, t, a_2, t, a_3, t, a_4, 0, a_5, t, a_6, t, a_7, t, a_8, 1, a_9, t, a_{10}, t, a_{11}, t, a_{12}, e, E.$$

It is important to notice that string $01$ could also be accepted by considering the following sequence of transitions: $(q_1, 0, q_3), (q_3, 1, q_3)$. This is not a problem for our reduction as the same path $\pi$ represents this sequence of transitions, but this time considering the following sequence of property paths from $p$: $s/t, t/t/0/t/t/t, 1/t/t/t$ and $e$.

By generalizing the argument presented in the above discussion, it is possible to prove that the number of strings accepted by $M$ of length exactly $|x|$ is equal to the number of paths from $S$ to $E$ in $G'$ that conform to $p$.

We now show how to extend the definition of $G'$ to count all the strings of length at most $|x|$ that are accepted by $M$. To do this, for each triple $(a, \alpha, b)$ in $G'$, where $\alpha \in \{0, 1\}$, we add a triple $(a, e, E)$. For example, Figure 6.5 shows the generated RDF graph $G$ for the NFA $M_0$ (shown in Figure 6.3) and a string $x$ such that $|x| = 2$.

Given the previous discussion, it is possible to prove that the number of strings accepted by $M$ of length at most $|x|$ is equal to the number of paths from $S$ to $E$ in $G$ that conform to $p$. Thus, given that $G$ is an acyclic RDF graph, we have that the number of strings accepted by $M$ of length at most $|x|$ is equal to $\text{card}_{\text{path}}(\mu_\emptyset)$. Hence, to conclude the proof, it only remains to show that $G$ and $p$ can be constructed from $M$ and $x$ in logarithmic space. To construct the triples of $G$, we need to have an index $i$ that indicates the number of the building block that we are considering ($1 \leq i \leq |x| + 1$), and we need to have a second index $j$ that
indicates the position in this building block \((1 \leq j \leq N + 1)\). We need logarithmic space in the size of \(M\) and \(x\) to store these indexes, since \(1 \leq i \leq |x| + 1\) and \(1 \leq j \leq N + 1\) (recall that \(N\) is the number of states of \(Q\)). Moreover, to construct \(p\), we need to consider one by one the transitions \((q_i, \alpha, q_j)\) of \(M\), and we need to store \(N\) so that we can generate \(t^{N-i}/\alpha/t^j\). This can be done in logarithmic space, which concludes the proof of the theorem. \(\square\)

Interestingly, our second complexity result shows that at least in terms of data complexity, the problem of evaluating property paths is tractable if their semantics is based on the usual notion of path.

**Theorem 6.2.** \(\text{COUNTPATH}(p)\) is in \(\#L\) for every property path \(p\). Moreover, there exists a property path \(p_0\) such that \(\text{COUNTPATH}(p_0)\) is \(\#L\)-complete.

**Proof. Membership:** Let \(p\) be a fixed property path. To prove that \(\text{COUNTPATH}(p) \in \#L\), we give a non-deterministic logarithmic space algorithm \(\text{CACYCLIC}_p\) that has as input an RDF graph \(G\) and elements \(a, b \in I\), and such that the number of valid executions of \(\text{CACYCLIC}_p\) with this input corresponds to \(\text{card}_{\{a,p,b\}^\text{path}}(\mu_\emptyset)\).

**Algorithm CACYCLIC\(_p\)(G, a, b)**

1: Construct a deterministic finite automaton (DFA) \(M_p = (Q, \Sigma, q_0, \delta, F)\) such that \(L(M_p) = L(p)\)
2: \(q_{\text{now}} := q_0\)
3: \(a_{\text{now}} := a\)
4: **while** \(a_{\text{now}} \neq b\) **do**
5: \(\text{guess a triple } (a_j, r, a_k) \in G\) such that \(a_j = a_{\text{now}}\)
6: \(q_{\text{now}} := \delta(q_{\text{now}}, r)\)
7: \(a_{\text{now}} := a_k\)
8: **end while**
9: **if** \(q_{\text{now}} \in F\) **then**
10: \(\text{accept}\)
11: **end if**
CA CYCLIC\textsubscript{p} is a non-deterministic procedure that first constructs a DFA \( M_p \) that accepts the same regular language as \( p \). Then CA CYCLIC\textsubscript{p} guesses triple-by-triple a path from \( a \) to \( b \) in \( G \), updating in each step the state of \( M_p \) according to the guessed triple. It is important to notice that if for some particular value of \( a_{\text{node}} \), there is no triple of the form \((a_{\text{now}}, r, a_k)\), then the execution of CA CYCLIC\textsubscript{p} terminates in a non-accepting state.

It is easy to see that CA CYCLIC\textsubscript{p} works in logarithmic space. In fact, the space needed to store \( M_p \) is constant because \( p \) is fixed and, thus, the space needed to store the state \( q_{\text{now}} \) is also constant. Moreover, \( a_{\text{now}} \) is a node of \( G \) and, thus, the space needed to store it is logarithmic in the size of \( G \) (if \( G \) contains \( n \) nodes, then each of these nodes can be stored as a string over the alphabet \( \{0, 1\} \) with \( O(\log n) \) bits). Therefore, to finish the proof it only remains to prove that the number of valid executions of CA CYCLIC\textsubscript{p}(\( G, a, b \)) corresponds to \( \text{card}_{(a,p,b)\text{path}}(\mu_\emptyset) \).

**Hardness:** We now prove that there exists a property path \( p_0 \) such that COUNTPATH(\( p_0 \)) is \#L-hard. In this proof, we reduce from the following counting problem. Given a DFA \( M \) over the alphabet \( \{0, 1\} \) and a string \( x \in \{0, 1\}^* \), count the number of words of length at most \(|x|\) accepted by \( M \). This problem is known to be \#L-complete (Álvarez & Jenner, 1993).

Fix an IRI \( c \in I \), and then define \( p_0 \) as the property path \(((0|1)^*/c)\). Next we show that given a DFA \( M = (Q, \{0, 1\}, q_0, \delta, F) \) over the alphabet \( \{0, 1\} \) and a string \( x \in \{0, 1\}^* \), one can construct in logarithmic space an acyclic RDF graph \( G_M \) and elements \( a, b \in I \) such that the number of words of length at most \(|x|\) accepted by \( M \) is equal to \( \text{card}_{(a,p_0,b)\text{path}}(\mu_\emptyset) \). More precisely, RDF graph \( G_M \) corresponds to the configurations-graph of \( M \), that is, \( G_M \) contains all the triples of the form \([t_1, q_1, \alpha, t_2, q_2]\), where \( \alpha \in \{0, 1\} \), \( q_1, q_2 \in Q, t_1, t_2 \in \{0, \ldots, |x|\}, q_2 = \delta(q_1, \alpha) \) and \( t_2 = t_1 + 1 \). That is, each triple \([t_1, q_1, \alpha, t_2, q_2]\) in \( G_M \) indicates that if \( M \)
is in state \( q_1 \) at time \( t_1 \) and reads the symbol \( \alpha \), then \( M \) will be in state \( q_2 \) at time \( t_2 = t_1 + 1 \). Notice that we impose the restriction \( 0 \leq t_1, t_2 \leq |x| \) as we are interested in counting words of length at most \( |x| \). Moreover, assuming that \( d \) is a fresh symbol from \( I \) (it was not used in any of the previous triples), for every final state \( q_f \) of \( M \) \((q_f \in F) \) and for every \( t \in \{0, \ldots, |x|\} \), RDF graph \( G_M \) contains triple \( ([t, q_f], c, d) \). These last triples are used to obtained a unique final node when counting paths in \( G \).

It is straightforward to prove that the number of words of length at most \( |x| \) accepted by \( M \) is equal to \( \text{card}_{[[0,q_0],[p_0,d]]}^\text{path}(\mu_0) \). Moreover, it is also easy to see that \( G_M \) can be constructed in logarithmic space as the space needed to store a triple of the form \( ([t_1, q_1], \alpha, [t_2, q_2]) \) is logarithmic in the size of \( M \) and \( x \) (recall that \( 0 \leq t_1, t_2 \leq |x|, \alpha \in \{0, 1\} \) and \( q_1, q_2 \in Q \)), and the triples in \( G_M \) can be generated one by one in lexicographic order. Thus, given that \( G_M \) is an acyclic graph, as \( t_1 < t_2 \) in every triple of the form \( ([t_1, q_1], \alpha, [t_2, q_2]) \), we conclude that the above reduction is correct.

Although COUNTPATH\((p)\) is tractable, it only considers acyclic RDF graphs, and thus leaves numerous practical cases uncovered.

### 6.2. Simple paths

We continue our investigation by considering the alternative semantics for property paths that is defined in terms of simple paths. This semantics has been considered in the public mailing list of the SPARQL 1.1 Working Group,\(^1\) as it is a natural way to solve the problem of having an infinite number of paths, as it is properly defined for every RDF graph. Notice that even for cyclic RDF graphs, the number of simple paths is finite, and thus, this semantics is properly defined for every RDF graph. Formally, assume that \( G \) is an RDF graph, \( t = (u, p, v) \) is a property-path triple and \( W = (\{u, v\} \cap V) \). The evaluation of \( t \) over \( G \) in terms of s-paths,

\(^1\)http://lists.w3.org/Archives/Public/public-rdf-dawg-comments
denoted by \( [t]_{G}^{\text{s-path}} \), is defined as:

\[
[t]_{G}^{\text{s-path}} = \{ \mu \mid \text{dom}(\mu) = W \text{ and there exists an s-path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p \},
\]

and for every \( \mu \in [t]_{G}^{\text{s-path}} \), the value \( \text{card}_{[t]_{G}^{\text{s-path}}} (\mu) \) is defined as the number of s-paths from \( \mu(u) \) to \( \mu(v) \) in \( G \) that conform to \( p \). For the case of s-paths, we define the problem \text{COUNTSIMPLEPath} as follows. The input of this problem is an RDF graph \( G \), elements \( a, b \in I \) and a property path \( p \), and its output is the value \( \text{card}_{[[a,p,b]]_{G}^{\text{s-path}}} (\mu_\emptyset) \). As for the previous problems, we define \text{COUNTSIMPLEPath}(p) as \text{COUNTSIMPLEPath} for a fixed property path \( p \). The following result shows that these problems are also intractable (see Section 5.2 for the definitions of \#P and \#P-completeness).

**Theorem 6.3.** \text{COUNTSIMPLEPath} is in \#P. Moreover, if \( c \in I \), then \text{COUNTSIMPLEPath}(c^*) is \#P-complete.

**Proof. Membership:** We first prove that \text{COUNTSIMPLEPath} \( \in \#P \), from which we immediately conclude that \text{COUNTSIMPLEPath}(p) \( \in \#P \) for every property path \( p \). Let \( a, b \) be elements from \( I \), \( p \) a property path and \( G \) and RDF graph. Then let \text{CSIMPLE} be the following procedure:

**Algorithm** \text{CSIMPLE}(a, b, p, G)

1: let \( n \) be the number of nodes in \( G \)
2: guess a path \( \pi \) from \( a \) to \( b \) in \( G \) of length at most \( n \)
3: if \( \pi \) is a simple path and \( \pi \) conforms to \( p \) then
   4: accept
5: end if

\text{CSIMPLE} is a non-deterministic procedure that guesses a path \( \pi \) from \( a \) to \( b \) in \( G \) with a length bounded by the number of nodes in \( G \), and then accepts if \( \pi \) is a simple path conforming to \( p \). Notice that \text{CSIMPLE} works in polynomial-time, as it is possible to verify in polynomial time
whether a string belongs to the regular language defined by a regular expression. Moreover, the number of accepting runs of CS\text{IMPLE}(a,b,p,G) is equal to the number of simple paths from \(a\) to \(b\) in \(G\) that conforms to \(p\), as the length of every such simple path is bounded by the number of nodes in \(G\). Thus, we conclude that \text{COUNTSIMPLEPATH} \in \#P.

**Hardness:** We now prove that \text{COUNTSIMPLEPATH}(\(c^*\)) is \#P-hard, where \(c\) is an arbitrary element of \(I\), from which we immediately conclude that \text{COUNTSIMPLEPATH} is \#P-hard. In this proof, we reduce from the following counting problem: Given a directed graph \(D = (V, E)\) and two distinct vertices \(v_1, v_2 \in V\), count the number of simple paths from \(v_1\) to \(v_2\) in \(D\). This problem is known to be \#P-complete (Valiant, 1979b).

Next we show how to generate, given a graph \(D = (V, E)\) and two distinct vertices \(v_1, v_2 \in V\), and RDF graph \(G\) and two nodes \(a, b\) such that the number of simple paths in \(D\) from \(v_1\) to \(v_2\) is equal to \(\text{card}_{[(a,c^*,b)]}^{\text{\#path}}(\mu_\emptyset)\). For the sake of readability, assume that the nodes of \(D\) are elements from \((I \setminus \{c\})\). Then define RDF graph \(G\) as \(\{(u, v) \mid (u, v) \in E\}\), and define \(a\) as \(v_1\) and \(b\) as \(v_2\).

It is straightforward to prove that the above reduction is correct. Thus, given that this reduction can be computed in polynomial time (in fact, in linear time), we conclude that \text{COUNTSIMPLEPATH}(\(c^*\)) is \#P-hard.

Notice that the data complexity of evaluating property paths according to the s-path semantics is the same as evaluating them according to the W3C semantics. The difference is in the combined complexity that is radically higher for the W3C semantics: for the case of the semantics based on s-paths the combined complexity is in \#P, while for the W3C semantics it is not in \#P.
7. AN EXISTENTIAL SEMANTICS TO THE RESCUE

We have shown in the previous section that evaluating property-path triples according to the semantics proposed in (Harris & Seaborne, 2011) is essentially infeasible, being the core of this problem the necessity of counting different paths. We have also shown that the version in which one counts simple-paths is infeasible too. A possible solution to this problem is to not use a semantics that requires counting paths, but instead a more traditional existential semantics for property-path triples. That is, one just checks if two nodes are connected (or not) by a path that conforms to a property-path expression. This existential semantics has been used for years in graph databases (Mendelzon & Wood, 1995; Calvanese et al., 1999; Barceló et al., 2010), in XML (Marx, 2005; Gottlob et al., 2005), and even on RDF (Alkhateeb et al., 2009; Pérez et al., 2010) previous to SPARQL 1.1. In this section, we introduce this semantics and study the complexity of evaluating property paths, and also SPARQL 1.1 queries, under it. We also compare this proposal with the current official semantics for property paths, and present some experimental results that validate our proposal.

The most natural way to define an existential semantics for property paths is as follows. Assume that \( u, v \in (I \cup V) \), \( W = (\{u, v\} \cap V) \), \( t = (u, p, v) \) is a property-path triple, and \( G \) is an RDF graph. Then define \( \llbracket t \rrbracket_G^{\exists(path)} \) as:

\[
\llbracket t \rrbracket_G^{\exists(path)} = \{ \mu \mid \text{dom}(\mu) = W \text{ and there exists a path from } \mu(u) \text{ to } \mu(v) \text{ in } G \text{ that conforms to } p \}\.
\]

Moreover, define the cardinality of every mapping \( \mu \) in \( \llbracket t \rrbracket_G^{\exists(path)} \) just as 1. Notice that with the semantics \( \llbracket t \rrbracket_G^{\exists(path)} \), we are essentially discarding all the duplicates from \( \llbracket t \rrbracket_G^{path} \). This allows us to consider general graphs (not necessarily acyclic graph as in Section 5). To study the complexity of evaluating property paths under this semantics, we define the decision problem \( \text{EXISTSPATH} \), whose input is an RDF graph \( G \), elements \( a, b \in I \) and a property-path triple \( t = (a, p, b) \), and whose output is the answer to the question: is \( \text{card} \llbracket t \rrbracket_G^{\exists(path)}(\mu_\emptyset) = 1 \)? That is, the problem \( \text{EXISTSPATH} \) is equivalent to checking whether \( \mu_\emptyset \in \llbracket t \rrbracket_G^{\exists(path)} \).
Notice that with \textsc{ExistsPath}, we are measuring the combined complexity of evaluating paths under the existential semantics. The following result shows that \textsc{ExistsPath} is tractable. This is a corollary of some well-known results on graph databases (e.g. see Section 3.1 in (Pérez et al., 2010)). In the result, we use $|G|$ to denote the size of an RDF graph $G$ and $|p|$ to denote the size of a property-path $p$.

\textbf{Proposition 7.1.} \textsc{ExistsPath} can be solved in time $O(|G| \cdot |p|)$.

\section{7.1. Discarding duplicates from the standard and simple-paths semantics}

A natural question at this point is whether there exists a relationship between the existential semantics defined in the previous section and the semantics that can be obtained by discarding duplicates from $[t]_G$ and $[t]_G^{s\text{-path}}$ for a property-path triple $t$. We formalize and study these two semantics in this section.

Assume that $G$ is an RDF graph and $t$ is a property-path triple. Then we define $[t]_G^{\exists}$ as having exactly the same mappings as in $[t]_G$, but with the cardinality of every mapping in $[t]_G^{\exists}$ defined just as 1. Similarly, we define $[t]_G^{\exists(s\text{-path})}$ as having exactly the same mappings as in $[t]_G^{s\text{-path}}$, but with the cardinality of every mapping in $[t]_G^{\exists(s\text{-path})}$ defined as 1. In this section, we study the decision problem \textsc{Existsw3c}, whose input is an RDF graph $G$, elements $a, b \in I$ and a property-path triple $t = (a, p, b)$, and whose output is the answer to the question: is $\text{card}_{[t]_G^{\exists}}(\mu_\emptyset) = 1$? We also study the complexity of the decision problem \textsc{ExistsSimplePath}, which is defined as \textsc{Existsw3c} but considering the semantics $[\cdot]_G^{\exists(s\text{-path})}$ instead of $[\cdot]_G^{\exists}$.

Our first result shows that, somehow surprisingly, the semantics $[\cdot]_G^{\exists}$ coincides with $[\cdot]_G^{\exists(s\text{-path})}$. Thus, even though the official semantics of property paths is given in terms of a particular procedure (Harris & Seaborne, 2011), when one does not count paths, it coincides with the classical existential semantics based on the usual notion of path.

\textbf{Theorem 7.1.} For every RDF graph $G$, mapping $\mu$ and property-path triple $t$: $\mu \in [t]_G^{\exists}$ if and only if $\mu \in [t]_G^{\exists(s\text{-path})}$.

\textbf{Proof.} We use the following two lemmas in the proof of the theorem.
Lemma 7.1. For every RDF graph $G$, elements $a, b \in I$ and property path $p$:

$$\text{If } \mu_0 \in \llbracket (a, p, b) \rrbracket^G_{\text{path}}, \text{ then } \mu_0 \in \llbracket (a, p, b) \rrbracket_G.$$  

Proof. We prove the lemma by induction on $p$.

- **Base case**: Assume that $p = c$ with $c \in I$. If $\mu_0 \in \llbracket (a, c, b) \rrbracket^G_{\text{path}}$, then the triple $(a, c, b)$ is in $G$ and, therefore, $\mu_0 \in \llbracket (a, c, b) \rrbracket_G$ by definition of the semantics of property paths.

- **Inductive step**: Assume that the property holds for property paths $r_1, r_2$.

  - **Union**: Assume that $p = r_1 | r_2$. If $\mu_0 \in \llbracket (a, p, b) \rrbracket^G_{\text{path}}$, then $\mu_0 \in \llbracket (a, r_1, b) \rrbracket^G_{\text{path}}$ or $\mu_0 \in \llbracket (a, r_2, b) \rrbracket^G_{\text{path}}$. Without loss of generality, assume that $\mu_0 \in \llbracket (a, r_1, b) \rrbracket^G_{\text{path}}$. Then we conclude by induction hypothesis that $\mu_0 \in \llbracket (a, r_1, b) \rrbracket_G$. Thus, by the definition of the semantics of $|$ in property paths, which is defined in terms of the UNION operator (see Section 4), we conclude that $\mu_0 \in \llbracket (a, r_1 | r_2, b) \rrbracket_G$ and, hence, $\mu_0 \in \llbracket (a, p, b) \rrbracket_G$.

  - **Concatenation**: Assume $p = r_1 / r_2$. If $\mu_0 \in \llbracket (a, p, b) \rrbracket^G_{\text{path}}$, then there exists $d \in I$ such that $\mu_0 \in \llbracket (a, r_1, d) \rrbracket^G_{\text{path}}$ and $\mu_0 \in \llbracket (d, r_2, b) \rrbracket^G_{\text{path}}$. Thus, we conclude by induction hypothesis that $\mu_0 \in \llbracket (a, r_1, d) \rrbracket_G$ and $\mu_0 \in \llbracket (d, r_2, b) \rrbracket_G$. Therefore, by the definition of the semantics of $/$ in property paths, which is defined in terms of the AND operator (see Section 4), we conclude that $\mu_0 \in \llbracket (a, r_1 / r_2, b) \rrbracket_G$ and, hence, $\mu_0 \in \llbracket (a, p, b) \rrbracket_G$.

  - **Kleene star**: Assume that $p = r_1^*$. If $a = b$, then we immediately conclude that $\mu_0 \in \llbracket (a, r_1^*, b) \rrbracket_G$ by definition of the procedures COUNT and ALP (see Figure 4.1). Thus, assume also that $a \neq b$.

    If $\mu_0 \in \llbracket (a, r_1^*, b) \rrbracket^G_{\text{path}}$, then there exist elements $a_1, a_2, \ldots, a_m$ in $I$, such that $m \geq 2$, $a_1 = a$, $a_m = b$ and for every $i \in \{1, \ldots, m - 1\}$, it holds that $\mu_0 \in \llbracket (a_i, r_1, a_{i+1}) \rrbracket^G_{\text{path}}$. Let $b_1, b_2, \ldots, b_n$ be a subsequence of $a_1, a_2, \ldots, a_m$ such that $m \geq 2$, $b_1 = a_1 = a$, $b_n = a_m = b$, $\mu_0 \in \llbracket (b_i, r_1, b_{i+1}) \rrbracket^G_{\text{path}}$ for every $i \in \{1, \ldots, n - 1\}$, and $b_i \neq b_j$ for every $i, j \in \{1, \ldots, n\}$ such that $i \neq j$ (notice that such a sequence exists since $a \neq b$). By induction hypothesis,
we have that \( \mu_0 \in [((b_i, r_1, b_{i+1})]_G \) for every \( i \in \{1, \ldots, n-1\} \). Therefore, by definition of procedures \textsc{Count} and \textsc{ALP}, we deduce that \( \mu_0 \in (((b_1, r_1^*, b_n)]_G \) and, hence, \( \mu_0 \in [((a, r_1^*, b)]_G \), which concludes the proof of the lemma.

\[ \square \]

**Lemma 7.2.** For every RDF graph \( G \), property-path triple \( t \) and mapping \( \mu \):

\[
\text{If } \mu \in [t]^{\exists\text{(path)}}_G, \text{ then } \mu \in [t]_G.
\]

**Proof.** Assume that \( t = (u, p, v) \), where \( u, v \in (V \cup I) \) and \( p \) is a property-path. If \( \mu \in [((u, p, v)]^{\exists\text{(path)}}_G \), then \( \mu_0 \in [[[\mu(u), p, \mu(v))]^{\exists\text{(path)}}_G \). Thus, we have by Lemma 7.1 that \( \mu_0 \in [((a, p, b)]_G \), from which we conclude that \( \mu \in [((u, p, v)]_G \).

\[ \square \]

We now have all the necessary ingredients to prove Theorem 7.1. Assume that \( G \) is an RDF graph and \( t = (u, p, v) \) is a property-path triple. First, we notice that if \( \mu \in [t]^{\exists\text{(path)}}_G \), then \( \mu \in [t]_G \) and, therefore, we have that \( \mu \in [t]^{\exists\text{(path)}}_G \) as each branch in the tree of recursive calls generated by invoking procedure \textsc{ALP} corresponds to a path in \( G \) that conforms to \( p \) (see Section 5.1 and the proof of Theorem 5.1 for an analysis of this tree of recursive calls). Second, assume that \( \mu \in [t]^{\exists\text{(path)}}_G \). Then we have by Lemma 7.2 that \( \mu \in [t]_G \), from which we conclude that \( \mu \in [t]^{\exists\text{(path)}}_G \).

\[ \square \]

As a corollary of Propositions 7.1 and Theorem 7.1, we obtain that:

**Theorem 7.2.** \textsc{ExistsW3C} can be solved in time \( O(|G| \cdot |p|) \).

The situation is radically different for the case of simple paths. From some well-known results on graph databases (Mendelzon & Wood, 1995), one can prove that \textsc{ExistsSimplePath} is an intractable problem, even for a fixed property-path. More precisely, for a fixed property-path \( p \), the decision problem \textsc{ExistsSimplePath}(\( p \)) has as input an RDF graph \( G \) and elements \( a, b \in I \), and the question is whether \( \text{card}_{[[(a, p, b)]^{\exists\text{(s-path)}}]_G} (\mu_0) = 1 \).
**Proposition 7.2.** \textsc{ExistsSimplePath} is in \textsc{NP}. Moreover, \textsc{ExistsSimplePath}\(((c/c)^*)\) is \textsc{NP-complete}, where \(c \in \mathcal{I}\).

**Proof. Membership:** To prove that \textsc{ExistsSimplePath} \(\in \textsc{NP}\), we give a non-deterministic polynomial time algorithm that, given an RDF graph \(G\), elements \(a, b \in \mathcal{I}\) and a property path \(p\), guesses an s-path from \(a\) and \(b\) in \(G\) that conforms to \(p\), which shows that \(\text{card}_{(\mathcal{G})}^\exists \langle a, p, b \rangle = 1\). This algorithm first guesses a list of triples from \(G\) of length at most \(|G|\). Then it checks whether this list corresponds to a path from \(a\) to \(b\) in \(G\) with no repeated nodes. Finally, it verifies whether the generated path conforms to the property path \(p\). It is clear that these steps can be executed in polynomial time because the size of the generated path is at most \(|G|\).

**Hardness:** The fact that \textsc{ExistsSimplePath}\(((c/c)^*)\) is \textsc{NP-hard} is a simple corollary of the fact that the following problem is \textsc{NP-complete}: Given a directed graph \(D\) and a pair \(a, b\) of nodes in \(D\), verify whether there exists a simple directed path from \(a\) to \(b\) in \(D\) of even length (Mendelzon & Wood, 1995). In fact, property path \((c/c)^*\) is used to look for paths of even length. \(\square\)

### 7.2. Existential semantics and SPARQL 1.1

We have shown that when bags are considered for the semantics of property paths, the evaluation becomes intractable, even in data complexity. However, the previous version of SPARQL, that did not include path queries, considered a bag semantics for the mapping operators (AND, OPT, UNION, FILTER and SELECT), which has proved to be very useful in practice. Thus, a natural question is whether one can construct a language with functionalities to express interesting queries about paths in RDF graphs, with bag semantics for the mappings operators, and that, at the same time, can be efficiently evaluated. In this section, we give a positive answer to this question. We show that if one combines existential semantics for property paths and bag semantics for the SPARQL 1.1 operators, one obtains the best of both worlds and still has tractable data complexity.
We start by formalizing this alternative way of evaluating SPARQL 1.1 queries that considers existential semantics for property-path triples. Given a SPARQL 1.1 query $Q$, define $[Q]_G^\exists$ exactly as $[Q]_G$ is defined in Sections 3 and 4, but evaluating property-paths triples according to the semantics $[\cdot]_G^\exists$ defined in Section 7.1 (that is, $[t]_G$ is replaced by $[t]_G^\exists$ if $t$ is a property-path triple), and likewise for $[Q]_G^\exists(s\text{-path})$ and $[Q]_G^\exists(path)$. Notice that for the three semantics $[Q]_G^\exists$, $[Q]_G^\exists(path)$ and $[Q]_G^\exists(s\text{-path})$, we are not discarding all duplicates but only the duplicates that are generated when evaluating property paths. Thus, these semantics are still bag semantics, and therefore we consider the following computation problems. We define first the computation problem $\text{EVALEXISTSW3C}(Q)$, whose input is an RDF graph $G$ and a mapping $\mu$, and whose output is the value $\text{card}_G[Q]_{\exists}(\mu)$. Moreover, we also consider the computation problems $\text{EVALEXISTSIMPLEPATH}(Q)$ and $\text{EVALEXISTSPATH}(Q)$, that have the same input as $\text{EVALEXISTSW3C}(Q)$ and are defined as the problems of computing $\text{card}_G[Q]_{\exists(s\text{-path})}(\mu)$ and $\text{card}_G[Q]_{\exists(path)}(\mu)$, respectively. Notice that in these three problems, we are considering the data complexity of SPARQL 1.1 under the respective semantics.

Notably, the next result shows that the just defined semantics $[\cdot]_G^\exists$ and $[\cdot]_G^\exists(path)$ are tractable, in terms of data complexity.

This result is a consequence of Theorem 7.2 and Proposition 7.1. In the formulation of this result we use the class FP, which is defined as the class of all functions that can be computed in polynomial time (and thus, it is a class of tractable functions).

**Theorem 7.3.** $\text{EVALEXISTSW3C}(Q)$ and $\text{EVALEXISTSPATH}(Q)$ are in FP for every SPARQL 1.1 query $Q$.

We conclude this section by showing that for the case of the semantics $[\cdot]_G^\exists(s\text{-path})$, the data complexity is unfortunately still high. To study this problem we need the complexity classes $\text{FP}^{\text{NP}}$ and $\text{FP}^{\text{NP}[O(\log n)]}$, which are defined in terms of oracles as for the case of the complexity class $\text{FP}^\#P$ used in Corollary 5.1. More precisely, the class $\text{FP}^{\text{NP}}$ contains all the functions that can be computed in polynomial time by a procedure that is equipped with an efficient subroutine (oracle) for an NP-complete problem, with the restriction that all the calls to the subroutine should be made *in parallel*, that is, no call to the subroutine can depend on the
result of a previous call to this subroutine (Wagner, 1987). The class $\text{FP}^{\text{NP}[O(\log n)]}$ is defined in the same way, but with the restriction that the subroutine for an NP-complete problem can be called only a logarithmic number of times. Both classes $\text{FP}^{\text{NP}[O(\log n)]}$ and $\text{FP}^{\text{NP}}$ are considered to be intractable. Moreover, it is known that $\text{FP}^{\text{NP}[O(\log n)]} \subseteq \text{FP}^{\text{NP}}$, but it is open whether this containment is strict (Selman, 1994).

**Theorem 7.4.** $\text{EVALEXISTSSIMPLEPATH}(Q)$ is in $\text{FP}^{\text{NP}}$ for every SPARQL 1.1 query $Q$. Moreover, there exists a query $Q_0$ such that $\text{EVALEXISTSSIMPLEPATH}(Q_0)$ is $\text{FP}^{\text{NP}[O(\log n)]}$-hard.

**Proof.** We show that the problem is in $\text{FP}^{\text{NP}}$. We now prove that there exists a query $Q$ for which $\text{EVALEXISTSSIMPLEPATH}(Q)$ is $\text{FP}^{\text{NP}[O(\log n)]}$-hard.

It is known that the following problem is $\text{FP}^{\text{NP}[O(\log n)]}$-complete (Papadimitriou, 1994).

Problem: **COMPUTEMAXCLIQUE**
Input: a graph $G$
Output: the size of the maximum clique in $G$

By using **COMPUTEMAXCLIQUE** it is easy to prove that the following problem is also $\text{FP}^{\text{NP}[O(\log n)]}$-complete

Problem: **COMPUTEMAXSAT**
Input: a sequence $\varphi_1, \varphi_2, \ldots, \varphi_k$ of SAT instances such that for $i \in \{1, \ldots, k - 1\}$ if $\varphi_i \notin \text{SAT}$ then $\varphi_{i+1} \notin \text{SAT}$
Output: the value $\max\{i \in \{1, \ldots, k\} \mid \varphi_i \in \text{SAT}\}$

We use function $f$ that reduce the problem SAT to **CHECKSIMPLE** (recall that $f(\varphi) = (G_\varphi, a_\varphi, b_\varphi)$ for an instance $\varphi$ of SAT). We have all the necessary to show the theorem. We use the query $Q$ given by the following expression:

$$\text{SELECT } ?Z \left( (?X, (u/u)^*, ?Y) \text{ AND } (?X, p, ?Y) \text{ AND } (?X, s, ?X') \text{ AND } (?Y, t, ?Y') \text{ AND } (?X', p, ?Y') \text{ AND } (?X', (u/u)^*, ?Y') \text{ AND } (?Z, q, ?Z) \right)$$
Now we prove that \textsc{EvalExistsSimplePath}(Q) is $\text{FP}^{\text{NP}[O(\log n)]}$-hard by using a reduction from \textsc{ComputeMaxSat}. Let $\varphi_1, \varphi_2, \ldots, \varphi_k$ be an instance of \textsc{ComputeMaxSat}. We use the function $f$ to construct an RDF graph $G$ as follows. To simplify the notation, we assume that for the formula $\varphi_i$ the result of $f(\varphi_i)$ is the tuple $(G_i, a_i, b_i)$. Moreover, we assume that for every $i \in \{1, \ldots, k\}$ the tuples $f(\varphi_i) = (G_i, a_i, b_i)$ satisfy that all the $G_i$’s are disjoint RDF graphs (does not have any node in common). Further assume that the graphs $G_i$ does not mention the values $0, a_0, b_0, c_0$. Then $G$ is the graph

$$
G = \left( \bigcup_{1 \leq i \leq k} G_i \right) \cup \{(a_i, s, a_{i+1}) \mid i \in \{1, \ldots, k - 1\}\}
\cup \{(b_i, t, b_{i+1}) \mid i \in \{1, \ldots, k - 1\}\}
\cup \{(a_i, p, b_i) \mid i \in \{1, \ldots, k\}\}
\cup \{(a_0, s, a_1), (b_0, t, b_1), (a_0, p, b_0), (a_0, u, c_0), (c_0, u, b_0), (0, q, 0)\}
$$

Next we show the following property which is enough to prove that \textsc{EvalExistsSimplePath}(Q) is $\text{FP}^{\text{NP}[O(\log n)]}$-hard:

$$
\mu = \{?Z \rightarrow 0\} \text{ has cardinality } n \text{ in } \llbracket Q \rrbracket_{G}^{\exists(s\text{-path})}
\text{iff with input } \varphi_1, \ldots, \varphi_k \text{ the output of ComputeMaxSat is } n. \quad (7.1)
$$

In the rest of the proof we denote by $R$ the following graph pattern:

$$
R = \left( (?X, (u/u)^*, ?Y) \text{ AND } (?X, p, ?Y) \text{ AND } (?X, s, ?X') \text{ AND } (?Y, t, ?Y') \text{ AND } (?X', p, ?Y') \text{ AND } (?X', (u/u)^*, ?Y') \text{ AND } (?Z, q, ?Z) \right)
$$

Since $R$ only has AND operators and we are using semantics $\llbracket_{G}^{\exists(s\text{-path})}$ to evaluate property path triples, it holds that every mapping in $\llbracket R \rrbracket_{G}^{s\text{-path}}$ has cardinality 1. Moreover, by the construction of $G$ it is easy to see that for every mapping $\nu \in \llbracket R \rrbracket_{G}^{s\text{-path}}$ it holds that $\nu(?Z) = 0$. By
the previous discussion we have that the following property implies (7.1):

$$\| [R]_G^{\exists (s \text{- path})} \| = n \text{ iff with input } \varphi_1, \ldots, \varphi_k \text{ the output of COMPUTEMAXSAT is } n. \quad (7.2)$$

Before proving property (7.2), we make a general observation regarding the evaluation of $R$ over $G$. Notice that given the construction of $G$ and since $R$ mentions the triple patterns $(?X, p, ?Y)$, $(?X, s, ?X')$, $(?Y, t, ?Y')$, $(?X', p, ?Y')$, and $(?Z, q, ?Z)$ we have that for every $\nu \in [P]_G^{\exists (s \text{- path})}$ there exists an element $i \in \{1, \ldots, k\}$ such that

$$\nu = \{ ?X \rightarrow a_{i-1}, ?Y \rightarrow b_{i-1}, ?X' \rightarrow a_i, ?Y' \rightarrow b_i, ?Z \rightarrow 0 \}. \quad (7.3)$$

We have all the necessary to prove (7.2). Assume first that with input $\varphi_1, \ldots, \varphi_k$ the output of COMPUTEMAXSAT is $n$. If $n = 0$ then we have that for every $i \in \{1, \ldots, k\}$ it holds that

$$\varphi_i \notin \text{SAT}. \quad \text{This implies that for every } i \in \{1, \ldots, k\} \text{ it holds that } \{ ?X' \rightarrow a_i, ?Y' \rightarrow b_i \} \notin \| [R]_G^{\exists (s \text{- path})} \| = \emptyset, \text{ implying that } \| [R]_G^{\exists (s \text{- path})} \| = 0 \text{ which shows that (7.2) holds.}$$

Assume now that $n \neq 0$, and let $i \in \{1, \ldots, n\}$. Since $\varphi_i \in \text{SAT}$, we have that $\{ ?X' \rightarrow a_i, ?Y' \rightarrow b_i \} \in \| [R]_G^{\exists (s \text{- path})} \|$ and that $\{ ?X \rightarrow a_{i-1}, ?Y \rightarrow b_{i-1} \} \notin \| [R]_G^{\exists (s \text{- path})} \|$. If $i - 1 \neq 0$ the property holds since $\varphi_{i-1} \in \text{SAT}$, and if $i - 1 = 0$ the property holds since $(a_0, u, c_0), (c_0, u, b_0)$ are triples in $G$). Thus we have that the mapping

$$\nu = \{ ?X \rightarrow a_{i-1}, ?Y \rightarrow b_{i-1}, ?X' \rightarrow a_i, ?Y' \rightarrow b_i, ?Z \rightarrow 0 \}. \quad (7.3)$$

is in $\| [R]_G^{\exists (s \text{- path})} \|$ for every $i \in \{1, \ldots, n\}$. Now if $j \in \{n+1, \ldots, k\}$ we have that $\{ ?X' \rightarrow a_j, ?Y' \rightarrow b_j \} \notin \| [R]_G^{\exists (s \text{- path})} \|$ from which we conclude that $\nu = \{ ?X \rightarrow a_{j-1}, ?Y \rightarrow b_{j-1}, ?X' \rightarrow a_j, ?Y' \rightarrow b_j, ?Z \rightarrow 0 \} \notin \| [R]_G^{\exists (s \text{- path})} \|$. Finally, since every mapping in $\| [R]_G^{\exists (s \text{- path})} \|$ is of the form (7.3), and given that the elements $a_0, a_1, \ldots, a_k$ (as well as $b_0, b_1, \ldots, b_k$) are pairwise distinct, we have that $\| [R]_G^{\exists (s \text{- path})} \|$ has exactly $n$ mappings. This was to be shown.
Assume now that $\| R \|^G G^{3(s\text{-path})} = n$. From (7.3), we know that there exists exactly $n$ values $i \in \{ 1, \ldots, k \}$ such that

$$\nu = \{ ?X \rightarrow a_{i-1}, ?Y \rightarrow b_{i-1}, ?X' \rightarrow a_{i}, ?Y' \rightarrow b_{i}, ?Z \rightarrow 0 \}. \quad (7.4)$$

is a mapping in $\| R \|^G G^{3(s\text{-path})}$. By using a similar argument as in the previous case it can be easily shown that if \( \varphi_1, \ldots, \varphi_k \) is the input of COMPUTE\textsc{MaxSat}, then the output is $n$. Just recall that for the input of COMPUTE\textsc{MaxSat} we assume that for each $i \in \{ 1, \ldots, k - 1 \}$ if $\varphi_i \notin \text{SAT}$ then $\varphi_{i+1} \notin \text{SAT}$, and $\{ ?X' \rightarrow a_i, ?Y' \rightarrow b_i \} \in \| (?X', (u/u)^*, ?Y') \|^G G^{3(s\text{-path})}$ if and only if $\varphi_i \in \text{SAT}$. □

Theorem 7.4 shows that simple paths are not a good option even if duplicates are not considered.

7.3. Experiments for the existential semantics

In the previous section, we showed that SPARQL 1.1 is tractable in terms of data complexity if one considers the existential semantics $\| \cdot \|^G G^{3}$ and $\| \cdot \|^G G^{3(s\text{-path})}$ for property paths. The goal of this section is to show the impact of using these semantics in practice, by conducting a final experiment with two implementations that extends SPARQL 1.0 with existential path semantics: Psparql (version 3.3) (ZZ5, 2011), and Gleen (version 0.6.1) (ZZ6, 2011). These two implementations evaluate SPARQL queries according to $\| \cdot \|^G G^{3(s\text{-path})}$, although they use a slightly different syntax for path queries.

In our experiments, we use the following result that allows us to compare SPARQL 1.1 implementations mentioned in Section 2.1 with Psparql and Gleen. It is important to notice that this result is of independent interest, as it shows that the implementations of SPARQL 1.1 that follow the official specification (Harris & Seaborne, 2011) can be highly optimized when using the \textsc{SELECT DISTINCT} feature.
Theorem 7.5. Let $P$ be a SPARQL 1.1 graph pattern, $G$ an RDF graph and $W$ a set of variables. Then we have that:

$$\llbracket \text{(SELECT DISTINCT } W \ P) \rrbracket_G = \llbracket \text{(SELECT DISTINCT } W \ P) \rrbracket^\exists(\text{path})_G$$

$$\llbracket \text{(SELECT DISTINGUISH } * \ P) \rrbracket_G = \llbracket \text{(SELECT DISTINGUISH } * \ P) \rrbracket^\exists(\text{path})_G$$

Proof. We show first that:

$$\llbracket \text{(SELECT DISTINGUISH } * \ P) \rrbracket_G = \llbracket \text{(SELECT DISTINGUISH } * \ P) \rrbracket^\exists(\text{path})_G.$$

First notice that every mapping in both sets has cardinality 1, and thus it is enough to show that for every mapping $\mu$ it holds that $\mu \in \llbracket \text{(SELECT DISTINGUISH } * \ P) \rrbracket_G$ if and only if $\mu \in \llbracket \text{(SELECT DISTINGUISH } * \ P) \rrbracket^\exists(\text{path})_G$. Recall that given a SPARQL 1.1 query $Q$ of the form (SELECT DISTINGUISH $* \ P$) we have that $\mu \in \llbracket Q \rrbracket_G$ if and only if $\mu \in \llbracket P \rrbracket_G$ (although both mapping can have different cardinalities), and similarly for the semantics $\llbracket \cdot \rrbracket^\exists(\text{path})_G$. Thus, we only need to prove that $\mu \in \llbracket P \rrbracket_G$ if and only if $\mu \in \llbracket P \rrbracket^\exists(\text{path})_G$. We do this by induction on the construction of $P$.

For the base case we need to consider triple patterns and property-path triples. For a triple pattern $t$ the property trivially holds since $\llbracket t \rrbracket^\exists(\text{path})_G$ is defined as $\llbracket t \rrbracket_G$ (the definition for triple patterns does not change). Now for the case of a property-path triple $t$, notice that by definition $\mu \in \llbracket t \rrbracket^\exists(\text{path})_G$ if and only if $\mu \in \llbracket t \rrbracket_G$. Moreover, from Theorem 7.1 we have that that $\mu \in \llbracket t \rrbracket^\exists(\text{path})_G$ if and only if $\mu \in \llbracket t \rrbracket^\exists(\text{path})_G$, thus implying that $\mu \in \llbracket t \rrbracket_G$ if and only if $\mu \in \llbracket t \rrbracket^\exists(\text{path})_G$, and the property holds. Now assume that $P_1$ and $P_2$ are patterns and such that $\mu_1 \in \llbracket P_1 \rrbracket_G$ if and only if $\mu_1 \in \llbracket P_1 \rrbracket^\exists(\text{path})_G$, and that $\mu_2 \in \llbracket P_2 \rrbracket_G$ if and only if $\mu_2 \in \llbracket P_2 \rrbracket^\exists(\text{path})_G$. We split the rest of the proof in cases:

- For the case of the AND operator, notice that a mapping $\mu$ is in $\llbracket (P_1 \text{ AND } P_2) \rrbracket_G$ if and only if $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in \llbracket P_1 \rrbracket_G$ and $\mu_2 \in \llbracket P_2 \rrbracket_G$. Similarly, we have that $\mu$ is in $\llbracket (P_1 \text{ AND } P_2) \rrbracket^\exists(\text{path})_G$ if and only if $\mu = \mu_1 \cup \mu_2$ with $\mu_1 \in \llbracket P_1 \rrbracket^\exists(\text{path})_G$ and $\mu_2 \in \llbracket P_2 \rrbracket^\exists(\text{path})_G$. Thus, by applying the induction hypothesis we have that $\mu \in \llbracket (P_1 \text{ AND } P_2) \rrbracket_G$ if and only if $\mu \in \llbracket (P_1 \text{ AND } P_2) \rrbracket^\exists(\text{path})_G$,
For the case of the UNION operator, notice that a mapping \( \mu \) is in \( \llbracket (P_1 \lor P_2) \rrbracket \) if and only if \( \mu \in \llbracket P_1 \rrbracket \cup \llbracket P_2 \rrbracket \). Similarly, we have that \( \mu \) is in \( \llbracket (P_1 \land P_2) \rrbracket \) if and only if \( \mu \in \llbracket P_1 \rrbracket \land \llbracket P_2 \rrbracket \). Thus, by applying the induction hypothesis we have that \( \mu \in \llbracket (P_1 \lor P_2) \rrbracket \) if and only if \( \mu \in \llbracket (P_1 \land P_2) \rrbracket \).

For the case of the OPT operator, notice that a mapping \( \mu \) is in \( \llbracket (P_1 \text{\ OPT} P_2) \rrbracket \) if and only if \( \mu \in \llbracket (P_1 \text{\ AND} P_2) \rrbracket \) or \( \mu \in \{ \mu_1 \in \llbracket P_1 \rrbracket \ | \ \forall \mu' \in \llbracket P_2 \rrbracket : \mu_1 \not\sim \mu' \} \). If \( \mu \in \llbracket (P_1 \text{\ AND} P_2) \rrbracket \), then we know that \( \mu \in \llbracket (P_1 \text{\ AND} P_2) \rrbracket \). Thus, assume that \( \mu \in \Omega \) = \( \{ \mu_1 \in \llbracket P_1 \rrbracket \ | \ \forall \mu' \in \llbracket P_2 \rrbracket : \mu_1 \not\sim \mu' \} \). Notice that by induction hypothesis, \( \Omega = \{ \mu_1 \in \llbracket P_1 \rrbracket \ | \ \forall \mu' \in \llbracket P_2 \rrbracket : \mu_1 \not\sim \mu' \} \). This shows that if \( \mu \in \llbracket (P_1 \text{\ OPT} P_2) \rrbracket \), then \( \mu \in \llbracket (P_1 \text{\ OPT} P_2) \rrbracket \). The other direction is similar.

For the case of the MINUS operator, notice that a mapping \( \mu \) is in \( \llbracket (P_1 \text{\ MINUS} P_2) \rrbracket \) if and only if \( \mu \in \llbracket P_1 \rrbracket \) and for every \( \mu' \in \llbracket P_2 \rrbracket \) either \( \mu \not\sim \mu' \) of \( \text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset \). By applying the induction hypothesis we conclude that \( \mu \) is in \( \llbracket (P_1 \text{\ MINUS} P_2) \rrbracket \) if and only if \( \mu \in \llbracket P_1 \rrbracket \) and for every \( \mu' \in \llbracket P_2 \rrbracket \) either \( \mu \not\sim \mu' \) of \( \text{dom}(\mu) \cap \text{dom}(\mu') = \emptyset \), which implies that \( \mu \in \llbracket (P_1 \text{\ MINUS} P_2) \rrbracket \). The other direction is similar.

For the case of the FILTER operator, let \( R \) be an arbitrary built-in condition. Notice that \( \mu \in \llbracket (P_1 \text{\ FILTER} R) \rrbracket \) if and only if \( \mu \in \llbracket P_1 \rrbracket \) and \( \mu \models R \). Thus, by applying the induction hypothesis, we have that \( \mu \in \llbracket (P_1 \text{\ FILTER} R) \rrbracket \) if and only if \( \mu \in \llbracket P_1 \rrbracket \) and \( \mu \models R \), and then \( \mu \in \llbracket (P_1 \text{\ FILTER} R) \rrbracket \) if and only if \( \mu \in \llbracket (P_1 \text{\ FILTER} R) \rrbracket \).

This completes the proof by induction.

We now show that:

\[
\llbracket (\text{SELECT DISTINCT } W \ P) \rrbracket_G = \llbracket (\text{SELECT DISTINCT } W \ P) \rrbracket_G^{\text{path}}.
\]

We have already shown that \( \mu \in \llbracket P \rrbracket \) if and only if \( \mu \in \llbracket P \rrbracket^{\text{path}} \). Thus we have that \( \mu' \in \{ \mu|_W \ | \ \mu \in \llbracket P \rrbracket \} \) if and only if \( \mu' \in \{ \mu|_W \ | \ \mu \in \llbracket P \rrbracket^{\text{path}} \} \). From
this it immediately follows that \( \mu \in \llbracket (\text{SELECT DISTINCT } W \ P) \rrbracket_G \) if and only if \( \mu \in \llbracket (\text{SELECT DISTINCT } W \ P) \rrbracket_G^{3(\text{path})} \), which was to be shown.

In view of this theorem, we consider all the queries in Section 2, but this time using the \texttt{SELECT DISTINCT} feature:

- **Cliq-1D**: \( \text{SELECT DISTINCT } * \text{ WHERE } \{ :a0 (:p)* :a1 \} \)
- **Cliq-2D**: \( \text{SELECT DISTINCT } * \text{ WHERE } \{ :a0 ((:p)*)* :a1 \} \)
- **Cliq-3D**: \( \text{SELECT DISTINCT } * \text{ WHERE } \{ :a0 (((:p)*)*)* :a1 \} \)
- **Foaf-1D**: \( \text{SELECT DISTINCT } * \text{ WHERE } \{ \text{axel:me (foaf:knows)* } ?x \} \)

Tables 1.2 and 7.1 show the results of this experiment. Although the tested systems return the same results for these queries, the differences in efficiency between the SPARQL 1.1 implementations and the implementations that use an existential semantics are dramatic.
Table 7.1. Time in seconds for queries Cliq-1D, Cliq-2D, and Cliq-3D (symbol “–” means one-hour timeout).

<table>
<thead>
<tr>
<th></th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Kgram</th>
<th>Sesame</th>
<th>Psparql</th>
<th>Gleen</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>1.68</td>
<td>32.61</td>
<td>1.39</td>
<td>9.08</td>
<td>0.18</td>
<td>1.24</td>
</tr>
<tr>
<td>9</td>
<td>2.00</td>
<td>213.99</td>
<td>5.34</td>
<td>166.82</td>
<td>0.20</td>
<td>1.23</td>
</tr>
<tr>
<td>10</td>
<td>3.65</td>
<td>2123.90</td>
<td>227.66</td>
<td>–</td>
<td>0.20</td>
<td>1.25</td>
</tr>
<tr>
<td>11</td>
<td>29.71</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.23</td>
<td>1.25</td>
</tr>
<tr>
<td>12</td>
<td>1394.06</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.24</td>
<td>1.24</td>
</tr>
<tr>
<td>13</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>0.27</td>
<td>1.24</td>
</tr>
</tbody>
</table>

**Cliq-1D**

<table>
<thead>
<tr>
<th></th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Psparql</th>
<th>Gleen</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.40</td>
<td>0.76</td>
<td>0.14</td>
<td>1.23</td>
</tr>
<tr>
<td>3</td>
<td>1.19</td>
<td>0.84</td>
<td>0.14</td>
<td>1.23</td>
</tr>
<tr>
<td>4</td>
<td>1.65</td>
<td>19.38</td>
<td>0.14</td>
<td>1.23</td>
</tr>
<tr>
<td>5</td>
<td>97.06</td>
<td>–</td>
<td>0.15</td>
<td>1.22</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>–</td>
<td>0.16</td>
<td>1.23</td>
</tr>
<tr>
<td>7</td>
<td>–</td>
<td>–</td>
<td>0.16</td>
<td>1.23</td>
</tr>
</tbody>
</table>

**Cliq-2D**

<table>
<thead>
<tr>
<th></th>
<th>ARQ</th>
<th>RDFQ</th>
<th>Psparql</th>
<th>Gleen</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1.18</td>
<td>0.77</td>
<td>0.14</td>
<td>1.24</td>
</tr>
<tr>
<td>3</td>
<td>1.41</td>
<td>6.78</td>
<td>0.14</td>
<td>1.23</td>
</tr>
<tr>
<td>4</td>
<td>–</td>
<td>–</td>
<td>0.15</td>
<td>1.24</td>
</tr>
<tr>
<td>5</td>
<td>–</td>
<td>–</td>
<td>0.15</td>
<td>1.24</td>
</tr>
<tr>
<td>6</td>
<td>–</td>
<td>–</td>
<td>0.16</td>
<td>1.24</td>
</tr>
</tbody>
</table>

**Cliq-3D**
8. CONCLUSIONS: OUR RESULTS IN PERSPECTIVE

Our results pose a strong argument against the current semantics of property paths, from both, theory and practice. We have made clear that the main problem is the necessity of counting paths imposed by the current SPARQL 1.1 specification. Our investigation raises several questions, being one of the most important whether there exists such a strong use case for counting paths that will make the designers of the language to stick with the current semantics, even knowing that in simple and natural cases it will lead to completely impractical evaluation procedures. We have searched in the official document and also in the discussions around the design of the language, and to the best of our knowledge, there is no strong use case for counting paths. It should also be noticed that this counting functionality has not been used as a primitive in previous navigational languages for graph structured data.

On the positive side, we have shown that a semantics based on checking the existence of paths (without counting them), has several advantages: it can be easily defined and understood, it is based on years of research and practical experience, and, most importantly, it can be efficiently evaluated. In view of our result that DISTINCT can be used to go from the counting semantics to the existential semantics, one might be tempted to think that users not interested in counting paths can use DISTINCT in queries. We strongly disagree with this view. Bag semantics for relational queries has proved to be essential in practice. Thus, a good language should be able to deal with both characteristics: being able to search for paths using an existential semantics, and, at the same time, having bag semantics for relational-like operators (such as AND and OPT). Our proposal is to have such a semantics for SPARQL 1.1, which corresponds to the semantics $\exists \cdot \subseteq \mathcal{G}$ defined in Section 7.1, and provide a special feature such as ALL-PATHS, for users that would like to count paths and know the implications of this action.

Although SPARQL is still in its infancy, the increasing interest in managing RDF data is making this language to become more and more popular. To maintain and even increase its adoption rate, the group in charge of the upcoming version of SPARQL should consider all the possible inputs regarding the language. We do think that for the case of property-path
evaluation, our proposal, as opposed to the current official semantics, would lead to a wide adoption of the language by practitioners, developers and theoreticians.
REFERENCES


