Signing RDF Graphs

Jeremy J. Carroll
Digital Media Systems Laboratory
HP Laboratories Bristol

HPL-2003-142
July 23rd , 2003*

E-mail: jjc@hpl.hp.com

semantic web, digital signatures, RDF, graph isomorphism, canonicalization

Assuming **P<GI<NP**, the creation and verification of a digital signature of an arbitrary RDF graph cannot be done in polynomial time. However, it is possible to define a large class of canonicalizable RDF graphs, such that digital signatures for graphs in this class can be created and verified in O (n log (n)). Without changing its meaning, an arbitrary RDF graph can be nondeterministically pre-canonicalized into a graph of this class, before signing. The techniques in this paper are key enablers for the use of digital signature technology in the Semantic Web.

* Internal Accession Date Only
ISWC 2003, Florida, US. Oct. 2003

Signing RDF Graphs

Jeremy J. Carroll

Hewlett-Packard Labs, Bristol, UK, BS34 12QZ
jjc@hpl.hp.com

Abstract. Assuming **P**<**GI**<**NP**, the creation and verification of a digital signature of an arbitrary RDF graph cannot be done in polynomial time. However, it is possible to define a large class of canonicalizable RDF graphs, such that digital signatures for graphs in this class can be created and verified in O(*n*log(*n*)). Without changing its meaning, an arbitrary RDF graph can be nondeterministically pre-canonicalized into a graph of this class, before signing. The techniques in this paper are key enablers for the use of digital signature technology in the Semantic Web.

1 Introduction

Digital signatures permit a user of a document to verify that some named party has vouched for it. This is a key technology for reducing the trust barriers to e-business and e-commerce.

For use within the Semantic Web, it is necessary for the user of an RDF graph [1] to be able to verify a signature, either for that graph or for a subgraph.

All obvious methods for doing this have exponential worst-case behaviour.

We present a method that involves:

- a class of canonical RDF documents,
- a canonicalization algorithm that runs in O(*n*log(*n*)) but fails on worst-case inputs, for which there is no corresponding canonical RDF document,
- a precanonicalization algorithm, applicable to any RDF graph, that makes nondeterministic modifications to the input graph, if there is a danger that it is one of the worst-case inputs.

Signing a graph then involves running the precanonicalization algorithm, which may modify the graph, canonicalization the modified graph and signing the resulting canonical RDF.

Verifying a signature involves extracting the signed subgraph, canonicalizing this subgraph, and verifying the signature for the canonical RDF. If the canonicalization algorithms fails, then the graph was not signed.

It is crucial that we can make significant syntactic modifications to the graph that are explicitly meaningless

Section 2 demonstrates that RDF canonicalization is graph isomorphism (GI) complete. Section 3 introduces some preliminary notions needed in sections 4 and 6 which contains the principle algorithm of this paper. Section 5 defines canonical RDF, on the basis of the naïve algorithm of section 4. Section 6 shows how to patch up the problems caused by the naivety. Section 7 details some more sophisticated techniques which are not necessary but may be desirable. Section 8 shows that all the techniques used are O(*n*log*n*). Section 9 focuses on practical use of the techniques and reports on some test results.

2 Verifying Signatures of Arbitrary RDF

In a Semantic Web application, there is likely to be an RDF graph. If we wish to verify that either it, or a subgraph of it, has been digitally signed in some way, then we have to solve at least one well-known hard problem. Note this way of stating the problem rules out one scenario which is that *B* has read the graph from a file signed by *A*. If *B* has done this, then *B* can simply verify the signature on the file, and has no need to verify the signature for the graph, as a graph.

We follow [2] and assume the graph isomorphism problem is strictly harder than polynomial time, and strictly easier than nondeterministic polynomial time. (i.e. P < GI < NP). We also note that the subgraph isomorphism problem is well known to be NP complete. These results are applicable to RDF, see section 2.3.

Thus, if *A* wishes to sign an RDF graph and *B* wishes to verify the signature, then *A* needs to serialize the graph in some form, sign the serialization, and *B* needs to verify that signature. This overall process of signature verification permits a comparison (either for isomorphism or for subgraph) between the graph that *A* signs, and the graph that *B* verifies. This overall process solves the graph isomorphism problem or the subgraph isomorphism problem. Thus either the algorithm for signing the graph, or the algorithm for verifying the signature of the graph must be of complexity class GI or NP. It would possibly be best to make the signer do the hard work. Messmer and Bunke [3] show that graph and subgraph isomorphism with respect to a fixed graph, of size m, comparing with a varying graph of size n, can be performed in O(n^3) time, but needs O(3^m) space for the verification algorithm, computed in O(m^m) time. Note that *A* would calculate this large algorithm, and *B* would have to execute it (without downloading all of it). This method could be used to give signature verification algorithms of cubic complexity; but creating the signature is of exponential complexity.

These worst-case scenarios are bad, and the solution for signing RDF needs to avoid them. The solution we present is based on not attempting to sign or verify signatures of *arbitrary* RDF.

2.1 Meaningless Changes

The key insight of this paper is that while in general signing RDF Graphs is GI complete, all interesting RDF graphs can be slightly modified (typically by adding a few, explicitly meaningless, triples) to be in a class of RDF graphs which can be canonicalized, and hence both signed and verified, in O(*n*log*n*). Moreover, the class of easily canonicalizable RDF graphs can be large enough so that 90% of ‘interesting’ RDF graphs lie within it, and most of the other interesting graphs require only slight modification.

For the purpose of signing an RDF graph, since such modifications are meaningless, the social act of signing the unmodified graph is the same as the social act of signing the modified graph. Thus we always sign modified graphs, that can be canonicalized in O(*n*log(*n*)). For signature verification, if the graph requires modification, then the signature to be checked was incorrect, since the only graphs that get signed have been modified and do not require further modification.

The reader should judge whether these modifications are merely a dirty hack or an elegant engineering compromise.

An alternative engineering approach of simply deleting problematic parts of the graph is also explored.

2.2 A Simple Example

Consider an RDF Graph with two triples, represented in N-Triples [4] as either:

```
# Here is a graph
_:aBlankNode <eg:prop>    “3” .
_:aBlankNode   <eg:prop> “5”   .
```

or

```
_:ax     <eg:prop> “5”   .
_:ax  <eg:prop> “3” .
```

The two files are different, but the graph they describe is the same. The differences between the files are *insignificant*. These include: the comment; the whitespace; the order of the lines; the choice of blank node identifiers (e.g. *aBlankNode* or *ax*)

Other aspects of the N-Triples are *significant* in that they represent the intended abstract RDF Graph [1]. Significant aspects include: the presence or absence of a triple; the string in each literal; the URIs for each property or resource.

Our approach to signing a graph is to: specify a class of canonical RDF documents; and to sign an RDF graph by signing a canonical RDF document which is a serialization of either that graph, or a different graph which both entails and is entailed by the graph to be signed.

Canonical RDF is chosen so that there is at most one representation of that graph in Canonical RDF. Thus, we can check that these two are the same by converting both into canonical RDF and doing a character-by-character comparison. This might fail, if the graph proves difficult (which the above example does not).

Of the insignificant differences highlighted, only two present any interesting difficulties: the order of the lines, and the blank node identifiers.

2.3 RDF C14N is GI Complete

The techniques in this paper cannot canonicalize arbitrary RDF graphs without making some modifications to them. This reflects the underlying difficulty of RDF canonicalization.

As discussed by Carroll [5], RDF graph equality and the graph isomorphism problem have equivalent complexity.

Any unlabelled undirected graph can be encoded in RDF by replacing each node with a blank node, and replacing each arc by two arcs (in each direction) always using a single property (e.g. `<eg:x>`). So a simple triangle (three nodes, three arcs) can be encoded in N-Triples as:

```
        _:a <eg:x> _:b .          _:a <eg:x> _:c .
        _:b <eg:x> _:a .          _:b <eg:x> _:c .
        _:c <eg:x> _:b .          _:c <eg:x> _:a .
```

If we could fully solve RDF canonicalization in polynomial time, then we could compare two RDF graphs for equality in polynomial time (by comparing their canonical representations). This would provide an (unlikely) polynomial time solution to the Graph Isomorphism problem [2], [6], [7].

2.4 What's the Difficulty?

The graph isomorphism problem is deceptive. It really doesn't look hard! We will informally explore the problem of a canonical representation of a simple graph, with two disconnected components shown in Figure 1.

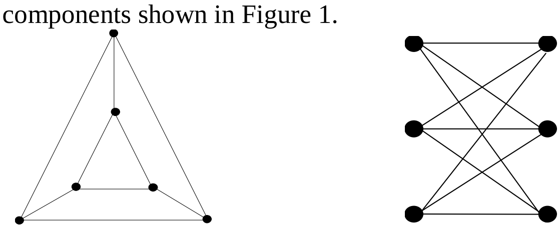

Figure 1 A 12 vertex graphs

The graph consists of 12 nodes and 18 edges. Each node has three neighbours, and two nodes at a distance of two away. Each node is in a connected component of six nodes.

If we try and canonicalize this, we will start by writing some node before all the other nodes. That node will either come from the component on the left or that on the right. The canonicalization algorithm needs to make a deterministic choice. There is no intuitively obvious rationale for choosing one or the other, and which ever choice we make seems to require considerations not just about the node, but also its neighbours, and their neighbours, and …

A good solution to this problem is provided by McKay [8], [9]. He uses an analysis of the automorphism group of a graph to construct canonical representatives. His solution is of non-polynomial complexity and complicated to program. It is not a plausible candidate for an infrastructural component within the semantic web.

3 Preliminaries

3.1 Syntactic

For clarity of exposition the algorithms described in this paper are based around the lexicographic sorting of N-Triples [4] files. N-Triples is an ASCII format.

We use numbered gensym identifiers during the algorithm. These are created with just sufficient leading zeros so that lexicographic ordering and integer ordering are the same. (The number of leading zeros required is computed from the number of blank nodes in the RDF graph being considered).

3.2 Semantic

The techniques in this paper rely on being able to make meaningless changes to an RDF graph. This is done in accordance with the RDF formal semantics [10], by using a special property, which we conventionally refer to as **`c14n:true`** defined here:

```
  <rdf:RDF xml:base=”&c14n;” xmlns:c14n=”&c14n;#”>
    <rdfs:Property rdf:ID=”true”>
       <rdfs:description>This property is true whatever
resource is its subject, and whatever literal is its object.
     Thus triples with literal objects, and c14n:true as
predicate, can arbitrarily be added to and deleted from an RDF
graph without changing its meaning. </rdfs:description>
    </rdfs:Property>
  <rdf:RDF>
```

(Omitting namespace declarations and the definition of the entity **`&c14n;`** as the URL `http://www-uk.hpl.hp.com/people/jjc/rdf/c14n`)

By specifying this predicate as being always true, adding or deleting triples with this predicate does not alter the entailments under the RDF model theory. Formally the semantics of the document are unchanged.

If desired, an alternative would be to use OWL (Full) to define a trivially true predicate which holds between all resources and all long integers. The following cardinality constraint, in the OWL abstract syntax [11], expresses this condition:

```
   class(rdfs:Resource complete restriction(c14n:true cardinality=2^64))
  dataValuedProperty( c14n:true range( xsd:long ) )
```

3.3 N-Triples to Canonical XML

In this paper, we concentrate on creating canonical representations of RDF graphs in N-Triples [4].

However, for full integration with tools built on Canonical XML [12] it is necessary to transform these files into XML, and, moreover, we wish the XML produced to canonically depend on the original RDF graph.

This needs to be done by specifying a canonical method for turning an N-Triple document into RDF/XML. This must: preserve the order of the triples; prohibit the many variants in the grammar; specify the whitespace precisely.

3.4 RDF C14N without Blank Nodes

To create a canonical N-Triples file for an RDF graph without any blank nodes we can simply reorder the lines in the N-Triples document to be in lexicographic order. (This could be implemented simply with Unix™ **`sort`**[1]). We may wish to canonicalise the lexical forms of typed literals first.

[1] **`sort`** in Unix uses a locale dependent ordering. To use US-ASCII order, it is necessary to set the environment variable `LC_ALL=C`.

4 Labelling Blank Nodes

If we have blank nodes in the graph then life is somewhat trickier.

In N-Triples blank nodes are represented using blank node identifiers, which can appear in subject or object position.

Unfortunately, these identifiers are gensyms created during the writing of the N-Triples, and are not an intrinsic part of the graph. Hence, it is an error if the canonicalization depends on these gensyms. In contrast, the canonicalization algorithm must deterministically choose new blank node identifiers.

In this part of our approach, we first write out the file using arbitrarily chosen blank node identifiers; then we sort the document (mostly) ignoring those identifiers. On the basis of this sorted document, we then rename all the blank nodes, in a (hopefully) deterministic fashion.

Since the level of determinism is crucial to the workings of the canonicalization algorithm, we start by defining a deterministic blank node labelling algorithm. This suffers from the defect of not necessarily labeling all the blank nodes.

Deterministic, in this context, means dependent only upon significant parts of the initial representation of the graph, and *nondeterministic* means dependent, in part, upon insignificant parts of the initial representation of the graph.

4.1 One-step Deterministic Labelling

In RDF [1], a single triple can contain zero, one or two blank nodes (the property must be specified by a URI reference).

We present an algorithm that labels some blank nodes on the basis of the immediate neighbors of that blank node (the one-step in the name of the algorithm).

1. Canonicalize the typed literals in the graph according to XML Schema [13].
2. Write the graph as an N-Triples document. Each line of the document is a complete distinct triple of the graph.
3. For each line with a blank node identifier in subject position (e.g. **`_:subj`**), replace the blank node identifier with **`“~”`** and add a comment **`“# _:subj”`** to the end of the line, indicating the original identifier.
4. For each line with a blank node identifier in object position (e.g. **`_:obj`**), replace the blank node identifier with **`“~”`** and add a comment **`“# _:obj”`** to the end of the line, indicating the original identifier.
5. Reorder the lines in the N-Triples document to be in lexicographic order. (This could be implemented with Unix™ **`sort`**).
6. Use a gensym counter, initialized to 1, and a lookup table, initially empty. Go through the file from top to bottom:
    a. If this line is the same as the next or previous line excluding any trailing comment, continue to the next line in the file.
    b. If there is a **`“~”`** in object position:
        i. Extract the blank node identifier from the final comment in the line. Remove the comment.
        ii. Look the identifier up in the table.

iii. If there is no entry, insert a new entry formed from **"_:g"** concatenated with the value of the gensym counter. Increment the counter.

iv. Replace the **"~"** with the value from the table.

c. If there is a **"~"** in subject position, use the same subprocedure to replace it with a consistently chosen gensym.

7. Using the same lookup table. Go through the file from top to bottom:

   a. If there is a **"~"** in object position:

      i. Extract the blank node identifier from the final comment in the line. Remove the comment.

      ii. Look the identifier up in the table.

      iii. If there is an entry, replace the **"~"** with the value from the table.

   b. If there is a **"~"** in subject position, use the same subprocedure to possibly replace it with a consistently chosen gensym.

8. Lexicographically sort the N-Triples again.

The only nondeterminism here is that the sort may depend on the blank node labels in pairs of triples for which the rest of the triple is identical. Such pairs are studiously avoided (step 6.a) in the assignment of labels, and so the order in which such pairs appear does not effect the labels chosen. Notice that step 7, which does deal with the incomparable lines, does not choose any labels and is hence deterministic.

The algorithm will deterministically label some of the blank nodes, for others it leaves them unlabelled. These nodes are referred to as *hard to label* nodes.

The operation of the deterministic labeling algorithm depends on triples that can distinguish one blank node from another. These *distinctive triples* are characterized by being unique in the graph even when all blank nodes are treated as identical.

The hard to label nodes do not participate in any distinctive triples.

Example 1, is fully labeled by this algorithm:

<eg:a> <eg:foo> _:a .	→	<eg:a> <eg:foo> _:g1 .
_:a <eg:prop> "val" .		<eg:b> <eg:prop> _:g2 .
<eg:b> <eg:prop> _:b .		_:g1 <eg:prop> "val" .

On the other hand, **example 2** is not fully labelled:

_:a <eg:zee> "why" .	→	<eg:b> <eg:prop> _:g1 .
_:a <eg:prop> "val" .		_:g2 <eg:prop> "val" .
<eg:b> <eg:prop> _:b .		_:g2 <eg:zee> "why" .
_:b3 <eg:prop> "val" .		~ <eg:prop> "val" . # _:b3

5 Canonical RDF

The algorithm above has the desired property of being of a low complexity class O(*n*log(*n*)), and hence optimizable to be sufficiently quick.

Thus we will define canonical RDF on the basis of this algorithm.

A canonical N-Triples document is one (without any comments) which is unchanged under the application of the one-step deterministic labeling algorithm.

That is canonical RDF in N-Triples has the following features:

- There are no hard to label nodes.

- Every blank node identifier has the form gNNN where NNN is some number of digits. The number of the digits is the same for every blank node identifier. At least one identifier has a non-zero first digit.
- After deleting all triples which are not distinctive, the first occurrences of each blank node identifier appear in numeric order, starting at 1, without gaps.
- The file is in lexicographic sort order.

Such files are unchanged under the one-step deterministic labeling.

If one-step deterministic labeling successfully labels all the nodes, then the resulting output will be canonical RDF. In such cases, one-step deterministic labeling is idempotent. So the resulting file in example 1 above is in canonical RDF.

6 Pre-canonicalization

We seek to make such a definition useable in all cases. Instead of defining the set of canonical RDF documents, such that *every* RDF document has a canonical representative, we set out to modify RDF graphs which do not have a canonical representative into ones that do, by ensuring that there are no hard to label nodes. This process of making nondeterministic modifications to the input in order to make it easier to canonicalize is not acceptable in a true canonicalization algorithm. Thus we refer to this step as precanonicalization. We try to minimize the changes made; and to be informed by the application needs as to what sort of changes are acceptable (this paper concentrates on the digital signature application).

Since many of the steps involved in precanonicalization are identical to those in the one step deterministic labeling algorithm already presented, we continue to consider complete algorithms involving both precanonicalization and the labeling.

6.1 Nondeterministic Pre-canonicalization

Applications of digital signatures usually require that the meaning of the document is fully captured in the object that is signed. A (somewhat tired) example would be signing a purchase order. In terms of the RDF semantics [10], the signed object should both entail and be entailed by the document presented for signature. The **`c14n:true`** predicate presented in section 3.2 has been designed so that arbitrary triples with this predicate can be added or deleted, without modifying the meaning. Using such triples we can make sure that none of the nodes are hard to label.

The version described here uses many passes of the file – the three important passes:

- identify the hard to label nodes;
- create additional distinctive triples for those nodes
- deterministically label the resulting graph.

Thus, we can perform the following steps:

A. Perform a one-step deterministic labeling
B. If there are no hard to label nodes, then stop. [Ensures idempotence]
C. Delete all triples with predicate **`c14n:true`** . [B ensures idempotence]

D. Perform a one-step deterministic labeling.

E. Using a new lookup table from step D, and a new counter, scan the file from top to bottom, performing these steps on each line:
    a. If there is a `“~”` in object position:
        i. Extract the blank node identifier from the final comment in the line. Remove the comment.
        ii. Look the identifier up in the table.
        iii. If there is no entry: add an entry `“x”` to the table; and add a new triple to the graph with subject being the blank node identified by the identifier from the comment, predicate being **`c14n:true`** and object being the string form of the counter; increment the counter.
    b. If there is a `“~”` in subject position, use the same subprocedure to possibly create a distinctive triple for the subject as well.

F. Perform a one-step deterministic labeling (of the new modified graph, with a new lookup table and counter). Since all nodes participate in a distinctive triple, every table lookup will find an entry.

This algorithm is nondeterministic in step E, which is hence the precanonicalization step, but that nondeterminism is fairly limited, because even with the rather naïve one-step deterministic labeling algorithm almost all nodes in almost all (practically occurring) RDF graphs will have been classified.

So again continuing with example 2, we find:

```
<eg:b> <eg:prop> _:g1 .
_:g2 <eg:prop> "val" .
_:g2 <eg:zee> "why" .
_:g3 <eg:prop> "val" .
_:g3 <http://www-uk.hpl.hp.com/people/jjc/rdf/c14n#true> "1" .
```

An application scenario in which this is useable is that of signing an OWL ontology [14]. The ontologist creates the ontology in a tool, and then asks the tool to generate a signature using a private key. The tool:

- applies this algorithm, possibly adding additional triples to the ontology.
- creates a canonical RDF graph with the same meaning as the original ontology.
- computes the signature for the canonical RDF graph.
- adds additional triple(s) to the graph with the ontology URI as subject, and the signature as object.

The resulting graph (with additional triples both as a result of canonicalization and reflecting the signature) then replaces the original ontology. It is this graph (which can be canonicalized without any changes) that the ontologist publishes. Users of this ontology can then:

- find the signature in the graph.
- find the public key using some public key infrastructure
- delete the triple(s) carrying the signature from the graph.
- apply the deterministic labeling to form a canonical representation of the graph
- verify the signature of this canonical RDF using the public key.

6.2 Further Discussion

The one-step deterministic labeling algorithm used had the following characteristics:

- Reasonably fast (subquadratic)
- Deterministic
- Labels ‘enough’ blank nodes in realistic RDF
- Easy to explain and implement

The last point is helpful in a paper, but not a hard requirement. Thus in a deployed system we can imagine using a variant of the algorithm here.

It is a practical engineering problem to choose a near-optimal deterministic classification algorithm that gets the best trade off between speed of classification and practical utility. The deterministic labeling algorithm used in this paper is probably a touch naïve, but only a touch. We can be sure that those used by Mackay [8] are too expensive, given their non-polynomial complexity.

7 Advanced Methods

The methods in the previous sections are sufficient to solve the problem of signing RDF graphs. However, some additional techniques may improve the solution. These are discussed in this section.

7.1 Multistep Deterministic Labelling

The one-step deterministic labeling only considers the immediate neighbours of any node. In some graphs, particularly those arising in OWL ontology definitions [14], this may leave too many nodes unlabelled.

A solution is to choose a fixed depth e.g. two or three, and to consider the neighbours of a node up to that distance away. Thus we can consider the two-step and three-step deterministic labeling algorithms. Since for a fixed k a k-step method will, in essence, be k applications of the one step method, we will expect a linear slow down as we increase k. The underlying complexity class remains O(*n*log*n*). (It is straightforward to optimize k-step methods to make the slow down much less than linear).

To consider neighbours at distance N+1 we resort the partial labeled output of a deterministic labeling based on neighbours at distance N.

Having resorted the output, which will take into account those labels already deterministically chosen, we can then apply the one pass deterministic labeling algorithm, which effectively considers neighbours at an additional step removed.

For example, the full two-step deterministic labeling is as follows:

A. Perform a one step deterministic labeling.
B. Repeat steps 6, 7, 8 from the one step deterministic labeling algorithm, without reinitializing the table or counter.

For a k-step labeling, we need to repeat B k-1 times.

7.2 Optimizations

This paper has avoided issues to do with optimizing the algorithm presented, but concentrates on the complexity class of the algorithms.

Systems would benefit from merging steps presented here as sequential to reduce the total number of passes of the graph required. Also, given that most canonicalization is for objects that are small enough to fit in memory, the exposition in terms of sorting files is probably misleading. In-memory structures such as b-trees [15] should be used, and maintaining them in sort order, may be more efficient than permitting them to become unsorted and resorting them as a separate algorithmic step.

The multistep methods in particular are susceptible to optimization. Often the first stage will completely label the blank nodes of the graph. The subsequent stages are then redundant. Moreover, during the first stage we can identify those triples in the graph that require more work. The ones that do not need further work, play no further role in the algorithm, except being included, in a final merge, into the output.

7.3 Deterministic Precanonicalization

Other applications of digital signatures have a greater need for determinism than the exact preservation of meaning. Lynch [16] argues that canonicalization is a "Fundamental Tool to Facilitate Preservation and Management of Digital Information." The scenario he discusses is the need to reformat a digital object which is being preserved in an electronic library. The need for reformatting occurs as the software and hardware required to view the original object become obsolete.

Lynch uses (lossy) canonicalization as a way of defining the essence of a document. This essence, if sufficient unimportant details are discarded, will be invariant with reformatting. Thus we can compute this essence of the original document, and the original author can sign this essence to vouch for its authenticity.

Later, after the death of the author, and a sequence of many transformations of the original byte sequence, the essence of the document is unchanged, and we can still verify that the author signed it.

With this application canonicalization is intended to be lossy, but must be deterministic. A variant precanonicalization is simply to delete all hard-to-label nodes. This is deterministic, but loses meaning.

7.4 Multiple Arcs in a Single Comparison

The methods presented, have been constrained for simplicity, to those that can easily be implemented using Unix **`sort`** and **`awk`** over N-Triples.

On larger graphs it may be the case that there are nodes that do not participate in distinctive triples, but for which the set of triples in which they participate is distinctive. A simple example is:

```
_:a <eg:eat> “apple” .     _:a <eg:eat> “pear” .
_:b <eg:eat> “apple” .     _:b <eg:eat> “banana” .
_:c <eg:eat> “pear” .      _:c <eg:eat> “banana” .
```

None of the triples is distinctive, but **`_:a`** is the only blank node eating both apples and pears.

For some real RDF data (see section 9.1) it is advantageous to collect all the triples in which a node participates and use the set of labels on those triples as a key to attempt to distinguish the node.

8 Complexity

All the steps in this paper either involve lexicographically sorting a file, for which the best known algorithms are O(*n*log*n*) [17] or involve stepping through a file line by line and doing either the same simple modification, or a modification involving a table lookup (e.g. step 6.b.ii in section 4). The former steps are O(*n*), the latter O(*n*log*n*) (table lookup is O(log*n*), see [15]). Since any particular variant of the techniques involves a constant number of such sort or modification steps the overall complexity is O(*n*log*n*) where *n* is the number of triples in the RDF graph.

9 Practical Issues

The nondeterministic algorithm potentially triples the size of the graph (worst case). This paper is predicated on that potential occurrence not actually happening. The deterministic part of the algorithm will label almost all the nodes in a typical RDF graph

Both of the preconicalization methods, (addition or deletion) change the RDF graph. In this sense the algorithms presented are not canonicalization algorithms. However, we have argued that for practical applications these changes are acceptable, as long as: they happen relatively rarely; they are expected; and that the choice between the deleting method or the adding method has been made appropriately for the application.

A further difficulty is found when canonicalizing multiple subsets of an RDF graph, using the nondeterministic method. The required changes for one such canonicalization will most likely be incompatible with the required changes for another. This can be addressed by using super-properties of **`c14n:true`** to create the distinctive triples.

9.1 Test Results

To assess how often these changes are made, we applied some of the algorithms in this paper to various test data sources. Specifically, we use the one step, two step and three step deterministic labeling algorithms[2] on the test data; and then further analyzed some of the problem cases.

[2] We used a small prototype implementation [18] written in AWK on a Unix system.

Source	**1**	**2**	**3**	*****
RDF test data [4]	130	0	0	0
OWL test data [19]	45	2	0	0
OWL guide [20]	0	0	0	1
DAML ontology library [21] (part)	93	10	5	40

The last line shows that of the files tested from the DAML ontology library 93 were deterministically labeled by the one-step algorithm. A further 10 were deterministically labeled using the two-step algorithm, but that 40 were not fully labeled even with the three step algorithm. The first two lines show that the N-Triples files include with test cases for RDF and OWL are not sufficiently challenging to be good tests for canonicalization.

The Baseball ontology [22] was one of the more difficult ontologies. The specific problem concerned the definition of `#LineUpEvent` and `#Start.` The four restrictions shown each have a blank node, and none of the triples used in the four are distinctive. This is a more complicated variant of the example in section 7.4.

```
<rdfs:Class
         rdf:about="#LineupEvent">
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty
      rdf:resource="#position"/>
     <daml:toClass
       rdf:resource ="#Position"/>
  </daml:Restriction>
 </rdfs:subClassOf>
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty
      rdf:resource="#player"/>
   <daml:toClass
      rdf:resource="#Player"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</rdfs:Class>
```

```
<rdfs:Class
         rdf:about="#Start">
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty
      rdf:resource="#position"/>
     <daml:toClass
       rdf:resource ="#Position"/>
  </daml:Restriction>
 </rdfs:subClassOf>
 <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty
      rdf:resource="#player"/>
   <daml:toClass
      rdf:resource="#Player"/>
  </daml:Restriction>
 </rdfs:subClassOf>
</rdfs:Class>
```

This observation prompted the use of a multiple arc comparison algorithm.

The algorithm: first did an N-step deterministic labeling (using single arcs); then performed a single multiple arc labeling, only on the as yet unlabelled nodes; then to finish off did a further M-step deterministic labeling. This algorithm was tested on the 56 files (from the DAML ontology library and the OWL Guide) that had proved difficult. The results were as in this table:

		Post (M)			
		0	1	2	3
Pre (N)	0	12	9	7	6
	1	9	7	6	6
	2	8	6	6	6
	3	7	6	6	6

The N and M values show the number of deterministic labellings before and after the multiple arc labeling. The entries show how many of the 56 files were not completely labeled by such an approach.

These show that the multiple arc method, even without the one step method, was effective in all but twelve of the difficult cases. Since these were about a third of the sample from the ontology library, the multiple arc method, used by itself, may be effective in 90% of the cases.

9.2 Redundancy

The test results pointed to a core of six difficult examples. These were analyzed further. One was the OWL Wine ontology from the OWL Guide [20]. The specific construct that caused the problem was the innocent looking:

```
<owl:Class rdf:ID="Wine">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasMaker" />
      <owl:cardinality
          rdf:datatype=
  "http://www.w3.org/2000/10/XMLSchema#nonNegativeInteger"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
```

The problem with this construct is that, presumably by error, it is repeated and occurs two times in the wine ontology. This gives rise to two blank nodes, one for each restriction. Each participates in four triples (with properties `rdfs:subClassOf, owl:onProperty, rdf:type, owl:cardinality`). These two blank nodes are indistinguishable.

The other five difficult cases all had examples of such repeated structures.

These repeated constructs add no meaning. The repeat, if detected, could simply be deleted. Unfortunately, detecting a repeated substructure within an RDF graph can be shown to be at least as hard as the graph isomorphism problem. Mugnier and Chein [23] show that the somewhat harder problem of removing all redundant arcs from a conceptual graph is NP complete. (In the RDF context this would include deletion of partial repeated substructures).

9.3 Problems concerning rdf:Collection

It is conjectured that lists formed with the `rdf:parseType=”Collection”` construct may present greater problems than has been revealed in this study.

The `rdf:parseType=”Collection”` syntax [24] when used for lists with two or more elements, creates a number of blank nodes which:

- Are the subject of a triple with predicate **`rdf:type`**, and object **`rdf:List`**.
- Are the subject of an **`rdf:first`** triple with object being a blank node.
- Are the subject of an **`rdf:rest`** triple with object being a blank node.
- Are the object of an **`rdf:rest`** triple with subject being a blank node.

Since none of these triples is distinctive, and the node does not participate in any others, the one pass deterministic labeling will not label such nodes. Moreover the multiple arc label (section 7.4) does not help.

Given the importance of OWL in the semantic web, and the desire to canonicalize and digitally sign ontologies, this is a disadvantage with one-step methods. Moreover, additional distinctive arcs on such nodes, added as a result of the nondeterministic precanonicalization algorithm, would cause the resulting graph to be one that cannot use the compact `rdf:parseType=”Collection”` syntax.

The multistep methods will often distinguish the blank nodes in a typical OWL ontology (possibly requiring the multiple arc label method). This is because while the elements of the list may all be blank, for example a list of restrictions, enough of these elements will participate in distinctive triples, for example, distinctive **`owl:hasValue`** triples. When adding further distinctive triples it is best to try and exploit the multistep canonicalization and to avoid adding them to the blank nodes carrying the list structure, but instead to add them to the members of the list (e.g. to a blank node of type **`owl:Restriction`**).

9.4 Summary of Test Results

The test results show that:

- The one step deterministic algorithm is usable but may require nondeterministic modification of many graphs arising from OWL.
- Methods involving comparisons between multiple arcs are applicable to almost all of the RDF data studied.
- Duplicate subgraphs could not be labeled deterministically, and cannot even be detected in polynomial time. The approach of nondeterministic graph modification to deal with these seems appropriate.

10 Conclusions

While we might expect that signing RDF graphs involves exponential worse case behaviour, this paper has shown that there are O(*n*log*n*) solutions.

The key is a preparedness to modify the graph prior to signing. Such modifications can be justified by the observation that the signature endorses the meaning of the graph, rather than the graph itself. The modifications can be chosen so as not to change the meaning of the RDF graph.

Judging from the test results, these modifications are, in practice, only needed for handling the problem of duplication within the graph to be signed. These results suggest that the naïve definition of canonical RDF in section 5 should be augmented with considerations of labels from multiple arcs (section 7.4).

References

1. Klyne, G., Carroll, J. J. (eds.): RDF Concepts and Abstract Syntax. W3C (2003)
2. Köbler, J., Schöning, U., Torán, J.: The Graph Isomorphism Problem: Its Structural Complexity. Birkhauser (1993)
3. Messmer, B.T., Bunke, H.: Subgraph Isomorphism Detection in Polynomial Time on Preprocessed Model Graphs. ACCV (1995) 373-382
4. Grant, J., Beckett, D. (eds.): RDF Test Cases. W3C Working Draft (2002) http://www.w3.org/TR/2002/WD-rdf-testcases-20021112/

5. Carroll, J.J.: Matching RDF Graphs. In: Horrocks, I. and Hendler, J. (eds.): ISWC 2002, LNCS 2342 (2002) 5-15.
6. Read, R.C., Corneil, D.G.: Graph Isomorphism Disease. Journal of Graph Theory (1977) 339-363
7. Fortin, S.:The Graph Isomorphism Problem, Technical Report TR 96-20, Department of Computer Science, University of Alberta (1996) ftp://ftp.cs.ualberta.ca/pub/TechReports/1996/TR96-20/TR96-20.ps.gz
8. McKay, B.D.: Practical Graph Isomorphism. Congressus Numerantium 30 (1981) 45-87 http://cs.anu.edu.au/~bdm/papers/pgi.pdf
9. McKay, B.D.: Nauty (1994) http://cs.anu.edu.au/~bdm/nauty/
10. Hayes, P. (ed.): RDF Semantics. W3C Working Draft (2003)
11. Patel-Schneider, P.F., Hayes, P., Horrocks, I. (eds.): OWL Semantics and Abstract Syntax. W3C Working Draft (2003)
12. Boyer, J. (ed.): Canonical XML. W3C Recommendation (2001)
13. Biron, P.V., Malhotra, A. (eds.): XML Schema Part 2: Datatypes. W3C Rec. (2001)
14. Dean, M., Schrieber, G.: OWL Reference. W3C Working Draft (2003)
15. Adel'son-Vel'skii, G.M., Landis, E.M.: An algorithm for the Organization of Information. Doklady Akademia Nauk SSSR, vol. 146 (1962) 263-266. English translation in Soviet Mathematics Doklady, vol. 3, pp. 1259-1263.
16. Lynch, C.: Canonicalization: A Fundamental Tool to Facilitate Preservation and Management of Digital Information. D-Lib 5(9) (1999) http://www.dlib.org/dlib/september99/09lynch.html
17. Knuth, D.E.: The Art of Computer Programming, V.3: Sorting and Searching (1973)
18. Carroll, J.J.: Awk scripts for canonicalizing N-Triples (2002) http://www-uk.hpl.hp.com/people/jjc/rdf/c14n-impl/
19. Carroll, J.J., de Roo, J. (eds.): Web Ontology Language (OWL) Test Cases, W3C Working Draft (2002) http://www.w3.org/TR/2002/WD-owl-test-20021024/
20. Smith, M.K., Welty, C., McGuinness, D. (eds): OWL Web Ontology Language, Guide. W3C Working Draft (2003) http://www.w3.org/TR/2003/WD-owl-guide-20030331/
21. Dean, M.(librarian): DAML Ontology Library http://www.daml.org/ontologies/
22. Dean, M.: Baseball Ontology, http://www.daml.org/2001/08/baseball/baseball-ont
23. Chien M., Mugnier, M.-L.: Conceptual Graphs: Fundamental Notions. Revue d’Intelligence Atificielle, Vol 5, n^{o} 4 (1992) 365-406
24. Beckett, D. (ed.): RDF/XML Syntax Specification (Revised). W3C Working Draft (2003)