NOTICE: this is the author's version of a work that was accepted for publication in Journal of Computer and System Sciences. Changes resulting from the publishing process, such as peer review, editing, corrections, structural formatting, and other quality control mechanisms may not be reflected in this document. Changes may have been made to this work since it was submitted for publication. A definitive version was subsequently published in Journal of Computer and System Sciences (Volume 76, Issue 7, November 2010, Pages 663-685) with DOI: 10.1016/j.jcss.2010.01.003.

When citing this work, please use the reference above.

Hashing and canonicalizing Notation 3 graphs☆

Jesus Arias Fisteus[*,a,1,2], Norberto Fernández García[a], Luis Sánchez Fernández[a], Carlos Delgado Kloos[a]

[a] *Depto. de Ingeniería Telemática, Universidad Carlos III de Madrid, Avda. de la Universidad, 30, 28911, Leganés (Madrid), Spain*

Abstract

This paper presents a hash and a canonicalization algorithm for Notation 3 (N3) and Resource Description Framework (RDF) graphs. The hash algorithm produces, given a graph, a hash value such that the same value would be obtained from any other equivalent graph. Contrary to previous related work, it is well–suited for graphs with blank nodes, variables and subgraphs. The canonicalization algorithm outputs a canonical serialization of a given graph (i.e. a canonical representative of the set of all the graphs that are equivalent to it). Potential applications of these algorithms include, among others, checking graphs for identity, computing differences between graphs and graph synchronization. The former could be specially useful for crawlers that gather RDF/N3 data from the Web, to avoid processing several times graphs that are equivalent. Both algorithms have been evaluated on a big dataset, with more than 29 million triples and several millions of subgraphs and variables.

Key words: Semantic Web, Notation 3, RDF, hash

---

1. Introduction

Semantic Web technologies describe resources and their relations using graphs, normally represented by means of Resource Description Framework (RDF) de-

☆This work has been partially funded by the Spanish Ministry of Science and Innovation in the context of the project ITACA (TSI2007-65393-C02-01).

*Corresponding author

Email addresses: `jaf@it.uc3m.es` (Jesus Arias Fisteus), `berto@it.uc3m.es` (Norberto Fernández García), `luiss@it.uc3m.es` (Luis Sánchez Fernández), `cdk@it.uc3m.es` (Carlos Delgado Kloos)

[1] Corresponding author's phone number: +34916245940; Fax: +34916248749

[2] The author would like to thank the Decentralized Information Group at the MIT Computer Science and Artificial Intelligence Laboratory for hosting him during the first stage of this research.

scriptions. More specifically, RDF is based on node– and edge–labelled directed graphs. Nodes of a graph represent resources, normally labelled with a URI (some resources like blank nodes and literals are not labelled). Edges model a relation between two nodes, and are directed. Edges are also labelled with a URI. In the rest of this paper, the term *graph* will be used to refer to this specific kind of graphs used in the Semantic Web.

RDF is usually serialized using the RDF/XML notation. However, RDF/XML is not the only notation available to describe resources in the Semantic Web. Other commonly used notations are N–triples and Notation 3 (N3). N3 [1] is a more compact and readable alternative to RDF/XML. In addition, it extends RDF with additional semantics that provide the ability to:

- Express triples about graphs, i.e., graphs with one or more statements can be subject or object of triples. They are referred to as subgraphs in this work. Doing the same with reification in RDF would be complex, because reification works on triples, but not on graphs.

- Add variables to a graph and quantify them, either universally or existentially.

The combination of subgraphs and variables makes it possible to express rules in N3.

The ability to produce a hash value from a graph can potentially simplify operations like checking graphs for equivalence, detecting duplicates in large collections of graphs, computing differences between graphs and implementing caches of graphs, among others.

A general purpose text–based hash function is not enough for that, mainly because of two reasons. Firstly, it depends on the specific serialization of the graph (ordering of triples, disposition of white–spaces, namespace prefixes, knowledge representation language, etc.). Different serializations of the same graph produce different hash values, precluding general purpose hash functions from being used for some applications like checking whether two serializations correspond to the same or different graphs. Secondly, some applications, like computing differences between graphs, need not only the whole hash of a graph but also partial hashes of its components (nodes, triples, etc.).

In this work we present an algorithm for computing hash values from the kind of graphs used in the Semantic Web. It produces the same hash value for equivalent graphs (i.e. any possible serialization of the same graph). Furthermore, hash values do not depend on the specific language used for serialization: the same graph produces the same hash value regardless if it is serialized with RDF/XML, N3 or N–triples. The algorithm also produces partial hashes for triples, variables and subgraphs, that may be used for other purposes like canonicalization and graph comparison.

In addition to the hash algorithm, a graph canonicalization algorithm derived from the former is presented. Graph canonicalization is, up to a point, related to hashing, because the hash algorithm can help to compute canonical graphs, as will be explained in Section 7. Given a graph G, a canonicalization algorithm must produce an equivalent graph that is canonical, i.e., the algorithm would produce exactly the same graph when applied to every graph G' equivalent to G.

The problems of hashing and canonicalizing graphs would be fairly straightforward if graphs were only a set of statements in which all subject, predicate and object were labelled. A naive solution for hashing would be computing the digest of the label of each component of each statement (using a hash function for strings), and mixing those digests with appropriate operators. A solution for canonicalizing would be concatenating the labels of subject, predicate and object of each statement and sorting the resulting strings lexicographically. However, that assumption is not true in general because:

- Blank nodes do not have a label, and variables have a label that is bound to a local scope (i.e. the label of a variable does not matter, as long as the same label is used in all the triples referring to it). However, both blank nodes and variables can be subject or object in triples. Therefore, their hash values cannot be based on a label, but on how those blank nodes or variables are related to other nodes in the graph.

- N3 allows subgraphs to be subject or object of statements. That makes hashing variables even trickier, because the same variable can appear in multiple (sub)graphs, and therefore not only its relations to other nodes have to be considered, but also in which graphs those relations happen.

- Sets and lists require special consideration when variables and blank nodes appear as members of those data structures.

The algorithms presented in this paper are designed to solve these problems. The subject of this work is related to the classical graph isomorphism problem from graph theory. Section 2 reviews that relation. Section 3 briefly introduces the syntax of N3. The basic definitions and notation used in the rest of the paper are presented in Section 4. The hash algorithm itself is presented in Section 5 (basic features) and 6 (advanced features). The canonicalization algorithm is explained in Section 7. The algorithms can be used for a number of purposes in Semantic Web applications. Some of them are discussed in Section 8. Finally, Section 9 shows the results of the experiments carried out with the purpose of validating the algorithms.

2. Equivalence and canonicalization in graph theory

The problems of graph hashing and canonicalization are closely related to the problem of graph isomorphism, a widely studied problem in the field of graph theory. This section briefly reviews the relation between graph theory and the work presented in this paper. The definitions and results about graph theory explained in this section are mainly taken from [2].

2.1. Basic definitions

According to its usual definition in the literature [2], a graph $G = (V, E)$ consists of two sets V and E. The elements of V are called vertices (or nodes). The elements of E are called edges (or arcs). Each edge has one or two vertices associated to it, which are called its endpoints. An edge is said to join its endpoints.

A directed graph (or digraph) is a graph whose edges are directed. A directed edge is an edge in which one endpoint is designated as the tail and the other as the head. The edge is directed from its tail to its head [2].

A labelled graph is a graph with labels associated to its vertices and/or edges [2].

2.2. Graph isomorphism

According to its definition in [2], two directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic* if there is a one–to–one, onto mapping $\phi : V_1 \to V_2$ such that, for any two vertices $x, y \in V_1$, there is a directed edge $(x \to y)$ in G_1 if an only if there is a directed edge $(\phi(x) \to \phi(y))$ in G_2. The mapping ϕ is called an *isomorphism*.

The definition of *isomorphic as labelled* graphs found in [2] considers labelled vertices, but not labelled edges. Due to that, it has to be extended with an additional condition about preservation of labels of edges. Therefore, two directed graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are *isomorphic as labelled* if there is an isomorphism $\phi : V_1 \to V_2$ such that for each $x \in V_1$, the vertices x and $\phi(x)$ have the same label, and for any two vertices $y, z \in V_1$ so that there is a directed edge $(y \to z)$ in G_1, the directed edge $(\phi(y) \to \phi(z))$ in G_2 has the same label.

As stated in [2], the problem of graph isomorphism is in NP (non-deterministic polynomial time), but no NP–completeness proof is known and no polynomial–time algorithm is known for general graphs. However, polynomial–time algorithms are known for some classes of graphs like, for example, rooted trees, planar graphs, graphs with bounded genus, degree or eigenvalue multiplicity.

As stated by Grohe [3], algorithms for graph isomorphism are often divided into combinatorial algorithms [4, 5, 6], normally based on color refining, and group–theoretic algorithms [7].

2.3. Relation between graph theory and the work presented in this paper

RDF graphs are labelled directed graphs. RDF nodes are the vertices of the graph, with the restriction that two different labelled nodes cannot have the same label (URI). Triples represent directed edges (subjects are their tails and objects are their heads). Both vertices and edges are labelled with a URI. Although blank nodes do not have a label, they can be considered as labelled with the same special label, different from the URI of any labelled node. Literal nodes can be considered to have a special label derived from their value. This way, it can be assumed that all the nodes in a RDF graph are labelled.

On the contrary, N3 graphs are not graphs according to the definition in Section 2.1, because in N3 a node of the graph may itself be another graph.

Moreover, nodes of the internal graph may be connected by edges to nodes outside that graph.

One of the key applications of graph hashing is graph identity, which can also be solved with graph isomorphism techniques. Two RDF graphs are equivalent, according to the definition of equivalence in Section 4.2, if both graphs are isomorphic as labelled. This is true because two graphs being isomorphic as labelled contain exactly the same triples (including labels). Blank nodes can also be processed, because the isomorphism maps them to their equivalent blank nodes in the other graph according to the edges going to and coming from them, provided that all the blank nodes are labelled with the same special label.

Graph isomorphism is also related to graph hashing and canonicalization because some graph isomorphism algorithms can solve both the hash and canonicalization problem. Several isomorphism algorithms, for example [8], can produce a canonical numbering for nodes in the graph, from which a canonical serialization of the graph can be derived. A hash value for the graph could be computed by applying a text–based hash function to that canonical serialization.

Since N3 graphs do not comply with the definition of graph in Section 2.1, direct application of the well-known graph isomorphism algorithms presented above is not possible.

Instead of trying to adapt those graph isomorphism algorithms to the characteristics of N3 graphs, we propose in this work new algorithms for both hashing and canonicalizing N3 graphs, with a competitive polynomial–time complexity between $O(N \log N)$ and, in the worst–case, $O(N^2)$. The counterpart is the possibility of hash collisions, which could prevent the canonicalization algorithm from being deterministic. Nevertheless, as our experiments show (see Section 9.3), the probability of that to happen is extremely low when 64–bit hash values are computed on graphs up to a several hundred million edges.

3. A brief introduction to the notation of N3

This section presents a brief introduction to the notation of N3, with special focus on the constructions used in the examples of this paper. More details about N3 can be found in [1].

Graphs are serialized in N3 as a list of statements (triples), each one finished

with a period. A statement is composed, in this order, by its subject, predicate and object. Subjects and objects are nodes of the graph. N3 defines several kinds of nodes (labelled nodes, which are represented by URI, literal values, lists, sets, variables, blank nodes and subgraphs), each one represented with a different syntax. Predicates and labelled nodes are denoted using their URI. Prefixes may be used to make the notation compact when many components share the beginning of their URIs. Literal nodes are directly represented by their value within quotation marks:

```
@prefix voc: <http://www.example.com/vocabulary/> .
@prefix dis: <http://www.example.com/disney/> .
@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
dis:mickey rdf:type voc:mouse .
dis:mickey voc:has-name "Mickey Mouse" .
dis:goofy voc:friend-of dis:mickey .
dis:goofy voc:friend-of dis:daisy .
```

Triples sharing the same subject can be represented in a compact way, separated by semicolons. If they share both the subject and predicate, their objects may be separated by commas:

```
dis:mickey rdf:type voc:mouse ;
           voc:has-name "Mickey Mouse" .
dis:goofy voc:friend-of dis:mickey, dis:daisy .
```

Anonymous nodes are represented by square brackets that enclose the predicates and objects of triples of which the anonymous node is subject. Because an anonymous node itself is actually a node, it appears in the subject or object of a statement. The following example can be read as *a certain cartoon friend of Mickey was featured on TV*:

```
[ rdf:type voc:cartoon;
  voc:friend-of dis:Mickey ] voc:featured-on voc:tv .
```

Variables may be declared either existentially or universally quantified with, respectively, the "`@forAll`" or "`@forSome`" keywords. The following example says that *there exists a certain node whose name is "Mickey Mouse" that is friend of Goofy*:

```
@forSome :y .
:y voc:has-name "Mickey Mouse" .
:y voc:friend-of dis:goofy .
```

A subformula (subgraph) is represented by its statements enclosed in braces. It may be the subject or object of a statement. The combination of subformulae and variables makes it possible to express rules. The following statement has predicate "`log:implies`" and both its subject and object are subformulae. Its meaning is *for all nodes* `:x` *and* `:y`*, if* `:y` *is parent of* `:x` *and* `:x` *is a female, then* `:x` *is daughter of* `:y`:

```
@forAll :x, :y .
{ :y voc:parent-of :x .
  :x voc:gender voc:female }
log:implies
{ :x voc:daughter-of :y } .
```

4. Definitions and notation

This section defines the concepts of *graph* and *graph equivalence* upon which this work is based, and introduces the notation that will be used in the following sections.

4.1. Definition of graph

Definition 1. *An* N3 graph $G = (M, S)$ *is a set* M *of nodes and a set* S *of edges where:*

- *Nodes are typed. Given a node, its type is one of the following: labelled node (a node labelled with a URI), literal node (a node labelled with a literal piece of text and, optionally, a data–type identifier and/or a language identifier), list (an ordered sequence of nodes from* M*), set (an unordered collection of nodes from* M*), variable (anonymous node, universally-quantified variable or existentially-quantified variable) or subgraph (defined below).*
- *The labelling of labelled nodes is injective, i.e. for all* a*,* b *labelled nodes in* M*, if the label of* a *is equal to the label of* b*, then* $a = b$*.*
- *Edges are directed and labelled with a URI, and have exactly two end–points in* M *(named subject and object). Contrary to labelled nodes, the labelling of edges does not need to be injective, i.e. different edges may have the same label.*

- *An N3 graph $G' = (M', S')$ is a subgraph of the graph G if G' is contained in the set M of nodes of G.*

In order to use the usual nomenclature of RDF and N3, an N3 graph / subgraph may be also referred to in this work as *formula / subformula*, and an edge as a *triple* or *statement*.

4.2. Definition of graph equivalence

Preserving graph equivalence is a basic requirement for the hash algorithm, i.e. the hash of two equivalent graphs must be the same. Different definitions of equivalence are possible, depending on the application. This work is based on the notion that two equivalent graphs express the same explicit meaning, independently of their specific serialization.

For the purpose of this work, two graphs are equivalent if they contain the same triples, independently of how they are serialized. Some of the differences in serializations can be hidden by parsers, including differences in white–spaces and line–breaks, the specific syntax used for serialization (e.g., RDF/XML, N3, N–triples) and the use of compact notations with N3 connectors like "," and ";". Hence, this work is focused on other changes in serializations that are not normally handled by parsers, like the order in which statements are serialized and how anonymous nodes are treated.

Definition 2. *Two nodes x and y are equivalent if and only if their type is the same and:*

- *If labelled nodes, their URIs are equal.*
- *If literal nodes, their value, data type and language are equal.*
- *If lists, being $x = (x_1, x_2, \dots, x_n)$ and $y = (y_1, y_2, \dots, y_m)$, $n = m$ and x_i equivalent to y_i for all $i = 1 \dots n$.*
- *If sets, being $X = \{x_1, x_2, \dots, x_n\}$ and $Y = \{y_1, y_2, \dots, y_m\}$, $n = m$ and there exists a bijection $f : X \to Y$ such that for all x_i in X the node $f(x_i)$ is equivalent to x_i.*
- *If variables, they have the same quantification (both blank nodes and existentially–quantified variables are considered to be existentially–quantified) and, being*

S_x and S_y the sets of triples in which, respectively, x and y appear either as subject or object, there exists a bijection $f : S_x \rightarrow S_y$ such that for all s in S_x, variable x appears in the same position of s as y in $f(s)$, predicates of s and $f(s)$ have the same URI, and subjects of s and $f(s)$ are equivalent (if x appears in the object of s) or objects of s and $f(s)$ are equivalent (if x appears in the subject of s) or x appears in both subject and object of s.

- *If graphs, both graphs are equivalent according to the definition of graph equivalence (Definition 4) shown below.*

Definition 3. *Two triples s and s' are equivalent if and only if their subjects are equivalent, their objects are equivalent and their predicates are labelled with the same URI.*

Definition 4. *Two graphs G and G' are equivalent if and only if, independently of their specific serialization (syntax, white–spacing, use of compact N3 connectors, etc.), being S the set of triples in G and S' the set of triples in G', there exists a bijection $f : S \rightarrow S'$ such that for every triple s in S, triples s and $f(s)$ are equivalent.*

Figure 1 shows an example of two equivalent N3 graphs. The first two triples of each graph are equal (although serialized in different order). Variable `:x` is equivalent to `:b`, `:y` to `:c` and `:z` to `:a`. The last triple of each graph contains subgraphs in both their subjects and objects. Those triples are also equivalent because their subgraphs differ only in variables that are equivalent.

Note that, in the example, the *meaning* of both graphs would be equivalent to the meaning of the graph created by adding the statement "`:john :issibling :bob`" to the former, because that statement can be inferred from the rule, i.e., it represents knowledge that is implicit in the original graph. However, from the point of view of this work, this graph is not considered equivalent to the original graphs, because only explicit knowledge is considered. If for a specific application it is needed to consider implicit knowledge in the graph, a reasoning tool should be applied first to the graph until no new knowledge can be discovered and, after that, computing the hash value of the graph extended with the new statements.

```
:bob :hasmother :alice .
:john :hasmother :alice .
@forall :x, :y, :z.
{ :x :hasmother :z . :y :hasmother :z } log:implies { :x :issibling :y }
```

```
:john :hasmother :alice .
:bob :hasmother :alice .
@forall :a, :b, :c.
{ :c :hasmother :a . :b :hasmother :a } log:implies { :b :issibling :c }
```

Figure 1: Two simple equivalent N3 graphs. Predicate `log:implies` has special semantics in N3 (logical implication).

4.3. Notation

A *formula* (N3 graph) will be usually denoted as f. The set of *statements* (triples) of the formula is denoted as $S = \{s_1, \ldots, s_n\}$. The set V is the subset of M containing all the nodes of type *variable*. It is denoted as $V = \{v_1, \ldots, v_m\}$. It does not include variables declared in upper or inner formulae.

A formula f' is said to be a subformula *at any depth* of f if there is a sequence of formulae $(f_1, f_2, \ldots, f_n)$ such that f' is a subformula of f_1, f_1 a subformula of f_2 and so on, and f_n a subformula of f.

A statement s is composed by two nodes, *subject* and *object*, and the label of the edge itself, *predicate*. They are denoted respectively as $subj(s)$, $obj(s)$ and $pred(s)$.

The hash value or partial hash value of an item (formula, statement, variable, etc.) x, as computed by the algorithm, is denoted as $h(x)$. For example, the hash value of a given formula f is $h(f)$.

The URI of a labelled node x is denoted as $uri(x)$. A literal node x is characterized by its textual value $text(x)$ and, optionally, its language $lang(x)$ and data type $dtype(x)$.

The hash algorithm for string values presented in Section 5.3 is denoted as $h_s(t)$ where t is the string to be hashed.

The XOR operator will be denoted as "$\otimes$". For example: $a \otimes b$. The modular product operator (modulo N) will be denoted with no explicit symbol. For example: $h(a)h(b)$ means product of the hash values of a and b modulo N.

Given a term a, a modulo N will be denoted as $\overline{(a)}_N$. However, in order to keep equations as concise as possible, products are always assumed to be computed modulo N, despite the modulo operator not being explicitly shown.

5. A first approach to the hash algorithm

This section shows a first approach to the hash algorithm. In order to improve readability, a simplification of the algorithm is presented here. The explanation of the algorithm is completed in Section 6, which presents the advanced features not covered in this section.

5.1. Operators

An intuitive way of computing a hash value for a graph is computing a partial hash value for each triple, and then mixing all the partial values into one. The result will be independent of the order of the triples if the mixing operator is commutative and associative.

The algorithm we propose in this work uses two operators for mixing hash values:

- XOR: the bit–wise exclusive OR operator is commutative and associative, and widely used for mixing hash values. Hash values generated by the algorithm must be between 0 and $N-1$ for a certain value of N that is not a power of 2 (see Section 5.2). Because N is not a power of two, their XOR could be out of that range even when both operators are between 0 and $N-1$. The problem is solved by computing results of XOR operations modulo N. However, modulo has always to be computed only with the final result of a sequence of XOR operations, because XOR modulo N is commutative but not associative. That is, the XOR of values $a_1, a_2, \ldots, a_n$ must be computed as $\overline{(a_1 \otimes a_2 \otimes \cdots \otimes a_n)}_N$ in order to both maintain associativity and keep the result between 0 and $N-1$.

- Multiplication (modulo N): this operator is also commutative and associative, and can spread values when computed in modulo.

The reason for using two different mixing operators is that commutativity and associativity are not always suitable. For example, an intuitive way of

hashing a triple is to separately compute partial hashes for subject, predicate and object, and mix them together later. In this case, the mixing operator cannot be commutative nor associative because if, for example, subject and object are swapped, the hash value of the whole triple would not be affected by the change.

A solution to the above–mentioned problem is taking advantage of the fact that XOR and multiplication have the property that, in general:

$$a(b \otimes c) \neq a \otimes (bc) \tag{1}$$

Because of that, the hash values of subject, predicate and object of a given triple s can be mixed to obtain the hash value $h(s)$ of the triple as follows:

$$h(s) = \overline{((h(subj(s))\, k_{subj}) \otimes (h(pred(s))\, k_{pred}) \otimes (h(obj(s))k_{obj}))}_N \tag{2}$$

Where $k_{subj}, k_{pred}, k_{obj}$ are constants. This is further explained in the following sections. The equation above produces different hash values if, for example, subject and object are swapped. The same principle is applied, as explained later, for other purposes in the algorithm, like making partial hashes inside a subgraph be opaque to the supergraph.

5.2. Selection of N

Hashes are computed modulo N, so that they are between 0 and $N-1$ (they are contained in the Galois field $\mathbb{Z}_N$). The obvious choice is to establish $N = 2^n$, where n is the number of bits used for representing hash values. However, it is not a good choice, because there are chances that the result of a modular multiplication is zero even if both operands are not zero (note that here integer multiplication is represented explicitly with the "·" symbol to avoid confusion with the modular multiplication, that is implicitly represented with no symbol throughout the paper):

$$\forall a, b \in \mathbb{Z}_N,\ \exists m \in \mathbb{N} \;/\; a \cdot b = m \cdot N \implies \overline{(a \cdot b)}_N = 0 \tag{3}$$

In fact, in the case of $N = 2^n$, it is easy to find pairs of non–zero numbers whose multiplication is zero. For example, $\overline{(4 \cdot 8)}_{16} = 0$.

Once a multiplication returns zero, further multiplications to mix it with other partial hashes result again in zero, and therefore the other partial hashes do not affect the global hash value, leading to probable collisions.

The problem does not happen, however, when N is prime. It is a well-known fact that when N is prime, if $\overline{(a \cdot b)}_N = 0$, with $a, b \in \mathbb{Z}_N$, then either $a = 0$ or $b = 0$.

As a consequence, it is convenient for n–bit hash values to choose N as the largest prime number that can be encoded with an unsigned n–bit integer. Given that 64–bit hashes are computed with the algorithms presented in this work, the value chosen for N is $2^{64} - 59$.

5.3. Hashing string values

Some steps of the hash algorithm rely on hashing text (string values). It is the case, for example, of labelled nodes, whose URIs are hashed using a text–hashing function. For the purpose of hashing an N3 graph, any good text–hashing algorithm may be chosen.

5.4. Constants

The hash algorithm uses a set of 16 different constants that are mixed with other hashes. For example, constants are used to make hashes of terms depend on their position in a triple. They are denoted as: k_{subj}, k_{pred}, k_{obj}, k_{list}, k_{set}, k_{dtype}, k_{lang}, k_{setitm}, k_{univ}, k_{exist}, k_{lab}, k_{lit}, k_{opq}, k_{lseed}, k_{lmul} and k_{fitm}.

Concrete values for those constants may be randomly chosen, although their quality should be experimentally validated (some combinations of values might increase the probability of hash collision). Tab. 1 in Section 9 lists the values selected for the experimental validation of the algorithm, which showed a good behavior.

5.5. Hashing a formula

Let f be a formula with set of statements S. Let V be the set of variables declared directly in f. The algorithm takes as input an initial hash value for every variable in V and for every variable declared in any inner subformula (at any depth). The output of the algorithm is the hash value of the formula, and the hash value of every statement and variable appearing in S and V.

The initial hash value for a variable v is computed as:

$$h(v) = \begin{cases} k_{univ}, \text{ if } v \text{ is universally–quantified,} \\ k_{exist}, \text{ if } v \text{ is existentially–quantified.} \end{cases} \quad (4)$$

The use of initial hash values for variables is related to the multi–step variant of the algorithm, and will be explained in Section 6.

Applying the hash algorithm to f comprises several steps: first, computing the hash value of the statements in S; then, computing the hash value of the variables in V; and, finally, mixing those hash values. If $S = \{s_1, \ldots, s_m\}$ and $V = \{v_1, \ldots, v_n\}$, the hash value of f is computed as:

$$h(f) = h(s_1)h(s_2)\cdots h(s_m)h(v_1)h(v_2)\cdots h(v_n) \quad (5)$$

The next sections explain how hash values of statements and variables are computed.

5.6. Hashing statements

The hash value of a statement s is computed from the hash value of its three terms: subject, predicate and object.

$$h(s) = \overline{((h(subj(s))k_{subj}) \otimes (h(pred(s))k_{pred}) \otimes (h(obj(s))k_{obj}) \otimes p_f)}_N \quad (6)$$

The value p_f is specific for the formula f in which the statement appears. It depends on the path from the root formula of the graph to the formula f. It is an advanced feature designed to avoid some kinds of hash collisions. Its computation is explained in Section 6.2.

The rationale behind equation 6 is:

- The hash value of each term is mixed with a constant to make the hash value of the statement change if positions of terms change in the statement (for example, if subject and object are swapped).
- The hash from each part of the statement is mixed with XOR because in the upper level (formula) hashes are mixed with a multiplication. This prevents collisions, as explained in Section 5.1.

The algorithm for computing the hash value of each term depends on its type (labelled node, literal value, list, formula or variable). If the term is a subformula, its hash value is computed recursively as explained in Section 5.5. The following sections explain how to compute the hash value of any other term type.

5.7. Hashing labelled nodes and predicates

The hash value of a labelled node is the hash value of its URI, as computed by the text–hashing function of Section 5.3. That hash value is mixed with the constant k_{lab} to avoid collisions with other kinds of nodes (for example, literal values) with the same textual value:

$$h(node) \quad = \quad \overline{(h_s(uri(node)) \otimes k_{lab})_N} \tag{7}$$

Hash values of predicates are computed the same way, using the URI of the predicate.

5.8. Hashing literal values

N3 literals comprise a data value and, optionally, a language declaration and a data type declaration. The string representation of the literal node is hashed using the string–hashing function. The same function is applied, if present, to the language declaration and the data type:

$$h(node) \quad = \quad \overline{((h_s(text(node))\ h_{lang}(node)\ h_{dtype}(node)) \otimes k_{lit})_N} \tag{8}$$

Where $h_{lang}(node)$ and $h_{dtype}(node)$ are computed as:

$$h_{lang}(node) = \begin{cases} 1, \text{ if no language is specified,} \\ \overline{(h_s(lang(node)) \otimes k_{lang})_N}, \text{ otherwise.} \end{cases} \tag{9}$$

$$h_{dtype}(node) = \begin{cases} 1, \text{ if no data type is specified,} \\ \overline{(h_s(dtype(node)) \otimes k_{dtype})_N}, \text{ otherwise.} \end{cases} \tag{10}$$

5.9. *Hashing lists*

The hash value for each term $l_1, \ldots, l_n$ in the list is computed recursively. Then, it is mixed with a code that depends on the position of the term in the list. Finally, all those figures are mixed to get the final hash of the whole list:

$$h(l) = \overline{\left(\left(\overline{(h(l_1) \otimes m_1)}_N \overline{(h(l_2) \otimes m_2)}_N \cdots \overline{(h(l_n) \otimes m_n)}_N\right) \otimes k_{list}\right)}_N \tag{11}$$

$$m_1 = k_{lseed} \tag{12}$$

$$m_i = m_{i-1} k_{lmul}, \ \forall i \geq 2 \tag{13}$$

Values m_i make the hash value change when nodes change their position in the list, because lists containing the same nodes in different order are not equivalent. Constants k_{lseed} and k_{lmul} have been tested to guarantee that m_i values are not repeated at least up to $i = 2^{32}$.

5.10. *Hashing sets*

The hash value for each term $l_1, \ldots, l_n$ in the set is computed recursively. Each hash value is mixed with a constant, and then all these hashes are combined into one:

$$h(l) = \overline{\left(\left(\overline{(h(l_1) \otimes k_{setitm})}_N \overline{(h(l_2) \otimes k_{setitm})}_N \cdots \overline{(h(l_n) \otimes k_{setitm})}_N\right) \otimes k_{set}\right)}_N \tag{14}$$

5.11. *Hashing variables and blank nodes*

Computing hash values for variables is challenging because those hash values cannot depend on their URI, as shown previously in Fig. 1. The same applies to blank nodes, which do not have a URI. This section explains how a hash value can be computed for variables and blank nodes based on their context, i.e., their relations to the rest of the nodes of the graph.

Blank nodes are almost equivalent to existentially quantified variables, with the only difference that the former do not have a URI. Considering that the URI of variables do not affect their hash value, the way of computing the hash value of a blank node and an existentially quantified variable is the same.

5.11.1. Dependency between hashes of statements and variables

In order to make the hash value of a variable be affected by its context, it is computed from the hash values of the statements in which the variable appears either as subject or object. However, the hash values of those statements are affected by the hash value of the variable itself (see Eq. 6). In other words, the hash of a statement has to influence the hash of the variables referenced by it, and vice versa.

Our solution to this mutual dependency is computing first the hash values of the statements, based on the initial hash values for variables of Eq. 4. Once hash values for every statement have been computed, hash values for variables are computed from the hash values of the statements referring to them.

With this solution, hash values of statements are not influenced by hash values of variables. This can be a problem in some occasions, because partial hashes of variables and statements may collide. The solution to this issue is explained later in Section 6. It is based on running the hash algorithm in several steps, using the final hash values of variables in one step as the initial values for the next step.

5.11.2. Computing the hash of variables

This section explains how to compute the hashes of variables, provided that hashes of statements have already been computed.

Let f be a formula with immediately nested subformulae $f_1, \ldots, f_k$. Let v be a variable declared at f that appears in at least one statement of f or any of its inner subformulae (at any depth). The partial hash of v at f is computed as:

$$h_{\{f\}}(v) = h_{\{f\}}^{local}(v)\overline{\left(h_{\{f_1\}}(v) \otimes h(f_1)\right)}_N \cdots \overline{\left(h_{\{f_k\}}(v) \otimes h(f_k)\right)}_N \tag{15}$$

The final hash value of v is obtained from its partial hash value at the formula f in which it is declared:

$$h(v) = h_{\{f\}}(v)h^{ext}(v) \tag{16}$$

Values $h_{\{f\}}^{local}(v)$ and $h^{ext}(v)$ for variables are computed by the recursive algorithm shown in Alg. 1 and 2. This algorithm avoids multiple causes of hash

collision for variables in graphs with nested subformulae, when those subformulae have similar structure. The objective is making the hash of the variable not only dependent on the inner structure of the subformulae it appears in but also on the location of those subformulae in the whole graph. In other words, variables appearing in subformulae that are equal will have a different hash value because those subformulae are in different locations of the whole graph.

Algorithm 1 Computation of $h^{local}_{\{f\}}(v)$ and $h^{ext}(v)$ for variables.

Require: For all v, $h^{ext}(v)$ must be initialized to 1 before the first call to this procedure.

procedure compute_hash_vars(f):
for each variable v declared in f or any upper formula and appearing in f or its subformulae (any depth) **do**
  if v is existentially–quantified or blank node **then**
    $h^{local}_{\{f\}}(v) \leftarrow k_{exist}$
  else
    $h^{local}_{\{f\}}(v) \leftarrow k_{univ}$
  end if
end for
for each statement s in f **do**
  call process_term($subj(s)$, $h(s)$, k_{subj}, f)
  call process_term($obj(s)$, $h(s)$, k_{obj}, f)
end for
end procedure

---

6. Advanced features of the hash algorithm

The hash algorithm presented in Section 5 has several limitations that cause undesirable hash collisions in some cases. This section explains the causes for these collisions and two mechanisms introduced in the algorithm to avoid them.

Almost all the collisions have their origin in the way variables and blank nodes are handled. Hashes of statements in which variables appear depend only on whether these variables are universal or existential, but not on their actual hash value (see Section 5.11.1). This may provoke collisions in the hash

Algorithm 2 Computation of $h^{local}_{\{f\}}(v)$ and $h^{ext}(v)$ for variables. Auxiliary procedure.

procedure process_term(t, $h(s)$, m_{path}, f):

if t is a variable **then**

  if v declared at f or any upper formula **then**

    $h^{local}_{\{f\}}(t) \leftarrow h^{local}_{\{f\}}(t)\overline{(h(s) \otimes m_{path})}_N$

  else

    $h^{ext}(t) \leftarrow h^{ext}(t)\overline{(h(s) \otimes m_{path})}_N$

  end if

else if t is a list **then**

  $m \leftarrow k_{lseed}$

  for *each* term t_i in t **do**

    call process_term(t_i, $h(s)$, $\overline{((m_{path}m) \otimes k_{opq})}_N$, f)

    $m \leftarrow mk_{lmul}$

  end for

else if t is a set **then**

  for *each* term t_i in t **do**

    call process_term(t_i, $h(s)$, $\overline{((m_{path}k_{setitm}) \otimes k_{opq})}_N$, f)

  end for

else if t is a formula **then**

  for *each* statement s_i of t **do**

    call process_term($subj(s_i)$, $h(s)$, $\overline{((m_{path}k_{fitm}k_{subj}) \otimes k_{opq})}_N$, f)

    call process_term($obj(s_i)$, $h(s)$, $\overline{((m_{path}k_{fitm}k_{obj}) \otimes k_{opq})}_N$, f)

  end for

end if

end procedure

```
@forAll :x, :y .
{
  :x a :Dog .
  :y a :Cat
}
log:implies
{
  :x a :Animal .
  :y a :Animal
} .
```

Figure 2: Example of collision in canonicalization. Although the algorithm produces a different hash for variables ":x" and ":y", the hash of the last two statements is the same.

of statements, as shown in Fig. 2. In the example, the last two statements have the same hash because they only differ in the variable, and both variables are universally quantified.

The problem in the example is that the final hash value produced for variables, which is different for ":x" and ":y", does not influence the hash value of the statements referring to them.

Fig. 3 shows another example of collision. The hash value of the graph should change if the labelled nodes "`:big`" and "`:small`" were swapped, but it does not. The cause is that there are two blank nodes that differ only on those nodes.

The solution proposed to avoid this kind of collisions consists of two complementary techniques: running the algorithm in multiple steps, and making partial hashes of statements and variables depend on the path from the root formula of the graph to them.

6.1. Multi–step hashing

The solution for avoiding collisions like the one in the example of Fig. 2 is to run the hash algorithm twice. The first step is as explained in Section 5. The second step is similar, with the only difference that the initial hash values of variables are those obtained at the end of the first step, instead of the constants for universal and existential variables of Eq. 4. This way, the hash values computed for ":x" and ":y" in the first step influence the hashes produced in the second step for the statements referring to them.

```
:something log_:arg1 [
    a log_:FunctionCall;
    log_:functionName :or;
    log_:arg1 [
        a log_:FunctionCall;
        log_:functionName :not;
        log_:arg1 :big
    ];
    log_:arg2 [
        a log_:FunctionCall;
        log_:functionName :or;
        log_:arg1 [
            a log_:FunctionCall;
            log_:functionName :not;
            log_:arg1 :small
        ]
    ]
] .
```

Figure 3: Example of hash collision due to blank nodes. If the labelled nodes "`:big`" and "`:small`" are swapped in this graph, the hash value of the new graph remains the same, although it is not equivalent to the original one.

```
[ :managed_by [ :knows [ :likes :Art ] ] ] .
[ :managed_by [ :knows [ :likes :Sports ] ] ] .
```

Figure 4: Example of a graph that needs three steps to resolve collisions.

Although the collision shown in Fig. 2 can be resolved in two steps, other collisions may require more steps. For example, the graph shown in Fig. 4 needs three steps.

The graph contains six blank nodes. In the first step, only the inner–most blank nodes do not collide (the ones that like art and sports). In the second step, they influence the hash of the two blank nodes in the middle (the ones that "know" them), but the first blank nodes (subject of "managed by") still collide. Finally, in the third step, the blank nodes in the middle influence the first blank nodes and no more collisions happen.

The example above can be generalized. When there are two or more chains of dependencies between variables, and the chains differ only in one of their endpoints, as many steps as nodes in the chains are needed to resolve collisions

```
@forSome :a, :b, :c, :d .

:a :likes :b .
:c :likes :d .
:a :dislikes :d .
:c :dislikes :b .
:a :likes :c .
:c :likes :a .
```

Figure 5: Example of a graph with a symmetry that has collisions independently of the number of steps the hash algorithm has executed.

in the variables in the opposite endpoint. Collisions are resolved step by step through the chain of dependency.

One conclusion is that the number of steps required depends on the structure of the graph. Furthermore, given a specific number of steps, it is possible to find a graph with collisions after those steps that would not have collisions with one more step (for example, in the example of Fig. 4, intermediate blank nodes may be inserted to increase the number of needed steps.

Therefore, the hash algorithm should be run, step by step, until no collisions happen. This can be done with a loop that checks if there are collisions at the end of every single step.

However, some kinds of graphs present collisions independently of the number of steps being run. This happens in trivial cases, like two different variables appearing in statements which only differ in the variables themselves. Symmetries involving several variables are less trivial cases. Fig. 5 shows an example of symmetry.

In that example, the hash of ":a" collides with the hash of ":c" and the hash of ":b" collides with the hash of ":d". This happens independently of the number of steps being run, because there is a symmetry.

If the hash algorithm were run until no collisions occur, it would never stop in case of symmetries like the one mentioned above. Therefore, it is necessary to introduce a new condition. After a given step, a new step has to be computed if:

- There is at least one collision, and only one step has been run.

```
@forAll :x, :y.
{:x :predicate :y} log:implies {:y :predicate :x} .
```

Figure 6: Example of a graph in which the component p_f of Eq. 6 is necessary to avoid a hash collision between two variables.

- Or there is at least one collision, more than one step has been run, and the number of collisions (number of variables and statements with the same hash as other variables and statements) is fewer than in the previous step.

With these modifications, the algorithm stops after two steps with the graph of Fig. 5. With the graph in Fig. 2 it stops after two steps. With the graph in Fig. 4 it stops after three steps.

The stop condition proposed in this section has been proven to be correct in absence of random collisions. The proof is shown in appendix A.

6.2. Computing the path component

When the hash of a statement is computed with Eq. 6, a term called p_f is used, being f the formula in which the statement appears. The objective of p_f is making the position of a statement (path from the root formula of the graph to the actual formula in which the statement appears) influence its hash value.

The rationale behind p_f is to avoid hash collisions between variables or blank nodes in graphs like the ones in Fig. 3 and Fig. 6. In the latter, variables "`:x`" and "`:y`" are different, because "`:x :predicate :y`" implies "`:y :predicate :x`", but not vice versa. Therefore, both variables should have a different hash, but they would not if p_f values were not used.

The value of p_f is different for each formula. It is computed as follows:

- The root formula of the graph has $p_{root} = 1$.

- For a given formula f, a direct child g (formula, list or set) of it has $p_g = \overline{((p_f m) \otimes k_{opq})}_N$, where m takes as value k_{subj} or k_{obj} depending on the position of g (subject or object) in the statement of f in which it appears.

- For a given list l, a direct child g (formula, list or set) of it has $p_g = \overline{((p_l m) \otimes k_{opq})_N}$, where m is the value m_i defined in Eq. 13 for the position i of g in the list.

- For a given set s, a direct child g (formula, list or set) of it has $p_g = \overline{((p_s k_{setitm}) \otimes k_{opq})_N}$.

When p_f is used, hashes of variables ":x" and ":y" in Fig. 6 do not collide because the two subformulae in the statement differ in their value of p_f, due to one of them being in the subject of the statement and the other one in the object. The collision in Fig. 3 is also avoided by using this mechanism.

Even though p_f values introduce big changes in hashes depending on the position of variables and subformulae, their use is consistent with the notion of equivalence. For example, other equivalent variations of the graph in Fig. 6, like `{:y :predicate :x} log:implies {:x :predicate :y}`, produce exactly the same hash value.

7. Using the hash algorithm for canonicalization

The *class of equivalence* of a given graph is defined as the set composed by the graph itself and all the possible graphs equivalent to it. One of the graphs in a class of equivalence will be selected to represent the class itself, and will be called the *canonical graph* for that class of equivalence.

The problem is designing the algorithm that chooses the canonical graph for each class of equivalence. The algorithm should be able to compute, given a graph, the canonical graph for its class of equivalence.

From the definition of equivalence, choosing the canonical representation of a graph consists in choosing:

- The canonical order of the declarations of variables.

- The canonical name for each variable of the graph.

- The canonical order for the statements of the graph.

- The canonical order for the items of sets.

- The canonical serialization in terms of syntax, white–spacing, etc.

This work is focused on the first four aspects, because the last one, serialization, can be trivially solved by establishing a strict output format.

If graphs did not contain variables, a trivial solution to the problem would be to sort statements and items of sets according to the lexicographical order of their N3 serialization. However, because graphs can actually contain variables and canonicalization cannot depend on their names, other kind of solution is necessary.

The hash algorithm can, as shown in this section, be used for canonicalizing graphs, by using the partial hashes it computes for each statement and variable in the graph:

- If a partial hash value is assigned to each variable in a graph, variable declarations can be sorted according to the ordering of their hash values.
- Variable names can be chosen from the position of the variable in that order.
- If a partial hash value is assigned to each statement, statements can be sorted according to their hash values.
- In a similar way, if hash values are assigned to each item in a set, those items can be sorted.

Therefore, the canonicalization algorithm that we propose comprises three steps:

1. Execute the hash algorithm on the graph and keep the partial hash of every single statement, node in a set and variable obtained from the last step of the algorithm.
2. Sort the variables by hash value (from lower to upper) and name them with relative URIs "`<#X_n__>`" where "`n`" is 0 for the variable with the lowest hash, 1 for the next, etc.
3. Sort the statements by hash value (from lower to upper).
4. Recurse on every direct subformula.
5. Serialize the graph. Variable declarations, statements and items of sets are serialized according to the ordering of their hashes.

The canonicalization algorithm sketched above does not work if two different variables, two different statements of the same formula or two different items of the same set have the same hash value, because more than one order is possible. Therefore, the algorithm, as presented above, is non–deterministic. Section 7.1 discusses the issue in depth and develops the algorithm further to solve the problem.

7.1. A solution to canonicalization collisions

There are two kinds of collisions that prevent the hash algorithm from producing a deterministic canonical graph:

- Collisions due to symmetries in the graph: graphs with symmetries related to variables cannot be canonicalized by using only the multi–step hash algorithm, because several variables (and statements in some cases) have the same hash value.
- Random collisions: hashes can collide, from a probabilistic point of view, in variables and statements with neither similarities nor symmetries.

Random collisions that only affect statements that do not contain variables can be solved by sorting the statements that collide according to the lexicographical order of their respective N3 serializations. However, this solution cannot be applied when the collision affects variables or statements containing variables.

Random collisions that cause canonicalization to fail have, however, an extremely low probability, because only collisions in variables or statements in the same formula are relevant. Considering 64–bit hashes, 2^{32} variables or statements in the same formula are necessary to get a probability of collision in that formula of approximately 0.5 (see Section 9.3 for an explanation). Therefore, it can be assumed that random collisions are very unlikely to happen with 64–bit hashes. Moreover, if a given application requires graphs bigger than that, hashes may be computed with more bits, just choosing a new value for N and new, bigger constants.

The other kind of collisions, those due to symmetries, although improbable in normal N3 graphs, can happen. The canonicalization algorithm should produce a deterministic canonical graph even in such cases.

```
# Possible serialization 1:
@forAll  <#X_0__> , <#X_1__> , <#X_2__> , <#X_3__> .
<#X_0__>   :likes   <#X_1__>   .
<#X_1__>   :likes   <#X_0__>   .
<#X_0__>   :likes   <#X_3__>   .
<#X_1__>   :likes   <#X_2__>   .
<#X_0__>   :dislikes   <#X_2__>   .
<#X_1__>   :dislikes   <#X_3__>   .

# Possible serialization 2:
@forAll  <#X_0__> , <#X_1__> , <#X_2__> , <#X_3__> .
<#X_0__>   :likes   <#X_1__>   .
<#X_1__>   :likes   <#X_0__>   .
<#X_0__>   :likes   <#X_2__>   .
<#X_1__>   :likes   <#X_3__>   .
<#X_0__>   :dislikes   <#X_3__>   .
<#X_1__>   :dislikes   <#X_2__>   .
```

Figure 7: Two different serializations of the graph in Fig. 5 that the (incomplete) canonicalization explained in section 7 might produce.

The graph in Fig. 5 is an example of symmetry. Variables "`:a`" and "`:c`" get the same hash values, as well as variables "`:b`" and "`:d`". The same happens to some statements that differ only on those variables, like "`:a :likes :b`" and "`:c :likes :d`". Therefore, the relative order of those variables and statements cannot be deterministically decided by the basic canonicalization algorithm. Fig. 7 shows the two possible canonical serializations of that graph. This section explains how to improve the basic canonicalization algorithm to make it deterministic in case of symmetries.

The cause of this problem is that, although variables "`:a`" and "`:c`" can be interchanged without altering the meaning of the graph, and the same occurs to "`:b`" and "`:d`", there is a relation across these pairs of variables that has to be preserved (in the example, "`:a`" likes "`:b`" and dislikes "`:d`", whereas "`:c`" likes "`:d`" and dislikes "`:b`"). This asymmetry of variable co–occurrences can be exploited to produce a deterministic canonical graph.

For every pair of variables v_1 and v_2 for which there is at least one statement with subject v_1 and object v_2, their co–occurrence hash value $h(v_1, v_2)$ is computed as follows:

$$h(v_1, v_2) = \overline{(h(pred(s_1)) \otimes h(pred(s_2)) \otimes \cdots \otimes h(pred(s_k)))}_N \qquad (17)$$

where $s_1, s_2, \ldots, s_k$ are all the statements that have v_1 as subject and v_2 as object.

Once all the necessary steps of the hash algorithm have been executed and variables have been sorted according to their hash value (there is a sorted sequence $v_1, v_2, \ldots, v_m$ of variables), every group of variables with the same hash value is arbitrarily sorted. In order to sort them deterministically, variables in colliding groups are rearranged within the group by using the co–occurrence hash values as follows.

Colliding groups are processed sequentially (beginning with the group with lowest hash value and ending with the one with highest hash value). Assume that v_k is the first variable in a given colliding group. The primary sorting criterion is the value $h(v, v_1)$ for every variable v in that group. The secondary sorting criterion is $h(v_1, v)$, then $h(v, v_2)$, and so on until $h(v, v_{k-1})$ and $h(v_{k-1}, v)$. Note that if k for the first group is 1, variables in that group are not rearranged at all, because there are no preceding variables. Nevertheless, this is an expected case that does not affect the performance of the algorithm, as shown later.

Some statements may have the same hash value because of variables appearing in them having the same hash value. Once variables have been sorted according to the criteria above, their ordering can be used to establish a deterministic ordering for those statements. The primary criterion for sorting statements is still their hash value. Then, if the statements have a variable in their subject, the position of that variable in the sequence of ordered variables is the second criterion (or -1 if the subject is not a variable). Finally, if the statements have a variable in their object, its position is the third criterion (or -1 if the object is not a variable).

Some variables may still collide (specially those that are close to the beginning of the sorted list) after they have been sorted with the criteria above. Nevertheless, it does not matter, because they are symmetrical and the canonical graph will still be deterministic. To understand this point, consider the example. Variables "`:a`" and "`:c`" have the lowest hash values. Even after the improved algorithm is applied, they are non–deterministically sorted. However,

it does not matter because now "`:b`" and "`:d`" are sorted accordingly: if "`:a`" is chosen as the first variable, "`:b`" is chosen as the third one, but if "`:c`" is chosen as the first variable instead, "`:d`" is chosen as the third one. Both situations lead to exactly the same serialization of the graph due to how statements are ordered. In fact, the second canonicalization shown in Fig. 7 is produced in both cases.

With the improvements explained above, the canonical output of the algorithm is deterministic even when symmetries occur.

8. Applications of the algorithms

The proposed hash and canonicalization algorithms can be applied to a number of scenarios. Some of them are discussed in this section.

8.1. Graph identity detection

Detecting whether several graphs are equivalent is useful for a number of applications like, for example, those in which processing the same graph several times leads to a waste of resources. For instance, Oren et al. [9] mention duplicate detection as an area of further research in the context of Sindice, an index for open linked data. Equivalence detection can also be useful to detect whether a new version of a graph contains relevant changes with respect to a previous version (regardless of changes in notation, ordering, naming of variables, etc.).

This kind of serialization–independent comparison of graphs is not straightforward, as shown in previous sections. The hash and canonicalization algorithms proposed in this work can solve the problem.

Graphs can be compared by comparing their hash values. If their hash values are different, the graphs are definitely not equivalent. If their hash values are equal, they are very likely to be equivalent, but further tests with other algorithms are necessary to assure that. One possibility is to serialize their canonical graphs, using the canonicalization algorithm (Section 7), and use a simple text–comparison tool. Only in the unlikely case that the canonicalization algorithm were not able to produce a deterministic canonical representation of both graphs, due to a statistical collision, other comparison algorithms would need to be considered.

This algorithm for graph comparison is notation–independent, in the sense that graphs serialized with different notations (e.g. RDF and N3) can be compared.

8.2. Computing differences between graphs and synchronization

For some applications, graphs have to be compared in order to detect their differences, for example for remote synchronization of RDF graphs [10] or to mix the triples of several graphs into one graph, avoiding duplicate statements.

In order to do this, it is necessary, basically, to detect which statements and variables of the graphs are equal, and which ones are different. An obvious solution is comparing every statement in one graph with all the statements in the other graph, but the complexity of this algorithm is quadratic with respect to the number of statements.

A solution with complexity $O(N \log N)$ would be applying the hash algoritm proposed in this work to compute the array of partial hashes of the statements of each graph, sorting those arrays, and comparing them with a simple linear–time algorithm.

Hash collisions might make the algorithm over–detect common hashes. However, there is an extremely low probability of that to happen. Applications having strict requirements about this can compare the statements with the same hash to avoid false positives. Note that even in this case the number of comparisons of statements is highly reduced because only statements suspected to be equal are compared.

If the algorithm has to deal with variables, the unit for computing hashes has to be a minimum self–contained graph, as explained in [10].

If the graphs contain subgraphs, the algorithm proposed works if subgraphs are treated as a unit and it is executed recursively on subgraphs appearing in statements that have the same hash value. If there is the need to identify also the differences between subgraphs statement by statement, the problem is not trivial and needs to be further studied.

8.3. Storing graphs in hash tables

Some Semantic Web applications may require graphs or parts of graphs to be stored in hash tables for later retrieval, being the graphs themselves the

keys of the hash tables. In this case, a hash algorithm for graphs is absolutely necessary. It is the case, for example, of caches of graphs implemented with hash tables.

9. Validation

As stated in [11], a good hash function should satisfy two requirements: (1) its computation should be very fast; and (2) it should minimize collisions.

The first requirement, fast computation, has implications in the performance of systems using the hash algorithm. Section 9.2 analyzes the computation time of the algorithm for graphs of different sizes.

The second requirement, minimization of collisions, is also very important for hash algorithms. For example, when hash values are used to compare graphs, a high rate of collisions leads to a high rate of false positives, which degrades the performance of the applications relying on the hash algorithm. It has also consequences for canonicalization, because semantic graphs with collisions in the hash of statements or variables defined at the same level cannot be deterministically canonicalized with the algorithm proposed in this work. Section 9.3 analyzes the probability of hash collision of the algorithm.

A prototype of the hash and the canonicalization algorithm was implemented in Python in order to test and validate them. It uses CWM [12] to parse input graphs. The experiments were run on this prototype.

The description of the hash algorithm left open the selection of a hash function for string values (Section 5.3) and the value of several constants (Section 5.4). The prototype uses the Python's built-in *hash* function and the constants listed in Tab. 1.

9.1. Data–set

In order to test the algorithm, a relatively big set of N3 and/or RDF files was needed. Particularly, N3 files with subgraphs were preferred because our experience during the initial design and testing phases of this algorithm showed that hashing subgraphs is a potential source of hash collisions in case of design errors in the algorithm. It is not difficult to find big amounts of RDF data or generate them automatically from data sources. However, big real–world

k_{subj}	41fbe48c045cc9ae	k_{univ}	5a9ee26bddc7fc70
k_{pred}	00bad7a94840f874	k_{exist}	c47fced69d144f22
k_{obj}	5724e0c64cf12be5	k_{lab}	418034d90ff93b33
k_{list}	e9bf7ebef4b0b2d9	k_{lit}	76de978e6b243c5d
k_{set}	b1b679fa7b11e586	k_{opq}	60c31fea734ab6b8
k_{dtype}	b9e474b819981c67	k_{lseed}	25189d055841d312
k_{lang}	72dffc38531a8870	k_{lmul}	01c4d4bbac73aa93
k_{setitm}	712fa2c0c5d65b16	k_{fitm}	f122de4aaf060e36

Table 1: Constants used by the hash algorithm (hexadecimal).

N3 data-sets with nested subgraphs and variables are more difficult to find. Generating random N3 graphs could be an option, but real–world data was preferred to avoid the potential problem of using data with biased properties.

For the purpose of validating this work, we built an N3 data–set based on the TPTP (Thousands of Problems for Theorem Provers) problem library [13].

The TPTP library of problems contains 9894 (as of version 3.3.0) problems used for testing automated theorem proving systems. They are specially interesting for testing the hash algorithm because they include propositions, universally and existentially quantified variables and rules. They were automatically translated into N3 to create a big data–set intensive in variables and subgraphs. As a result, 9894 N3 files were obtained. The main figures for this dataset are:

- Number of statements: 29,122,676.
- Number of subformulae: 6,100,441.
- Number of variables: 5,744,262.
- Total file-size: 1,9 GB

9.2. Complexity and computation time

In order to test the requirement of fast computation time, the evolution of the computation time of the algorithm with respect to the size of the input N3 graph was analyzed, both theoretically and empirically with the data–set.

9.2.1. Theoretical analysis

The multi–step hash algorithm consists of several steps of the basic hash algorithm presented in Section 5, each one followed by an algorithm to detect partial hash collisions. This section shows how computation time grows when the size (number of statements and variables) of the input graph is increased.

The time needed for computing one step of the basic hash algorithm is the sum of the time needed for:

- Computing the hash of every statement: it is linear, because the hash of a statement is computed in constant time.
- Computing the hash of every variable: due to the recursive Algorithm 1, complexity is quadratic in the worst case (N3 graphs in which the tree of nested subformulae is extremely unbalanced, i.e. it forms a sequence instead of a tree). For the usual graphs containing just some subformulae, complexity is almost linear. For graphs not containing subformulae at all (like RDF graphs), it is definitely linear, because computation time depends only on the number of statements that contain a variable as subject or predicate.
- Mixing hashes of statements and variables: it is linear, because they are mixed by multiplying the hashes of all the statements and variables.

Therefore, a basic hash step has linear complexity ($O(N)$) for RDF graphs, and linear or almost linear complexity for the usual N3 graphs. Complexity is $O(N^2)$ in the worst case, but that case seems likely to seldom appear in normal applications.

The algorithm to detect partial hash collisions can be implemented by sorting the partial hashes and a linear–time algorithm for detecting adjacent equal hash values. Its complexity is $O(N \log N)$ because of the sort algorithm.

The number of steps of the algorithm to be run does not depend on the number of statements or variables of the graph, and is, therefore, a constant from the point of view of the complexity analysis.

Therefore, the global complexity of the hash algorithm is between $O(N \log N)$, for RDF and usual N3 graphs, and $O(N^2)$, for worst–case N3 graphs. The

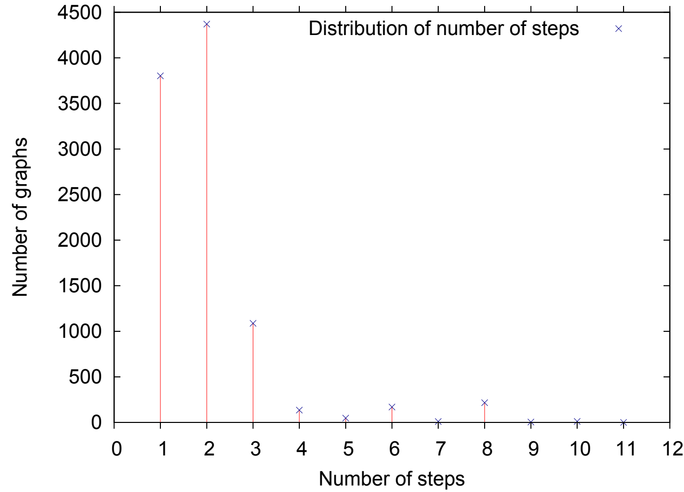

Figure 8: Distribution of the number of steps run per input N3 graph in the TPTP data–set.

complexity of the canonicalization algorithm is the same because its essential structure is the same.

9.2.2. Experimental results

The previous theoretical analysis was complemented with an empirical evaluation. An experiment was run with the data–set in order to measure the rate of growth of the computation time of the hash algorithm with respect to the size of the graph. This dataset was selected because it uses subformulae very extensively, and can therefore help to measure the behavior of the algorithm with graphs that are closer to the worst–case.

In order to avoid the distortion that a variable number of steps would introduce in measuring the effect of the size of the graph in its computation time, times displayed in figures are computed as the average time per step of the algorithm for each N3 file. Almost all the graphs require between one and three steps, as shown in Fig. 8.

Figures 9, 10 and 11 show, in different scales, the relation between the average hash computation time and the size of the graph. Figure 9 shows the computation time for all the graphs. Figure 10 zooms into the region with more data, at the left–bottom of the graph. Figure 11 zooms even more into that area, where almost all the graphs can be found.

Fig. 12 shows the distribution of computation time in the dataset. As shown

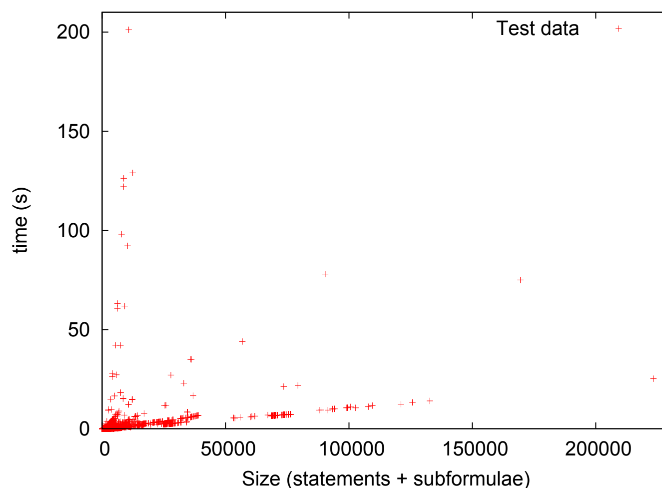

Figure 9: Computation time of the hash algorithm for the TPTP dataset. Full view.

in the figure, it is less than 100ms for 60% the graphs in the dataset, and less than 1s for more than 90% the graphs. The x axis is represented in logarithmic scale to improve resolution in the range of lower computation times.

As expected from the theoretical analysis, a variety of growth patterns appear, depending on the structure of each N3 graph. Many of those patterns seem to be linear. This apparently linear rate of growth of the experiments does not contradict the theoretical lower bound of $O(N \log N)$. The explanation is that the graphs of the experiment, although large, are not large enough to make the time needed for sorting bigger than the time needed for the other tasks of the algorithm.

Computation times may seem big, a portion of them in the order of seconds (almost 10% the graphs are above 1s per step, as shown in Fig. 12). The cause is that they were obtained in a low–profile computer with a non–optimized Python prototype of the algorithm. They are subject to considerable improvements in production–quality implementations. Nevertheless, what is relevant in this analysis is the rate of growth of computation time with respect to the size of the graph, which does not depend on the degree of optimization of the implementation.

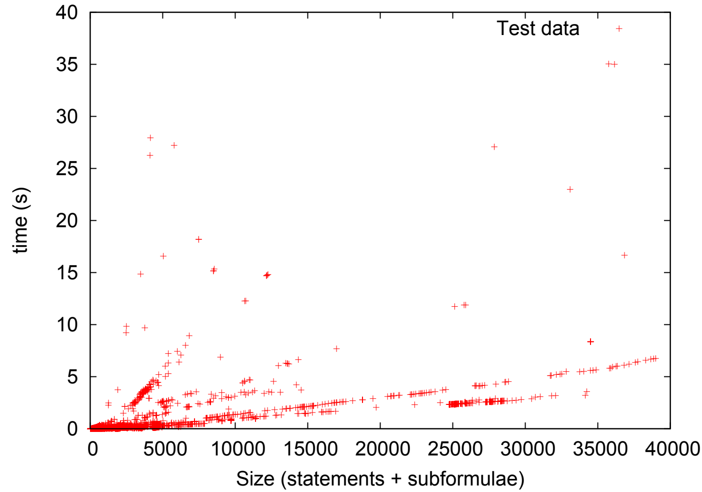

Figure 10: Computation time of the hash algorithm for the TPTP dataset. Zoomed view.

9.3. Probability of hash collisions

Let us focus on the problem of hash collisions. Suppose a hash function with 2^m equiprobable outputs (i.e., an m-bit output), and a set of k hash values obtained by applying the hash function to k different random inputs. According to the *birthday paradox*, the probability of hash collision (i.e., the probability that there are at least two equal hash values in the set) is greater than 0.5 for $k = 1.18 \cdot 2^{m/2} \approx 2^{m/2}$ (see [14, Appendix 11A] for the proof and further explanation).

Therefore, with a 64-bit hash function like the one proposed in this work, supposing that it produces uniformly distributed hashes, the probability of collision is greater than 0.5 when it is applied to 2^{32} different inputs.

However, for non–uniform distributions of hash values, that limit could be remarkably lower. The objective of the experiments carried out in this work is mainly to confirm that the output of the hash function we propose is sufficiently uniform and therefore the limit of 2^{32} applies.

In order to verify empirically the uniformity of the hash values it produces, the algorithm was run with the graphs of the data–set. The hash values obtained were processed with a statistical analyzer and a collision test, as described in the next subsections.

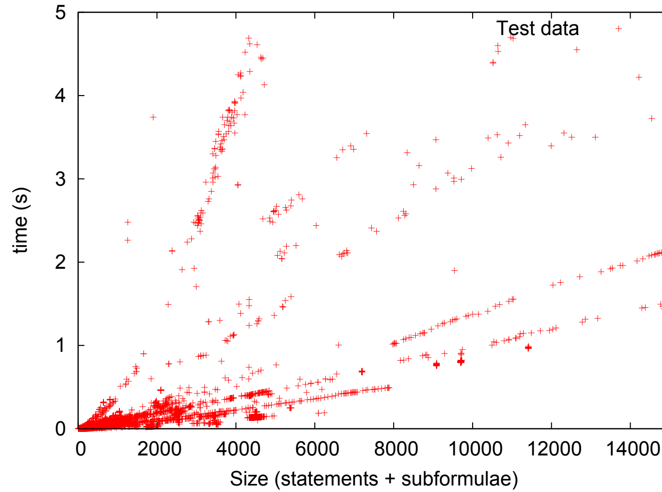

Figure 11: Computation time of the hash algorithm for the TPTP dataset. Zoomed view.

9.3.1. Statistical analysis

Several statistical tests were run on the hash values obtained from the N3 graphs in the TPTP data–set, in order to estimate by different means their likelihood to be uniformly distributed. A sequence of bytes was obtained from the hash values (each hash value produces 8 bytes). Ent [15], a pseudo-random number sequence test program, was run on that sequence of bytes. The main results for this test were:

- Entropy: 7.997813 bits per byte. This value is quite close to the theoretical value of 8 expected for a uniform sequence of bytes.
- Arithmetic mean: 127.1884. This value is close to the value of 127.5 expected for a uniform sequence of bytes.
- Value of π computed with the Monte Carlo method: 3.129567086. The error is 0.38% with respect to the actual value of π.
- Serial correlation coefficient: -0.002159, very close to the value of 0 for a totally uncorrelated sequence.
- Chi–square distribution test: 236.47; randomly would exceed this value 79.15% of the times, which means that the chi–square test gives no evidence of non–randomness, because the result is in the $(10\%, 90\%)$ interval.

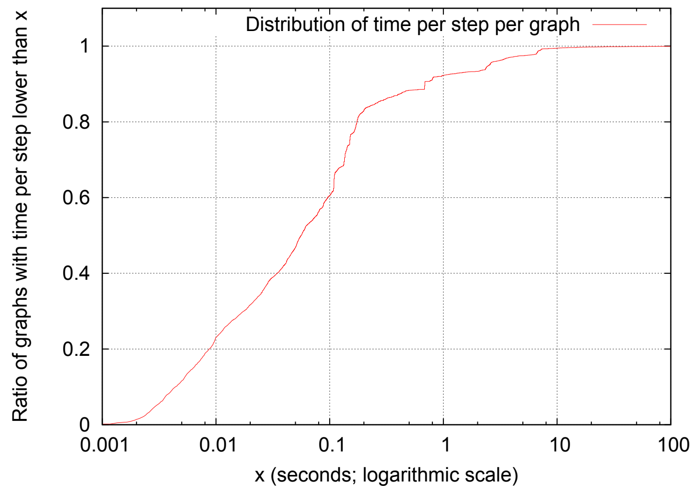

Figure 12: Distribution of computation time per step per graph in the TPTP dataset.

As a conclusion, the sequence of bytes obtained from the hash values of the TPTP files does not show any evidence of non–randomness in the tests run by Ent.

9.3.2. Collision test

The collision test is one of the empirical tests for randomness proposed by Knuth in [16]. It is well–suited for testing hash functions because it allows testing randomness with less data values than possible hash values, whereas other tests like the chi–square test need several data instances for each hash value.

The test consists in throwing n balls randomly into m urns that are initially empty. The number of collisions is the number of urns with two or more balls at the end of the experiment. The test is passed if the number of collisions is not too high or not too low with respect to the expected value.

Given the number of collisions c that actually occurred in the experiment, the theoretical probability p that c or less collisions occur for a random sequence is computed with the algorithm proposed in [16], or the normal approximation proposed in [17]. If the resulting theoretical probability is too low ($p < 0.05$) or too high ($p > 0.95$), the sequence of data is suspicious of not being random, because the number of collisions is either too low or too high compared to the expected number of collisions for a uniformly–distributed random sequence.

Because it can happen even for a truly random sequence to be in those ranges of probabilities, the test has to be run several times with different data sequences. If several of them turn to be suspicious, then the sequence of data fails the test, and is considered to be non–random.

For the purpose of these experiments, we selected $m = 2^{16}$, instead of $m = 2^{64}$, in order to keep n (in this case the number of graphs) close enough to m. Given that the hash values have 64 bits, different 16 bit data sequences were extracted from the 64–bit hash values of the TPTP data–set:

- Positional sequences: 4 sequences were produced, each one by taking a specific 16–bit part from each hash value (bits 1–16, 17–32, 33–48 and 49–64).
- Random position sequences: 16 sequences were produced by taking a random 16–bit block from each hash value.

Fig. 13 shows the histogram of the values of probability obtained for each sequence. The hashes of the TPTP data–set passed the collision test, because the value of probability obtained for all the sequences was between 0.05 and 0.95, which are the thresholds proposed by Knuth for this test. In other words, the collision test showed that the number of collisions that happen in the sequences produced from the random values is in the range expected for a uniformly–distributed random sequence.

9.4. Conclusions of the evaluation

Hash values produced by the algorithm were found in the experiments of Section 9.3 to be uniformly–distributed enough, because none of the tests run on them showed evidence of non–uniformity. Due to this reason, the probability of collision is expected to be close to 0.5 for a set of 2^{32} hash values, according to the *birthday paradox*. This value, above $4 \cdot 10^9$, is very high for almost all the practical applications of the algorithm, thus making the probability of collision extremely low for applications working with a reasonable number, even tens of millions, of graphs. As reported by Ding et al. [18], almost all the semantic Web documents that could be found in the Web in 2006 had less than 100 triples. The largest document found had over 1 million triples. Moreover, in

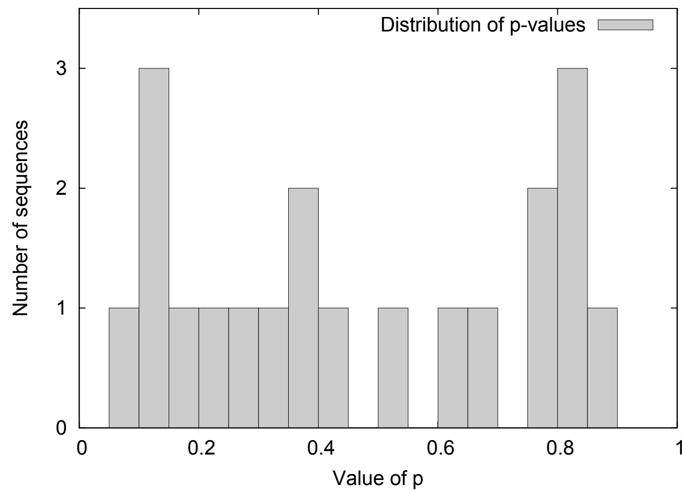

Figure 13: Histogram of probabilities for the different sequences in the collision test.

case an application required to work with bigger graphs, the algorithm could be extended to produce bigger hash values, by selecting a bigger value N and new constants.

Computation time has a tendency of growth between $O(N \log(N))$ and $O(N^2)$. Nevertheless, most of the graphs (especially those with a few subgraphs) exhibit an almost–linear growth up to several tens of thousands of statements, when the sort algorithm ($O(N \log(N))$) begins to dominate computation time. Only some scarce graphs with many subgraphs nested one into another would exhibit the worst case $O(N^2)$ complexity.

10. Related work

There are not many works related to hashing or canonicalizing Semantic Web graphs in the scientific literature. The most relevant up to now are [19] and [20]. Both works are specific for RDF, and therefore do not support subformulae, an important N3–specific feature.

Carroll [19] proposes an algorithm for canonicalizing RDF graphs, with a similar notion of equivalence of graphs to the one proposed in this article. Despite that work does not consider hashing, a hash value can be trivially obtained by applying a general purpose hash function to the canonical representation of the graph. The canonicalization algorithm sorts triples according to their lexico-

graphic order. As expected, blank nodes make this tricky. The author proposes a solution for handling blank nodes similar to the one proposed in this work:

- Blank nodes are labelled initially with a generic label, equal for all the nodes.
- Triples are sorted lexicographically (using the generic labels for blank nodes). After sorting, variables are re–labelled with a numerical identifier according to the position of the triple in which they appear first. Collisions may appear in this step (consecutive triples that are equal).
- Finally, triples are sorted again, now with the label computed for each variable.

Some graphs cannot be canonicalized, because of collisions. The author explains that multiple steps of that algorithm can fix some of the collisions. For those graphs that cannot be deterministically canonicalized, the author proposes a pre–canonicalization step that modifies the RDF graph by introducing new triples that help to differentiate variables but do not modify the meaning of the graph. However, this step is non–deterministic, and therefore some graphs cannot be deterministically canonicalized anyway. As a consequence, the hash value resulting for certain graphs would also be non–deterministic, provoking some equivalent graphs to produce different hashes.

The main drawbacks of the work of Carroll compared to this work are:

- It does not support subformulae, and therefore can only be applied to a subset of N3 graphs.
- The hash value of the RDF graphs that cannot be deterministically canonicalized does not preserve the equivalence relation, i.e., some equivalent graphs can produce different hashes. The author states that this can happen only with a very limited proportion of practically occurring RDF graphs. On the contrary, the hash algorithm presented in this work can produce a consistent hash value for every possible graph.

Sayers and Karp [20] propose a hash algorithm for N3 graphs that does not need to canonicalize the graph before computing its hash. The algorithm computes a partial hash for each triple, and then all the hash values are mixed to

produce the hash of the whole graph. In that work, blank nodes are supported by explicitly labelling them with new labels. Authors do not explain how labels for blank nodes are selected. Taking into account that labels can be assigned non–deterministically, hash values produced for graphs are also non–deterministic.

The main limitations of the work of Sayers and Karp compared to this work are:

- It does not support subformulae, and therefore can only be applied to a subset of N3 graphs.
- Hashes are non–deterministic if there are blank nodes, because authors do not propose a deterministic way of selecting labels for blank nodes. For some applications, like graph signing, this is not a limitation in general, because labels are included in the RDF file whose signature has to be checked. However, this non–determinism precludes the algorithm from being used for other applications, like checking equivalence of graphs.

11. Conclusions

Two related algorithms that work on N3 graphs have been presented in this paper. The first algorithm computes the hash value of a graph. The second one, which is based on the partial hash values produced by the hash algorithm, computes the canonical representation of a graph. Although designed for N3, both algorithms are also able to work with RDF graphs represented with RDF/XML notation, which has been widely adopted by the Semantic Web community.

The ability to properly handle blank nodes, variables and subgraphs is the main contribution of this work with respect to the related work found in the literature. The mechanisms designed to handle those three N3 features led, in fact, to most of the complexity of the algorithms.

The experiments carried out on a big N3 dataset, using a prototype implementation of the algorithms, show that the probability of hash collision is very low (approximately 2^{32} hash values are needed to get a probability of collision of 0.5). This result is supported by the *birthday paradox*, which can be applied to this case because the hash values produced by the hash algorithm seem to

be random and uniformly–distributed (no evidence of non–uniformity has been found using a collection of several randomness tests).

A theoretical analysis of the algorithms shows that their complexity is between $O(N \log N)$ and, in the worst case, $O(N^2)$. These results have been confirmed experimentally using the dataset.

A. Proof of correctness of the stop condition of the multi–step algorithm

This section proves the correctness of the stop condition proposed for the multi–step hash algorithm in Section 6.1. Because of lemma 3 (see below), in absence of random collisions it is guaranteed that if the number of collisions is the same in two consecutive steps, more steps will no lower that number of collisions. Lemmas 1 and 2 are intermediate results needed to prove lemma 3.

Definition 5. *A hash collision of two items (variables, statements, etc.) is said to be* non–random *iff it is caused by both hashes being computed with the same equations from the same input data.*

Let us represent the effects of the structure of the graph on the hash value of an item x by means of a function $f_x \colon V \times V \times \cdots \times V \to \mathbb{Z}_N$. Given the hashes at the previous step of those variables that affect the hash of x, it maps them to the hash value of x at the current step. The function f_x is, in general, different for different items. However, given a specific item, it is the same for all the steps of the hash algorithm.

The hash values of two items x and x' at step k are therefore computed using f_x and $f_{x'}$ from the hashes at step $k-1$ of the variables that affect x (variables $v_1, \ldots, v_n$) and x' (variables $v'_1, \ldots, v'_{n'}$) as follows:

$$h_x^k = f_x(h_{v_1}^{k-1}, \ldots, h_{v_n}^{k-1}) \quad (18)$$

$$h_{x'}^k = f_{x'}(h_{v'_1}^{k-1}, \ldots, h_{v'_{n'}}^{k-1}) \quad (19)$$

By Def. 5, if the hashes of x and x' collide, and the collision is not random, then:

$$f_x = f_{x'} \tag{20}$$

$$n = n' \tag{21}$$

and there exists an ordering of variables v_i and v'_i such that:

$$h^{k-1}_{v_1} = h^{k-1}_{v'_1}, \ldots, h^{k-1}_{v_n} = h^{k-1}_{v'_n} \tag{22}$$

Lemma 1. *In absence of random collisions, if two variables or two statements collide at step k, they must collide also at its previous step $k-1$.*

Proof. Suppose two variables v and v' such that they collide at step k but not at step $k-1$:

$$h^k_v = h^k_{v'} \tag{23}$$

$$h^{k-1}_v \neq h^{k-1}_{v'} \tag{24}$$

$$\tag{25}$$

The hash values of v and v' at step k are computed using f_v and $f_{v'}$ from the hashes at step $k-1$ of some of the variables of the graph as follows:

$$h^k_v = f_v(h^{k-1}_{v_1}, \ldots, h^{k-1}_{v_n}) \tag{26}$$

$$h^k_{v'} = f_{v'}(h^{k-1}_{v'_1}, \ldots, h^{k-1}_{v'_{n'}}) \tag{27}$$

Because the collision at k is not random, it must be due to both variables having exactly the same relations to other components of the graph, that is:

$$f_v = f_{v'} \tag{28}$$

$$n = n' \tag{29}$$

and f_v and $f_{v'}$ depending on equal hash values:

$$h^{k-1}_{v_1} = h^{k-1}_{v'_1}, \ldots, h^{k-1}_{v_n} = h^{k-1}_{v'_n} \tag{30}$$

Given that the hashes of v and v' do not collide at step $k-1$, it necessarily must exist $1 \leq i \leq n$ such that the pair of variables v_i and v'_i do not collide at step $k-2$. But taking into account that according to Eq. 30 those variables collide at step $k-1$, and applying the same reasoning by induction on v_i and v'_i, there must finally exist a pair of variables that collide at step 1 but not at step 0 (initial hash values from Eq. 4). However, if the collision at step 1 is not random, both variables must have the same quantification, and therefore their initial hashes must be equal, which leads to a contradiction.

If a pair of statements s and s' is considered instead of variables, and they collide at step k but not at step $k-1$, by the same reasoning applied above to v and v' it would imply that there must be a pair of variables that collide at step $k-1$ but not at step $k-2$. However, that has been proven to be impossible. □

Lemma 2. *In absence of random collisions, if the number of collisions at two consecutive steps k and $k+1$ is the same, any pair of variables v and v' whose hash value is equal at step k (i.e. $h_v^k = h_{v'}^k$) have also equal hash value at step $k+1$ (i.e. $h_v^{k+1} = h_{v'}^{k+1}$).*

Proof. Because of lemma 1, variables and statements that do not collide at step k cannot collide at step $k+1$. Consequently, no new (non–random) collisions may happen at step $k+1$.

Because no new collisions may happen, if the number of collisions at step k is equal to the number of collisions at step $k+1$, variables and statements that collide at step k must necessarily collide also at step $k+1$. □

Lemma 3. *In absence of random collisions and duplicate triples, if the number of collisions in a given step k is equal to the number of collisions in its following step $k+1$, there will be the same number of collisions in any other future step $k' > k+1$.*

Proof. If there were less collisions at step $k+2$ than at step $k+1$, at least two variables or two statements that collided at step $k+1$ might not collide at step $k+2$.

Let us suppose that at least two variables v and v' collide at step $k+1$ but do not at step $k+2$. Due to lemma 1 they must also collide at step k:

$$h_v^k = h_{v'}^k \tag{31}$$
$$h_v^{k+1} = h_{v'}^{k+1} \tag{32}$$
$$h_v^{k+2} \neq h_{v'}^{k+2} \tag{33}$$

Their hash values at steps k and $k+1$ depend, respectively, on the hashes of some of the variables of the graph at steps $k-1$ and k:

$$h_v^k = f_v(h_{v_1}^{k-1}, \ldots, h_{v_n}^{k-1}) \tag{34}$$
$$h_{v'}^k = f_{v'}(h_{v'_1}^{k-1}, \ldots, h_{v'_{n'}}^{k-1}) \tag{35}$$
$$h_v^{k+1} = f_v(h_{v_1}^{k}, \ldots, h_{v_n}^{k}) \tag{36}$$
$$h_{v'}^{k+1} = f_{v'}(h_{v'_1}^{k}, \ldots, h_{v'_{n'}}^{k}) \tag{37}$$

Because the collisions happening at steps k and $k+1$ are not random, it must be due to functions f_v and $f_{v'}$ being equal and depending on equal hash values:

$$f_v = f_{v'} \tag{38}$$
$$n = n' \tag{39}$$
$$h_{v_1}^{k-1} = h_{v'_1}^{k-1}, \ldots, h_{v_n}^{k-1} = h_{v'_n}^{k-1} \tag{40}$$
$$h_{v_1}^{k} = h_{v'_1}^{k}, \ldots, h_{v_n}^{k} = h_{v'_n}^{k} \tag{41}$$

Given that $f_v = f_{v'}$, the only way to avoid a collision between v and v' at step $k+2$ is that at least one of the equalities above no longer holds for step $k+1$. That is, there must exist v_i and v'_i with $1 \leq i \leq n$ such that:

$$h_{v_i}^{k-1} = h_{v'_i}^{k-1} \tag{42}$$
$$h_{v_i}^{k} = h_{v'_i}^{k} \tag{43}$$
$$h_{v_i}^{k+1} \neq h_{v'_i}^{k+1} \tag{44}$$

However, it is impossible by lemma 2, because the number of collisions in steps k and $k+1$ is the same. Therefore, it can be concluded that v and v' must still collide at step $k+2$.

If two statements colliding at step $k+1$ but not at step $k+2$ are taken instead of two variables, the reasoning above is equally valid for them, leading also to a contradiction with lemma 2.

Consequently, the number of collisions at step $k+2$ has to be the same as at steps k and $k+1$. By induction, the same is true also for any other step $k' > k+1$. □

References

[1] T. Berners-Lee, Notation 3: A Readable RDF Syntax, http://www.w3.org/DesignIssues/Notation3, (Visited May 2009).

[2] J. L. Gross, J. Yellen (Eds.), Handbook of Graph Theory, Discrete Mathematics and its Applications, CRC Press, 2003.

[3] M. Grohe, Isomorphism testing for embeddable graphs through definability, in: STOC '00: Proceedings of the thirty-second annual ACM symposium on Theory of computing, ACM, New York, NY, USA, 2000, pp. 63–72. doi:http://doi.acm.org/10.1145/335305.335313.

[4] D. G. Corneil, C. C. Gotlieb, An efficient algorithm for graph isomorphism, Journal of the ACM 17 (1970) 51–64.

[5] B. Weistfeiler (Ed.), On construction and identification of graphs, Lecture Notes in Mathematics, Springer, 1976.

[6] L. Babai, P. Erdös, S. M. Selkov, Random graph isomorphism, SIAM Journal on Computing 9 (1980) 628–635.

[7] C. Hoffmann, Group-theoretic algorithms and graph isomorphism, Lecture Notes in Computer Science 136.

[8] D. Corneil, M. Goldberg, A non–factorial algorithm for canonical numbering of a graph, Journal of Algorithms 5 (1984) 345–362.

[9] E. Oren, R. Delbru, M. Catasta, R. Cyganiak, H. Stenzhorn, G. Tummarello, Sindice.com: a document-oriented lookup index for open linked data, International Journal of Metadata, Semantics and Ontologies 3 (1) (2008) 37–52.

[10] G. Tummarello, C. Morbidoni, R. Bachmann-Gmür, O. Erling, RDFSync: efficient remote synchronization of RDF models, in: The Semantic Web, 6th International Semantic Web Conference, Busan, Korea, November 11-15, 2007, Vol. 4825 of Lecture Notes in Computer Science, Springer, 2007.

[11] D. E. Knuth, Sorting and searching, 2nd Edition, Vol. 3 of The Art of Computer Programming, Addison Wesley, 1997.

[12] T. Berners-Lee, Cwm, http://www.w3.org/2000/10/swap/doc/cwm.html, (Visited May 2009).

[13] G. Sutcliffe, C. Suttner, The TPTP problem library: CNF release v1.2.1, Journal of Automated Reasoning 21 (2) (1998) 177–203.

[14] W. Stallings, Cryptography and Network Security, 4th Edition, Pearson Prentice Hall, 2006.

[15] J. Walker, Ent: A pseudorandom number sequence test program, http://www.fourmilab.ch/random/, (Visited May 2009).

[16] D. E. Knuth, Seminumerical algorithms, 2nd Edition, Vol. 2 of The Art of Computer Programming, Addison Wesley, 1997.

[17] W. W. Tsang, Tuning the collision test for stringency, Tech. Rep. TR-2000-05, Computer Science & Information Systems, Hong Kong University (2000).

[18] L. Ding, T. Finin, Characterizing the semantic web, in: Proceedings of the 5th International Semantic Web Conference, Vol. 4273 of Lecture Notes in Computer Science, Springer, 2006.

[19] J. J. Carroll, Signing RDF graphs, in: The SemanticWeb - ISWC 2003, Second International Semantic Web Conference, Vol. 2870 of Lecture Notes in Computer Science, Springer, 2003, pp. 369–384.

[20] C. Sayers, A. Karp, Computing the digest of an RDF graph, Tech. Rep. HPL-2003-235 (R. 1), Hewlett Packard Laboratories (2004).

Authors' biographies

Jesús Arias Fisteus received a MSc degree in Telecommunication Engineering from the University of Vigo, Spain, in 2001, and a PhD degree in Communication Technologies from the Carlos III University of Madrid, Spain, in 2005. In his PhD. thesis he proposed a formalism that uses model-checking for verifying business processes. His current research interests are Web engineering, Semantic Web, Web services and business processes. Currently he works as assistant professor at the Carlos III University of Madrid.

Norberto Fernández García received a MSc degree in Telecommunication Engineering from University of Vigo, Spain, in year 2002 and a PhD degree in Telematics from Carlos III University of Madrid in year 2007. His current research interests are mainly related with Semantic Web, more in detail, with semantic annotation, the convergence of Semantic Web and Web 2.0, and with the application of Semantic Web technologies into the journalism domain.

Luis Sánchez Fernández graduated as a telecommunications engineer from Universidad Politcnica de Madrid, Spain, in 1992 and received a PhD in Telecommunications Engineering, from the same university in 1997. In October 1997 he joined Universidad Carlos III de Madrid where he is currently a full professor in the Dept. of Telematics Engineering. He has participated and/or led a number of national research projects and one European project related to web technologies, including Semantic Web technologies, and has authored more than 50 publications in national and international conferences and journals as well as a number of chapters in scientific books. His current research activities are focused on the Semantic Web (semantic annotation, ontologies) and Web 2.0.

Carlos Delgado Kloos received the Ph.D. degree in Computer Science from the Technical University of Munich and in Telecommunications Engineering from the Technical University of Madrid in 1986. He is Full Professor of Telematics Enginering at the Carlos III University of Madrid, where he is the director of the on-line Master's programme on e-Learning <learn.uc3m.es>, of the Nokia Chair <www.it.uc3m.es/nokia>, and of the GAST research group <www.gast.it.uc3m.es>. He is also Associate Vice-rector of International Relations. His main interests include Internet-based applications and in particular e-learning. He has been involved in more than 20 projects with European, na-

tional or bilateral funding. He has published more than 200 articles in national and international conferences and journals. He has further written a book and co-edited five. He is the Spanish representative at IFIP TC3 on Education and Senior Member of IEEE.