Draft 13th July 2001 — *please send comments to jjc@hpl.hp.com*

Matching RDF Graphs

Jeremy J. Carroll
Hewlett-Packard Laboratories Bristol, UK
jjc@hpl.hp.com

July 2001

Abstract

The Resource Description Framework (RDF) describes graphs of statements about resources. This paper explores the equality of two RDF graphs in light of the graph isomorphism literature. We consider anonymous resources as unlabelled vertices in a graph, and show that the standard graph isomorphism algorithms, developed in the 1970's, can be used effectively for comparing RDF graphs.

1. Introduction

The resource description framework (RDF) specification [6] defines a model and a syntax. The syntax is defined on top of the XML syntax [2]. The model is defined in terms of resources, often identified with URIs [1], and literals. Some of the resources are "anonymous". The model is a set of triples.

This paper concerns the comparison of two models — read (or otherwise created) from different files.

If the two models consist of identical sets of triples then the two models are equal. As we shall see in the next section, to limit equality to only such cases would be counter-intuitive in its treatment of anonymous resources.

We explore that issue in some depth, and show that this makes RDF model equality to be equivalent to graph isomorphism. We show that standard algorithms for graph isomorphism, particularly iterative vertex classification from Read and Corneil [10] (section 6, pp 346-347), are applicable.

We describe the algorithm as used in Jena 1-1-0 [7].

We finally consider whether there are other aspects of model equality and equivalence.

2. Some examples

Consider a simple RDF file:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:t="http://example.org/brothers#"
    xml:base="http://example.org/brothers">
    <rdf:Description  rdf:about="#John" t:name="John">
      <t:child  rdf:resource="#Robert" t:name="Robert"/>
      <t:child  rdf:resource="#Jeremy" t:name="Jeremy"/>
      <t:child  rdf:resource="#Terry" t:name="Terry"/>
    </rdf:Description>
</rdf:RDF>
```

This creates a model with four triples:

```
<#John> <http://example.org/brothers#child> <#Robert> .
<#Robert> <http://example.org/brothers#name> "Robert" .
<#John> <http://example.org/brothers#child> <#Terry> .
<#John> <http://example.org/brothers#name> "John" .
<#Terry> <http://example.org/brothers#name> "Terry" .
<#Jeremy> <http://example.org/brothers#name> "Jeremy" .
<#John> <http://example.org/brothers#child> <#Jeremy> .
```

The syntax we use for such triples is the "N-triple" syntax being used by the RDF working group [4].

Suppose we produce two RDF models by reading this in twice. They are clearly the same. Indeed, even if we reorder the XML we get the same model.

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:t="http://example.org/brothers#"
    xml:base="http://example.org/brothers">
    <rdf:Description  rdf:about="#John" t:name="John">
      <t:child  rdf:resource="#Jeremy" t:name="Jeremy"/>
      <t:child  rdf:resource="#Terry" t:name="Terry"/>
      <t:child  rdf:resource="#Robert" t:name="Robert"/>
    </rdf:Description>
</rdf:RDF>
```

If we consider the same examples with anonymous resources, it is much less clear. Typically our RDF processing environment assigns gensyms to each anonymous resource. And so we may get sets of triples like:

```
_:a3 <http://example.org/brothers#name> "Robert" .
_:a1 <http://example.org/brothers#name> "John" .
_:a1 <http://example.org/brothers#child> _:a9 .
_:a1 <http://example.org/brothers#child> _:a3 .
_:a9 <http://example.org/brothers#name> "Terry" .
_:a6 <http://example.org/brothers#name> "Jeremy" .
_:a1 <http://example.org/brothers#child> _:a6 .
```

corresponding to:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:t="http://example.org/brothers#"
    xml:base="http://example.org/brothers">
    <rdf:Description  t:name="John">
      <t:child  t:name="Robert"/>
      <t:child  t:name="Jeremy"/>
      <t:child  t:name="Terry"/>
    </rdf:Description>
</rdf:RDF>
```

and:

```
_:a3 <http://example.org/brothers#name> "Jeremy" .
_:a6 <http://example.org/brothers#name> "Terry" .
_:a1 <http://example.org/brothers#name> "John" .
_:a1 <http://example.org/brothers#child> _:a9 .
_:a1 <http://example.org/brothers#child> _:a3 .
_:a9 <http://example.org/brothers#name> "Robert" .
_:a1 <http://example.org/brothers#child> _:a6 .
```

corresponding to:

```
<rdf:RDF
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:t="http://example.org/brothers#"
    xml:base="http://example.org/brothers">
    <rdf:Description  t:name="John">
      <t:child  t:name="Jeremy"/>
      <t:child  t:name="Terry"/>
      <t:child  t:name="Robert"/>
    </rdf:Description>
</rdf:RDF>
```

A simple notion of equality suggests these are unequal, because the anonymous nodes have been given different gensyms (for example that with name `"Jeremy"` is `_:a6` in the first and `_:a3` in the second.

This does not seem consistent with the intended reading of anonymous resources being like resources but without a name. Hence we favour an

understanding of anonymous resources that gives equality between these models.

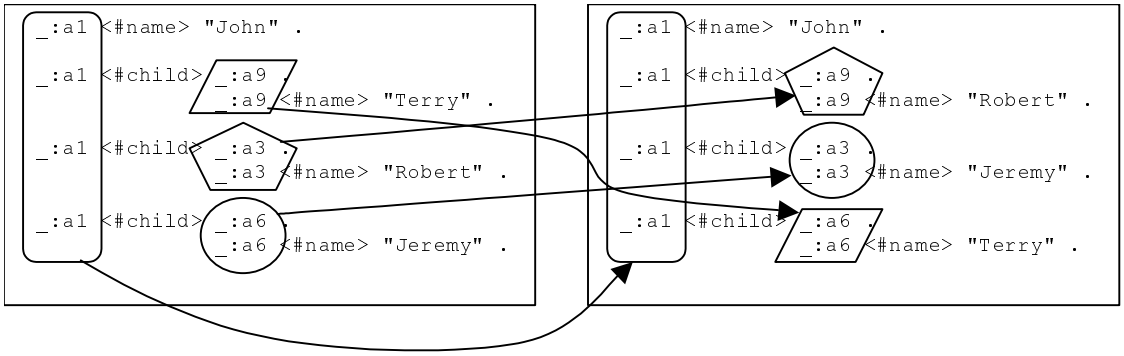
3. Anonymous resources are named with local scope

Despite being called "anonymous" it is necessary to name anonymous resources in some way if we wish to process them in any way. An example name might be an object reference within a program execution. A frequent name is the result of a gensym. Other resources have names with explicitly global scope: i.e. their names are URIs. The names of anonymous resources, in contrast, should not be meaningful outside the immediate environment.

This suggests that an appropriate naming for anonymous resources is one that gives them names with local scope: specifically the scope of the single RDF model, such as that built from a single RDF/XML file. The N-triple syntax [4] is clear, the names for the anonymous resources have file scope, linking the appearance of the resource in one triple with that in another triple *in the same file*.

Since such names are not externally visible, it is an implementation detail quite how they are generated.

Then in order to show that the two models are equal we have to show which anonymous resources correspond with which:



Such a correspondence needs to link each anonymous resource in the first model with an anonymous resource in the second, in such a way that all the triples in both graphs correspond. Technically, this is a bijection between the anonymous resources which induces a labelled digraph isomorphism.

4. Graph isomorphism theory

In the graph isomorphism literature (e.g. [5], [10]) a graph typically consists of a set of unlabelled nodes or vertices, with a set of undirected unlabelled pairs of vertices called edges. The graph isomorphism problem is: "Given two graphs, are they the same?" and "If they are, which vertices from one correspond to which vertices in the other?"

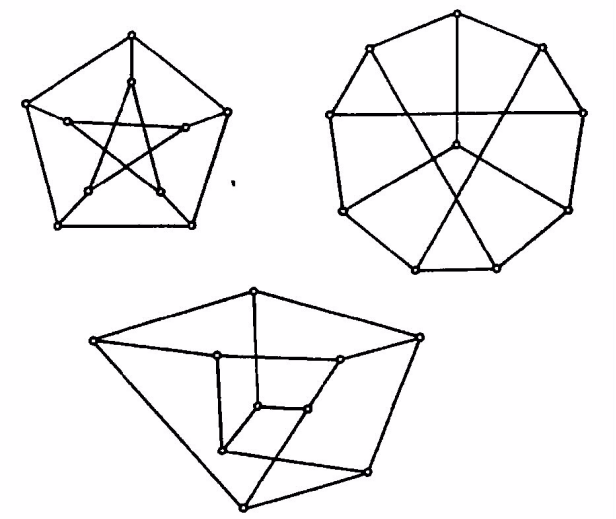


Figure 1 isomorphic graphs from [10]

Figure 1 shows three isomorphic graphs; note each has ten vertices shown by the small circles.

Most of the many variants of graphs have equivalent isomorphism problems. These included labelled digraphs: in which the edges have a label and a direction.

Within RDF models it is possible to encode an unlabelled digraph by using a single property label (e.g. rdf:value) for the edges and anonymous resources for each vertex. Undirected graphs can be encoded by encoding each edge of the graph as two RDF triples, one in each direction.

In this way it can be seen that RDF model equality and the graph isomorphism problem are equivalent from a theoretical point of view. However, in practice RDF model equality is significantly easier because:

- most of the vertices are labelled with the URI of a resource.
- most of the edges have distinctive labels from the URI of the property of the triple.
- the XML syntax imposes significant (and unmotivated) restrictions on where anonymous resources can occur.

We view the third point as an error that should be corrected; and regard the other two points as important factors in the design of an effective algorithm.

5. Iterative vertex classification algorithms

Standard graph isomorphism algorithms are non-deterministic, i.e. they involve guessing, e.g. (from [10], section 2).

1. Label the vertices V_1 of G_1.
2. Label the vertices V2 of G2.
3. If |V1|=|V2| set n = |V1| else the graphs are not isomorphic.
4. Guess a mapping from V_1 to V_2 (note: n! choices)
5. Check all the edges are the same. (at most, n^2 checks).

This is a slow method. There are n! different guesses to make, and maybe only one of them is correct. An implementation of this algorithm needs to use backtracking or some similar technique to consider the other guesses in the usual case that step 5 finds that the edges are not the same.

It is possible to greatly reduce the amount of guessing by classifying the vertices. The underlying idea of this method is to look for distinctive characteristics of the vertices, and then to only guess a mapping (in step 4) which maps any vertex in a class with some given characteristics to a vertex in the other graph of the equivalent class with the same characteristics. For example if a vertex is adjacent to three other vertices (i.e. it is at the end of three edges), then it can only map to a vertex adjacent to three further vertices (this is a classification by *'degree'*).

If the two graphs do not have equal numbers of vertices with each class of characteristics then the two graphs are not isomorphic.

This process of classifying vertices is also known in the graph theory literature as *colouring* the vertices, in deference to the four colour problem.[1]

Now we can make better guesses, we modify the algorithm above to be:

1. Label the vertices V_1 of G_1.
2. Label the vertices V2 of G2.
3. If |V1|=|V2| set n = |V1| else the graphs are not isomorphic.
4. Classify the vertices of both graphs.
5. For each class c in the classification
    a. Find the sets $V_{1,c}$ and $V_{2,c}$ of nodes which are in c
    b. If $|V_{1,c}|=|V_{2,c}|$ set $n_c = |V_{1,c}|$ else the graphs are not isomorphic.
    c. Guess a mapping from $V_{1,c}$ to $V_{2,c}$ (note: $n_c!$ choices)
6. Check all the edges are the same. (at most, n^2 checks).

This is an improvement because the total number of different guesses has been (substantially) reduced. (We make a number of small guesses instead of one large one). We can improve performance again by evaluating each of the checks of step 6 as early as possible, during step 5, as soon as both vertices involved in an edge have had their mapping assigned.

Iterative vertex classification (also known as *partition refinement*, in e.g. [8]) is when we use the information from our current classifications to reclassify the vertices producing smaller sets of each classification. In this we don't see a vertex classification as only a function of the vertex and the graph, but also of the current classification of the vertices of the graph. So for example, iterating on the degree classification above, we can classify a vertex by e.g. "This is adjacent to four vertices which have degree three," (or in more words, "This is adjacent to four vertices which are, in turn, adjacent to three vertices"). The typical classification is formed by AND-ing lots of classifications like that together.

[1] How many colours are needed to colour a map so that no two adjacent countries have the same colour.

Once we have made one guess aligning two vertices, we can re-classify the other vertices as to whether they are adjacent to the aligned vertices or not.

This can also apply after we have guessed. The full algorithm looks like:

1. Label the vertices V_1 of G_1.
2. Label the vertices V_2 of G_2.
3. If $|V_1|=|V_2|$ set $n = |V_1|$ else the graphs are not isomorphic.
4. Classify all the vertices of both graphs into a single class.
5. Repeat:
    a. Repeat – generate a new classification from the current classification
        i. Reclassify each vertex by the number of vertices of each class in the current classifcation it is adjacent to.
        ii. If the new classification is the same as the current classification go to 5(b)
        iii. If any of the new classes has different numbers of members from the two graphs then fail and backtrack to the last guess [step 5(c)].
        iv. If any of the new classes is small enough (e.g. size 2) go to 5(b)
        v. Set the current classification as the new classifcation and go to 5(a)i
    b. If every class has one element then this defines an isomorphism and we are finished.
    c. Choose the smallest class with more than one vertex and guess one vertex from each, when we run out of guesses, we backtrack to the last guess.
    d. Generate a new classification from the current classification by putting the vertex in each graph guessed in 5(c) into its own class and otherwise leaving everything unchanged.
6. If we backtrack through all the guesses in 5 then we have failed and the graphs are not isomorphic.

This is substantially more complicated than the original algorithm but gives much, much better performance. Yet better solutions to the graph isomorphism problem can be found [8], [9]; typically they use more sophisticated invariants than the adjaceny one described here, and they use the 'automorphism group' of one of the graphs to eliminant many redundant guesses. However, for RDF graphs the above algorithm will generally be sufficient.

6. Vertex classification for RDF

The code found in Jena [3] is based on the iterative vertex classification algorithm above. It classifies each non-anonymous resource by its URI and each literal by its string. It classifies each anonymous resource on the basis of the statements in which it appears. The classification considers the role in which an anonymous resource appears in a statement, and the other items in the statement.

This allows substantial use to be made of the labelled vertices and edges. The non-deterministic parts will not be used except when the labels do not allow us to directly distinguish one anonymous node from another.

The graph isomorphism algorithm above is then used, with minor variation[2]. The principle variation is the use of hash codes in the reclassification process.

An anonymous resource can play three different roles in an RDF statement, it can be subject, object or both. The ModelMatcher code [3] goes further and will allow anonymous resources in the predicate position. This gives a further four possibilities of where the anonymous resource occurs in the triple.

The iterative vertex classification then amounts to the following:

- the reclassification of a statement depends on the current classification of the resources in the statement.
- the reclassification of an anonymous resource depends on the reclassifications of all the statements it appears in, and the role it plays in each appearance.
- the reclassification of a non-anonymous resource or a literal is its original classification.

7. Partition refinement by hashcode

The invariants discussed above seem to have quite complicated representations; which suggests that comparing them may be slow. A simple way to proceed is always use hash-codes for each invariant value, combining them in commutative and associative or non-commutative fashion depending on whether we are discussing a set or a sequence at that point.

Thus the code in Jena ModelMatcher proceeds in this fashion:

- The code of an anonymous resource is the sum of its relative codes with respect to each triple it participates in. Note this means that an anonymous resource that participates in two triples of a certain class is distinguished from one that participates in three triples of that class.
- The relative code of an anonymous resource with respect to a triple is the sum of a multiplier times the secondary code of the triple's subject, predicate and object excluding those positions filled by the anonymous resource. The multiplier is chosen to distinguish the subject, predicate and object.
- The secondary code of a non-anonymous resource or literal is its Java hashCode.
- The secondary code of an anonymous resource is its code from the previous iteration (which identifies the current classification).

[2] A minor variation, that is probably an error, is that an emphasis is placed on finding singleton classes and we use a small maximum number of iterations not finding one.

The anonymous resources are classified on the basis of their codes. We may, of course, get a hash collision. This will have the consequence of combining two partitions. While this will decrease the efficiency of the algorithm it does not impact its correctness.

8. Other equivalences

We may wish to ask if two RDF graphs are equivalent with a notion of vertex equivalence that allows non-anonymous resources with different URIs to be identified, or that allows non-anonymous resources to be identified with anonymous ones.

In these cases we need to use a similar approach, the underlying problem is still graph isomorphism, but we use a different classification procedure. For example if we wish to allow the identification of different reifications of a statement, we would initially classify all reifications in a single class, and otherwise use the above algorithm.

Another natural example comes from the use of `rdf:Bag` which is defined as an unordered container, yet the container membership statements are distinguished `rdf:_1`, `rdf:_2` etc. This suggests that a statement equivalence that maps all of these to the same class would be natural for a wide class of applications.

9. Conclusions

It is possible to use techniques from the graph isomorphism literature to compare RDF graphs while equating anonymous resources.

It is not necessary to use some of the more sophisticated techniques suggested, due to the large amount of labelling found in RDF graphs. Performance problems may be experienced if graph theorists use RDF tools to store and communicate pathological examples; but standard usages of RDF are not pathological.

These techniques could be extended to cope with a richer notion of equivalence between resources.

Even if we restrict ourselves to RDF graphs in which all resources are labelled with URIs we will need to use these graph matching techniques to address natural issues of equivalence.

10. References

[1] Berners-Lee, Fielding, Masinter, 1998 *Uniform Resource Identifiers (URI): Generic Syntax* Internet Draft Standard August, IETF, **RFC 2396**.

[2] Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler, 2000, *Extensible Markup Language (XML) 1.0 (Second Edition)*, World Wide Web Consortium, http://www.w3.org/TR/2000/REC-xml-20001006.

[3] Jeremy Carroll 2001 *ModelMatcher.java* found in [7]

[4] Dan Connolly, Jan Grant, Aaron Schwartz et al. 2001, e-mail thread: *Test cases: format of input and output* [*(uri/node/resource/entity too)*] http://lists.w3.org/Archives/Public/w3c-rdfcore-wg/2001May/0188.html

[5] Scott Fortin, 1996, *The Graph Isomorphism Problem*, Technical Report TR 96-20, Department of Computer Science, University of Alberta. ftp://ftp.cs.ualberta.ca/pub/TechReports/1996/TR96-20/TR96-20.ps.gz

[6] Ora Lassila, Ralph R. Swick, 1999, *Resource Description Framework (RDF) Model and Syntax Specification*, World Wide Web Consortium, http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.

[7] *Brian McBride 2001, Jena 1-1-0,* http://www-uk.hpl.hp.com/people/bwm/rdf/jena/jena-1-1-0.zip

[8] Brendan D. McKay 1981 *Practical Graph Isomorphism*, Congressus Numerantium 30, pp45-87. http://cs.anu.edu.au/~bdm/papers/pgi.pdf

[9] Brendan D. McKay 1994 *Nauty* http://cs.anu.edu.au/~bdm/nauty/

[10] Ronald C. Read, Derek G.Corneil, 1977, Graph Isomorphism Disease