NautiLOD: A Formal Language for the Web of Data Graph

VALERIA FIONDA, Department of Mathematics and Computer Science, University of Calabria
GIUSEPPE PIRRÒ, WeST, University of Koblenz
CLAUDIO GUTIERREZ, DCC, University of Chile

The Web of Linked Data is a huge graph of distributed and interlinked datasources fueled by structured information. This new environment calls for formal languages and tools to automatize navigation across datasources (nodes in such graph) and enable semantic-aware and Web-scale search mechanisms. In this article we introduce a declarative navigational language for the Web of Linked Data graph called NautiLOD. NautiLOD enables one to specify datasources via the intertwining of navigation and querying capabilities. It also features a mechanism to specify actions (e.g., send notification messages) that obtain their parameters from datasources reached during the navigation. We provide a formalization of the NautiLOD semantics, which captures both nodes and fragments of the Web of Linked Data. We present algorithms to implement such semantics and study their computational complexity. We discuss an implementation of the features of NautiLOD in a tool called `swget`, which exploits current Web technologies and protocols. We report on the evaluation of `swget` and its comparison with related work. Finally, we show the usefulness of capturing Web fragments by providing examples in different knowledge domains.

Categories and Subject Descriptors: H.2.3 [**Database Management**]: Languages; H.2.4 [**Database Management**]: Systems

General Terms: Design, Algorithms

Additional Key Words and Phrases: Navigation, graph languages, Web of data, linked data, semantic Web

ACM Reference Format:
Valeria Fionda, Giuseppe Pirrò, and Claudio Gutierrez. 2015. NautiLOD: A formal language for the Web of data graph. ACM Trans. Web 9, 1, Article 5 (January 2015), 43 pages.
DOI: http://dx.doi.org/10.1145/2697393

1. INTRODUCTION

There is an increasing availability of structured data on the Web. A vast portion of the Web can now be thought of as a large graph where nodes represent datasources describing *entities* (e.g., people, places) and edges *semantic relations* (e.g., born in, located in) between them. Such an enhanced version of the Web is usually referred to as the Web of Linked Data (WLOD). There are different projects that contribute to the

A preliminary version of this article titled "Semantic navigation on the Web of data: Specification of routes, Web fragments and actions" appeared in *Proceedings of the 21st World Wide Web Conference*, pp. 281–290, ACM Press, New York, NY, 2012.
V. Fionda's work was partially supported by the European Commission, the European Social Fund, and the Calabria region. G. Pirrò was partially supported by the EU Framework Programme for Research and Innovation under Grant Agreement No. 611242 (Sense4us). C. Gutierrez was supported by projects FONDECYT No. 1110287 and Millenium Nucleus CIWS, NC120004.
Authors' addresses: V. Fionda, Department of Mathematics and Computer Science, University of Calabria; email: fionda@mat.unical.it (contact author); G. Pirrò, WeST, University of Koblenz-Landau; email: pirro@uni-koblenz.de; C. Gutierrez, Computer Science Department, University of Chile; email: cgutierr@dcc.uchile.cl.

spread of the WLOD. On one hand, Linking Open Data[1] [Heath and Bizer 2011] aims at setting some principles for the publishing and interlinking of data on the Web by using the Resource Description Framework (RDF) [Klyne et al. 2004]. One key aspect of such project is the decentralized nature of data. On the other hand, applications like the Google Knowledge Graph (KG)[2] and the Facebook Graph (FG)[3] allow to access structured descriptions of entities from centralized gateways.

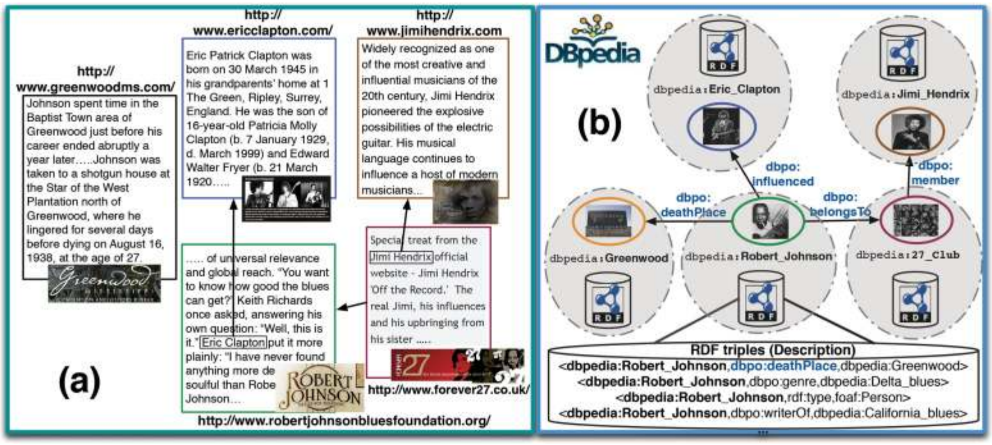
Although different in their goals, the common technical basis of all these approaches are large graphs that model structured information and emphasize the functionalities of *discovery* and *exploration* that are achieved by (manual) *navigation* from the entity (node in the graph) currently being visited toward semantically related entities, via labeled edges. The querying capabilities in these approaches do not enable one to go beyond asking "tell me friends living in my same city" or matching keywords to specific entities. It is not possible to perform simple information requests involving, for instance, the checking of paths, although questions like "find people up to three hops away from me that like music and live in my same city" sound very natural when dealing with graphs. Little flexibility is given to Web developers and users in terms of languages and tools capable of harnessing and putting together the power of the huge amount of structured data available in the WLOD [Weaver and Tarjan 2013].

In this article we introduce a formal graph navigational language for the WLOD called NautiLOD. It enables one to write navigational expressions reflecting conceptual specifications of both nodes and fragments (subgraphs) of interest in the WLOD. These formal specifications are expressed in a declarative way and enable one to navigate across datasources in the WLOD without human intervention. NautiLOD also introduces the functionality of specifying *actions*. Actions allow one to send alerts, mails, and open a whole new world of functionalities that contribute to the automation of information search on the WLOD. The technological underpinnings of our proposal are open and standard technologies such as the RDF data format and the SPARQL query language for RDF [Harris and Seaborne 2013]. In the remainder of this introductory section, we more formally present the WLOD; the notion of navigation on the Web, which takes advantage of the semantics of data on the Web; an overview of the general design of NautiLOD; and the structure of the article.

1.1. From the Web of Documents to the Web of Linked Data

The Web is usually modeled as a graph of links between pages [Brin and Page 1998]. This model has some peculiarities. First, links between pages are unlabeled and can only be expressed at one of the two endpoints.[4] For example, in Wikipedia a link from the HTML page about Rome (i.e., `wiki:Rome`[5]) to the page about Italy (i.e., `wiki:Italy`) can only be set in the page `wiki:Rome`. In the WLOD links can be set and be part of any datasource in the spirit of Tim Berners Lee's words "Anyone can say anything about any topic and publish it anywhere" [Berners-Lee 1998]. For instance, a link between the DBpedia *resources* Rome (i.e., `dbpedia:Rome`) and Italy (i.e., `dbpedia:Italy`) can be part of *any* RDF datasource. Indeed, an arbitrary datasource d identified by the URI `u` can contain the triple (`dbpedia:Rome`, `dbpo:country`, `dbpedia:Italy`). The preceding triple will be then available when dereferencing `u`. Note that differently from the case of HTML pages, now the link includes a label that specifies one of the possible semantic relations between `dbpedia:Rome` and `dbpedia:Italy`. Labels carry a semantic meaning

[1]http://www.w3.org/wiki/SweoIG/TaskForces/CommunityProjects/LinkingOpenData.
[2]http://www.google.com/insidesearch/features/search/knowledge.html.
[3]http://www.facebook.com/about/graphsearch.
[4]X-Link (http://www.w3.org/TR/xlink11/) was proposed to go beyond such limitation.
[5]The prefixes used in the article are those available at http://prefix.cc.


http://www.greenwoodms.com/
http://www.ericclapton.com/
http://www.jimihendrix.com
http://www.forever27.co.uk/
http://www.robertjohnsonbluesfoundation.org/
(a)
DBpedia
(b)
dbpedia:Eric_Clapton
dbpedia:Jimi_Hendrix
dbpedia:Greenwood
dbpedia:Robert_Johnson
dbpedia:27_Club
dbpo: influenced
dbpo: member
dbpo: deathPlace
dbpo: belongsTo
RDF
RDF triples (Description)

Johnson spent time in the Baptist Town area of Greenwood just before his career ended abruptly a year later.....Johnson was taken to a shotgun house at the Star of the West Plantation north of Greenwood, where he lingered for several days before dying on August 16, 1938, at the age of 27.

Eric Patrick Clapton was born on 30 March 1945 in his grandparents' home at 1 The Green, Ripley, Surrey, England. He was the son of 16-year-old Patricia Molly Clapton (b. 7 January 1929, d. March 1999) and Edward Walter Fryer (b. 21 March 1920.....

Widely recognized as one of the most creative and influential musicians of the 20th century, Jimi Hendrix pioneered the explosive possibilities of the electric guitar. His musical language continues to influence a host of modern musicians...

Special treat from the Jimi Hendrix official website - Jimi Hendrix 'Off the Record.' The real Jimi, his influences and his upbringing from his sister .....

..... of universal relevance and global reach. "You want to know how good the blues can get?" Keith Richards once asked, answering his own question: "Well, this is it." Eric Clapton put it more plainly: "I have never found anything more de soulful than Robe Johnson....

<dbpedia:Robert_Johnson,dbpo:deathPlace,dbpedia:Greenwood>
<dbpedia:Robert_Johnson,dbpo:genre,dbpedia:Delta_blues>
<dbpedia:Robert_Johnson,rdf:type,foaf:Person>
<dbpedia:Robert_Johnson,dbpo:writerOf,dbpedia:California_blues>

Fig. 1. Web of documents versus Web of Linked Data (WLOD).

as being part of conceptual specifications (e.g., ontologies, thesauri) used to model knowledge domains.

Second, although Web pages are created and kept distributively, their small size and lack of structure stimulated the idea to view searching and querying through single and centralized repositories. These repositories are built via crawlers that starting from a set of *seed* pages *navigate* the Web graph to reach Web pages whose content will be processed and stored. The WLOD can be seen as a *semantic* graph where nodes are autonomous datasources identified by URIs and maintaining sets of RDF triples some of which provide links toward other datasources. This model allows one to better express the distributed creation and maintenance of data, and the fact that the structure of the WLOD is provided by dynamic and distributed datasources. In particular, it reflects the fact that at each moment in time, and for each particular agent, the whole graph of data on the Web is unknown [Mendelzon et al. 1997].

Figure 1(a) shows an excerpt of the Web of documents with some HTML pages and (syntactic) links between them. For instance, the page about the R. Johnson Blues Foundation links to the (official) Web page about E. Clapton. Note that there is no link between the page about R. Johnson and the page about the city of Greenwood where he passed away. Figure 1(b) shows information about the same entities reported in Figure 1(a) taken from DBpedia, the counterpart of Wikipedia in the WLOD. Here, note the availability of structured information in RDF such as the triple that links R. Johnson to the city of Greenwood. The label of this link, that is, `dbpo:deathPlace` carries a semantic meaning as being formally defined in the DBpedia ontology.[6]

1.2. Semantic Navigation on the Web of Data

The availability of structured data in the form of graphs calls for adequate languages that go beyond keyword-based mechanisms. The traditional tool to get information or knowledge from structured data is a query language. In the world of RDF, there is a standard query language called SPARQL [Harris and Seaborne 2013]. Querying in SPARQL implies the access to one or more datasources to satisfy an information need. However, the language offers limited support in terms of capabilities related to (dynamic) "exploration" and "discovery" of datasources that characterize all the emerging graph applications. One fundamental ingredient toward this goal is the support for

[6]http://wiki.dbpedia.org/Ontology.

graph *navigation* functionalities. Generally speaking, navigation is the process of going, guided by some driving directions, from the known to the unknown in a given space. In the case of RDF graphs, the space is set by the graph topology (encoded by RDF triples) and navigation occurs by traversing edges. Consider, for instance, the discovery of knowledge about musicians influenced by R. Johnson in Figure 1(b); starting from the node R. Johnson in DBpedia (the known) it is possible to navigate toward the nodes of other musicians (the unknown) by traversing edges labeled as `dbpo:influenced` (the driving directions). SPARQL (via property paths [Harris and Seaborne 2013]) and other extensions (e.g., nSPARQL [Pérez et al. 2010]) offer graph navigational functionalities that restrict their scope to *local* RDF graphs; they offer little support in terms of navigation in the WLOD graph.

In the WLOD, navigation can go beyond the classical crawling that consists in traversing all the edges toward other nodes; it can be driven by some high-level semantic specification that encodes a reachability test, that is, the checking whether from a given node there exists a path (defined by considering edge labels) toward other nodes. In graph query languages, the specification is usually given by using regular expressions over the alphabet of edge labels. Reachability as well as other approaches to query information from graphs have been largely studied in graph data management [Angles and Gutierrez 2008; Wood 2012]; however, no proposal has been formalized for distributed graphs such as the WLOD. In this setting, neither querying nor navigation alone are enough; navigation can actually be complemented with querying. Navigation is necessary since the topology of the space of datasources to be queried is (from a practical point of view) unbounded, not known in its entirety, and dynamic. Querying is important since each node in the WLOD is an RDF datasource that can be queried upon to filter and drive the subsequent steps of the navigation. The idea developed in this article is to have a navigational language that makes usage of querying to both (i) drive the navigation by filtering datasources according to the pieces of information they store and (ii) dynamically retrieve data from datasources encountered during the navigation and use such data to perform some basic actions (e.g., send notification messages). The navigation model proposed in this article has the unique feature of enabling the retrieval of Web fragments, that is, graphs including nodes and edges visited when evaluating an expression. This goes beyond the classical navigational model focused on retrieving nodes. These considerations are at the basis of our proposal for a navigational language for the WLOD.

1.3. The NautiLOD Language: An Overview

We see a navigational language for the WLOD graph as a way of providing instructions in the form of navigational expressions. Such expressions enable one to navigate from a given node toward nodes of interest by adjusting the navigation according to dynamically discovered edges and nodes. Our desideratum is to define a simple language that can help developers and users in performing at least the following basic tasks in a declarative and integrated manner: (i) specification of driving directions, that is, semantic descriptions of routes allowing one to traverse and retrieve nodes and fragments of the WLOD; (ii) semiautomatic navigation across datasources "driven" by locally found information; (iii) specification of actions to be performed over the data.

This article presents a formal language for the WLOD that we call NautiLOD (Navigational Language for Linked Open Data). It is based on regular expressions over RDF predicates plus tests using Boolean SPARQL queries performed over RDF datasources. Regular expressions enable the posing of complex information needs involving navigation along the nodes of the WLOD graph, while tests enable the selection and filtering of relevant datasources from which to continue the navigation. NautiLOD also features a mechanism to specify actions (e.g., send notification messages) that may

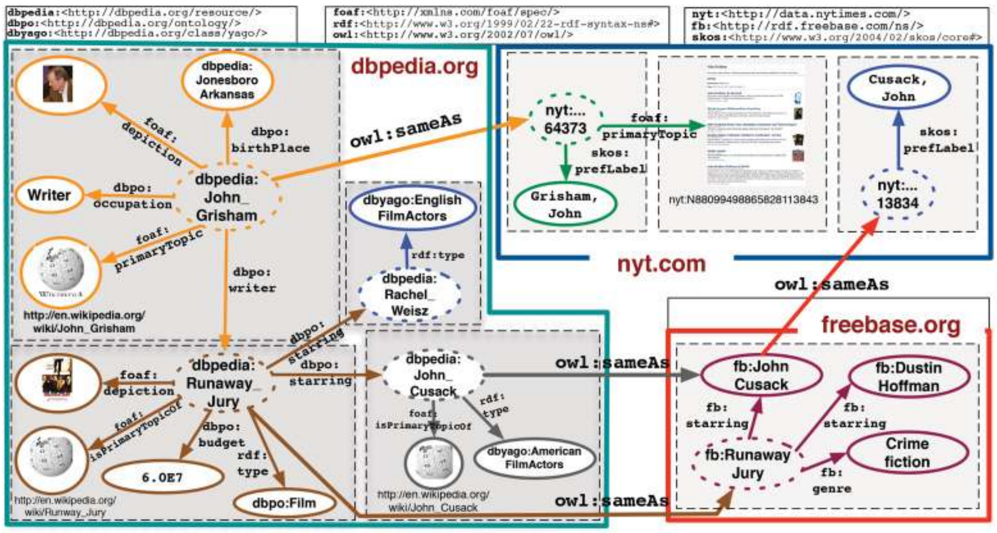

dbpedia:<http://dbpedia.org/resource/>
dbpo:<http://dbpedia.org/ontology/>
dbyago:<http://dbpedia.org/class/yago/>
foaf:<http://xmlns.com/foaf/spec/>
rdf:<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
owl:<http://www.w3.org/2002/07/owl/>
nyt:<http://data.nytimes.com/>
fb:<http://rdf.freebase.com/ns/>
skos:<http://www.w3.org/2004/02/skos/core#>
dbpedia.org
nyt.com
freebase.org
dbpedia:Jonesboro Arkansas
dbpo:birthPlace
foaf:depiction
dbpedia:John_Grisham
dbpo:occupation
Writer
foaf:primaryTopic
http://en.wikipedia.org/wiki/John_Grisham
dbpo:writer
owl:sameAs
dbyago:EnglishFilmActors
rdf:type
dbpedia:Rachel_Weisz
dbpo:starring
dbpedia:Runaway_Jury
dbpo:budget
6.0E7
rdf:type
dbpo:Film
foaf:isPrimaryTopicOf
http://en.wikipedia.org/wiki/Runaway_Jury
dbpedia:John_Cusack
dbyago:AmericanFilmActors
http://en.wikipedia.org/wiki/John_Cusack
nyt:...64373
foaf:primaryTopic
skos:prefLabel
Grisham, John
nyt:N88099498865828113843
Cusack, John
nyt:...13834
fb:John Cusack
fb:Dustin Hoffman
fb:starring
fb:Runaway Jury
Crime fiction
fb:genre

Fig. 2. An excerpt of data that can be navigated from `dbp:John_Grisham`.

use data encountered during the navigation. Although there exist several languages to process and query RDF data, the type of dynamic and open high-level specifications featured by NAUTILOD cannot be systematically simulated by these languages (see Related Work). Some approaches enhance SPARQL with navigational features (e.g., Alkhateeb et al. [2009], Pérez et al. [2010], and Harris and Seaborne [2013]) over fixed sets of datasources (typically a single RDF graph) but do not address Web scale navigation. Crawlers like LDSpider [Isele et al. 2010] offer limited semantic control over the navigation; in other words, it is not possible to select only (a subset of) relevant datasources. NAUTILOD shares some features with approaches that extend the scope of SPARQL queries over the WLOD [Hartig 2011; Umbrich et al. 2014]. Here, the traversal of edges is introduced as an extension of the SPARQL semantics, thus limiting explicit control over the navigation. Finally, none of the approaches mentioned previously incorporates the declarative specification of actions to be triggered over datasources.

1.4. NAUTILOD by Example

The tenet of our proposal is a language to provide high-level specifications to drive the navigation in the WLOD graph toward precise destinations. To show the potentialities of NAUTILOD we present some examples using the excerpt of real-world data shown in Figure 2. The formal syntax and semantics are introduced in Section 3. For the sake of space we will use prefixes, instead of full URIs, as defined at http://prefix.cc.

Example 1.1 (*Aliases via* `owl:sameAs`). Specify and retrieve documents associated with John Grisham in DBpedia and his possible aliases in other datasources.

In this example, the idea is to consider `owl:sameAs`-paths that originate from Grisham's URI in DBpedia, the starting point of the navigation in the WLOD graph. The predicate `owl:sameAs` states that two resources of the WLOD represent the same entity. Recursively, for each URI `u` reached, check in its corresponding datasource triples of the form (`u`, `owl:sameAs`, `v`); select all `v`'s found. Finally, for each such `v`, return all URIs `w` in triples of the form (`v`, `foaf:primaryTopic`, `w`). The specification can be given via the following NAUTILOD expression:

```
dbpedia:John_Grisham (owl:sameAs)*/foaf:primaryTopic
```

In Figure 2, when evaluating this expression starting from the URI `dbpedia:John_Grisham` we get all the different representations of Grisham provided by `dbpedia.org` and `nyt.com`. From these nodes, the expression `foaf:primaryTopic` is evaluated. The results of the evaluation is the set {`wiki:John_Grisham`, `nyt:N88099498865828113843`}. Note that the search for documents associated to Grisham, if restricted only to DBpedia, would not have allowed one to include documents available in the *New York Times* datasource. The high-level specification given in this expression underlines the main feature of NautiLOD: the capability to deal with distributed and dynamically discovered datasources.

We would like to point out that there is a substantial difference between NautiLOD and other approaches (e.g., crawlers like LDSPider [Isele et al. 2010]) that could retrieve portions of the WLOD by "crawling" all the data. It consists in the fact that NautiLOD's focus is on providing a declarative way to express driving directions that enable one to locate precisely and selectively specific parts of the WLOD graph. Moreover, NautiLOD goes beyond the scope of current navigational languages (e.g., SPARQL property paths, nSPARQL [Pérez et al. 2010], PSPARQL [Alkhateeb et al. 2009], RPL [Zauner et al. 2010]); while these languages are meant to be evaluated over *local* graphs NautiLOD targets navigation on the WLOD graph. A more complex example, which extends the scope of current navigational languages with capabilities for specifying actions and cannot be simulated by any of the existing proposals is as follows:

Example 1.2. Specify American actors (and their aliases) that have played in at least one movie based on a book written by J. Grisham. Send by email the Wiki pages of such actors.

This request involves paths of the form `dbpo:writer/dbpo:starring` and aliases as in the previous example; tests (expressed in NautiLOD using ASK-SPARQL queries) over the datasource associated with a given URI (if somebody that played in a movie written by J. Grisham is found, check if s/he is an American actress/actor); and actions to be performed using data from the datasource. The specification in NautiLOD is

```
dbpedia:John_Grisham  dbpo:writer/dbpo:starring[Test]/ACT[Act]/(owl:sameAs)*
```

where the test and the action are specified as follows:

```
Test=ASK {FILTER EXISTS{?ctx rdf:type dbyago:AmericanFilmActors}}
Act=sdEmail("x@y.z","SELECT ?p WHERE {?ctx foaf:isPrimaryTopicOf ?p.}")
```

The traversal of the `dbpo:writer` predicate enables one to reach movies whose script has been written by J. Grisham; whence actors of these movies are reached via the predicate `dbpo:starring`. The (implicit) variable `?ctx` in `Act` is bound to the set of URIs dereferenced at the current step; actors that played in a movie written by J. Grisham in this case. At this point, the test on the nationality of such actors via the ASK SPARQL query enables one to rule out actors that are not American. This filter leaves in this case only `dbpedia:John_Cusack`. Over the elements of this set (one element in this case), the action will send via email the Wiki page (obtained via the SELECT query). The action `sdEmail`, implemented by an ad hoc programming procedure, does not influence the navigation process. Thus, the evaluation will continue from the URI `dbpedia:John_Cusack`, by traversing the edge `owl:sameAs` (found in the set of triples obtained by dereferencing `dbpedia:John_Cusack`), similarly to what is already seen in

Example 1.1. The final result of the evaluation is (i) the set {dbpedia:John_Cusack, fb:John_Cusack, nyt:...13834}, that is, the URIs identifying John Cusack in dbpedia.org, freebase.org and nyt.com; (ii) the set of actions performed—in this case one email sent. The previous example underlines two of the main features of NautiLOD. First, it enables via tests to declaratively specify routes in the WLOD graph that have to be explored. Second, navigation can be complemented with high-level specification of actions to be performed over data. The combination of these two features is a powerful support for Web developers toward building applications that consume structured data on the WLOD.

We are now ready to show an example of how with NautiLOD it is possible not only to specify a set of nodes conforming to an expression, but also to keep information about the Web fragment where these nodes are located. Connectivity information available in Web fragments is crucial in several application scenarios such as social networks and citation networks. On the Web, connectivity is also useful to track provenance, that is, reconstructing the path that led to a particular datasource.

Example 1.3 (*Capturing Web Fragments*). Specify the coauthor network of Tim Berners-Lee (TBL) by considering coauthorship relations on papers published between 2010 and 2013 only. The specification in NautiLOD is

```
dblp:Tim_Berners-Lee (foaf:maker[Test]/foaf:maker)*
```

where the test is defined as follows:

```
Test=ASK {?ctx dc:issued ?y. FILTER( ?y>‘‘2010’’^^<xsd:gYear>).
                FILTER(?y<‘‘2013’’^^<xsd:gYear>).}
```

The purpose of Example 1.3 is that of constructing a *network* starting from the URI of TBL in the RDF version of the DBLP bibliography database.[7] Hence, the goal is twofold: (i) identifying nodes (coauthors of TBL) that satisfy the expression; an (ii) keeping track of the connections among these nodes. As we will discuss in Section 4, this request goes beyond the scope of current navigational languages (e.g., SPARQL property paths [Harris and Seaborne 2013] and nSPARQL [Pérez et al. 2010]) that mainly focus on identifying nodes.

In terms of expressiveness, we want to point out that SPARQL property paths cannot express Example 1.3 since it lacks capabilities to perform tests à la XPath [Clark and DeRose 1999]; in other words, it is not possible while evaluating a path to check conditions over the nodes (in this case the year of the publication) encountered in the path. An excerpt of the coauthor network for Example 1.3 is shown in Figure 3. To provide a user-friendly visualization, the swget tool implementing the NautiLOD language features a Graphical User Interface (GUI) with a rich set of functionalities to zoom and explore Web fragments and search for specific nodes and edges.

1.5. Contributions

The following are the main contributions of this article:

(1) We present a navigational language for the WLOD graph called NautiLOD. It provides a unique combination of navigational and querying features and incorporates a declarative mechanism to command actions over data.

[7]http://dblp.l3s.de.

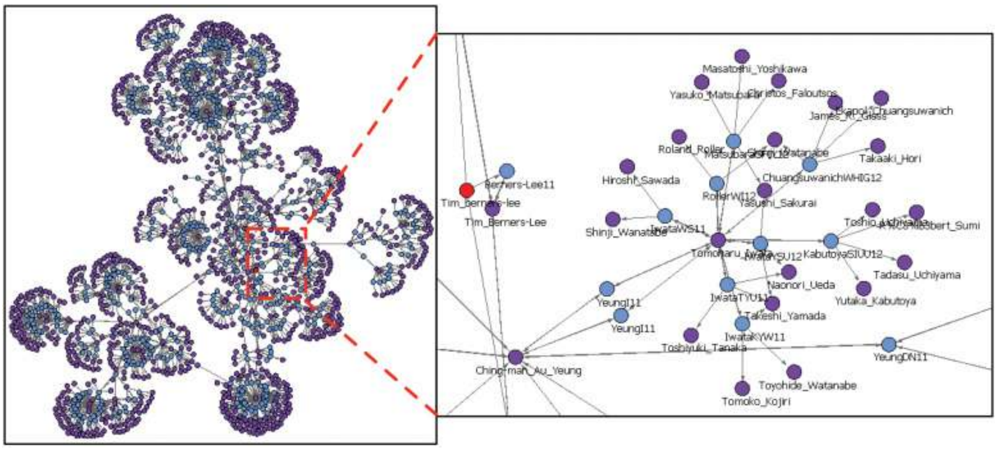


Fig. 3. The fragment of the WLOD retrieved for Example 1.3.

(2) We define a simple syntax and a formal semantics that associates with a NautiLOD expression the set of nodes (i.e., URIs) in the WLOD graph that satisfy the expression plus the set of actions performed.
(3) We formalize an enhancement of the semantics of NautiLOD to capture Web fragments, that is, graphs composed by nodes visited when evaluating expressions along with edges traversed to reach these nodes.
(4) We provide algorithms and study the complexity of evaluating NautiLOD expressions according to both semantics.
(5) We discuss an implementation of the language in the `swget` tool, which is readily available on the WLOD. `swget` is available as an Application Programming Interface (API), a GUI, and a Web portal.
(6) We analyze how the different features (i.e., navigation, tests, actions) affect the cost of the evaluation of NautiLOD expressions.
(7) We compare NautiLOD (and `swget`) with the state of the art.

Some of the ideas presented in this article appeared in proceedings of the WWW2012 conference [Fionda et al. 2012]. This article considerably expands our previous work in the following respects:

(1) We deepen the motivations underlying the NautiLOD language (also in the light of recent proposals such as the Google Knowledge Graph and Facebook Graph) and improve its presentation.
(2) We introduce two new formal semantics for the NautiLOD language that allow one to capture Web fragments.
(3) We present algorithms for evaluating NautiLOD expressions according to the new semantics, study their complexity and discuss their implementation.
(4) We perform a detailed evaluation of `swget` and compare it with related research.
(5) We discuss examples of Web fragments with real data.

1.6. Organization of the Article

Section 2 provides some background. Section 3 introduces the NautiLOD language; Here the syntax and the semantics returning sets of nodes are introduced. Section 4 discusses the new semantics of NautiLOD capturing Web fragments. Section 5 presents an implementation of NautiLOD in the `swget` tool. An experimental evaluation of `swget` is discussed in Section 6. Section 7 discusses related research. Finally, in Section 8 and Section 9 we draw some conclusions.

2. PRELIMINARIES

This section provides some background on the key notions underlying our proposal.

2.1. RDF and the Linking Open Data Project

The Resource Description Framework (RDF) is a metadata model introduced by the W3C for representing information about Web resources. RDF leverages Uniform Resource Identifiers (URIs) to identify resources; URIs represent global identifiers in the Web and enable access to the *descriptions* of resources according to specific protocols (e.g., HTTP). RDF builds upon the notion of *statement*. A statement defines the *property* p holding between two resources, the *subject* s and the *object* o. A statement is denoted by (s, p, o), and thus called *triple* in RDF. As an example, in Figure 2 (`dbpedia:John_Grisham`, `dbpo:writer`, `dbpedia:Runaway_Jury`) is an RDF triple. RDF triples make usage of vocabularies (ontologies) that model domains of interest. In the previous triple, the predicate `dbpo:writer` is defined in the DBpedia ontology and expresses the relation between a person and a piece of work. A collection of triples is referred to as *RDF graph*. As discussed in Section 1.1, the availability of structured information (in RDF) enables a new (semantic) space where objects are *linked* and looked-up by using (Semantic) Web languages and technologies. This space can be thought of as a *giant global graph* [Berners-Lee 2006] that we will refer to as the Web of Linked Data (WLOD). In order to publish and interlink data in the WLOD, the Linking Open Data (LOD) project [Berners-Lee 2006] sets some informal principles:

(1) Real-world objects or abstract concepts must be assigned names in the form of URIs.
(2) HTTP URIs have to be used so that people can look them up by using existing technologies.
(3) When someone looks up a URI, associated information has to be provided in a standard form (e.g., RDF).
(4) Interconnections among URIs have to be provided via references to other URIs.

A central notion in this context is that of *dereferenceable* URI, that is, a URI that when looked up (via an HTTP GET) provides the representation of the resource it identifies in a standard data format (e.g., RDF). Data in the WLOD are provided by publishers that maintain thousands of interlinked datasets containing billions of facts covering diverse domains, such as general knowledge provided by the *New York Times*, DBpedia, and Freebase/Google; music provided by the BBC; science provided by *Nature*; geospatial information provided by Geonames and OpenStreetMap; or public-sector information provided by the US and UK governments and European agencies.

Despite the huge availability, there is still scant consumption of this enormous wealth of structured, freely available information. Our proposal of a navigational language for the WLOD is meant to help in filling this gap by providing developers and Web users with a mechanism to specify, retrieve, and act over structured information on the Web.

2.2. Data Model

This section introduces a minimal abstract model of the WLOD that highlights the main features required in the subsequent discussion. Let $\mathcal{U}$ be the set of all URIs and $\mathcal{L}$ the set of literals, that is, strings or special datatypes. We distinguish between two types of RDF triples. *RDF links* $(s, p, o) \in \mathcal{U} \times \mathcal{U} \times \mathcal{U}$ that encode connections among resources in the WLOD and *literal triples* $(s, p, o) \in \mathcal{U} \times \mathcal{U} \times \mathcal{L}$ that are used to state descriptions or features of the resource identified by the subject s. Note that

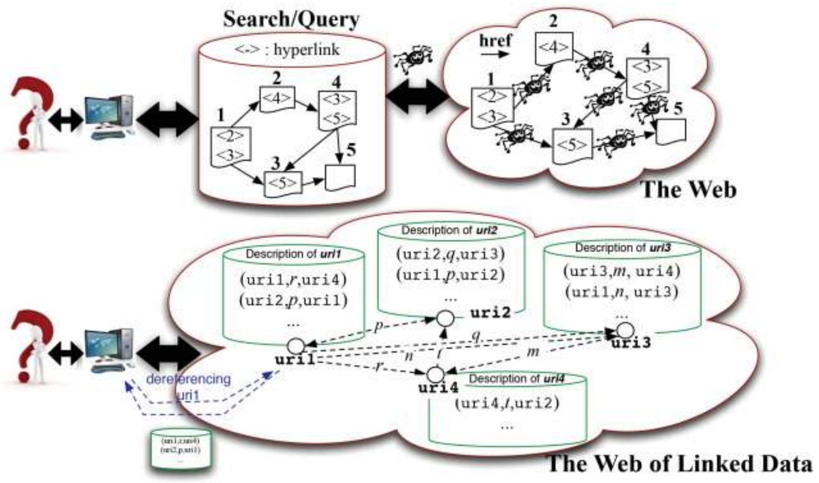


Fig. 4. The Web vs. the Web of Linked Data.

the subject/object of a triple, in the general case, can also be a blank node, which for most purposes could be thought of as an existential variable. However, here we will not consider them (note also that the usage of blank nodes is discouraged [Heath and Bizer 2011]). The following three notions will be fundamental.

Definition 2.1 (*Web of Linked Data* $\mathcal{T}$). Let $\mathcal{U}$ and $\mathcal{L}$ be as described previously (infinite sets). The Web of Linked Data (over $\mathcal{U}$ and $\mathcal{L}$) is the set of triples (s, p, o) in $\mathcal{U} \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{L})$. We will denote it by $\mathcal{T}$.

Definition 2.2 (*Description Function* $\mathcal{D}$). The description function $\mathcal{D} : \mathcal{U} \rightarrow \mathcal{P}(\mathcal{T})$ associates with each URI $u \in \mathcal{U}$ a subset of triples of $\mathcal{T}$. By $\mathcal{D}(u)$, we denote the set of triples obtained by dereferencing u.

Definition 2.3 (*Web of Linked Data Instance* $\mathcal{W}$). A WLOD instance is a pair $\mathcal{W}$=$\langle\mathcal{U}, \mathcal{D}\rangle$, where $\mathcal{U}$ is the set of all URIs and $\mathcal{D}$ is a description function.

In a WLOD instance what matters are those $\mathtt{u} \in \mathcal{U}$ for which $\mathcal{D}(\mathtt{u}) \neq \emptyset$. Note that the description function $\mathcal{D}$ could return the empty set for some u (e.g., if u is not dereferenceable). To give a concrete example of how the description function models the process of dereferencing, consider the URI `fb:Runaway_Jury` in `freebase.org` shown Figure 2. We have that $\mathcal{D}$(`fb:Runaway_Jury`) returns the set of triples {(`fb:Runaway_Jury, fb:starring, fb:John_Cusack`), (`fb:Runaway_Jury, fb:starring, fb:Dustin_Hoffman`), (`fb:Runaway_Jury, fb:genre,‘‘Crime fiction’’`)}.

From now on we will denote a WLOD *instance* simply via WLOD. Figure 4 provides a pictorial representation of the Web of documents and the WLOD. The Web is traditionally modeled as a graph of syntactic links between pages that maintain unstructured information. The traditional way of accessing information in such graph is via crawlers. The WLOD can be represented as a set of nodes plus data describing their semantic structure attached to each node and their interlinks. Effectively, a node represents a datasource (set of RDF triples) identified by a URI with links to other datasources. These new features enable one to evolve crawling toward more sophisticated forms of semantic navigation that can be specified via formal languages such as the NautiLOD language introduced in this article.

3. NAUTILOD: SEMANTIC NAVIGATION ON THE WEB OF LINKED DATA

This section presents the NautiLOD (*Navigational Language for Linked Open Data*) language. The first goal of NautiLOD is to enable the declarative specification of nodes of the WLOD graph by leveraging semantic information available in both nodes and edges. NautiLOD incorporates two main features. First, given a high-level conceptual specification of some semantic nodes in the WLOD graph (e.g., Italian directors influenced by S. Kubrick), it provides a way to drive the navigation toward the relevant places where to find them. Second, NautiLOD enables one to command actions over data encountered during the navigation (e.g., send via email the home pages of all the French directors encountered). Although there have been several proposals for accessing data in the WLOD (see Section 7), they differ from our proposal in the following main respects: (i) they treat navigation as a second-order citizen in the sense that such proposals do not feature either explicit constructs to perform semantic navigation on the WLOD or functionalities to retrieve *fragments* of it; (ii) none of them incorporate mechanisms to combine navigation and actions over data. The formalization of NautiLOD has been inspired by two nonrelated proposals, that is, `wget`, a tool to automatically navigate and retrieve Web pages, and XPath [Clark and DeRose 1999], a language to specify parts of a document in the world of semistructured data.

3.1. Syntax of NautiLOD

The syntax of NautiLOD is defined according to the grammar reported in the following box. The navigational core of NautiLOD is based on regular path expressions [Wood 2012], similarly to Web query languages (e.g., [Mendelzon et al. 1997; Abiteboul and Vianu 1997]) and XPath [Clark and DeRose 1999]. The navigation in the WLOD can be controlled via existential tests in the form of ASK-SPARQL queries [Harris and Seaborne 2013]. This mechanism allows one to redirect the navigation based on the information present at each node of the navigational path. The language also allows one to command actions during the navigation according to decisions based on the original specification and data available in the datasources (nodes in the WLOD) visited.

```
  path ::=pred | pred^ | action | path/path |path* |
          path|path | path[test] | path⟨l-h⟩
  pred ::=RDF predicate | ⟨_⟩
  test ::=ASK-SPARQL query
action ::=ACT[procedure(target, SELECT-SPARQL query)]
```

NautiLOD is based on *Path Expressions*; it accepts concatenations of basic and complex types of expressions. Basic expressions are predicates (`path`) and actions (`action`); complex expressions are concatenations and disjunctions of expressions; expressions involving repetitions using the features of regular languages [Hopcroft et al. 2000] (i.e., the Kleene * operator corresponding to zero or more repetitions and the ⟨l-h⟩ operator meaning at least l and at most h repetitions); and expressions followed by a `test`. The building blocks of an expression reflect the classical functionalities of navigational languages as described in what follows:

(1) *Predicates:* `path` is an RDF predicate or the wild card <_> denoting *any* predicate.
(2) *Test Expressions:* A `test` is given in the form of an ASK-SPARQL query used to perform an existential test at a datasource (node) in the WLOD.

NautiLOD expressions also include the additional feature of actions:

(1) *Action Expressions:* An `action` is a procedural specification of a command (e.g., send a notification message). Actions neither influence the navigational process nor change the underlying data.

Let us elaborate a bit more on the support of actions. NautiLOD expressions enable one to specify certain basic types of actions, more in the line of side effects than updates. Indeed, in addition to computing the result (set of nodes or Web fragments) an expression (via actions) can act on the outside world (e.g., users or devices). It is important to point out that the kind of actions supported by NautiLOD are not meant to transform datasources (e.g., as in Abiteboul et al. [2002]); their main purpose is to play the role of *metacommunication*, which means enabling one to perform basic communications (e.g., sending emails with some data). Parameters values of an action are obtained from the datasource (node) reached during the navigation via a SELECT-SPARQL query. Moreover, the special parameter *target* is specified (by the user) to indicate the communication channel (e.g., an email address, a mobile phone number, a printer name).

A scenario where actions result useful is when one not only is interested in retrieving particular pieces of information (e.g., her coauthors, her coauthor network) but also in receiving additional information that has been "encountered" during the evaluation and is not part of the result. An example of such reasoning is the list of nationalities of coauthors that can be obtained when building a coauthor network where only the names of coauthors (and their links) are specified to be part of the network retrieved.

If restricted to (1) and (2), NautiLOD can be seen as a declarative language to specify sets of datasources (nodes) or fragments of the WLOD conform to some semantic specification. We now introduce the formal semantics of NautiLOD capturing nodes.

3.2. NautiLOD Semantics Returning Set of Nodes

NautiLOD expressions are evaluated against a WLOD instance $\mathcal{W}=\langle\mathcal{U},\mathcal{D}\rangle$ from a `seed` node (i.e., a URI `u`) that represents the starting point of the navigation in the WLOD graph. The evaluation of an expression produces the set of nodes of the WLOD that are reachable from the `seed` via paths satisfying the expression, plus the set of actions triggered during its evaluation. The fragment of the language without actions follows the lines of formalization of XPath by Wadler [1999]. The semantics of NautiLOD is reported in Table I, where $\mathcal{W}$ is omitted for the sake of conciseness. The semantics has the following modules:

—$Sem[\![\texttt{path}]\!](\texttt{u})$: This function takes as input a `path` (i.e., an expression defined according to the the syntax in Section 3.1) and a URI `u`. The function gives as output the ordered pair of two sets: the set of URIs reached by the evaluation of `path` and the set of actions performed during the evaluation. The function *Sem* is the first to be called when evaluating a NautiLOD expression. It uses the function U to obtain the set of URIs while the set of actions performed is obtained via the function E_A.

—$U[\![\texttt{path}]\!](\texttt{u})$: This function takes as input a `path`, which is evaluated starting from the URI `u` in the WLOD instance $\mathcal{W}$, and gives as output the set of URIs reachable via sequences of edges satisfying `path`. In Table I, rules R2–R10 describe the result produced by U when considering the different types of `path` that can be defined according to the NautiLOD syntax reported in Section 3.1.

—$E_A[\![\texttt{path}]\!](\texttt{u})$: This function is responsible for the execution of the actions generated during the evaluation of a `path` starting from the URI `u`. In Table I, rule R11 shows that the function E_A calls an additional function $\texttt{Exec}(a, D(\texttt{u}))$. The function `Exec` in

Table I. Semantics of NAUTILOD returning nodes. The rules reflect all the possible forms of syntactic expressions that can be given according to the NAUTILOD syntax in Section 3.1. The evaluation of an expression occurs by calling the function *Sem* (rule R1). It produces two sets: (i) the set of URIs of $\mathcal{W}$ satisfying the expression, obtained via the function U and (ii) the set of performed actions obtained, via the function E_A, when evaluating the expression. $\texttt{Exec}(a, \mathcal{D}(\mathrm{u}))$ (rule R11) denotes the execution of action a over the datasource identified by u

R1	$Sem[\![\texttt{path}]\!](\mathrm{u}, \mathcal{W})$	=	$(U[\![\texttt{path}]\!](\mathrm{u}), E_A[\![\texttt{path}]\!](\mathrm{u}))$
R2	$U[\![\texttt{p}]\!](\mathrm{u})$	=	$\{\mathrm{u}' \mid (\mathrm{u}, \texttt{p}, \mathrm{u}') \in \mathcal{D}(\mathrm{u})\}$
R3	$U[\![\texttt{p\^{}}]\!](\mathrm{u})$	=	$\{\mathrm{u}' \mid (\mathrm{u}', \texttt{p}, \mathrm{u}) \in \mathcal{D}(\mathrm{u})\}$
R4	$U[\![\texttt{<\_>}]\!](\mathrm{u})$	=	$\{\mathrm{u}' \mid \exists \texttt{p}, (\mathrm{u}, \texttt{p}, \mathrm{u}') \in \mathcal{D}(\mathrm{u})\}$
R5	$U[\![\texttt{action}]\!](\mathrm{u})$	=	$\{\mathrm{u}\}$
R6	$U[\![\texttt{path}_1/\texttt{path}_2]\!](\mathrm{u})$	=	$\{\mathrm{u}'' \in U[\![\texttt{path}_2]\!](\mathrm{u}') : \mathrm{u}' \in U[\![\texttt{path}_1]\!](\mathrm{u})\}$
R7	$U[\![(\texttt{path})*]\!](\mathrm{u})$	=	$\{\mathrm{u}\} \cup \bigcup_1^\infty U[\![\texttt{path}_i]\!](\mathrm{u}) : \texttt{path}_1 = \texttt{path} \wedge \texttt{path}_i = \texttt{path}_{i-1}/\texttt{path}$
R8	$U[\![(\texttt{path})\langle l\text{-}h\rangle]\!](\mathrm{u})$	=	$\{\mathrm{u}\} \cup \bigcup_l^h U[\![\texttt{path}_i]\!](\mathrm{u}) : \texttt{path}_1 = \texttt{path} \wedge \texttt{path}_i = \texttt{path}_{i-1}/\texttt{path}$
R9	$U[\![\texttt{path}_1 \vert \texttt{path}_2]\!](\mathrm{u})$	=	$U[\![\texttt{path}_1]\!](\mathrm{u}) \cup U[\![\texttt{path}_2]\!](\mathrm{u})$
R10	$U[\![\texttt{path[test]}]\!](\mathrm{u})$	=	$\{\mathrm{u}' \in U[\![\texttt{path}]\!](\mathrm{u}) : \texttt{test}(\mathrm{u}') = \textbf{true}\}$
R11	$E_A[\![\texttt{path}]\!](\mathrm{u})$	=	$\{\texttt{Exec}(a, \mathcal{D}(\mathrm{u})) : (\mathrm{u}, a) \in A[\![\texttt{path}]\!](\mathrm{u})\}$
R12	$A[\![\texttt{p}]\!](\mathrm{u})$	=	$\emptyset$
R13	$A[\![\texttt{p\^{}}]\!](\mathrm{u})$	=	$\emptyset$
R14	$A[\![\texttt{<\_>}]\!](\mathrm{u})$	=	$\emptyset$
R15	$A[\![\texttt{action}]\!](\mathrm{u})$	=	$\{(\mathrm{u}, \texttt{action})\}$
R16	$A[\![\texttt{path}_1/\texttt{path}_2]\!](\mathrm{u})$	=	$A[\![\texttt{path}_1]\!](\mathrm{u}) \cup \bigcup_{\mathrm{u}' \in U[\![\texttt{path}_1]\!](\mathrm{u})} A[\![\texttt{path}_2]\!](\mathrm{u}')$
R17	$A[\![(\texttt{path})*]\!](\mathrm{u})$	=	$\bigcup_1^\infty A[\![\texttt{path}_i]\!](\mathrm{u}) : \texttt{path}_1 = \texttt{path} \wedge \texttt{path}_i = \texttt{path}_{i-1}/\texttt{path}$
R18	$A[\![(\texttt{path})\langle l\text{-}h\rangle]\!](\mathrm{u})$	=	$\bigcup_l^h A[\![\texttt{path}_i]\!](\mathrm{u}) : \texttt{path}_1 = \texttt{path} \wedge \texttt{path}_i = \texttt{path}_{i-1}/\texttt{path}$
R19	$A[\![\texttt{path}_1 \vert \texttt{path}_2]\!](\mathrm{u})$	=	$A[\![\texttt{path}_1]\!](\mathrm{u}) \cup A[\![\texttt{path}_2]\!](\mathrm{u})$
R20	$A[\![\texttt{path[test]}]\!](\mathrm{u})$	=	$A[\![\texttt{path}]\!](\mathrm{u})$

rule R11 takes two parameters: an action a of the form `procedure(target, SELECT-SPARQL query)` and the description of a datasource. `Exec` proceeds as follows: runs the SELECT-SPARQL query over $\mathcal{D}(\mathrm{u})$ and communicates the results obtained to the *target*. We again want to emphasize that actions are side effects and do not update datasources. Note also that rule R11 calls the function A to obtain the pairs (u, a) to be given as input to `Exec`.

—$A[\![\texttt{path}]\!](\mathrm{u})$: This function takes as input the different types of `path` (see Section 3.1) and gives as output the set of actions associated with the URIs visited during the evaluation of `path` starting from the URI u. Rules R12–R20 in Table I cover all the possible cases. Note that not all the types of `path` produce actions; indeed, rules R12–R14 return the empty set $\emptyset$ since no action is syntactically specified. On the other hand, rule R15 is the particular type of `path` that returns an action. For collecting the actions inside a complex path, rules R16–R20 are used.

3.3. Examples of Expressions Returning Nodes

To clarify the semantics of NAUTILOD, we now discuss some examples. Consider the expression consisting in the sole predicate `rdf:type` evaluated starting from the URI u. The evaluation starts by applying rule R1 in Table I. Such rule requires one to use the function U (to obtain the set of URIs) and the function E_A (to trigger the set of actions). The function U uses rule R2 since it matches the syntactic expression in input, which consists of the predicate `rdf:type`. Rule R2 returns the set of URIs reachable from u by traversing edges labeled as `rdf:type`. This is done by inspecting triples of the form $(\mathrm{u}, \texttt{rdf:type}, \mathrm{u}')$ included in $\mathcal{D}(\mathrm{u})$. As for actions, rule R1 calls rule R11, which in its turn calls rule R12; even in this case rule R12 is the only rule that matches the expression in input (i.e., `rdf:type`). Note that since no actions have been specified, rule R12 returns the empty set.

Consider now the evaluation of the expression `rdf:type[q]`, which includes a test `q` (an ASK-SPARQL query). The first part is similar to the previous example; that is, it uses rule R1 and then rules R10 and R2. Rule R2 evaluates `rdf:type` as discussed previously and returns a set of URIs. This set is filtered by rule R10, which is used because it matches the syntactic expression in input containing the test [q]. The filtering performed by using rule R10 discards those URIs u′ obtained via rule R2 for which the ASK query q evaluated on their descriptions $\mathcal{D}(\mathrm{u}')$ returns false. Also in this example the set of actions is empty.

Finally, consider a more complex expression `rdf:type[q]/a`, which also includes the specification of an action a. The evaluation starts again by considering the rule R1. The set of URIs that satisfy the expression is obtained by applying the rules R6, R10, R2, and R5. The subexpression `rdf:type[q]` is evaluated as previously discussed (rules R10 and R2). The evaluation of the action a is done by using rule R5, which simply returns the whole set of URIs obtained from the evaluation of the previous subexpression. This behavior points out how actions do not interfere with the navigation.

The evaluation of actions occurs by applying the rules R16, R20, R12, and R15, and their execution is managed via rule R11. By looking at Table I, it can be noted that before applying rule R15 the set of actions is the empty set. Then, according to rule R15 the action a is executed for each URI in the evaluation of `rdf:type[q]`. Overall, the evaluation of the expression `rdf:type[q]/a` consists in (i) the set of URIs obtained by evaluating `rdf:type[q]` *and* (ii) the action a performed on each URI u′ belonging to the previous set (possibly using some data from $\mathcal{D}(\mathrm{u}')$). We are now ready to introduce an automaton-based algorithm for the evaluation of NAUTILOD expressions.

3.4. Evaluating NAUTILOD expressions: Algorithms and Complexity

This section describes an automaton-based algorithm for the evaluation of NAUTILOD expressions according to the semantics shown in Table I. The usage of automata has a twofold benefit. First, since NAUTILOD is based on regular expressions, for each NAUTILOD expression e we can construct the Nondeterministic Finite-state Automaton (NFA) [Hopcroft et al. 2000] that recognizes strings belonging to the language defined by e. Second, the NFA associated with an expression can be given an intuitive graphical representation, which simplifies the presentation of the algorithm. Before getting into the technicalities, we give a high-level overview of the algorithm.

Let e be a NAUTILOD expression and $\mathcal{W}$ a WLOD instance. Given the automaton $\mathcal{A}_e$ associated with an expression e, the evaluation algorithm builds the product $\mathcal{W} \times \mathcal{A}_e$ between the automaton $\mathcal{A}_e$ and the data graph (the WLOD instance $\mathcal{W}$). Note that since the data graph $\mathcal{W}$ is not available in its entirety, $\mathcal{W} \times \mathcal{A}_e$ has to be built incrementally while traversing $\mathcal{W}$. The result of the evaluation of a NAUTILOD expression e is the set of nodes in $\mathcal{W} \times \mathcal{A}_e$ marked with a final state of $\mathcal{A}_e$.

3.4.1. Algorithm: Incremental Product Automaton Construction. Let Σ_e be the set of RDF predicates that appear in e and $|e|$ the size of the expression being evaluated. We now introduce the notion of core expression. The expression e_T is the core expression obtained from e by replacing each test with a fresh symbol $\alpha \notin \Sigma_e$ and each action with a fresh symbol $\beta \notin \Sigma_e$. Let Λ be the set of symbols used for tests and Ω the set of symbols used for actions (the sets Σ_e, Λ, and Ω are pairwise disjoint). The cost of the translation of an expression e into a core expression e_T is $O(|e|)$ (it can be done with a scan of the input expression e).

The NFA $\mathcal{A}_e$ corresponding to e_T is a tuple $(Q, \Sigma_e \cup \Lambda \cup \Omega, \delta, q_0, F)$, where Q is the set of states of $\mathcal{A}_e$, $\delta : Q \times (\Sigma_e \cup \Lambda \cup \Omega \cup \{\varepsilon\}) \rightarrow 2^Q$ the transition function, $q_0 \in Q$ the initial state, and $F \subseteq Q$ the set of accepting (final) states. $\mathcal{A}_e$ can be built with costs $O(|e_T|) = O(|e|)$ following the Thomson's construction rules [Hopcroft et al. 2000]. We

ALGORITHM 1: construct(NAUTILOD expression e, URI seed)

```
Input: NAUTILOD expression $e$, seed URI seed
Build: $\mathcal{W} \times \mathcal{A}_e = (Q', \Sigma_e \cup \Lambda \cup \Omega, \delta', (\texttt{seed}, q_0), F')$ product automaton
1  compute $e_T$, and let $\Lambda$ and $\Omega$ be the sets of labels used for tests and actions
2  build the NFA $\mathcal{A}_e = (Q, \Sigma_e \cup \Lambda \cup \Omega, \delta, q_0, F)$
3  initialize the set of states of $\mathcal{W} \times \mathcal{A}_e$ to $\{(\texttt{seed}, q_0)\}$
4  initialize the set of transitions of $\mathcal{W} \times \mathcal{A}_e$ to $\emptyset$
5  for each state $(\mathrm{u}, q) \in \mathcal{W} \times \mathcal{A}_e$ do
6    for each transition $\delta(q, x) \in \mathcal{A}_e$ do
7      if $x \in \Sigma_e$ then
8        for each $q' \in \delta(q, x)$ AND $(\mathrm{u}, x, \mathrm{u'}) \in \mathcal{D}(\mathrm{u})$ do
9          add $(\mathrm{u}', q')$ to the set of states of $\mathcal{W} \times \mathcal{A}_e$
10         add $(\mathrm{u}', q')$ to $\delta'((\mathrm{u}, q), x)$
11         if $q' \in F$ then
12           add $(\mathrm{u}', q')$ to $F'$
13     else if ($x \in \Lambda$ and evalTest($x$,$\mathcal{D}(\mathrm{u})$)=true) or
       ($x = \varepsilon$) or ($x \in \Omega$) then
14       for each $q' \in \delta(q, x)$ do
15         add $(\mathrm{u}, q')$ to the set of states of $\mathcal{W} \times \mathcal{A}_e$
16         add $(\mathrm{u}, q')$ to $\delta'((\mathrm{u}, q), x)$
17         if $q' \in F$ then
18           add $(\mathrm{u}', q')$ to $F'$
19 return $\mathcal{W} \times \mathcal{A}_e$
```

assume that $\mathcal{A}_e$ is stored by using two adjacency lists that make explicit the navigation of transitions and their inverses. This representation uses space $O(|\mathcal{A}_e|) = O(|e|)$. We also assume that given an element $q \in Q$, we can access its associated list of transitions in time $O(1)$.

Given a WLOD instance $\mathcal{W}$, we indicate with $\mathcal{W}_e$ the portion of $\mathcal{W}$ *navigated* when evaluating the expression e. We denote by uris($\mathcal{W}_e$) the set of URIs in $\mathcal{W}_e$ and by triples($\mathcal{W}_e$) its set of triples. Moreover, seed is the node in $\mathcal{W}$ (i.e., a URI) where the navigation, for the evaluation of e, starts. The product automaton $\mathcal{W} \times \mathcal{A}_e$ is built according to Algorithm 1. In what follows we clarify the phases of the algorithm.

(1) Initialization: $\mathcal{W} \times \mathcal{A}_e = (\{(\texttt{seed}, q_0)\}, \Sigma_e \cup \Lambda \cup \Omega, \emptyset, (\texttt{seed}, q_0), \emptyset)$; the product automaton at the beginning only contains the initial state $(\texttt{seed}, q_0)$ and its transition function is empty (lines 3–4).
(2) For each state (u, q) of $\mathcal{W} \times \mathcal{A}_e$ (line 5), all the transitions originating from q in $\mathcal{A}_e$ are considered (line 6). Each state of $\mathcal{W} \times \mathcal{A}_e$ is considered exactly once.
(3) The state (u', q') and the transition $((\mathrm{u}, q), \mathrm{p}, (\mathrm{u}', q'))$ are added to $\mathcal{W} \times \mathcal{A}_e$ if and only if $q' \in \delta(q, \mathrm{p})$, $\mathrm{p} \in \Sigma_e$ and the triple $(\mathrm{u}, \mathrm{p}, \mathrm{u}') \in \mathcal{D}(\mathrm{u})$, where $\mathcal{D}(\mathrm{u})$ is the description of u obtained by dereferencing u (lines 8–12).
(4) The state (u, q') and the transition $((\mathrm{u}, q), \varepsilon, (\mathrm{u}, q'))$ are added to $\mathcal{W} \times \mathcal{A}_e$ if and only if $q' \in \delta(q, \varepsilon)$ (lines 14–18).
(5) The state (u, q') and the transition $((\mathrm{u}, q), \texttt{test}, (\mathrm{u}, q'))$ are added to $\mathcal{W} \times \mathcal{A}_e$ if and only if $q' \in \delta(q, \texttt{test})$, $\texttt{test} \in \Lambda$, and test evaluates true over $\mathcal{D}(\mathrm{u})$ (lines 14–18).
(6) The state (u, q') and the transition $((\mathrm{u}, q), \texttt{action}, (\mathrm{u}, q'))$ are added to $\mathcal{W} \times \mathcal{A}_e$ if and only if $q' \in \delta(q, \texttt{action})$, $\texttt{action} \in \Omega$ (lines 14–18).

Example 3.1 (*Incremental Product Automaton*). Figure 5(a) shows an excerpt of real-word data ($\mathcal{W}$). Figure 5(b) shows an expression e along with the associated NFA $\mathcal{A}_e$. Figures 5(c)–5(f) describe the steps necessary to build the product automaton $\mathcal{W} \times \mathcal{A}_e$.

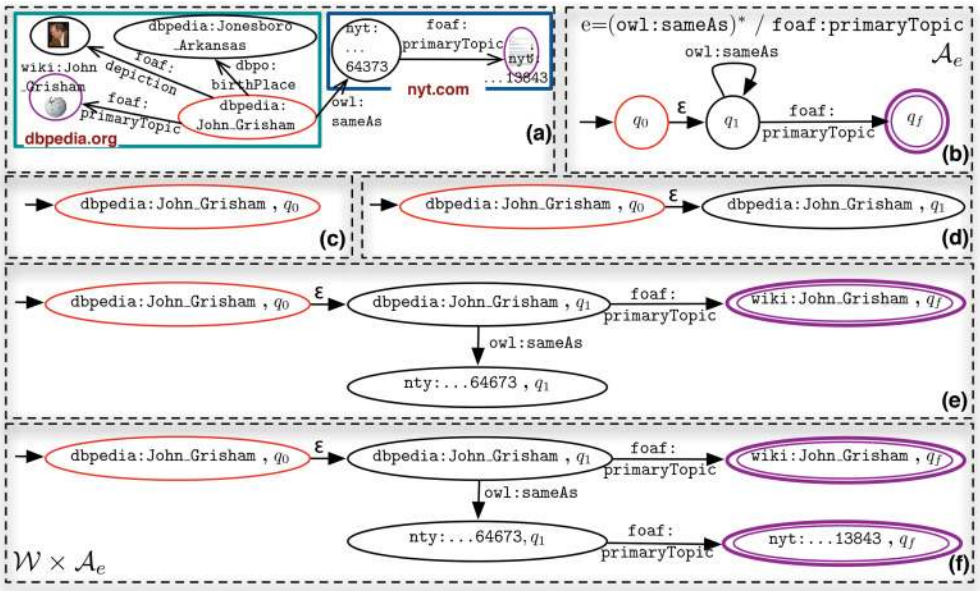


Fig. 5. (a) A WLOD instance $\mathcal{W}$. (b) The automaton $\mathcal{A}_e$ for the expression in Example 1.1. (c)–(f) Steps necessary for the construction of the product automaton $\mathcal{W}\times\mathcal{A}_e$ starting from the seed node dbpedia:John_Grisham.

Figure 5(c) shows the product automaton after the initialization (Algorithm 1, lines 3–4). Figure 5(d) show the partial automaton after the first iteration of the `for` (lines 5–18) when the state (dbpedia:John_Grisham, q_0) is considered. In particular, note that the only transition in $\mathcal{A}_e$ starting from q_0 is labeled by ε and leads to the state q_1. This corresponds to add to $\mathcal{W}\times\mathcal{A}_e$ the state (dbpedia:John_Grisham, q_1) and one ε transition. When the state (dbpedia:John_Grisham, q_1) is considered, the two transitions in $\mathcal{A}_e$ from q_1 to q_f labeled with foaf:primaryTopic and from q_1 to itself labeled by owl:sameAs allow one to add the two states (wiki:John_Grisham, q_f) and (nyt:...64673, q_1) and the respective transitions. The partial result is shown in Figure 5(e). When (nyt:...64673, q_1) is considered the transition labeled with foaf:primaryTopic allows one to add the state (nyt:...13843, q_f), while the transition labeled with owl:sameAs does not have any effect since there are no outgoing owl:sameAs-edges originating from nyt:...13843 in $\mathcal{W}$. The result is reported in Figure 5(f). Note that when the two states (wiki:John_Grisham, q_f) and (nyt:...13843, q_f) are considered, $\mathcal{W}\times\mathcal{A}_e$ remains unchanged since the state q_f has no outgoing transitions in $\mathcal{A}_e$.

THEOREM 3.2. *The product automaton $\mathcal{W}\times\mathcal{A}_e$ can be built in time $O(|\mathcal{W}_e| \times |e|) + C_T$ using Algorithm 1, where C_T is the cost of the evaluation of tests.*

PROOF. At the end of the execution of the Algorithm 1, we have that $Q' \subseteq \texttt{uris}(\mathcal{W}_e)\times Q$. Since in the construction of the product automaton each state of $\mathcal{W}\times\mathcal{A}_e$ is considered only once, the first `for` loop (line 5) costs $O(|Q'|)=O(|\texttt{uris}(\mathcal{W}_e)| \times |Q|)$. For each state $(\texttt{u}, q)$ of $\mathcal{W}\times\mathcal{A}_e$ the outgoing transitions of q in $\mathcal{A}_e$ are considered so that the inner `for` loop (line 6) is executed a different number of times for each state. In particular, each transition (q, x, q') is examined only when considering the states $(\_, q) \in Q'$. Since the same state of Q can be examined at most once for each URI in $\mathcal{W}_e$, each transition (q, x, q') is examined at most $|\texttt{uris}(\mathcal{W}_e)|$ times. Hence, the code inside the two `for` loops (lines 7–18) is executed at most $O(|\texttt{uris}(\mathcal{W}_e)| \times |\delta|)$ times. Inside the `for` loops, all the states q' reached by the current transition are considered. When considering the transitions altogether, the entire automaton $\mathcal{A}_e$ is visited. Moreover, triples belonging to the description of the current URI u are used to compute the states and transitions to be added to the product automaton $\mathcal{W} \times \mathcal{A}_e$. This can be done in time $O(|\mathcal{D}(\mathrm{u})|)$ (the first time a URI is encountered, triples in $\mathcal{D}(\mathrm{u})$ are stored so that every subsequent check for a specific predicate can be done in constant time). When considering the URIs of $\mathcal{W}_e$ altogether this step costs $O(|\texttt{triples}(\mathcal{W}_e)|)$. Thus, the total cost of the algorithm is $O((|\texttt{uris}(\mathcal{W}_e)| + |\texttt{triples}(\mathcal{W}_e)|) \times |\mathcal{A}_e|) = O(|\mathcal{W}_e| \times |\mathcal{A}_e|) = O(|\mathcal{W}_e| \times |e|)$.

As for the transitions in δ labeled with a symbol $\alpha \in \Lambda$ (i.e., the tests) the previous complexity bound has to be refined by considering the cost C_T of evaluating tests. Each test is considered at most once for each URI. Note that at most a number of transitions equal to the number of triples in $|\texttt{triples}(\mathcal{W}_e)| \times |Q|$ is added to $\mathcal{W} \times \mathcal{A}_e$ when dealing with predicates (lines 9–10) and at most additional $|\texttt{uri}(\mathcal{W}_e)| \times |\mathcal{A}_e|$ when dealing with tests, ε transitions, or actions (lines 15–16). Thus, the maximum number of transitions in the product automaton is $|\delta'| = O(|\mathcal{W}_e| \times |\mathcal{A}_e|) = O(|\mathcal{W}_e| \times |e|)$. □

The complexity of the algorithm is parametric in C_T, that is, the cost of evaluating tests. NAUTILOD tests are ASK-SPARQL queries; hence, their cost can be controlled by choosing a particular fragment of SPARQL [Pérez et al. 2009]. However, note that the usage of tests enables one to (possibly) reduce the size of the set of nodes visited during the evaluation. After having constructed $\mathcal{W} \times \mathcal{A}_e$, the results of the evaluation of the NAUTILOD expression can be obtained by running Algorithm 2, which checks the final states. The final result is given in terms of (i) the set of URIs reachable by paths satisfying the expression and (ii) the set of actions that have been executed. The cost C_A of executing actions has to be added to the cost of building $\mathcal{W} \times \mathcal{A}_e$ via Algorithm 1 and checking final states via Algorithm 2. Finally, note that $\mathcal{W} \times \mathcal{A}_e$ contains $O(|\texttt{uri}(\mathcal{W}_e)| \times |\mathcal{A}_e|) = O(|\texttt{uri}(\mathcal{W}_e)| \times |e|)$ transitions labeled with $\alpha \in \Omega$ (i.e., actions).

Example 3.3 (*Computation of result URIs*). By considering the product automaton $\mathcal{W} \times \mathcal{A}_e$ in Figure 5(f), the results are `wiki:John_Grisham` and `nyt:...13843`, that is, the URIs of the WLOD reported in Figure 5(a) reached at some final state of $\mathcal{W} \times \mathcal{A}_e$. These results were discussed in Example 1.1.

The algorithms introduced previously enable one to identify (and return) the sets of nodes in the WLOD that conform to a specification given in the NAUTILOD language and execute the set of actions specified. Note that the same results could have also been computed without using Algorithm 2; in this case, results can be collected as soon as a final state is added to $\mathcal{W} \times \mathcal{A}_e$ (see Algorithm 1). Moreover, actions can be handled in different ways: either by collecting the pairs (u, `action`) when the corresponding state is added to $\mathcal{W} \times \mathcal{A}_e$ and executing the whole set as a last step, or by executing each

ALGORITHM 2: evaluate(NAUTILOD expression e, URI `seed`)

```
Input: NAUTILOD expression $e$, seed URI seed
Build: $R$ set of URIs
Uses: $\mathcal{W} \times \mathcal{A}_e = (Q', \Sigma_e \cup \Lambda \cup \Omega, \delta', (\mathrm{seed}, q_0), F')$

1 $\mathcal{W} \times \mathcal{A}_e$ =construct($e$, seed) /* see Algorithm 1 */
2 for each state $(\mathrm{u}, q) \in F'$ do
3     add u to $R$
4 for each transition $\delta'((\mathrm{u}, q), \mathrm{action})$ such that action $\in \Omega$ do
5     execute action over $\mathcal{D}(\mathrm{u})$
6 return $R$
```

action (u, action) as soon as the corresponding state is added to $\mathcal{W} \times \mathcal{A}_e$. In the next section we extend the scope of the semantics of NautiLOD to capture Web fragments.

4. CAPTURING WEB FRAGMENTS

We have described NautiLOD as a language that enables the specification of (i) sets of nodes in the WLOD graph and (ii) actions to be performed over data encountered during the navigation. We have presented a formal semantics (see Section 3.2) and algorithms (see Section 3.4) to evaluate NautiLOD expression. The possibility to specify (and retrieve) sets of nodes misses information about the fragment of the Web navigated to reach these nodes; in other words, connectivity information is lost. Indeed, navigation is only referenced in the semantics as the means to get the resulting set of nodes. Such behavior is common to many navigational languages (e.g., XPath [Clark and DeRose 1999] and nSPARQL [Pérez et al. 2010]). However, connectivity information is crucial in some contexts [Fionda et al. 2014a, 2014c].

Consider the Web user Nick who wants to collect a fragment of the Friend-Of-A-Friend (FOAF) social graph consisting in people he knows directly or indirectly that have his same musical preferences. While it would not be a problem to specify and retrieve this set of people with NautiLOD, it is not possible to provide information about the structure (connections) of the fragment of the FOAF graph where Nick is linked with these people. Another domain where connectivity is crucial is that of bibliographic/citation networks. Consider a researcher interested in a new topic who wants to collect relevant bibliographic references. She can specify papers that cite a given seed paper and are published at a specific conference. While it is possible to automatically retrieve via NautiLOD the set of papers linked with the seed, connectivity information is lost.

In a distributed environment such as the Web, keeping connectivity information is very useful for provenance tracking [Gil and Groth 2011], that is, understanding from which datasources pieces of information in a path come from. Motivated by these scenarios, we formalize two semantics of the NautiLOD language to capture fragments of the Web, that is, graphs including the sets of nodes satisfying an expression along with edges connecting these nodes. From a high-level perspective, the main challenge that we are going to face is the fact that now the evaluation of a NautiLOD expression has to deal with graphs representing Web fragments instead of a set of nodes, thus requiring the definitions of both the structure of such graphs and new operators to combine them. Although our discussion is centered on NautiLOD and the WLOD graph, the following results can also be generalized to other navigational languages.

4.1. Multipointed Graphs

In this section we introduce a particular type of structure called a Multipointed Graph (MPG). Essentially, an MPG is the structure that we use to formally capture a Web fragment. MPGs, as will be discussed later on in this section, have an algebraic structure that allows composition, manipulation, and reuse.

An MPG Γ is a quadruple (V, E, s, T) where (V, E) is a standard directed graph, $s \in V$ is a seed node (starting node), and $T \subseteq V$ is a set of ending nodes (result nodes). To access the elements of Γ we use the notation $\Gamma.x$, where $x \in \{V, E, s, T\}$. Let R be the set of nodes (URIs) obtained by evaluating a NautiLOD expression starting from the node (URI) *seed* according to the semantics returning sets of nodes; this semantics can be captured via the MPG $(\{seed\} \cup R, \emptyset, seed, R)$. The advantage of MPGs is that they can additionally store the graph visited while evaluating an expression.

The semantics of NautiLOD described in Section 3.2 deals with sets of nodes; indeed, rules in Table I, which consider all the types of syntactic expressions defined according to the NautiLOD syntax presented in Section 3.1, construct sets and manipulate such sets by using standard operators such as set membership (e.g., rule R6 in Table I) and union (e.g., rule R8 in Table I). In order to define the semantics of NautiLOD returning fragments, it is necessary to deal with MPGs and associate with each syntactic expression in the displayed box in Section 3.1 a set of MPGs. Moreover, when dealing with MPGs (and sets of MPGs) it is necessary to (i) define operations over MPGs reflecting the standard operations over sets of nodes and (ii) devise specific additional functionalities.

Definition 4.1 (*Operations over MPGs*). Let $\Gamma_i = (V_i, E_i, s_i, T_i)$, $i = 1, 2$ be MPGs and $\Gamma_\perp = (\emptyset, \emptyset, \perp, \emptyset)$ denote the empty MPG, where $\perp$ is a special symbol not in the universe of nodes.

(1) **Concatenation** $\circ$

$$\Gamma_1 \circ \Gamma_2 \quad = \quad \begin{cases} \Gamma_\perp & \text{if } s_2 \notin T_1, \\ (V_1 \cup V_2, E_1 \cup E_2, s_1, T_2) & \text{if } s_2 \in T_1. \end{cases}$$

(2) **Union** $\cup$

$$\Gamma_1 \cup \Gamma_2 \quad = \quad \begin{cases} (V_1 \cup V_2, E_1 \cup E_2, s_1, T_1 \cup T_2) & \text{if } s_1 = s_2, \\ \Gamma_1 & \text{if } s_1 \neq s_2 \wedge \Gamma_2 = \Gamma_\perp, \\ \Gamma_2 & \text{if } s_1 \neq s_2 \wedge \Gamma_1 = \Gamma_\perp, \\ \text{not defined} & \text{if } s_1 \neq s_2 \wedge \Gamma_1, \Gamma_2 \neq \Gamma_\perp. \end{cases}$$

As can be observed, the preceding operators applied to two MPGs produce another MPG and need to be extended over sets of MPGs. Definition 4.2 formalizes such extension; here, the binary operator $\mathsf{op} \in \{\circ, \cup\}$ is applied to all pairs Γ_1, Γ_2 such that Γ_1 belongs to the first set and Γ_2 to the second one.

Definition 4.2 (*Operations over sets of MPGs*). Let S_1 and S_2 be two sets of MPGs.

(1) For each $\mathsf{op} \in \{\circ, \cup\}$ we define

$$S_1 \ \mathsf{op}\ S_2 = \{\Gamma_1 \mathsf{op}\ \Gamma_2 \mid \Gamma_1 \in S_1, \Gamma_2 \in S_2\}.$$

(2) (Disjoint union, direct sum) $\oplus$,

$$S_1 \oplus S_2 = \{\Gamma \mid \Gamma \in S_1 \vee \ \Gamma \in S_2\}.$$

The definition of concatenation over sets of MPGs serves the purpose of encoding concatenation of NautiLOD expressions. Disjoint union reflects the case of the union operation defined for the semantics in Table I; the crucial difference is that now the result is in terms of MPGs instead of nodes. As for the union, it encodes the standard graph union operation with an additional condition on the starting nodes. The basic (atomic) elements are singleton sets of MPGs. We will call them MPG atomic expressions.

Proposition 4.3. *The following algebraic identities between MPG atomic expressions over a universe Un hold:*

(1) *The binary operation $\oplus$ is commutative, associative, and idempotent.*
(2) *The binary operation $\cup$ is commutative, associative, and idempotent.*
(3) *The binary operation $\circ$ is associative.*
(4) *The following identities hold: $(a \oplus b) \circ c = a \circ c \oplus b \circ c$ and $c \circ (a \oplus b) = c \circ a \oplus c \circ b$; $(a \oplus b) \cup c = (a \cup c) \oplus (b \cup c)$ (same left side); $(a \cup b) \circ c = a \circ c \cup b \circ c$ (same left side).*

The identities follow easily from the definitions; using them we can get normal forms for MPG expressions:

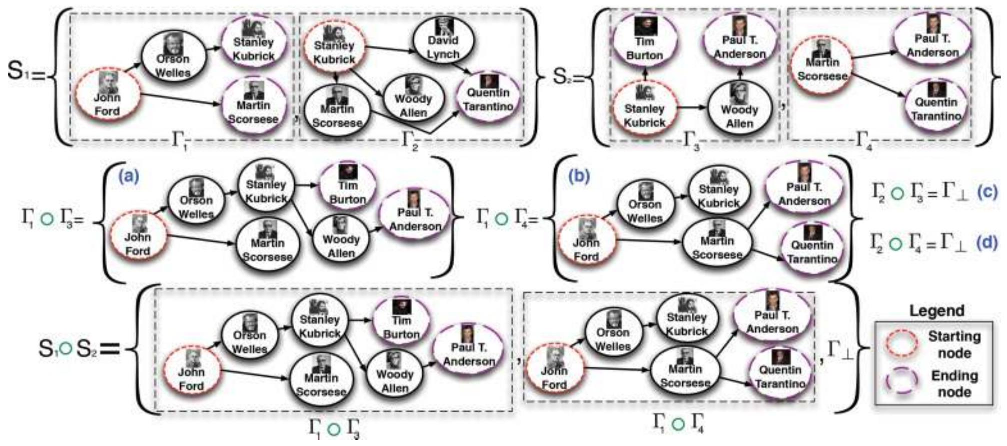


Fig. 6. MPGs and operations over (sets of) MPGs. Edges represent `influenced` relations from DBpedia.

PROPOSITION 4.4. *Any algebraic expression of MPGs over a fixed universe Un together with the preceding operators can be written in the following normal form, which is crucial to enhance the semantics of* NAUTILOD *to capture Web fragments:*

$$\bigoplus_{i=1}^{n}\left(\bigcup_{j=1}^{k_i}(e_{i_1}\circ\cdots\circ e_{i_{k_i}})\right)$$

where the e_{i_j} *are atomic MPG expressions.*

PROOF. The proof is a check that the rewriting system obtained by ordering the identities in (4) from left to right, plus $a \oplus a \to a$ and $a \cup a \to a$, form a confluent and terminating system modulo associativity and commutativity of the operators $\oplus$ and $\cup$ [Baader and Nipkow 1999]. The check of the critical pairs is straightforward in this case. □

Examples of operations over MPGs. Figure 6 shows examples of MPGs and operations over (sets of) MPGs. Recall that the definition of MPGs along with their operators is motivated by the fact that we want NAUTILOD expressions to deal with sets of MPGs instead of sets of nodes. In the example of Figure 6 it is shown the concatenation $\circ$ between the two sets of MPGs S$_1$=$\{\Gamma_1,\Gamma_2\}$ and S$_2$=$\{\Gamma_3,\Gamma_4\}$. Such operation, as is shown in Table II (rule R7), is necessary to deal with NAUTILOD expressions involving path concatenation (expressed in the NAUTILOD syntax with the symbol /). According to Definition 4.2, it is necessary to apply the operator $\circ$ to the elements in the two sets pairwise, thus obtaining a total of four concatenation operations as reported in Figures 6(a)–6(d).

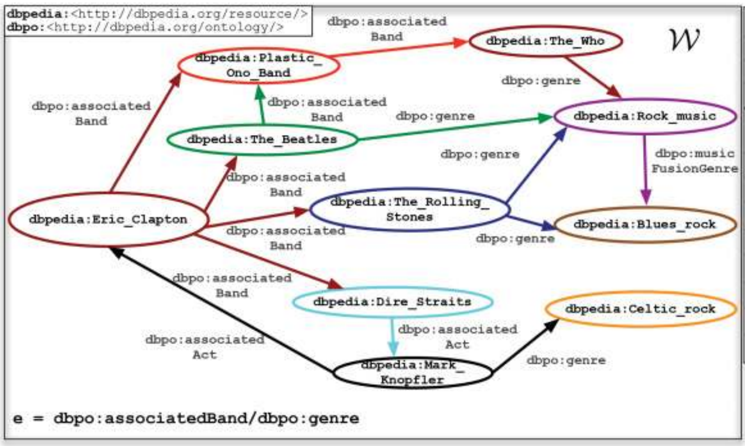
Consider the computation of $\Gamma_1\circ\Gamma_3$. Here, $\Gamma_1.s$=J. Ford, $\Gamma_1.T$={S. Kubrick, M. Scorsese}, and $\Gamma_3.s$=S. Kubrick, $\Gamma_3.T$={T. Burton, P. T. Anderson}, respectively. According to Definition 4.1, the result of the operation is *not* empty if $\Gamma_3.s \in \Gamma_1.T$ (which is the case). The resulting MPG (reported in Figure 6(a)) will be composed by the union of the two node and edge sets (i.e., $\Gamma_1.V \cup \Gamma_3.V$ and $\Gamma_1.E \cup \Gamma_3.E$), the node $\Gamma_1.s$ as starting node and the set $\Gamma_3.T$ as ending nodes. If we now consider the computation of $\Gamma_2\circ\Gamma_3$, by looking at Definition 4.1 and Figure 6, it is easy to see that the result will be empty since $\Gamma_3.s \notin \Gamma_2.T$. The operations between the remaining pairs can be computed similarly. The final result (see Definition 4.2) will be the set of MPGs $\{\Gamma_1\circ\Gamma_3, \Gamma_1\circ\Gamma_4, \Gamma_\perp\}$.

Table II. The VISITED (V) and SUCCESSFUL (S) semantics. The rules reflect all the possible forms of syntactic expressions that can be given according to the NAUTILOD syntax in Section 3.1. The rules for the syntactic expression path⟨l-h⟩ are omitted since they are simple adaptations of rules R8, R16, and R25 where the $\bigcup$ or the $\bigoplus$ is defined for the interval $[l, h]$ instead of $[1, \infty]$ as already shown for the semantics returning nodes shown in Table I. To access the elements of an MPG Γ we use the notation $\Gamma \cdot x$ where $x \in \{V, E, s, T\}$.

R1	$E_V\llbracket \mathtt{path} \rrbracket(\mathrm{u}, \mathcal{W})$	=	$\left\{ \bigcup_{\Gamma \in V\llbracket \mathtt{path} \rrbracket(\mathrm{u})} \Gamma, E_A\llbracket \mathtt{path} \rrbracket(\mathrm{u})) \right\}$
R2	$E_S\llbracket \mathtt{path} \rrbracket(\mathrm{u}, \mathcal{W})$	=	$\left\{ \bigcup_{\Gamma \in S\llbracket \mathtt{path} \rrbracket(\mathrm{u}) \mid \Gamma.T \neq \emptyset} \Gamma, E_A\llbracket \mathtt{path} \rrbracket(\mathrm{u})) \right\}$
R3	$V\llbracket \mathrm{p} \rrbracket(\mathrm{u})$	=	$\bigcup_{(\mathrm{u,p,v}) \in \mathcal{D}(\mathrm{u})} (\{\mathrm{u, v}\}, \{(\mathrm{u, p, v})\}, \mathrm{u}, \{\mathrm{v}\})$
R4	$V\llbracket \mathrm{p\hat{}} \rrbracket(\mathrm{u})$	=	$\bigcup_{(\mathrm{v,p,u}) \in \mathcal{D}(\mathrm{u})} (\{\mathrm{u, v}\}, \{(\mathrm{v, p, u})\}, \mathrm{u}, \{\mathrm{v}\})$
R5	$V\llbracket <\_> \rrbracket(\mathrm{u})$	=	$\bigcup_{\mathrm{p} \in \mathcal{U}} \bigcup_{(\mathrm{u,p,v}) \in \mathcal{D}(\mathrm{u})} (\{\mathrm{u, v}\}, \{(\mathrm{u, p, v})\}, \mathrm{u}, \{\mathrm{v}\})$
R6	$V\llbracket \mathtt{act} \rrbracket(\mathrm{u})$	=	$(\{\mathrm{u}\}, \emptyset, \mathrm{u}, \{\mathrm{u}\})$
R7	$V\llbracket \mathtt{path}_1/\mathtt{path}_2 \rrbracket(\mathrm{u})$	=	$V\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u}) \circ \left(\bigoplus_{\mathrm{v} \in \Gamma.T \mid \Gamma \in V\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u})} (V\llbracket \mathtt{path}_2 \rrbracket(\mathrm{v}) \oplus (\{\mathrm{v}\}, \emptyset, \mathrm{v}, \emptyset)) \right)$
R8	$V\llbracket (\mathtt{path})^* \rrbracket(\mathrm{u})$	=	$(\{\mathrm{u}\}, \emptyset, \mathrm{u}, \{\mathrm{u}\}) \cup \left(\bigcup_{i=1}^{\infty} V\llbracket (\mathtt{path}_i) \rrbracket(\mathrm{u}) \mid \mathtt{path}_1 = \mathtt{path} \wedge \mathtt{path}_i = \mathtt{path}_{i-1}/\mathtt{path} \right)$
R9	$V\llbracket \mathtt{path}_1 \vert \mathtt{path}_2 \rrbracket(\mathrm{u})$	=	$V\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u}) \cup V\llbracket \mathtt{path}_2 \rrbracket(\mathrm{u})$
R10	$V\llbracket \mathtt{path[test]} \rrbracket(\mathrm{u})$	=	$\bigoplus_{\Gamma \in V\llbracket \mathtt{path} \rrbracket(\mathrm{u})} (\Gamma.V, \Gamma.E, \Gamma.s, \{\mathrm{v} \in \Gamma.T \mid \mathtt{Evaluate(test, v) = true}\})$
R11	$S\llbracket \mathrm{p} \rrbracket(\mathrm{u})$	=	$\bigoplus_{(\mathrm{u,p,v}) \in \mathcal{D}(\mathrm{u})} (\{\mathrm{u, v}\}, \{(\mathrm{u, p, v})\}, \mathrm{u}, \{\mathrm{v}\})$
R12	$S\llbracket \mathrm{p\hat{}} \rrbracket(\mathrm{u})$	=	$\bigoplus_{(\mathrm{v,p,u}) \in \mathcal{D}(\mathrm{u})} (\{\mathrm{u, v}\}, \{(\mathrm{v, p, u})\}, \mathrm{u}, \{\mathrm{v}\})$
R13	$S\llbracket <\_> \rrbracket(\mathrm{u})$	=	$\bigoplus_{\mathrm{p} \in \mathcal{U}} \bigoplus_{(\mathrm{u,p,v}) \in \mathcal{D}(\mathrm{u})} (\{\mathrm{u, v}\}, \{(\mathrm{u, p, v})\}, \mathrm{u}, \{\mathrm{v}\})$
R14	$S\llbracket \mathtt{act} \rrbracket(\mathrm{u})$	=	$(\{\mathrm{u}\}, \emptyset, \mathrm{u}, \{\mathrm{u}\})$
R15	$S\llbracket \mathtt{path}_1/\mathtt{path}_2 \rrbracket(\mathrm{u})$	=	$S\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u}) \circ \left(\bigoplus_{\mathrm{v} \in \Gamma.T \mid \Gamma \in S\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u})} S\llbracket \mathtt{path}_2 \rrbracket(\mathrm{v}) \right)$
R16	$S\llbracket (\mathtt{path})^* \rrbracket(\mathrm{u})$	=	$(\{\mathrm{u}\}, \emptyset, \mathrm{u}, \{\mathrm{u}\}) \oplus \left(\bigoplus_{i=1}^{\infty} S\llbracket (\mathtt{path}_i) \rrbracket(\mathrm{u}) \mid \mathtt{path}_1 = \mathtt{path} \wedge \mathtt{path}_i = \mathtt{path}_{i-1}/\mathtt{path} \right)$
R17	$S\llbracket \mathtt{path}_1 \vert \mathtt{path}_2 \rrbracket(\mathrm{u})$	=	$S\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u}) \oplus S\llbracket \mathtt{path}_2 \rrbracket(\mathrm{u})$
R18	$S\llbracket \mathtt{path[test]} \rrbracket(\mathrm{u})$	=	$\bigoplus_{\Gamma \in S\llbracket \mathtt{path} \rrbracket(\mathrm{u})} (\Gamma.V, \Gamma.E, \Gamma.s, \{\mathrm{v} \in \Gamma.T \mid \mathtt{Evaluate(test, v) = true}\})$
R19	$E_A\llbracket \mathtt{path} \rrbracket(\mathrm{u})$	=	$\{\mathtt{Exec(act}, \mathcal{D}(\mathrm{u})) : (\mathrm{u}, \mathtt{act}) \in A\llbracket \mathtt{path} \rrbracket(\mathrm{u})\}$
R20	$A\llbracket \mathrm{p} \rrbracket(\mathrm{u})$	=	$\emptyset$
R21	$A\llbracket \mathrm{p\hat{}} \rrbracket(\mathrm{u})$	=	$\emptyset$
R22	$A\llbracket <\_> \rrbracket(\mathrm{u})$	=	$\emptyset$
R23	$A\llbracket \mathtt{act} \rrbracket(\mathrm{u})$	=	$\{(\mathrm{u}, \mathtt{act})\}$
R24	$A\llbracket \mathtt{path}_1/\mathtt{path}_2 \rrbracket(\mathrm{u})$	=	$A\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u}) \cup \bigcup_{\mathrm{v} \in \Gamma.T \mid \Gamma \in (S \vert V)\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u})} A\llbracket \mathtt{path}_2 \rrbracket(\mathrm{v})$
R25	$A\llbracket (\mathtt{path}){*} \rrbracket(\mathrm{u})$	=	$\bigcup_1^{\infty} A\llbracket \mathtt{path}_i \rrbracket(\mathrm{u}) : \mathtt{path}_1 = \mathtt{path} \wedge \mathtt{path}_i = \mathtt{path}_{i-1}/\mathtt{path}$
R26	$A\llbracket \mathtt{path}_1 \vert \mathtt{path}_2 \rrbracket(\mathrm{u})$	=	$A\llbracket \mathtt{path}_1 \rrbracket(\mathrm{u}) \cup A\llbracket \mathtt{path}_2 \rrbracket(\mathrm{u})$
R27	$A\llbracket \mathtt{path[test]} \rrbracket(\mathrm{u})$	=	$A\llbracket \mathtt{path} \rrbracket(\mathrm{u})$

4.2. NAUTILOD Semantics Returning Web Fragments

We are now ready to introduce the two formal semantics of NAUTILOD that deal with MPGs and operations over (sets of) MPGs. Abstractly speaking, an MPG captures both nodes and edges traversed while evaluating an expression. In what follows we distinguish between V (VISITED) semantics and S (SUCCESSFUL) semantics. The first semantics returns the MPG consisting of all the nodes and edges *navigated* during the evaluation of an expression while the SUCCESSFUL semantics returns the MPG obtained as the union of all paths that satisfy the expression (i.e., all the successful paths). Table II shows the two formal semantics.



Abbreviations	
dbpedia:Eric_Clapton	EC
dbpedia:Plastic_Ono_Band	POB
dbpedia:The_Beatles	TB
dbpedia:The_Rolling_Stones	TRS
dbpedia:Dire_Straits	DS
dbpedia:The_Who	TW
dbpedia:Rock_music	RM
dbpedia:Blues_rock	BR
dbpedia:Celtic_rock	CR
dbpo:associatedBand	AB
dbpo:associatedAct	AA
dbpo:genre	G
dbpo:musicFusionGenre	MFG

Fig. 7. An excerpt of the WLOD taken from DBpedia.

The evaluation of an expression e over the WLOD instance $\mathcal{W}$ according to the new semantics is performed starting from a seed node u according to the rules R1–R27 in Table II. The evaluation returns an ordered pair composed of the MPG that captures the fragment of $\mathcal{W}$ (according to either the VISITED or the SUCCESSFUL semantics) and the set of actions performed during the evaluation. The main modules are the functions E_V (rule R1) and E_S (rule R2) that represent the starting points for the evaluation of a NAUTILOD expression according to the VISITED or the SUCCESSFUL semantics, respectively. The definition and application of the rules in Table II follows the same reasoning described in Section 3.2 with the difference that now the functions E_V, V, E_S, and S deal with (sets of) MPGs. Indeed, the V and S semantics are given by considering all the types of syntactic expressions that can be defined according to the NAUTILOD syntax (see Section 3.1).

It is worth pointing out that the evaluation of an expression always returns an MPG. Indeed, rules R1 and R2, which are the main modules, always perform the union $\cup$ of all resulting MPGs, which gives an MPG as a result (see Definition 4.1). Moreover, also note that the union is always defined since all resulting MPGs have the same starting node (i.e., the seed node u). Indeed, the final MPG has u as the starting node; moreover, the ending nodes (i.e., the set T) are the nodes reachable by paths that satisfy the expression. Actions are handled by the two functions E_A (execution) and A (collection). The crucial aspect of the new semantics is that, as we will show in Section 4.4, the cost of obtaining the MPG is the same as that of the traditional semantics outputting only the nodes reached by evaluating an expression.

4.3. Examples of NAUTILOD Expressions Returning Web Fragments

We now provide an example of evaluation for the NAUTILOD expression $e=$ dbpo:associatedBand/dbpo:genre according to both the VISITED (V) and SUCCESSFUL (S) semantics on the excerpt of WLOD shown in Figure 7. We consider the node dbpedia:Eric_Clapton as seed. Moreover, for sake of space, we will adopt the following URI shorthands (also reported in Figure 7): dbpedia:Eric_Clapton (EC), dbpedia:Plastic_Ono_Band (POB), dbpedia:The_Beatles (TB), dbpedia:The_Rolling_Stones (TRS), dbpedia:Dire_Straits (DS), dbpedia:Rock_music (RM), dbpedia:Blues_rock (BR), dbpo:associatedBand (AB), and dbpo:genre (G).

VISITED *semantics.* The starting point for the evaluation of e is rule R1, which will then trigger rules R7, R19, and R24. Here, it is necessary to evaluate the two parts

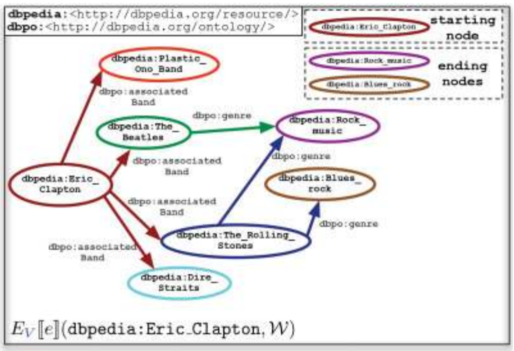


Fig. 8. The MPG as for the VISITED semantics.

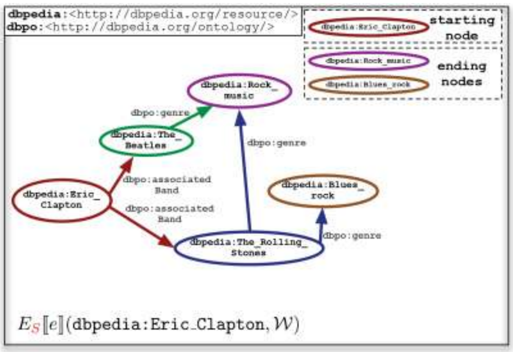


Fig. 9. The MPG as for the SUCCESSFUL semantics.

of the concatenation both for building the MPG (rule R7) and collecting actions (rule R24). To build the MPG, the subexpression `Associated Band` (`AB`) is evaluated by using Rule R3 starting from `Eric Clapton` (`EC`):

$$V[\![\mathtt{AB}]\!](\mathtt{EC}) = \{(\{\mathtt{EC}, \mathtt{POB},\ \mathtt{TB},\ \mathtt{TRS},\ \mathtt{DS}\}, \{(\mathtt{EC}, \mathtt{AB}, \mathtt{POB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TRS}), (\mathtt{EC}, \mathtt{AB}, \mathtt{DS})\}, \mathtt{EC}, \{\mathtt{POB},\ \mathtt{TB},\ \mathtt{TRS},\ \mathtt{DS}\})\}.$$

Then, from each ending node of each MPG in $V[\![\mathtt{AB}]\!](\mathtt{EC})$ (i.e., `POB, TB, TRS, DS`) the subexpression `dbpo:genre` (`G`) is evaluated by applying again rule R3, thus obtaining

$$\begin{aligned}
&V[\![\mathtt{G}]\!](\mathtt{POB}) = \Gamma_\perp,\\
&V[\![\mathtt{G}]\!](\mathtt{TB}) = \{(\{\mathtt{TB},\mathtt{RM}\}, \{(\mathtt{TB},\ \mathtt{G},\ \mathtt{RM})\}, \mathtt{TB}, \{\mathtt{RM}\})\},\\
&V[\![\mathtt{G}]\!](\mathtt{TRS}) = \{(\{\mathtt{TRS},\mathtt{RM},\ \mathtt{BR}\}, \{(\mathtt{TB},\ \mathtt{G},\ \mathtt{RM}), (\mathtt{TB},\ \mathtt{G},\ \mathtt{BR})\}, \mathtt{TRS}, \{\mathtt{RM},\ \mathtt{BR}\})\},\\
&V[\![\mathtt{G}]\!](\mathtt{DS}) = \Gamma_\perp.
\end{aligned}$$

Then, according to rule R7, the results of $V[\![\mathtt{G}]\!](\mathtt{POB})$, $V[\![\mathtt{G}]\!](\mathtt{TB})$, $V[\![\mathtt{G}]\!](\mathtt{TRS})$, and $V[\![\mathtt{G}]\!](\mathtt{DS})$ are combined through the disjoint union (i.e., $\oplus$ operator) with the MPGs $(\{\mathtt{POB}\}, \emptyset, \mathtt{POB}, \emptyset)$, $(\{\mathtt{TB}\}, \emptyset, \mathtt{TB}, \emptyset)$, $(\{\mathtt{TRS}\}, \emptyset, \mathtt{TRS}, \emptyset)$, $(\{\mathtt{DS}\}, \emptyset, \mathtt{DS}, \emptyset)$; this gives the set

$$\begin{aligned}
V_\Gamma = \{&(\{\mathtt{TB},\mathtt{RM}\}, \{(\mathtt{TB},\ \mathtt{G},\ \mathtt{RM})\}, \mathtt{TB}, \{\mathtt{RM}\}),\\
&(\mathtt{TRS},\mathtt{RM},\ \mathtt{BR}\}, \{(\mathtt{TB},\ \mathtt{G},\ \mathtt{RM}), (\mathtt{TB},\ \mathtt{G},\ \mathtt{BR})\}, \mathtt{TRS}, \{\mathtt{RM},\ \mathtt{BR}\}),\\
&(\{\mathtt{POB}\}, \emptyset, \mathtt{POB}, \emptyset), (\{\mathtt{TB}\}, \emptyset, \mathtt{TB}, \emptyset), (\{\mathtt{TRS}\}, \emptyset, \mathtt{TRS}, \emptyset), (\{\mathtt{DS}\}, \emptyset, \mathtt{DS}, \emptyset)\}
\end{aligned}$$

Then, the set $V[\![\mathtt{AB}]\!](\mathtt{EC})$ is composed (i.e., $\circ$ operator) with V_Γ, thus giving

$$\begin{aligned}
V[\![\mathtt{AB/G}]\!](\mathtt{EC}) = \{&(\{\mathtt{EC}, \mathtt{POB},\mathtt{TB},\mathtt{TRS},\mathtt{DS},\mathtt{RM}\}, \{(\mathtt{EC}, \mathtt{AB}, \mathtt{POB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TB}),\\
&(\mathtt{EC}, \mathtt{AB}, \mathtt{TRS}), (\mathtt{EC}, \mathtt{AB}, \mathtt{DS}), (\mathtt{TB},\ \mathtt{G},\ \mathtt{RM})\}, \mathtt{EC}, \{\mathtt{RM}\}),\\
&(\{\mathtt{EC}, \mathtt{POB},\mathtt{TB},\mathtt{TRS},\mathtt{DS},\mathtt{RM},\mathtt{BR}\}, \{(\mathtt{EC}, \mathtt{AB}, \mathtt{POB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TB}),\\
&(\mathtt{EC}, \mathtt{AB}, \mathtt{TRS}), (\mathtt{EC}, \mathtt{AB}, \mathtt{DS}), (\mathtt{TRS},\mathtt{G},\mathtt{RM}), (\mathtt{TRS},\mathtt{G},\mathtt{BR})\}, \mathtt{EC}, \{\mathtt{RM},\mathtt{BR}\}),\\
&(\{\mathtt{EC},\mathtt{POB},\mathtt{TB},\mathtt{TRS},\mathtt{DS}\}, \{(\mathtt{EC}, \mathtt{AB}, \mathtt{POB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TRS}),\\
&(\mathtt{EC}, \mathtt{AB}, \mathtt{DS})\}, \mathtt{EC}, \emptyset)\}.
\end{aligned}$$

Finally, according to rule R1, the union ($\cup$ operator) of all MPGs in $V[\![\mathtt{AB/G}]\!](\mathtt{EC})$ is computed, thus returning the following MPG (reported in Figure 8):

$$\begin{aligned}
E_V[\![e]\!](\mathtt{EC}, \mathcal{W}) = \{&(\{\mathtt{EC}, \mathtt{POB},\mathtt{TB},\mathtt{TRS},\mathtt{DS},\mathtt{RM},\mathtt{BR}\}, \{(\mathtt{EC}, <\mathtt{AB}>, \mathtt{POB}), (\mathtt{EC}, \mathtt{AB}, \mathtt{TB}),\\
&(\mathtt{EC}, \mathtt{AB}, \mathtt{TRS}), (\mathtt{EC}, \mathtt{AB}, \mathtt{DS}), (\mathtt{TB},\mathtt{G},\mathtt{RM}), (\mathtt{TRS},\mathtt{G},\mathtt{RM}),\\
&(\mathtt{TRS},\mathtt{G},\mathtt{BR})\}, \mathtt{EC}, \{\mathtt{RM},\mathtt{BR}\})\}
\end{aligned}$$

A similar process is performed to collect and execute actions. In this case, since the expression does not contain actions the final set of actions is the empty set.

ALGORITHM 3: visited(NautiLOD expression e, URI seed)

Input: NautiLOD expression e, seed URI s
Build: $\mathcal{W}_V$ = MPG visited during the evaluation of e

```
1  build the NFA $\mathcal{A}_e = (Q, \Sigma_e \cup \Lambda \cup \Omega, \delta, q_0, F)$
2  $\mathcal{W} \times \mathcal{A}_e$ = construct $(e, \texttt{seed})$
3  set seed as the starting node of $\mathcal{W}_V$
4  for each state $(\mathtt{u}, q) \in \mathcal{W} \times \mathcal{A}_e$ do
5    add u to the set of nodes of $\mathcal{W}_V$
6    if $q \in F$ then
7      add u to the ending nodes of $\mathcal{W}_V$
8    for each transition $\delta'((\mathtt{u}, q), x) \in \mathcal{W} \times \mathcal{A}_e$ do
9      for each $(\mathtt{u}', q') \in \delta'((\mathtt{u}, q), x)$ do
10       if $x \in \Sigma_e$ then
11         add u' to the set of nodes of $\mathcal{W}_V$
12         add $(\mathtt{u}, x, \mathtt{u}')$ to $\mathcal{W}_V$
13 return $\mathcal{W}_V$
```

Successful *semantics*. The evaluation is similar to the previous case, but the starting point for the evaluation of e is rule R2. According to the Successful semantics the resulting MPG (reported in Figure 9) is

$$E_S[\![e]\!](\mathtt{EC}, \mathcal{W}) = \{(\{\mathtt{EC,TB,TRS,RM,BR}\}, \{(\mathtt{EC, AB, TB}), (\mathtt{EC, AB, TRS}), (\mathtt{TB,G,RM}), (\mathtt{TRS,G,RM}), (\mathtt{TRS,G,BR})\}, \mathtt{EC}, \{\mathtt{RM,BR}\})\}$$

4.4. Capturing Web Fragments: Algorithms and Complexity

We are now ready to present algorithms to capture Web fragments according to the Visited and Successful semantics. We recall the following notation: $\mathcal{W}$ denotes a WLOD instance; $\mathcal{W}_e$ denotes the fragment of $\mathcal{W}$ navigated when evaluating an expression e while $\mathtt{uris}(\mathcal{W}_e)$ and $\mathtt{triples}(\mathcal{W}_e)$ are its set of URIs and triples, respectively. Σ_e is the set of RDF predicates that appear in e and $|e|$ the size of the expression. e_T is the core expression associated with e and Λ and Ω the set of symbols used in lieu of tests and actions, respectively. Moreover, $\mathcal{A}_e = (Q, \Sigma_e \cup \Lambda \cup \Omega, \delta, q_0, F)$ is the NFA that accepts the language generated by e_T. Now, we introduce some definitions.

Definition 4.5 (*MPG of a Web fragment*). Given a WLOD instance $\mathcal{W}$, a NautiLOD expression e, and a seed node, $\mathcal{W}_V$ is the MPG obtained by evaluating e over $\mathcal{W}$ starting from seed according to the Visited semantics. $\mathcal{W}_S$ is the MPG obtained by evaluating e over $\mathcal{W}$ starting from seed according to the Successful semantics.

Theorem 4.6. *The MPG $\mathcal{W}_V$ of a* NautiLOD *expression e can be computed in time $\mathcal{O}(|\mathcal{W}_e| \times |e|) + C_T$, where C_T is the cost of evaluating tests.*

Proof. The MPG corresponding to the Visited semantics is obtained by executing Algorithm 3. $\mathcal{W}_V$ can be computed by building the product automaton $\mathcal{W} \times \mathcal{A}_e = (Q', \Sigma_e \cup \Lambda \cup \Omega, \delta', q_0', F')$ with cost $\mathcal{O}(|\mathcal{W}_e| \times |\mathcal{A}_e|) + C_T$ (see Theorem 3.2). Starting with an empty MPG, $\mathcal{W}_V$ can be built by visiting $\mathcal{W} \times \mathcal{A}_e$ (lines 4–12). In particular, for each transition $((\mathtt{u}, \_), x, (\mathtt{u}', \_))$ in δ'such that $x \in \Sigma_e$, the edge $(\mathtt{u}, x, \mathtt{u}')$ and the nodes u and u' are added to $\mathcal{W}_V$. It is also necessary to specify the starting node and the set of ending nodes. In particular, the starting node of $\mathcal{W}_V$ is the seed URI (line 3) and the set of ending nodes is $T = \{\mathtt{u} \mid (\mathtt{u}, q_f) \in Q' \wedge q_f \in F\}$ (lines 6–7). Given the product automaton $\mathcal{W} \times \mathcal{A}_e$, $\mathcal{W}_V$ can be computed by visiting each transition and each node exactly once with a cost $O(|Q'| + |\delta'|) = O(|\mathtt{uris}(\mathcal{W}_e)| \times |Q| + |\mathcal{W}_e| \times |\mathcal{A}_e|) = O(|\mathcal{W}_e| \times |e|)$. The total cost of the algorithm (also considering the cost of building the product automaton) is

ALGORITHM 4: successful(NautiLOD expression e, URI `seed`)

```
Input: NautiLOD expression $e$, seed URI seed
Build: $\mathcal{W}_S$ = MPG of all the paths in $\mathcal{W}_V$ that spell $e$
1  build the NFA $\mathcal{A}_e$; assume $q_0$ as its initial state and $F$ as the set of final states
2  $\mathcal{W} \times \mathcal{A}_e$ = construct ($e$, seed)
3  set seed as the starting node of $\mathcal{W}_S$
4  initialize toVisit with all state $(u, q_f) \in \mathcal{W} \times \mathcal{A}_e$, such that $q_f \in F$
5  put each u such that $(u, q_f) \in \mathcal{W} \times \mathcal{A}_e$, with $q_f \in F$ in the set of ending nodes of $\mathcal{W}_S$
6  while toVisit is not empty do
7     remove from toVisit a pair $(u, q)$
8     add u to the set of nodes of $\mathcal{W}_S$
9     for each transition $\delta((u, q), x) \in \mathcal{W} \times \mathcal{A}_e$ do
10       for each $(u', q') \in \delta((u, q), x)$ such that $(u', q')$ has not been already checked do
11          add u' to the set of nodes of $\mathcal{W}_S$
12          if $x \in \Sigma_e$ then
13             add $(u, x, u')$ to $\mathcal{W}_S$
14          add $(u', q')$ to toVisit
15 return $\mathcal{W}_S$
```

$O(|\mathcal{W}_e| \times |e|) + C_T$. Note that $\mathcal{W}_V$ could also be built without any overhead with respect to the construction of $\mathcal{W} \times \mathcal{A}_e$; this could be done by adding on-the-fly to $\mathcal{W}_V$ nodes and edges encountered during the navigation by slightly modifying Algorithm 1. □

Theorem 4.7. *$\mathcal{W}_S$ can be computed in time $O(|\mathcal{W}| \times |e|) + C_T$ by navigating the product automaton backward (from the final states to the initial state).*

Proof. $\mathcal{W}_S$ can be built by using $\mathcal{W} \times \mathcal{A}_e$ according to Algorithm 4. In particular, again, the idea is to start with an empty MPG and navigate the product automaton backward (from the final states to the initial) by adding nodes and edges to $\mathcal{W}_S$ (lines 8, 11, and 13). Each state and each transition (in the opposite direction) is visited at most once with cost $O(|Q'| + |\delta'|) = O(|\texttt{uris}(\mathcal{W}_e)| \times |Q| + |\mathcal{W}_e| \times |\mathcal{A}_e|) = O(|\mathcal{W}_e| \times |e|)$. Thus, the total cost, when also considering the cost of building the product automaton, is $O(|\mathcal{W}_e| \times |e|) + C_T$. □

5. AN IMPLEMENTATION OF NautiLOD

This section presents an implementation of NautiLOD in the `swget` tool. `swget` has been implemented in Java in different versions: (i) a developer release, which includes a command-line tool and an API; (ii) an end user release, which features a GUI; (iii) a Web portal[8] [Fionda et al. 2014b]. `swget` uses Jena[9] to deal with RDF data. Besides the features presented in Sections 3 and 4, `swget` adds a set of ad hoc options to further control the navigation from a network-oriented perspective.

Such options include, among others, the possibility to limit the size of data transferred and enable/disable *caching*. In particular, `swget` can be configured to use a cache that keeps Jena *Model* objects associated to dereferenced URIs. The cache has the lifetime of the evaluation of an expression. However, it could also be made persistent, for instance, via a hashtable-like structure with *keys* being URIs and *values* Jena *Model* objects built from data obtained by dereferencing such URIs. This will improve the runtime performance of the system by avoiding dereferencing the same URI multiple times. However, the cache introduces problems of data freshness (e.g., how often the

[8]http://swget.inf.unibz.it.
[9]http://jena.apache.org.

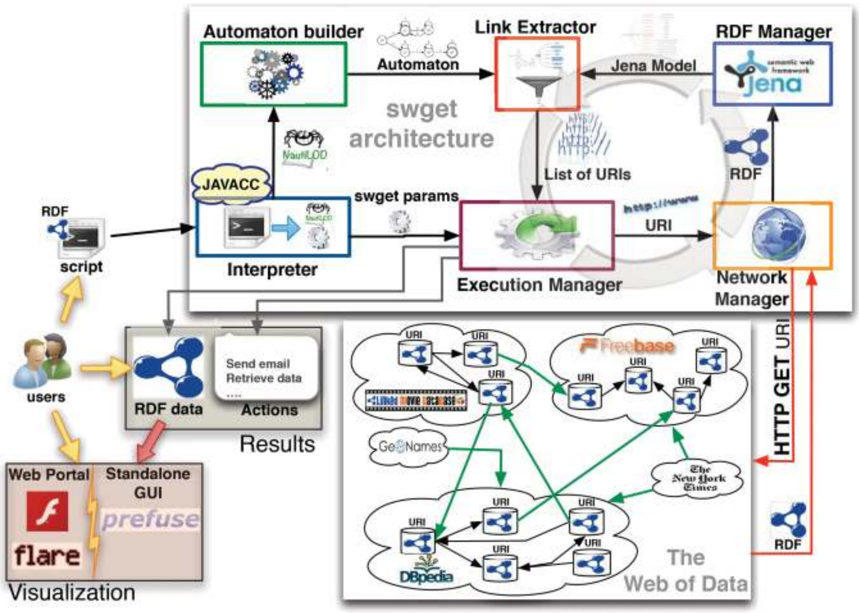


Fig. 10. The swget high-level architecture.

cache has to be updated). This issue can be addressed either by setting a cache lifetime or by looking at the header of HTTP connections when dereferencing a URI and getting data only if changes are detected with respect to data in the cache. Further information and examples are available at the swget website.[10]

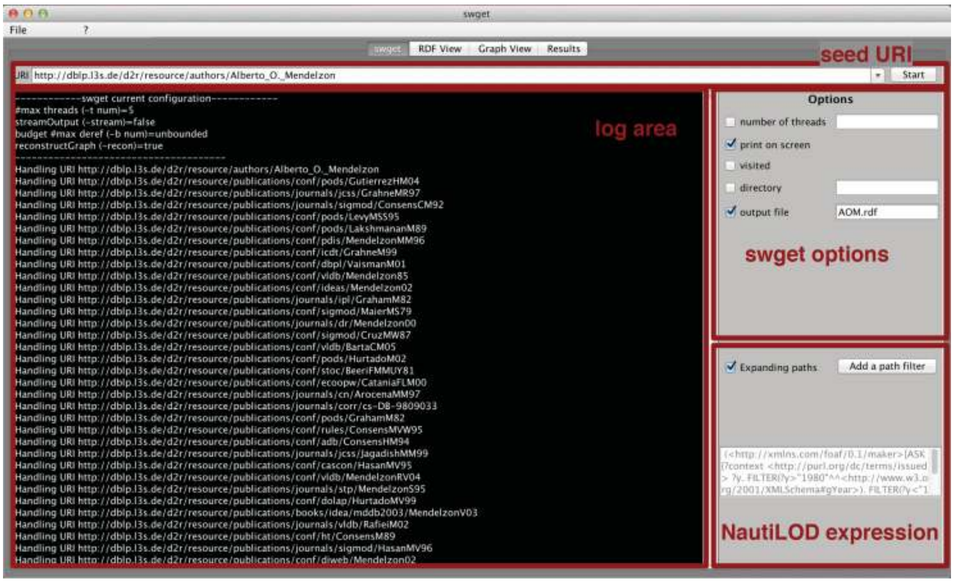
5.1. The swget Tool

We now provide an overview of the implementation of NautiLOD in swget. The conceptual architecture of swget is reported in Figure 10, which also shows the flow of information. The user submits a script that contains a NautiLOD expression plus other metadata (e.g., parameters, comments in natural language). The script is received by the *Interpreter* module, which checks the syntax and initializes both the *Execution Manager* (with the *seed* URI) and the *Automaton Builder*.

The *Automaton Builder*, via a JavaCC-based parser,[11] generates the automaton associated with the expression; it will be used to drive the evaluation of such expression on the WLOD. The *Execution Manager* controls the execution and passes the URIs to be dereferenced (i.e., ids of nodes encountered when navigating the WLOD) to the *Network Manager* that performs the dereferencing of URIs via HTTP GET calls. This set of links is given to the *Execution Manager*, which starts over the cycle. The execution ends either when some navigational parameter imposes it (e.g., a threshold on the network traffic has been reached) or when there are no more URIs to be dereferenced.

At the end of the execution, the results include (i) the Web fragment (represented in RDF) obtained according to the Visited/Successful semantics (see Section 4) and (ii) the execution of the actions fired during the navigation. The Web fragment can be locally stored and manipulated via third-party tools or can be accessed via the swget GUI or the Web portal. The GUI makes usage of the Prefuse API[12] for the visualization

[10]http://swget.wordpress.com.
[11]http://javacc.java.net.
[12]http://prefuse.org/.



URI http://dblp.l3s.de/d2r/resource/authors/Alberto_O._Mendelzon

```
------------swget current configuration------------
#max threads (-t num)=5
streamOutput (-stream)=false
budget #max deref (-b num)=unbounded
reconstructGraph (-recon)=true
--------------------------------------
Handling URI http://dblp.l3s.de/d2r/resource/authors/Alberto_O._Mendelzon
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/pods/GutierrezHM04
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/jcss/GrahneMR97
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/sigmod/ConsensCM92
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/pods/LevyMSS95
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/pods/LakshmananM89
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/pdis/MendelzonMM96
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/icdt/GrahneM99
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/dbpl/VaismanM01
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/vldb/Mendelzon85
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/ideas/Mendelzon02
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/ipl/GrahamM82
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/sigmod/MaierMS79
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/dr/Mendelzon00
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/sigmod/CruzMW87
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/vldb/BartaCM05
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/pods/HurtadoM02
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/stoc/BeeriFMMUY81
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/ecoopw/CataniaFLM00
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/cn/ArocenaMM97
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/corr/cs-DB-9809033
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/pods/GrahamM82
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/rules/ConsensMVW95
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/adb/ConsensHM94
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/jcss/JagadishMM99
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/cascon/HasanMV95
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/vldb/MendelzonRV04
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/stp/MendelzonS95
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/dolap/HurtadoMV99
Handling URI http://dblp.l3s.de/d2r/resource/publications/books/idea/mddb2003/MendelzonV03
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/vldb/RafieiM02
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/ht/ConsensM89
Handling URI http://dblp.l3s.de/d2r/resource/publications/journals/sigmod/HasanMV96
Handling URI http://dblp.l3s.de/d2r/resource/publications/conf/diweb/Mendelzon02
```

number of threads: [ ]
[✓] print on screen
[ ] visited
[ ] directory
[✓] output file: AOM.rdf
[✓] Expanding paths

```
(<http://xmlns.com/foaf/0.1/maker>[ASK
{?context <http://purl.org/dc/terms/issued
> ?y. FILTER(?y>"1980"^^<http://www.w3.o
rg/2001/XMLSchema#gYear>). FILTER(?y<"1
```

Fig. 11. The swget GUI. On the left-hand side, the log area prints messages about the current evaluation. On the right-hand side, it is possible to input the NAUTILOD expression and set network parameters.

and manipulation of graphs. The Web portal has been developed by using the Flash technology (i.e., the Flex application framework) and the Flare visualization library.[13]

The command line implementation of swget has been thought for developers that need low-level access to the swget functionality. This implementation is in the same spirit of tools like wget[14] and curl.[15] Our implementation features a Java API that can be used to embed swget's features into third-party applications. As an example, a user could leverage this API to embed in a standard HTML page structured information taken from the WLOD about his/her favorite movies, books, or scientific papers. In this respect, the swget expression used to achieve this goal can be thought of as a *view* over the WLOD that can be run on a regular basis to keep the page up-to-date.

The swget GUI facilitates the writing of swget scripts and provides a visual overview of the results of the execution of such scripts. Figure 11 shows the interface for the creation of scripts. As can be observed, it is possible to specify the seed URI plus the other parameters (e.g., the NAUTILOD expression). Figure 12 shows the fragment visualization tab of the swget GUI. In this particular case, the coauthor network of Alberto O. Mendelzon (the node in red) when considering papers published between 1980 and 1990 is shown. Besides the ending nodes (nodes in violet), the fragment also contains other nodes (nodes in blue) that belong to paths from Mendelzon to his coauthors satisfying the expression. Note that the GUI enables one to search for nodes/edges, zoom in/out, and rearrange the visualization of the Web fragment.

6. EXPERIMENTAL EVALUATION

We have introduced the NAUTILOD language and described three formal semantics: one that captures sets of nodes (Section 3.2) and two that capture Web fragments (Section 4). We have also described automaton-based algorithms to evaluate NAUTILOD expressions according to the different semantics and discussed their complexity

[13]http://flare.prefuse.org/.
[14]http://www.gnu.org/software/wget/.
[15]http://curl.haxx.se/.

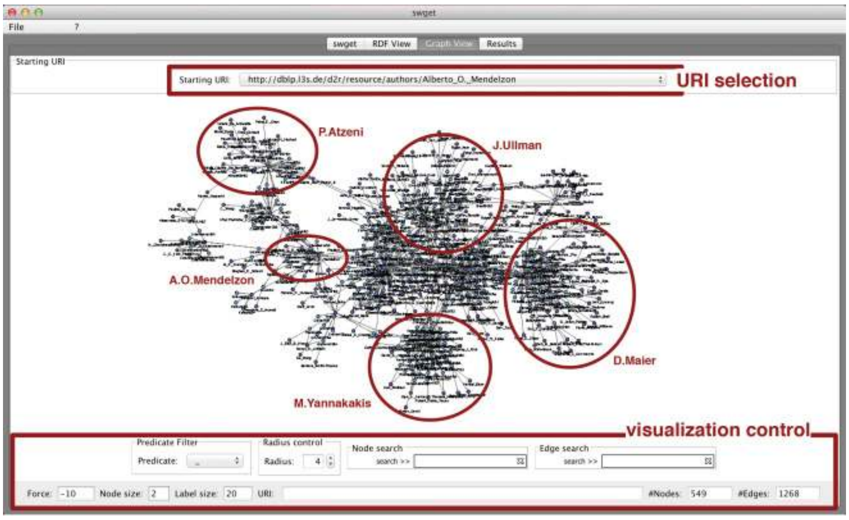


Fig. 12. Exploring the results of the evaluation of a NAUTILOD expression.

(Sections 3.4.1 and Section 4.4). In addition, in Section 5 we introduced an implementation of NAUTILOD in the swget tool, which is available in three different releases. In this section, we discuss an experimental evaluation aimed at assessing the cost of evaluating NAUTILOD expressions (via swget) on the WLOD. All the experiments have been performed on a PC with an Intel 2.5GHz Core i5 processor and 8GBs of RAM memory. Running times are the average of 5 runs; moreover, the number of results and the number of dereferenced URIs are rounded to the next integer value. The objective of the evaluation was to measure (i) execution time, (ii) the number of URIs dereferenced, (iii) the number of results retrieved, and (iv) performance of swget as compared to related systems.

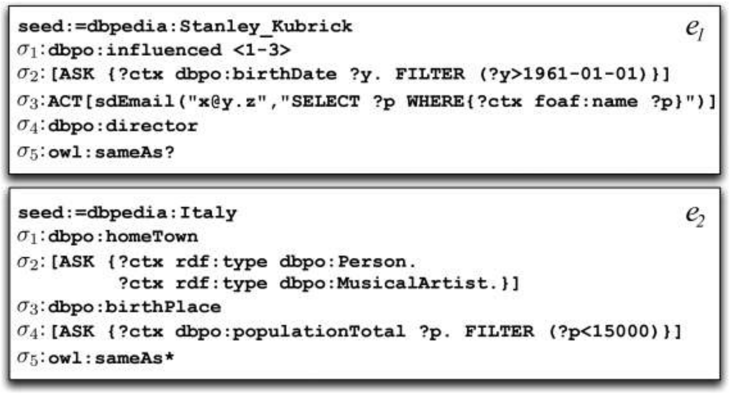
6.1. Experiment 1: Semantics Returning Nodes

The first experiment focuses on the semantics of NAUTILOD returning nodes introduced in Section 3.2 and the associated algorithms (i.e., Algorithm 1 and Algorithm 2).

6.1.1. The Cost of Evaluating NAUTILOD *Expressions.* In this section we discuss the various components that affect the cost of the evaluation of a NAUTILODexpression. We first introduce some notation. We denote by e a NAUTILOD expression, A the set of actions, and T the set of tests specified in e that have to be performed when evaluating it, and by $\bar{e}$ the action-and-test-free expression obtained by removing from e all tests in T and actions in A. Abstractly, we can separate the cost of evaluating e over a WLOD instance $\mathcal{W}$ in three parts:

$$\text{cost}(e, \mathcal{W}) = \text{cost}(\bar{e}, \mathcal{W}) + \text{cost}(A) + \text{cost}(T). \tag{1}$$

The first component considers the cost of evaluating the expression by taking out actions and tests. Actions do not affect the navigation process and then we can treat their cost separately. Moreover, tests are ASK-SPARQL queries having a different structure from the pure navigational path expressions of the language; even in the case of tests we can treat their cost independently. The cost of actions has essentially two components: *execution* and *transmission*. The execution cost boils down to the cost of evaluating the SELECT SPARQL query that gives the action's parameters. As for transmission costs, a typical example is the sending of an email message including



```
seed:=dbpedia:Stanley_Kubrick                                    e1
σ1:dbpo:influenced <1-3>
σ2:[ASK {?ctx dbpo:birthDate ?y. FILTER (?y>1961-01-01)}]
σ3:ACT[sdEmail("x@y.z","SELECT ?p WHERE{?ctx foaf:name ?p}")]
σ4:dbpo:director
σ5:owl:sameAs?
```

```
seed:=dbpedia:Italy                                              e2
σ1:dbpo:homeTown
σ2:[ASK {?ctx rdf:type dbpo:Person.
         ?ctx rdf:type dbpo:MusicalArtist.}]
σ3:dbpo:birthPlace
σ4:[ASK {?ctx dbpo:populationTotal ?p. FILTER (?p<15000)}]
σ5:owl:sameAs*
```

Fig. 13. Expressions used in the evaluation. Expressions have been executed incrementally and as a whole.

some data. In this case the cost is that of sending such email. Note that what really matters for our discussion is not the whole $\mathcal{W}$, but only the set of nodes visited when evaluating the expression; we will refer to such set as $\mathcal{W}_{\overline{e}}$. Thus, we have

$$\text{cost}(e, \mathcal{W}) = \text{cost}(\overline{e}, \mathcal{W}_{\overline{e}}) + \text{cost}(A) + \text{cost}(T). \tag{2}$$

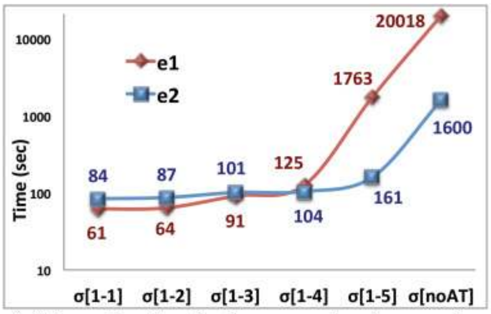
Note that the usage of tests possibly reduces the size of the set of nodes visited during the evaluation. Thus, the cost$(\overline{e}, \mathcal{W}_{\overline{e}})$ has to be reduced to take into account the effective subset of nodes reachable after the filtering performed via tests. Let $\mathcal{W}_T$ be the portion of $\mathcal{W}_{\overline{e}}$ when taking into account this filtering. We have

$$\text{cost}(e, \mathcal{W}) = \text{cost}(\overline{e}, \mathcal{W}_T) + \text{cost}(A) + \text{cost}(T). \tag{3}$$

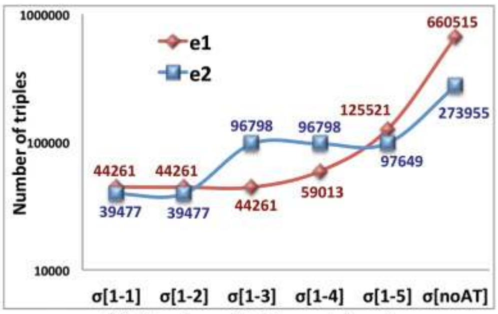
Finally, we want to point out that there is a relation between the diffusion (portion of linked datasources using a given predicate) and selectivity (the number of RDF links with a given predicate for a given URI) of the predicates used in the NAUTILOD expression and its evaluation cost. In particular, the usage of predicates having lower diffusion and higher selectivity allow one to reduce the size of the portion of the WLOD visited during the evaluation and the generated network traffic. For example, we noticed that some predicates (e.g., `dbpedia:influenced`) have a high diffusion and a low selectivity thus allowing one to span a larger portion of the WLOD and causing the dereferencing of several URIs. Some other predicates (e.g., `dbpedia:director`) have a lower diffusion allowing one to reduce the portion of the WLOD that is reached during the evaluation of an expression. Finally, other predicates such as `dbpedia:birthDate` have a high diffusion but very high selectivity (having at most one occurrence for each URI visited). An interesting study about these aspects has been recently performed [Schmachtenberg et al. 2014].

6.1.2. Experiments. We evaluate the impact of the different components of a NAUTILOD expression according to the equations introduced previously. We chose two complex expressions (shown in Figure 13) and measured execution time (t), number of URIs dereferenced (d), and number of triples retrieved (n). To investigate how the various components affect the parameters t, d, and n, each NAUTILOD expression has been divided into five parts (i.e., $\sigma_i,\ i \in \{1\ldots5\}$). Such parts have been executed as a whole (i.e., $\sigma_{[1-5]}$) and as action-and-test-free expressions (i.e., $\sigma_{[\text{no AT}]}$), for which we use the notations $\overline{e}_1$ and $\overline{e}_2$, respectively. Moreover, the various subexpressions (i.e., $\sigma_{[1-i]},\ i \in \{1\ldots4\}$) have also been executed. This leads to a total of 12 expressions.

The results, in logarithmic scale, are reported in Figures 14 and 15. In particular, in the x axis are reported from left to right: the four subexpressions, the full expression

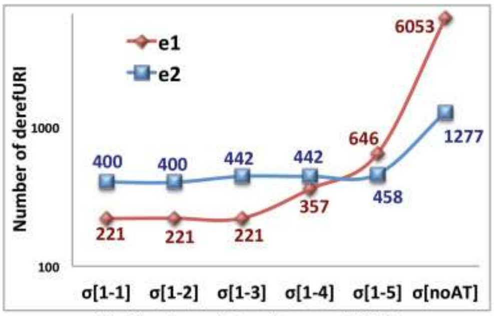


(a) Execution time for the expressions in seconds.

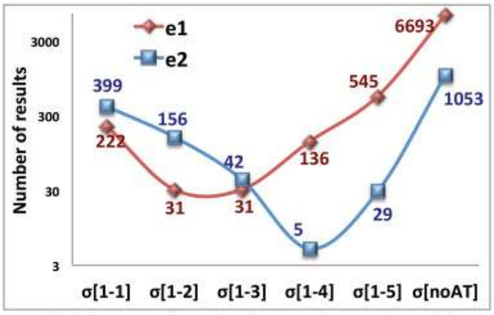


(b) Number of triples retrieved.

Fig. 14. Evaluation of swget on e_1 and e_2.



(a) Number of dereferenced URIs.



(b) Number of results (expression endpoints).

Fig. 15. Evaluation of swget on e_1 and e_2.

(i.e., $\sigma_{[1-5]}$), and the action-and-test-free expression (i.e., $\sigma_{[\text{no AT}]}$). The first expression (e_1) starts by looking for people influenced by Stanley Kubrick up to distance 3 (subexpression $\sigma_{[1\text{-}1]}$). This operation requires ∼61 s, for a total of 221 URIs dereferenced. On the description of each of these 221 URIs, an ASK query is performed to select only those people that were born after 1961 (subexpression $\sigma_{[1-2]}$). The execution time of the queries was about 4 s (i.e., ≃0.02 sec, per query) with 31 entities selected. Then, an action is performed on the descriptions of these 31 entities by selecting the `foaf:name` to be sent via email (subexpression $\sigma_{[1-3]}$). The select operation, the rendering of the results in HTML format, and the transmission of the emails had a cost of about 25 s. The navigation continues from the 31 entities found before the triggering of the action to retrieve movies via the RDF property `dbpo:director` (subexpression $\sigma_{[1-4]}$). The cost was about 34 s, for a total of 136 movies. Finally, for each movie only one level of possible additional descriptions is searched by the `owl:sameAs` property (the whole expression $\sigma_{[1-5]}$) whose cost is 1638 sec, for a total of 409 new URIs available from multiple servers (e.g., `linkedmdb.org`, `freebase.org`) of which only 289 were dereferenceable.

Thus, we have that $\text{cost}(e_1, \mathcal{W}) = \text{cost}(\overline{e}_1, \mathcal{W}_{T_1}) + \text{cost}(A_1) + \text{cost}(T_1) = 1763$; with $\text{cost}(A_1) \simeq 25$ s, $\text{cost}(T_1) \simeq 4$ s and $\text{cost}(\overline{e}_1, \mathcal{W}_{T_1}) \simeq 1738$ s. When considering the test-and-action-free expression, $\text{cost}(\overline{e}_1, \mathcal{W}) \simeq 20018$ secs. Note that the evaluation of the ASK-SPARQL queries had a cost of about 4 s, and allowed one to reduce the portion of the WLOD navigated by $\overline{e}_1$; this enabled one to save about 20,018 − 1,738 = 18,280 s. Such a larger difference in the execution time is justified by the fact that the 221 initial URIs, selected by $\sigma_{[1-1]}$ were not filtered in the case of $(\overline{e}_1, \mathcal{W})$ and then caused a larger amount of paths to be followed at the second step of the evaluation. Indeed,

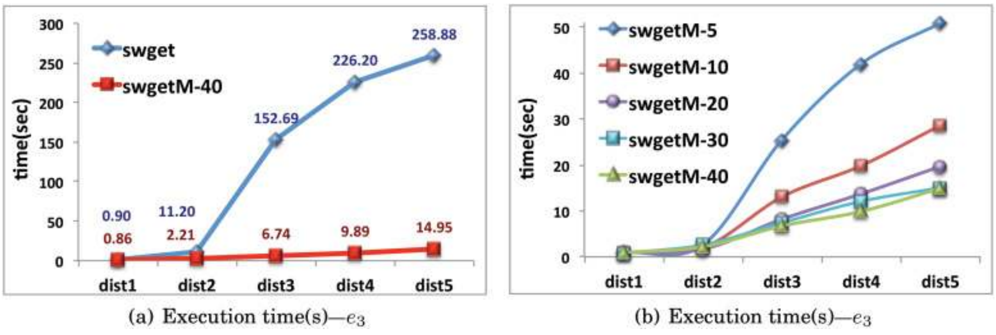


Fig. 16. Comparison between swget and SWGETM on e_3.

the total number of dereferenced URIs for $(\overline{e}_1, \mathcal{W})$ was 6,053, with about 660 K triples retrieved, while for $(\overline{e}_1, \mathcal{W}_{T_1})$ it was 646, with 125 K triples retrieved.

The second expression e_2 specifies the navigation of the property `dbpo:homeTown` to find entities living in Italy (subexpression $\sigma_{[1\text{-}1]}$); this had an execution time of about 84 s with a total of 400 dereferenced URIs (one seed plus 399 additional URIs). On the description of each URI, an ASK query filters entities that are of type `dbpo:Person` and `dbpo:MusicalArtist` (subexpression $\sigma_{[1-2]}$). All the queries were performed in about 3.8 s, with an average time per query of 0.01 s; this enabled one to select 156 people. At the second step, the navigation continues via the RDF property `dbpo:birthPlace` to find the places where these people were born (subexpression $\sigma_{[1-3]}$); this had a cost of about 14 s. In total, 42 new URIs were reached. The navigation continued with a second ASK query to select only those places where less than 15,000 habitants live (subexpression $\sigma_{[1-4]}$). The cost of performing the 42 ASK queries was of about 3 s and five places were selected. Finally, for each of the five places additional descriptions were searched by navigating the `owl:sameAs` property (the whole expression $\sigma_{[1\text{-}5]}$). This allowed one to reach a total of 29 URIs, some of which were external to `dbpedia.org`. The cost for this operation was of about 57 s.

As for the total cost, we have $\text{cost}(e_2, \mathcal{W}) = \text{cost}(\overline{e}_2, \mathcal{W}_{T_2}) + \text{cost}(A_2) + \text{cost}(T_2) = 161$. The factor $\text{cost}(A_2) = 0$ since e_2 does not contain any action whereas $\text{cost}(T_2) \simeq 6$ s, which gives $\text{cost}(\overline{e}_2, \mathcal{W}_{T_2}) = 155$ s. The cost of the action-and-test-free expression (i.e., $\overline{e}_2$) over $\mathcal{W}$ is $\text{cost}(\overline{e}_2, \mathcal{W}) \simeq 1600$ s, with 1277 URIs dereferenced. This is because the expression is not selective since it performs a sort of "semantic" crawling only based on RDF predicates. In fact, the number of triples retrieved (see Figure 14(b)) is almost three times higher than in the case of the expression with tests. By including the tests, the evaluation of $\overline{e}_2$ is 1445 s faster.

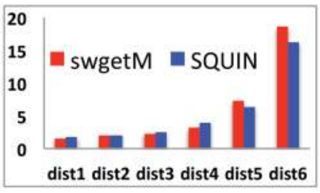
6.2. Experiment 2: swget vs SWGETM

We now compare `swget` against a new implementation that leverages multithreading (referred to as SWGETM). For this comparison we used the following expression:

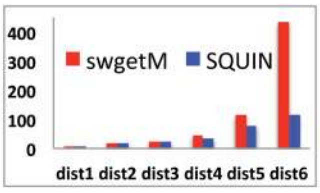
$$\boxed{\texttt{(foaf:knows)<1-X>}} \qquad (e_3)$$

The notation `<1-X>` is a shorthand for the concatenation of up to X `foaf:knows` edges. In the experiment, we considered values of X ranging from 1 to 5 and used A. Polleres' FOAF profile[16] as the starting node. The results of the comparison are reported in Figure 16. In particular, Figure 16(a) compares `swget` with SWGETM when

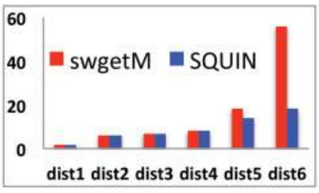
[16]http://www.polleres.net/foaf.rdf#me.



(a) Execution time(s)—e_4



(b) Number of URI deref.—e_4



(c) Number of results—e_4

Fig. 17. Comparison between SWGETM and SQUIN on $\boldsymbol{e_4}$.

using 40 threads. As can be observed, there is a major improvement especially for long running queries.

As an example, at dist5 in Figure 16(a), SWGETM is ~17 times faster than `swget`. In Figure 16(b) it can be observed that for SWGETM, a sensible speedup is obtained by increasing the number of threads up to 30. To test the benefit of SWGETM, we have also executed expressions e_1 and e_2 (see Section 6.1.2) with a pool of 30 threads; even in this case we obtained a significant speedup (~$5x$).

6.3. Experiment 3: Comparison with Related Work

We compared SWGETM against SQUIN[17] and SPARQL 1.1's property paths (for which we considered the DBpedia SPARQL endpoint[18]). Since SQUIN is based on a multithread implementation, we compared it with SWGETM only. Further details about the experimental setting and the complete query set are available at `http://swget.wordpress.com/evaluation`. We considered three different sets of expressions. The first set (e_4) includes the influence network of Tim Berners-Lee (TBL) in DBpedia restricted to scientists only. The seed node (i.e., TBL's URI in DBpedia) is `dbpedia:Tim_Berners-Lee` while the NAUTILOD expression is

```
(dbp:influenced)<1-X>[Test]
```
(e_4)

where the test is defined as follows:

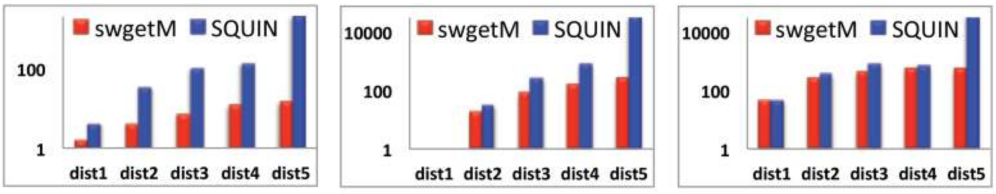
```
Test=ASK {?ctx rdf:type dbpo:Scientist.}
```

In the expression, the notation `<1-X>` is shorthand for the concatenation of up to X `dbp:influenced` while the `test` is used to filter ending nodes identifying scientists. In the experiment we considered values of X ranging from 1 to 6.

The execution times for e_4 are shown in Figure 17(a). As for SWGETM, we used a thread pool of size 5 to avoid many simultaneous connection requests that may overload servers and generate errors that would result in the loss of results. As compared to SQUIN, SWGETM reported a higher running time at dist6. However, by looking at Figure 17(b) it can be noted that SWGETM performs a much larger number of dereferencing operations (433 vs. 117).

This latter aspect had an impact on the number of (valid) results shown in Figure 17(c). Indeed, SWGETM retrieved 56 results while SQUIN retrieved 18. The reason for these differences stems from the link-traversal-based query execution mechanism based on nonblocking iterators implemented by SQUIN [Hartig 2011]. This approach cannot guarantee to discover all reachable URIs that may contribute to the final results [Hartig 2011] (a more detailed comparison is provided in Section 7). On

[17]http://www.squin.org.
[18]http://dbpedia.org/snorql/.



(a) Execution time(s)—e_5 (b) Number of URI deref.—e_5 (c) Number of results—e_5

Fig. 18. Comparison between swget multithread (SWGETM) and SQUIN over e_5.

the other hand, NAUTILOD at the core of SWGETM guarantees that all the parts of the WLOD graph that may contribute to the results are visited. We also want to point out that with SQUIN it was not possible to consider the closure via (owl:sameAs)* that would enable one to find aliases of TBL in other datasources. To make the comparison possible, we considered an additional expression (not reported here) where up to three levels of owl:sameAs were considered. The running times of SWGETM and SQUIN were comparable (~45 s at distance 6 with three levels of owl:sameAs). However, the number of results provided by SWGETM was higher.

The second set of expressions used in the comparison (i.e., e_5) considers the expressions previously discussed in Experiment 2. The comparison between SWGETM and SQUIN over e_5 is reported in Figure 18 (results are in log scale). The number of results varies from 43 to 2.6 K. The number of results between SWGETM and SQUIN varies due to the fact that SQUIN can return blank nodes identifiers as results while NAUTILOD discards blank nodes. Also for this experiment, the size of the thread pool has been set to five. As for SQUIN, the running time ranges from ~4 s to ~250 s. As for SWGETM, the running time varies from ~1 s to ~50 s. During the evaluation of e_5, the running times have been affected by I/O exceptions due to missing FOAF profiles.

Comparison with SPARQL 1.1's Property Paths. In this experiment, we used another expression (***e*₆**) that looks for Stanley Kubrick's codirectors at distance 6. The seed node is dbpedia:Stanley_Kubrick while the NAUTILOD expression is

```
(dbpo:director/dbpo:director)<6-6>
```
(e_6)

On e_6, we compared SWGETM with SPARQL 1.1's Property Paths (PPs) and SQUIN. It is crucial to note that PPs are a means to deal with paths that link RDF triples available in a *local* graph while SWGETM and SQUIN target the WLOD graph where paths exist between distributed (and a priori unknown) datasources. Nevertheless, we compared SWGETM against PPs on DBpedia's SPARQL endpoint (DBse). This expression took ~50 s on DBse, ~180 s on SWGETM, while we stopped the execution of SQUIN after ~2 h. It is interesting to observe that DBse uses a local triplestore, while SWGETM and SQUIN work on the Web. Indeed, SWGETM performed a total of ~3,400 dereferencing operations.

The last set of expressions (e_7) asks for the influence closure of TBL ((dbp:influenced)* ($e_{7.1}$) and the influence closure by only considering scientists (dbp:influenced[ASK {?ctx rdf:type dbpedia:Scientist}])* ($e_{7.2}$). As for $e_{7.1}$, SQUIN cannot express such request since it does not support PPs. And, even if this would be possible, PPs cannot be evaluated over the WLOD graph. On this query, DBse did not provide any result because of a memory overflow after ~2 min, while SWGETM provided 1,792 results in ~172 s. As for $e_{7.2}$, note that this request cannot be expressed either by PPs or by SQUIN due to the lack of tests when using closure operators (i.e.,*, +). SWGETM provided eight results with 21 dereferencing operations in ~2.5 s.

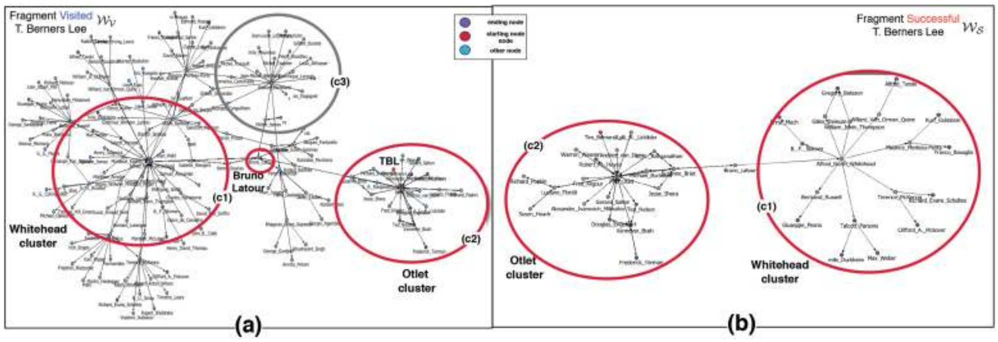


Fig. 19. Fragments obtained according to the VISITED (a) and SUCCESSFUL semantics $\mathcal{W}_S$ (b) for TBL.

6.4. Experiment 4: Semantics Returning Web Fragments

In this section we provide some examples of Web fragments that can be specified with NAUTILOD. We report results according to both the VISITED semantics V and SUCCESSFUL semantics S indicated as $\mathcal{W}_V$ and $\mathcal{W}_S$, respectively.

6.4.1. Influence Networks. An influence network is a graph where nodes are different kinds of persons and edges represent influence relations. We leverage information taken from `dbpedia.org` where the notion of influence is captured via the property `dbp:influenced` defined in the DBpedia ontology. We consider two different fragments: a fragment built starting from TBLand another starting from Stanley Kubrick (SK). Figures in higher quality along with the representation in RDF of the Web fragments are available at the `swget` Website.[19] Such fragments can be loaded and explored by using the `swget` GUI.

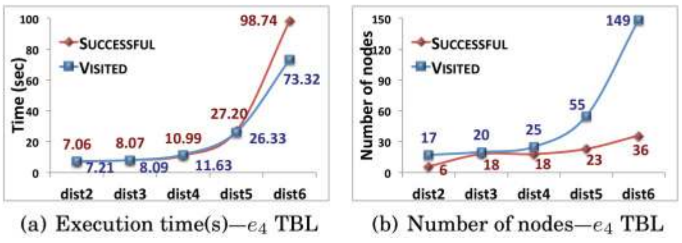
Example 6.1. Specify two Web fragments including scientists that have influenced or have been influenced up to distance 6 by TBL and SK, respectively.

The Web fragments can be specified via the NAUTILOD expression e_4 in Section 6.3 by considering the URIs of TBL and SK in DBpedia as seed nodes.

Influences of TBL. Figure 19(a) reports the Web fragment obtained with the VISITED semantics for TBL. This fragment contains 149 nodes and 236 edges. Recall that according to the specification of the fragment only scientists have to be considered as ending nodes. As can be observed, there are two main influence clusters: the first (c1) around Whitehead and the second (c2) around Otlet. Note also that Latour is the bridge between the two clusters but he is not a scientist. In the fragment there are several nodes that were visited during the evaluation of the expression but did not lead to any result. For instance, the cluster of nodes (c3) did not contribute to reach any scientist. Moreover, observe that while Otlet is an ending node, Whitehead is not; however, via Whitehead other scientists (e.g., Peano) have been reached. TBL belongs to the influence cluster (c2). By looking at the fragment it is possible to reconstruct the whole influence path (also including nonscientists) between any two nodes. As an example it is possible to get the the full path between TBL and G. Peano, that is, TBL$\rightarrow$P. Outlet$\rightarrow$S. Briet$\rightarrow$B. Latour$\rightarrow$A. N. Whitehead$\rightarrow$B. Russel$\rightarrow$G. Peano. The `swget` GUI facilitates the discovery and exploration of such paths.

Figure 19(b) reports the fragment obtained with the SUCCESSFUL semantics for TBL; the fragment contains 40 nodes and 70 edges. Here, the two clusters identified before

[19]http://swget.wordpress.com.



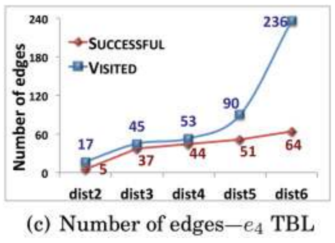


Fig. 20. Comparison between Successful and Visited semantics on e_4 starting from TBL.

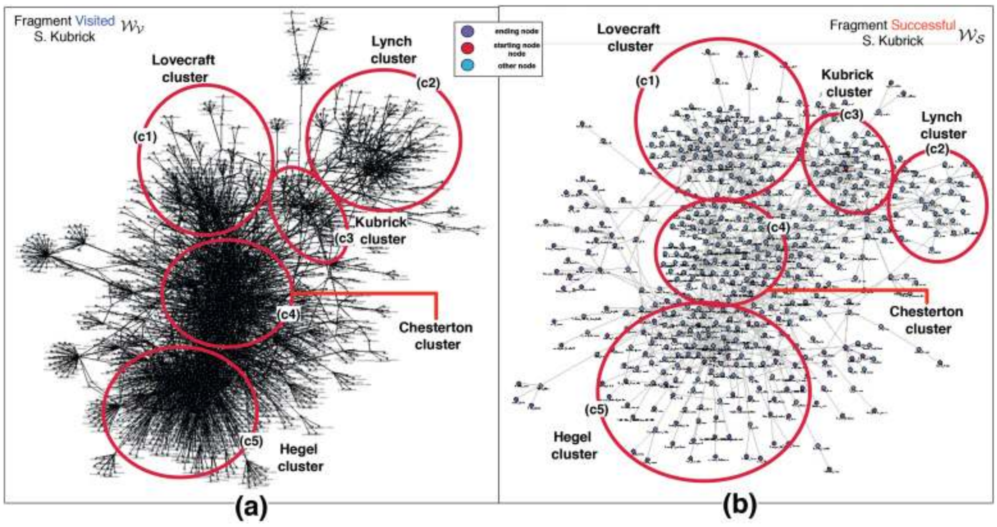


Fig. 21. Fragments obtained according to the Visited (a) and Successful semantics (b) for SK.

are clearer, and the cluster (c3) disappeared since it did not lead to any result. The flexibility of NautiLOD to provide the two semantics is useful to understand why a node has been considered or not in the fragment. As an example, in clusters (c1) and (c2) identified before it was possible to understand the importance of B. Latour in the influence network of TBL. Figure 20 provides a comparison between the two semantics on e_4. In particular, Figure 20(a) shows the difference in time to build the fragment according to the Successful and Visited semantics. Figures 20(b) and 20(c) show the difference in terms of the number of nodes and edges retrieved with the two semantics. As can be observed, the Successful semantics enables one to reduce the number of nodes by ~75% and edges by ~73% at distance 6.

Influences of SK. Figure 21(a) reports the Visited Web fragment associated with the influence network of SK. The fragment contains 2,981 nodes and 7,893 edges. In the figure, five clusters of nodes can be recognized centered around SK and four other hub nodes (i.e., Lovecraft, Lynch, Chesterton, and Hegel). However, clusters are too big to single out nodes satisfying the expression and influence paths among them. This is also true for hub nodes. In this particular case, none of the four hub nodes is a scientist. Hence, one may wonder whether these four hub nodes belong to some influence path connecting S. Kubrick with ending nodes. This question can be answered by looking at the Successful fragment shown in Figure 21(b), which contains 530 nodes and 1,375 edges. The same clusters, now containing a significant lower number of nodes are

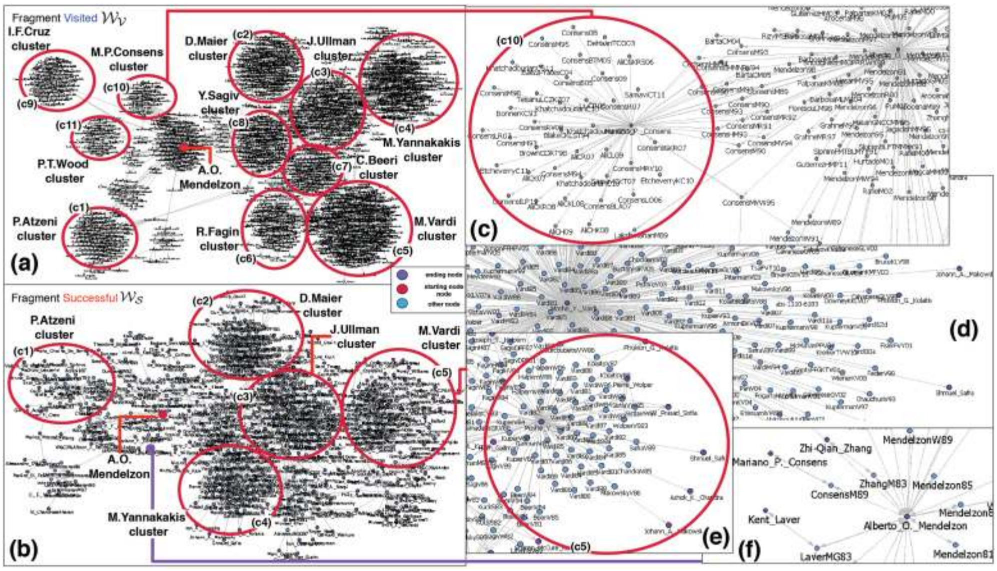


Fig. 22. Fragments obtained according to the VISITED (a) and SUCCESSFUL semantics (b).

shown. Since all the previous four hub nodes belong to the SUCCESSFUL fragment they allow one to reach some scientist indirectly influenced by Kubrick. Indeed, by exploring such a fragment one can discover that from Kubrick we can reach C. Segan (a scientist) by passing through H. P. Lovecraft; O. Rank by passing through D. Lynch; S. Freud by passing through G. K.Chesterton; and J. Kepler and N. Tesla by passing through G. W. F. Hegel.

6.4.2. Coauthorship Web Fragments. In this scenario, nodes represent authors and papers and edges authorship relations. To give an example, we leverage information in RDF taken from `dblp.l3s.de` where the notion of authorship is captured via the property `foaf:maker`. We have built the network starting from Alberto O. Mendelzon (AOM) by considering his coauthors up to distance 2 only on papers published between 1980 and 1990. The Web fragments can be specified via the following NAUTILOD expression and using the URI of AOM in the `dblp.l3s.de` datasource as the starting node:

```
dblp:Alberto_O._Mendelzon (foaf:maker[Test]/foaf:maker)<1-2>
```

where the test is defined as follows:

```
Test=ASK{?ctx dc:issued ?y. FILTER( ?y>''1980''^^<xsd:gYear>).
                FILTER(?y<''1990''^^<xsd:gYear>).}
```

Figure 22(a) reports the fragment associated to AOM according to the VISITED semantics. The fragment contains 2,110 nodes and 2,929 edges. The SUCCESSFUL fragment is shown in Figure 22(b), which contains 549 nodes and 1,268 edges.

Some of the clusters present in the VISITED fragment disappear in the SUCCESSFUL fragment due to the test in the NAUTILOD expression (i.e., clusters c6–c11). As an example, Mariano P. Consens belongs to the ending nodes of the NAUTILOD expression because he coauthored a paper with AOM in 1989 and also his papers belong to the VISITED fragment (Figure 22(c)). However, the cluster to which he belongs disappears from the SUCCESSFUL fragment (see Figure 22(f)) since all the other papers he coauthored (that belong to the VISITED fragment) have been published after 1990. On the contrary, the cluster c5 around M. Y. Vardi belongs both to the VISITED and SUCCESSFUL fragments since he published papers in the decade 1980–1990 that allow one to reach other ending nodes (e.g., S. Salfra and P. G. Kolaitis). However, also in this case, the cluster in the VISITED fragment (Figure 22(d)) contains many more nodes representing papers. These nodes disappear from the SUCCESSFUL fragment (see Figure 22(e)) since they have been published after 1990.

7. RELATED WORK

Many of the ideas underlying our proposal have been around in particular settings. We owe our inspiration to several of them. We will organize the discussion of the different influences and related works by topics.

Browsers. The first class of tools developed to access data in the WLOD were browsers such as Tabulator [Berners-Lee et al. 2006] and OpenLink Data Explorer;[20] such tools enabled *manual* navigation on the WLOD graph in the sense that they require human intervention to perform the traversal of edges. These tools have evolved to include support for *faceted* navigation ([Hildebrand et al. 2006], BrowseRDF [Oren et al. 2006]) where information is classified in facets that can be used to filter RDF data. Other proposals such as Explorator [Araujo and Schwabe 2009] and RExplorator [Cohen and Schwabe 2012] have been proposed to also support direct manipulation of the pieces of RDF information being navigated. The departure point of NAUTILOD is different; it formalizes and features a declarative and *automatic* navigation mechanism for the WLOD. NAUTILOD enables one to express high-level specifications of information on the WLOD that can be retrieved without human intervention. Such specification can also include actions to be performed using data encountered during the navigation.

Retrieval of Sources (crawling). Specification (and retrieval) of collections of sites was addressed earlier, and a good example is the `wget` GNU tool.[21] Besides being nondeclarative, it is restricted to almost purely syntactic features. The execution philosophy of `wget` was a source of inspiration for the incorporation of actions into NAUTILOD and for the design of `swget`. At the semantic level, Isele et al. [2010] proposed LDSpider, a crawler for the Web of Data able to retrieve RDF data by following RDF links according to different crawling strategies. LDSpider has little flexibility and is not declarative. After NAUTILOD, another language, called LDPath [Schaffert et al. 2012], has been proposed to enable navigation on the WLOD. Differently from NAUTILOD, LDPath does not have a formal semantics and does not allow the specification of actions.

Graph Navigational and Query Languages. Languages for the *navigation and specification* of nodes in graphs have a long tradition. Among them, we recall XPath (for XML) and some proposals extending SPARQL with navigational features (e.g., Alkhateeb et al. [2009], Kochut and Janik [2007], Pérez et al. [2010], Zauner et al. [2010], and SPARQL 1.1). Also several query languages for the Web have been proposed (see Florescu et al. [1998] for a survey) but these are not grounded on semantic technologies and languages. Nonetheless, all these languages have inspired the navigational core of NAUTILOD. A recent survey on graph query languages is given by Wood [2012].

In general, languages to query the Web enable the evaluation of path expressions also including tests, and NAUTILOD uses the core functionality of them (very much like XPath). In almost all such languages the evaluation of an expression returns a set of nodes. In our case, NAUTILOD expressions can also return graphs (i.e., fragments of the

[20]http://demo.openlinksw.com/ode.
[21]www.gnu.org/software/wget/.

Table III. Comparison with Related Research in Graph Query/Navigational Langauges

System/Language	Scope	Output
NautiLOD	WLOD	Regions
LDSpider [Isele et al. 2010]	WLOD	Nodes
LDPath [Schaffert et al. 2012]	WLOD	Nodes
SPARQL 1.1 SERVICE	Federated queries	Nodes
PSPARQL [Alkhateeb et al. 2009]	Local graph	Nodes
nSPARQL [Pérez et al. 2010]	Local graph	Nodes
Property paths [Harris and Seaborne 2013]	Local graph	Nodes
Zauner et al. [Zauner et al. 2010]	Local graph	Nodes
SPARQLeR [Kochut and Janik 2007]	Local graph	Path expressions
ECRPQ [Barceló et al. 2012]	Local graph	Regular expressions

Web). Recently (in parallel to the development of NautiLOD) there has been research on graph languages returning graphs. For example, in the formalism ECRPQ [Barceló et al. 2012] the output are paths found by the query. Also another work that introduced path variables was SPARQLeR. The fundamental difference with respect to NautiLOD is that, while they return the expressions formed by the labels of the subgraph traversed (e.g., regular expressions in ECRPQ), NautiLOD returns the actual nodes of the subgraphs traversed plus their labels, that is, actual pieces of the Web subgraph. Finally, NautiLOD does not assume data stored in a central repository (typically a single RDF graph), but target distributed graphs such as the WLOD. NautiLOD incorporates a basic type of actions, technically side effects, to allow basic communication to users and shipping of data. Table III summarizes the comparison between NautiLOD and related graph/navigational languages.

Distributed and active approaches. Active XML [Abiteboul et al. 2002] is a declarative framework for peer-to-peer data and service integration for the Web. It defines a datatype for documents, which has embedded calls to web services, plus a language to define them. Its goals are different from NautiLOD, in the sense that NautiLOD is designed to take advantage of the current data over the WLOD (which do not have calls incorporated), and actions (calls) do not transform datasources, but have the meaning of metacommunication channels; in particular, sending data to final users and performing basic communication (e.g., sending emails).

In the Semantic Web world, Papamarkos et al. [2004] proposed an “Event-Condition-Action” language for RDF called RDFLT. It was motivated by the need of having, in distributed environments, timely notifications of metadata changes and “mechanisms for monitoring and processing such a changes.” It has similarities with NautiLOD in the notion of navigation. But RDFLT is oriented to catch events and fire insertion or deletion of a set of nodes defined by a navigational expression, and NautiLOD was designed to specify/retrieve expressions of regions of the Web. Another language, called Data-Fu, was proposed by Stadtmüller et al. [2013] for interactions in the WLOD based on a RESTful approach. Data-Fu points to a different goal than NautiLOD. While the former works to allow interactions between multiple providers with read/write data functionalities (assuming data extended with RESTful descriptions), NautiLOD points to read, filter, and integrate standard data from multiple distributed sources on the WLOD. Finally, regarding distribution, there have been many proposals in the Semantic Web landscape. Rakhmawati et al. [2013] surveys different models of federation that use endpoints. Again, NautiLOD was used to take advantage of standards for data on the Web, thus does not consider endpoints (although it could be extended to that functionality).

Querying/Navigation on the WLOD. Here we identify three main lines of research:

(1) Load the *desired* data into a single RDF store (by *crawling* parts of the WLOD) and process queries in a centralized way [Hose et al. 2011]. There have also been developments in indexing techniques for semantic data like Swoogle [Ding et al. 2004], Sindice [Oren et al. 2008], and Watson [d'Aquin and Motta 2011]. An approximate index structure has been proposed in Harth et al. [2010].

(2) Process queries in a distributed form by using a federated query processor. The SERVICE feature of SPARQL 1.1 and proposals like DARQ [Quilitz and Leser 2008] and FedX [Schwarte et al. 2011] provide mechanisms for transparently query answering on multiple query services. The query is split into subqueries that are forwarded to the individual datasources and their results are processed together. An evaluation of federated query approaches can be found in Haase et al. [2010].

(3) Extend SPARQL with navigational features to discover datasources relevant for answering SPARQL queries on the WLOD. This approach is referred to as "Link Traversal-Based Query Execution" (LTQBE). SQUIN [Hartig et al. 2009; Hartig 2011] and LiDaQ [Umbrich et al. 2014] are two implementations (and formalizations) of such an approach. At their core, there is the possibility to navigate toward distributed datasources while executing a query. SQUIN[22] is an iterator-based implementation of the LTQBE paradigm, while LiDaQ [Umbrich et al. 2014] also considers lightweight reasoning extensions to find additional sources on-the-fly and generate further answers. Recently, an analysis of subwebs that are reachable by LTQBE engines was performed [Hartig and Ozsu 2014] giving some hints about why results obtained with these engines are not complete. We discuss the differences between LTQBE engines and `swget` both in terms of scope and expressiveness.

First, we want to point out that LTQBE and `swget` have different departure points. LTQBE uses "implicit" (noncontrolled) navigation to collect information from the WLOD to perform query answering (i.e., obtain a set of variable mappings). Hence, navigation is a "means" to retrieve such information. Approaches based on LTQBE do not mean to be (pure) navigational languages. `swget` leverages NAUTILOD, which is a pure navigational language that uses (Boolean ASK SPARQL) queries to select datasources. Hence, navigation (not querying) is the main actor. While NAUTILOD uses queries to enhance/control the navigation, LTQBE proceeds in the reverse order. For example, the SQUIN implementation of the LTQBE mechanism based on nonblocking iterators cannot guarantee that all reachable URIs that may contribute to the final results are discovered. This is because the building blocks of a SQUIN query (i.e., SPARQL BGPs) are evaluated in a fixed order; when an iterator obtains an intermediate solution from the previous iterator, it replaces the previously obtained intermediate solution. Hence, the (replaced) intermediate solution cannot be combined with any data that arrives later. As for SPARQL 1.1's, its navigational core (i.e., property paths) is meant to deal with paths that link RDF triples available in a *local* graph. On the other hand, NAUTILOD (and `swget`) deals with the WLOD graph where paths exist between distributed (and a priori unknown) datasources.

Overall, there are some substantial differences between our proposal and the previously mentioned strands of research in terms of querying/navigating the WLOD (as also summarized in Table IV): NAUTILOD focuses on *navigational* functionalities, thus departing from querying as in (2); emphasizes specification of autonomous distributed sources, as opposed to (1); uses SPARQL querying to enhance navigation, while (3) proceeds in the reverse direction; and incorporates actions that in some sense generalize procedures implicit in the evaluation over the Web (e.g., "get data" in crawlers and "return data" in query languages).

[22]http://squin.org.

Table IV. Comparison with Related Research about Querying/Navigation on the WLOD

System/Language	Scope	Techniques and Features
NautiLOD	WLOD	Controlled Navigation, Actions, Web fragments
SQUIN [Hartig et al. 2009; Hartig 2011]	WLOD	Navigation for Querying
LiDaQ [Umbrich et al. 2014]	WLOD	Navigation for Querying
Swoogle [Ding et al. 2004]	Crawled data	Indexing for querying
Sindice [Oren et al. 2008]	Crawled data	Indexing for querying
Watson [d'Aquin and Motta 2011]	Crawled data	Indexing for querying
Harth et al. [Harth et al. 2010]	Federated queries	Indexing for querying
DARQ [Quilitz and Leser 2008]	Federated queries	Querying
FedX [Schwarte et al. 2011]	Federated queries	Querying
RDFTL [Papamarkos et al. 2004]	Local graph	Navigation, Events
Data-Fu [Stadtmüller et al. 2013]	WLOD	Actions

8. DISCUSSION

We now deepen some aspects underlying the practical usage of NautiLOD and `swget`.

Datasource navigation. Similarly to other navigational and query languages for RDF, the usage of NautiLOD requires some familiarity with the schema of the data. In particular, the user should have some knowledge of which RDF predicates can help in expressing a conceptual specification. An increasing level of difficulty with NautiLOD is the fact that the evaluation of an expression can span over multiple (and different) schemata. Fortunately, there is an increasing adoption of metaschemata such as `schema.org` whose underlying idea is that of providing a common vocabulary to structure knowledge domains. Indeed, mappings from `schema.org` already exist with other schemata such as DBpedia.[23] Hence, NautiLOD users should only learn one schema as expressions written in `schema.org` will be dynamically translated into the schemata of the datasources encountered during the navigation. Another way of circumventing such problem would be the analysis of metadata about datasets (e.g., VoID descriptions [Alexander et al. 2010]). Last but not least, a challenging research goal is that of providing translations from natural language questions to NautiLOD expressions as done for SPARQL queries by systems like DEANNA [Yahya et al. 2012].

Performance. `swget` expressions are evaluated over the live WLOD without the need of SPARQL endpoints. Performance (in terms of execution time) depends on different factors: (i) load of the servers (and polite crawling), (ii) quality of the Internet connection, (iii) use of cache (as described in Section 5), and (iv) complexity of the expression. As for point (iv), the availability of statistics about the usage of predicates can help in determining their selectivity; this will help in estimating the cost of the evaluation of NautiLOD expressions. We want to point out that the `swget` engine can be configured (by passing a flag) to stream results as they are available. Hence, for queries that may take long, users can stop the execution and still get some results. Finally, the system also enables one to specify a budget in terms of maximum number of dereferencing operations and stop the execution when reaching the budget.

9. CONCLUDING REMARKS

Knowledge in the form of semantic graphs pervades everyday life. Initiatives such as Facebook Graph, Google Knowledge Graph, and Linking Open Data are examples of this trend. In this new world of structured data at Web scale, navigation becomes a relevant part of the toolbox to query and extract knowledge from the Web. As thousands

[23]http://schema.rdfs.org/mappings.html.

of distributed and interlinked semantic datasources are becoming available, navigation can be raised to the level of semantic navigation. We showed the importance of having a declarative language for the automatic navigation on the Web. This language, called NAUTILOD, and its implementation in swget show the feasibility and potentialities of semantic and automatic navigation at a Web scale. Our experience indicates that there are still some pieces missing to bring this potential to a planetary level. Among the most important there is the need for standardized semantic metadata about datasources and protocols to publish data.

REFERENCES

S. Abiteboul, O. Benjellourn, I. Manolescu, T. Milo, and R. Weber. 2002. Active XML: Peer-to-peer data and web services integration. In *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB Endowment, 1087–1090.

S. Abiteboul and V. Vianu. 1997. Queries and computation on the web. In *Proceedings of the International Conference on Database Theory*, Vol. 1186. 262–275.

K. Alexander, R. Cyganiak, M. Hausenblas, and J. Zhao. 2010. Describing Linked Datasets with the voiD Vocabulary. Retrieved from http://www.w3.org/2001/sw/interest/void/.

F. Alkhateeb, J.-F. Baget, and J. Euzenat. 2009. Extending SPARQL with regular wxpression patterns (for querying RDFsparq). *J. Web Semantics* 7, 2 (2009), 57–73.

R. Angles and C. Gutierrez. 2008. Survey of graph database models. *Comput. Surveys* 40, 1 (Feb. 2008), 1–39.

S. Araujo and D. Schwabe. 2009. Explorator: A tool for exploring RDF data through direct manipulation. In *Linked Data on the Web (CEUR Workshop Proceedings)*, Vol. 538.

F. Baader and T. Nipkow. 1999. *Term Rewriting and All That*. Cambridge University Press.

P. Barceló, L. Libkin, A. W Lin, and P. T Wood. 2012. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst. (TODS)* 37, 4 (2012), 31.

T. Berners-Lee. 1998. What the Semantic Web Can Represent. Retrieved from http://www.w3.org/Design Issues/RDFnot.html.

T. Berners-Lee. 2006. Linked Data Design Issues. Retrieved from http://www.w3.org/DesignIssues/Linked Data.html.

T. Berners-Lee, Y. Chen, L. Chilton, D. Connolly, R. Dhanaraj, J. Hollenbach, A. Lerer, and D. Sheets. 2006. Tabulator: Exploring and analyzing linked data on the semantic web. In *Proceedings of the International Semantic Web User Interaction Workshop*.

S. Brin and L. Page. 1998. The anatomy of a large-scale hypertextual web search engine. *Comput. Netw. ISDN Syst.* 30, 1–7 (1998), 107–117.

J. Clark and S. DeRose. 1999. XML path language (XPath) version 1.0. W3C Recommendation 16 November 1999. Retrieved from http://www.w3.org/TR/xpath/.

M. Cohen and D. Schwabe. 2012. Support for reusable explorations of linked data in the semantic web. In *SeCO Book*, Stefano Ceri and Marco Brambilla (Eds.). Lecture Notes in Computer Science, Vol. 7538. Springer, 176–190.

M. d'Aquin and E. Motta. 2011. Watson, more than a Semantic Web Search Engine. *Semantic Web* 2, 1 (2011), 55–63.

L. Ding, T. Finin, A. Joshi, R. Pan, R. S. Cost, Y. Peng, P. Reddivari, V. C. Doshi, and J. Sachs. 2004. Swoogle: A search and metadata engine for the semantic web. In *Proceedings of the ACM International Conference on Information and Knowledge Management*. ACM, 652–659.

V. Fionda, C. Gutierrez, and G. Pirrò. 2012. Semantic navigation on the web of data: Specification of routes, web fragments and actions. In *International World Wide Web Conference*. ACM Press, New York, NY, 281–290.

V. Fionda, C. Gutierrez, and G. Pirrò. 2014a. Knowledge maps of web graphs. In *International Conference on Principles of Knowledge Representation and Reasoning (KR)*. AAAI Press.

V. Fionda, C. Gutierrez, and G. Pirrò. 2014b. The swget portal: Navigating and acting on the Web of Linked Data. *J. Web Semantics* 26 (2014), 29–35.

V. Fionda, G. Pirrò, and C. Gutierrez. 2014c. The map generator tool. In *Proceedings of the ISWC 2014 Posters & Demonstrations Track*. 81–84.

D. Florescu, A. Levy, and A. O. Mendelzon. 1998. Database techniques for the World-Wide Web: A survey. *SIGMOD Rec.* 27, 3 (1998), 59–74.

Y. Gil and P. Groth. 2011. Using provenance in the semantic web. *J. Web Semantics* 9, 2 (2011), 147–148.

P. Haase, T. Mathäb, and M. Ziller. 2010. An evaluation of approaches to federated query processing over linked data. In *I-SEMANTICS*.

S. Harris and A. Seaborne. 2013. SPARQL 1.1 Query Language, W3C Recommendation. (2013). http://www.w3.org/TR/sparql11-query/.

A. Harth, K. Hose, M. Karnstedt, A. Polleres, K. Sattler, and J. Umbrich. 2010. Data summaries for on-demand queries over linked data. In *Proceedings of the International World Wide Web Conference*. ACM, 411–420.

O. Hartig. 2011. Zero-knowledge query planning for an iterator implementation of link traversal based query execution. In *Proceedings of the Extended Semantic Web Conference*. 154–169.

O. Hartig, C. Bizer, and J.-C. Freytag. 2009. Executing SPARQL queries over the web of linked data. In *Proceedings of the International Semantic Web Conference*. 293–309.

O. Hartig and M. T. Ozsu. 2014. Reachable subwebs for traversal-based query execution. In *Proceedings of the International World Wide Web Conference (Companion Volume)*. 541–546.

T. Heath and C. Bizer. 2011. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool.

M. Hildebrand, J. van Ossenbruggen, and L. Hardman. 2006. Facet: A browser for heterogeneous semantic web repositories. In *Proceedings of the International Semantic Web Conference*. 272–285.

J. E. Hopcroft, R. Motwani, and J. D. Ullman. 2000. *Introduction to Automata Theory, Languages and Computability* (2nd ed.). Addison-Wesley Longman, Boston, MA.

K. Hose, R. Schenkel, M. Theobald, and G. Weikum. 2011. Database foundations for scalable RDF processing. In *Reasoning Web.* Lecture Notes in Computer Science, Vol. 6848. 202–249.

R. Isele, A. Harth, J. Umbrich, and C. Bizer. 2010. LDspider: An open-source crawling framework for the Web of Linked Data. In *Proceedings of the International Semantic Web Conference*.

G. Klyne, J. J. Carroll, and B. McBride. 2004. Resource Description Framework (RDF): Concepts and Abstract Syntax. Retrieved from http://www.w3.org/TR/rdf-concepts.

K. Kochut and M. Janik. 2007. SPARQLeR: Extended SPARQL for semantic association discovery. In *Proceedings of the European Semantic Web Conference*. 145–159.

A. O. Mendelzon, G. A. Mihaila, and T. Milo. 1997. Querying the World Wide Web. *Int. J. Digital Libraries* 1, 1 (1997), 54–67.

E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, and G. Tummarello. 2008. Sindice.com: A document-oriented lookup index for open linked data. *Int. J. Metadata Semant. Ontol.* 3, 1 (2008).

E. Oren, R. Delbru, and S. Decker. 2006. Extending faceted navigation for RDF data. In *Proceedings of the International Semantic Web Conference*. 559–572.

G. Papamarkos, A. Poulovassilis, and P. T. Wood. 2004. RDFTL: An event-condition-action language for RDF. In *Proceedings of the 3rd International Workshop on Web Dynamics*.

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

J. Pérez, M. Arenas, and C. Gutierrez. 2010. nSPARQL: A navigational language for RDF. *J. Web Semantics* 8, 4 (2010), 255–270.

B. Quilitz and U. Leser. 2008. Querying distributed RDF data sources with SPARQL. In *Proceedings of the European Semantic Web Conference*. 524–538.

N. A. Rakhmawati, J. Umbrich, M. Karnstedt, A. Hasnain, and M. Hausenblas. 2013. Querying over federated SPARQL endpoints—A state of the art survey. *CoRR* (2013).

S. Schaffert, C. Bauer, T. Kurz, F. Dorschel, D. Glachs, and M. Fernandez. 2012. The linked media framework: Integrating and interlinking enterprise media content and data. In *I-SEMANTICS*. 25–32.

M. Schmachtenberg, H. Paulheim, and C. Bizer. 2014. Adoption of linked data best practices in different topical domains. In *Proceedings of the International Semantic Web Conference*.

A. Schwarte, P. Haase, K. Hose, R. Schenkel, and M. Schmidt. 2011. FedX: Optimization techniques for federated query processing on linked data. In *Proceedings of the International Semantic Web Conference*. 601–616.

S. Stadtmüller, S. Speiser, A. Harth, and R. Studer. 2013. Data-fu: A language and an interpreter for interaction with read/write linked data. In *Proceedings of the 22nd International Conference on World Wide Web*. 1225–1236.

J. Umbrich, A. Hogan, A. Polleres, and S. Decker. 2014. Link traversal querying for a diverse web of data. *Semantic Web—Interoperability, Usability, Applicability* (2014).

P. Wadler. 1999. Two semantics for XPath. Retrieved from http://www.cs.bell-labs.com/who/wadler/topics/xml.html.

J. Weaver and P. Tarjan. 2013. Facebook linked data via the graph API. *Semantic Web* 4, 3 (2013), 245–250.

P. T. Wood. 2012. Query languages for graph databases. *SIGMOD Record* 41, 1 (2012), 50–60.

M. Yahya, K. Berberich, S. Elbassuoni, M. Ramanath, V. Tresp, and G. Weikum. 2012. Deep answers for naturally asked questions on the web of data. In *Proceedings of the International World Wide Web Conference (Companion Volume)*. ACM, 445–449.

H. Zauner, B. Linse, T. Furche, and F. Bry. 2010. A RPL through RDF: Expressive navigation in RDF graphs. In *Web Reasoning and Rule Systems*. 251–257.

Received April 2014; revised August 2014; accepted October 2014