Matching RDF Graphs

Jeremy J. Carroll

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

Abstract. The Resource Description Framework (RDF) describes graphs of statements about resources. RDF is a fundamental lower layer of the semantic web. 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. The techniques presented are useful for testing RDF software.

1 Introduction[1]

The semantic web is being built on top of an RDF [1] layer. This paper concerns a technique useful for testing and debugging within that RDF layer.

The RDF specification [1] defines a data model and a syntax. The syntax is defined on top of the XML syntax [2]. The data model is defined in terms of resources, often identified with URIs [3], and literals. Some of the resources are "anonymous". The data model is a set of triples, often thought of as a graph. The anonymous resources correspond to blank nodes in the graph [4].

The processing of RDF graphs occurs in the lower layers of semantic web processing. In practice the correctness of implementations requires the ability to perform unit tests within the RDF layer. The ability to compare two RDF graphs for equality is a fundamental component of such unit tests. For example, the RDF Test Cases Working Draft [5] gives many examples of tests requiring that the graphs read in from two different files should be equal.

Fortunately, the problem of graph equality, usually referred to with the mathematical term "graph isomorphism" is a well-understood one, that is solved for practical use. Less fortunately, the literature is not very accessible. Mathematical texts on graph theory (e.g. [6]) define the concept of graph isomorphism, but do not address the algorithmics. There is an excellent study on the problem from the point of complexity theory [7], again not a practical guide. Graph isomorphism does appear in Skiena's book of algorithms [8] but space considerations only allows a sketch solution. Fortin's technical report [9] gives an in-depth account of algorithms for the graph isomorphism problem.

[1] Thanks to anonymous referees and others who have given valuable feedback on earlier versions of this paper.

I. Horrocks and J. Hendler (Eds.): ISWC 2002, LNCS 2342, pp. 5–15, 2002.
© Springer-Verlag Berlin Heidelberg 2002

A further difficulty presented by RDF graphs is that they do not fit any of the standard graph theoretic categories. They are directed graphs with labelled edges and partially labelled nodes. The partial node labelling is not addressed in prior work.

So, the intended contribution of this paper is as a "how to" guide, for developers of RDF based systems who need to provide a graph equality function, typically for test and debugging purposes.

Graph equality is not usually required or useful for end users, for whom it is believed that inference and entailment are more useful concepts. The model theory of Hayes [4] shows that subgraph isomorphism is the important concept for simple entailment between RDF graphs. This is, of course, a different concept from graph isomorphism. In particular two RDF graphs are semantically equivalent, under Hayes' model theory if they entail one another. This is a weaker condition than that of being isomorphic, which is the condition explored in this paper.

The paper shows how the iterative vertex classification of Read and Corneil [10] (section 6, pp 346-347) is applicable to RDF graphs.

We describe the algorithm and its use within Jena 1-3-0 [11].

2 An Example

If the two data models consist of identical sets of triples then the two data models are equal. This is particular useful for graphs with no blank nodes. However, when there are blank nodes in the RDF graph it is a mistake to limit equality to only such cases.

We explore this with a simple RDF/XML file with anonymous resources:

```
<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>
```

An RDF processor may produce a corresponding set of triples such as:

```
_: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 .
```

The syntax[2] we use for such triples is the “N-triple” syntax being used by the RDF working group [5]. The gensyms such as "_:a9", are identifiers for the blank nodes in the corresponding graph.

With a different gensym algorithm, or by a semantic-free reordering of the XML input, the same processor may give a different set of triples:

```
_: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 .
```

A naive 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 is not consistent with the intended reading of anonymous resources being like resources but without a name. Nor is it consistent with either the N-triple definition [5], and the newer RDF Model Theory [4]. Both are clear that the blank node identifiers have file scope, and such cross-file comparisons are inappropriate. Indeed, the abstract syntax for RDF is a graph from which the blank node identifiers have been erased, thus no identifier of a blank node is significant. So the graph isomorphism problem, in this example, amounts to finding the bijection between the blank node identifiers that makes the two sets of triples equivalent. The bijection being:

```
a1 → a1.
a6 → a3.
a9 → a6.
a3 → a9.
```

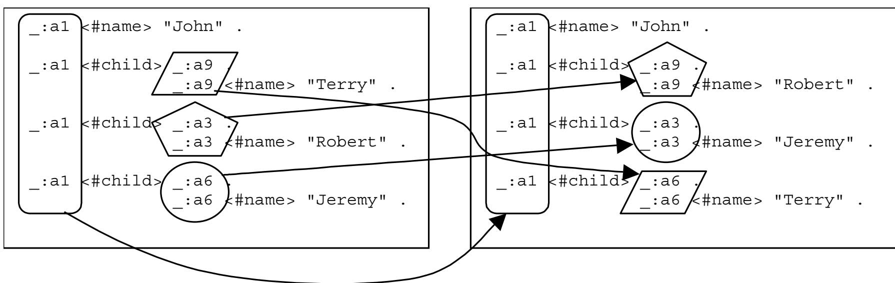

Fig. 1. An equivalence mapping between blank nodes

In very small examples, such as this one, it is plausible that a brute force search over all such permutations of anonymous resources will suffice. This has a factorial

[2] We use relative fragment URIs for compactness; these are not legal N-triple.

complexity and even with a dozen anonymous nodes ceases to provide the interactive feedback that is useful in debugging and testing.

The bijection between the two sets of blank nodes induces a labelled digraph isomorphism.

3 Graph Isomorphism Theory

In the graph isomorphism literature (e.g. [9], [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?"

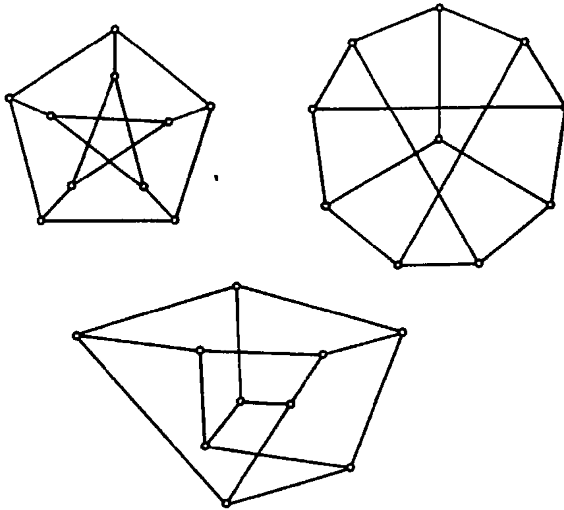

Fig. 2. Isomorphic graphs from [10].

Figure 2 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 data 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 data model equality and the graph isomorphism problem are equivalent from a theoretical point of view. However, in practice RDF data 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.

4 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 V_2 of G_2.
3. If $|V_1|=|V_2|$ set $n = |V_1|$ 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: brute force search over all the permutations of the vertices. 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 that is also 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.

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 V_2 of G_2.
3. If $|V_1|=|V_2|$ set $n = |V_1|$ 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. [12]) 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.

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 classification 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 classification and go to 5(a)i
   b. If every class has one element from each graph then this defines an isomorphism and we are finished.
   c. Choose a smallest class with more than two vertices. Select an arbitrary vertex from V_1 in this class. (Non-deterministically) guess a vertex from V_2 in this class, hence picking a pair of vertices; when we run out of guesses, we backtrack to the last guess.
   d. Generate a new classification from the current classification by putting the pair of vertices, selected and 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 [12], [13]; typically they use more sophisticated invariants than the adjacency one described here, and they use the 'automorphism group' of one of the graphs to eliminate many redundant guesses. However, for RDF graphs the above algorithm will generally be sufficient.

5 Vertex Classification for RDF

The code found in Jena [14] 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[3]. 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 [14] 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.

6 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).

---

[3] A minor variation is that an emphasis is placed on finding singleton classes.

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.

7 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 many applications.

A further natural equivalence between RDF graphs is given by the model theory [4]. Here the relevant notion of equivalence is, "do the two graphs entail one another?" This is a weaker condition than graph isomorphism, and the techniques described here are not suitable for this problem.

8 Use of Graph Isomorphism within Jena

As indicated in the introduction, the primary motivation for graph equality testing is for unit testing and debugging of underlying RDF infrastructure. Thus the major use of this code in Jena is in testing code:

- It is used by the RDF Core WG to apply the test cases to a suite of RDF/XML parsers [15]. This involves using a parser to convert RDF/XML into N-triple and then comparing this N-triple document with a reference N-triple document using graph isomorphism.
  It is used within the unit test code for ARP to check its conformance with the RDF Test Cases. This involves loading an RDF/XML file with the parser as one graph and loading a second graph from the reference N-triple document, and comparing the two graphs for isomorphism.
- It is used for additional parser tests within Jena.
- It is used for RDF writer tests of the form:
  - take an RDF graph
  - write it out
  - read it in as a new graph
  - compare new with old, if they are not the same then there is an error.

Further use can be made whenever operations within an RDF platform are meant to leave a graph unchanged.

The implementation within Jena is itself tested using some pathological cases based on slightly distorted unlabelled hypercubes (both directed and undirected). Unlabelled graphs are represented within RDF by:

- always using a label `rdf:value` for every edge
- always using blank nodes

Undirected edges are represented by using two directed edges (one in each direction).

As an example consider the 3-dimensional directed hypercube below (the vertex labels are only part of the diagram, not part of the graph):

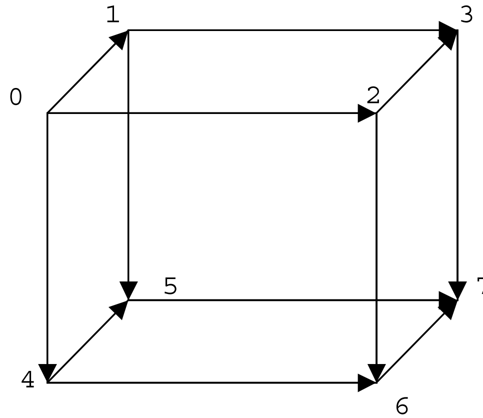

Fig. 3. A directed hypercube

We can distort this by duplicating a vertex:

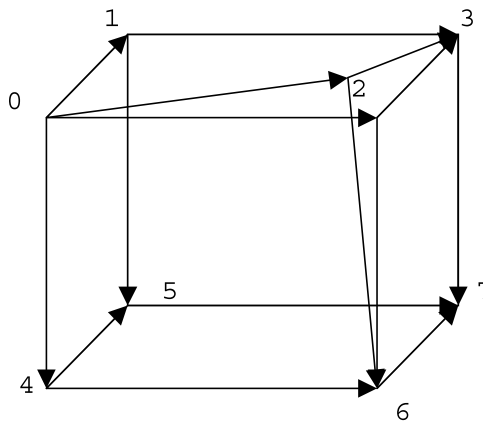

Fig. 4. A distorted directed hypercube

Before this distortion vertices 1, 2, and 4 were in the same class. After this distortion, the new vertex is indistinguishable from vertex 2, and the distinction between vertex 2 and vertex 4 is quite subtle.

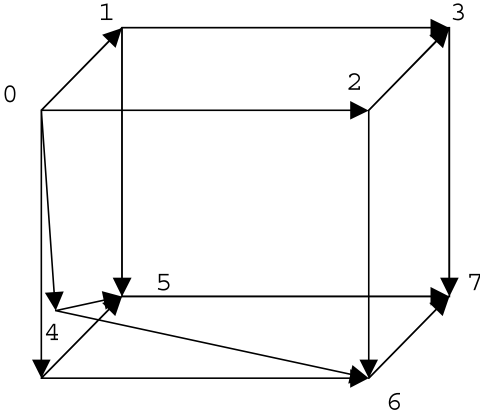

Fig. 5. An isomorphic distorted directed hypercube

Moreover, given two graphs differently distorted in this fashion, we know that they are isomorphic if and only if the number of bits in the (informal) node label of the duplicated nodes is the same. Compare figure 4 with figure 5 and figure 6.

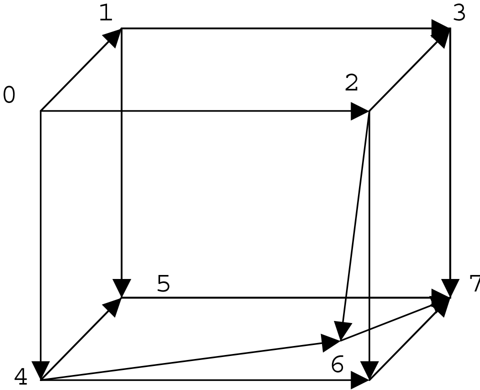

Fig. 6. A non-isomorphic distorted directed hypercube.

Thus we can produce a number of different, moderately difficult test cases for graph isomorphism, for which the correct result (isomorphic or not) is known.

Working on 8 dimensional hypercubes with 256 vertices each test takes less than a second on an off-the-shelf PC and Java 1.3. (The measured results are between 75 and 670 milliseconds, depending on the exact details of the deformity).

Since realistic uses of this functionality involve RDF graphs for which the variation in edge and node labels is much greater, resulting in a much better first vertex classification, the algorithm performs adequately for its intended purpose.

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.

References

1. Ora Lassila, Ralph R. Swick: Resource Description Framework (RDF) Model and Syntax Specification. World Wide Web Consortium (1999) http://www.w3.org/TR/1999/REC-rdf-syntax-19990222.
2. Tim Bray, Jean Paoli, C.M. Sperberg-McQueen, Eve Maler: Extensible Markup Language (XML) 1.0 (Second Edition). World Wide Web Consortium (2000) http://www.w3.org/TR/2000/REC-xml-20001006.
3. T. Berners-Lee, R. Fielding, U.C. Irvine, L. Masinter: Uniform Resource Identifiers (URI): Generic Syntax. RFC 2396. IETF (1998).
4. Patrick Hayes (ed): RDF Model Theory. W3C Working Draft (2001) http://www.w3.org/TR/rdf-mt/
5. Art Barstow, Dave Beckett (eds): RDF Test Cases. W3C Working Draft (2001) http://www.w3.org/TR/rdf-testcases/
6. Reinhard Diestel: Graph Theory. 2nd edition, Springer (2000).
7. J. Köbler, U Schöning, J. Torán: The Graph Isomorphism Problem, Its Structural Complexity. Birkhauser (1993).
8. Steve Skiena: The Algorithm Design Manual. Springer (1998).
9. Scott Fortin: 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
10. Ronald C. Read, Derek G.Corneil: Graph Isomorphism Disease. Journal of Graph Theory 1 (1977) 339-363.
11. Brian McBride, Jeremy Carroll, Ian Dickenson, Dave Reynolds, Andy Seaborne: Jena 1-3-0. Hewlett-Packard Labs (2002) http://www.hpl.hp.com/semweb/jena-top.html
12. Brendan D. McKay: Practical Graph Isomorphism. Congressus Numerantium 30 (1981) 45-87. http://cs.anu.edu.au/~bdm/papers/pgi.pdf
13. Brendan D. McKay: Nauty. (1994) http://cs.anu.edu.au/~bdm/nauty/
14. Jeremy Carroll: ModelMatcher.java found in [11] (2001)
15. Aaron Swarz: Test Case Results. http://lists.w3.org/Archives/Public/w3c-rdfcore-wg/2001Sep/0001.html