Skolemising Blank Nodes while Preserving Isomorphism

Aidan Hogan [*]
Department of Computer Science
University of Chile
ahogan@dcc.uchile.cl

ABSTRACT

In this paper, we propose and evaluate a scheme to produce canonical labels for blank nodes in RDF graphs. These labels can be used as the basis for a Skolemisation scheme that gets rid of the blank nodes in an RDF graph by mapping them to globally canonical IRIs. Assuming no hash collisions, the scheme guarantees that two Skolemised graphs will be equal if and only if the two input graphs are isomorphic. Although the proposed scheme is exponential in the worst case, we claim that such cases are unlikely to be encountered in practice. To support these claims, we present the results of applying our Skolemisation scheme over a diverse collection of 43.5 million real-world RDF graphs (BTC–2014); we also provide results for some nasty synthetic cases.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; G.2.2 [**Discrete Mathematics**]: Graph Theory—*Graph labeling*

Keywords

RDF; blank nodes; Skolemisation; isomorphism

1. INTRODUCTION

Blank nodes have been a thorny issue in the Semantic Web standards for well over a decade [10]. In the original 1999 specification of RDF [14], blank nodes (then simply called "anonymous nodes") made the assignment of URIs to nodes in an RDF graph optional. This enabled, e.g., convenient shortcuts for containers and reification in the RDF/XML syntax of the time, allowing parsers to generate anonymous nodes where necessary when reconstructing the RDF graph.

Blank nodes were later formalised as local existential variables in the 2004 RDF Semantics specification [7]. Unlike IRIs or literals, they do not point to something specific in the world described. Rather they denote the *existence* of something. Thus one can, for example, perform a one-to-one relabelling of the blank nodes in an RDF graph without affecting its meaning: two RDF graphs that are the same modulo a one-to-one rewriting of blank nodes are called *isomorphic*. Existential blank nodes also give rise to other semantic conditions involving simple entailment (checking if one RDF graph permits a subset of the interpretations of another RDF graph), simple equivalence (checking if two RDF graphs permit the same interpretations), leanness (checking if an RDF graph contains redundancy), and so forth [8, 6].

Blank nodes thus add theoretical complexity to RDF. No polynomial-time algorithms are known for deciding whether or not two RDF graphs are isomorphic. Likewise, checking simple entailment, simple equivalence and leanness of RDF graphs have all been proven to be intractable specifically due to the presence of existential blank nodes [6].

Blank nodes also introduce practical problems when dealing with RDF [10]. Although theoretical worst-case scenarios are unlikely to be encountered in real-world data [10], algorithms and tools handling RDF must still either account for all such cases or apply a pragmatic but non-standard fudge with respect to blank nodes. Fundamental operations, like checking what changed between two versions of an RDF graph [21], are often greatly complicated by blank nodes. In SPARQL, blank nodes cannot be directly referenced by a query, making them difficult to get data about [10]. In Linked Data, blank nodes cannot be externally linked and their use is thus discouraged [9]. And so forth [10].

Although blank nodes add complexity for consumers, their convenience for publishers means that they are widespread in real-world data. Our recent survey [10] found that in the BTC–2012 corpus – a crawl of 8.4 million RDF documents from the Web – 44.9% of the documents mentioned at least one blank node, 25.9% of the unique RDF terms were blank nodes, and 66.2% of pay-level-domains used blank nodes. Hence even if blank nodes were to be deprecated, the question of what to do with legacy data would remain open.

Ten years on from the previous release, an updated set of RDF 1.1 W3C Recommendations were recently published. The issue of blank nodes was a core topic within the working group. Since blank nodes have been in widespread use for

[*]This work was supported by the Millennium Nucleus Center for Semantic Web Research under Grant No. NC120004 and by Fondecyt Grant No. 11140900. The author would also like to thank Marcelo Arenas and Carsten Lutz for their valuable input as well as the anonymous referees whose detailed comments helped improve the paper.

WWW 2015, May 18–22, 2015, Florence, Italy.
ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741653.

over a decade, major changes would have lead to too drastic a cost in terms of current adoption [10]. Instead, for scenarios where blank nodes are considered undesirable, RDF 1.1 now endorses an abstract scheme whereby blank nodes can be replaced with "fresh" IRIs, called *Skolem IRIs* [5, § 3.5].

The name "Skolem" refers to the process of *Skolemisation* whereby existential variables appearing in a first-order formula can be replaced with a function whose function symbol is "fresh" and whose arguments include the terms upon which the existential variable (loosely speaking) depends. If the variable being replaced does not depend on another term – as is the case in RDF – it can be replaced by a constant (a zero-arity function). Although Skolemisation can change the possible interpretations of the formula, it does not affect the satisfiability of the formulae in question.[1]

Likewise, RDF 1.1 systems creating Skolem IRIs "SHOULD *mint a new, globally unique IRI for each blank node so replaced*" [5].[2] Using this process Skolemising an RDF graph does not affect its satisfiability. However, guaranteeing *equi-satisfiability* – that the Skolemised RDF graph is satisfiable if and only if the input graph is satisfiable – is vacuous since all RDF graphs are satisfiable until semantic conditions for datatypes or built-in vocabularies like RDFS or OWL are considered [8]. Skolemisation is rather a practical heuristic rubber-stamped by the recommendation for use in scenarios where blank nodes would otherwise be problematic.

However, the Skolemised versions of two isomorphic RDF graphs (or two copies of the same graph) are no longer isomorphic, nor do they entail each other. This means that it could not be known if two Skolemised graphs originated from isomorphic RDF graphs or even the same RDF graph.[3]

In this paper, we propose a method for producing a canonical labelling of blank nodes in an RDF graph that preserves isomorphism. We foresee two main use-cases for this scheme:

1. checking the isomorphism of RDF graphs or identifying groups of isomorphic RDF graphs from a large collection without requiring pair-wise isomorphism checks;
2. Skolemising RDF graphs such that the output graphs are equal if and only if the input graphs are isomorphic.

This paper continues with some preliminaries on RDF isomorphism. We then introduce a cheap but naive algorithm for computing a canonical labelling. Thereafter, we present illustrative cases where the naive algorithm fails and introduce a complete but exponential solution inspired by an existing canonical labelling algorithm that we adapt for RDF. We then address some issues relating to the generation of Skolem IRIs. Later evaluating our approach, we show that although the proposed scheme is efficient over a large collection of real-world RDF graphs, it can – as expected for an exponential algorithm – struggle for some modestly-sized synthetic cases. We wrap-up the contribution with discussion of related work and a brief conclusion.

[1]Which is used in FOL, e.g., to simplify theorem proving.

[2]In fact, minting globally unique IRIs is an unfeasible guarantee to make since any Web location may mention any IRI, but one can at least mint IRIs that are almost certainly unique, at least at the time of creation.

[3]One option would be to convert Skolem IRIs back to blank nodes, but this is problematic since there is nothing to restrict using Skolem-like IRIs in the input graph.

2. BACKGROUND

We first provide some formal preliminaries for RDF graphs, with a particular focus on blank nodes and the problem of checking if two RDF graphs are isomorphic.

Definition 1. Let $\mathbf{I}$ be the set of *IRIs*, $\mathbf{B}$ the set of *blank nodes*, and $\mathbf{L}$ the set of *literals*. The sets $\mathbf{I}$, $\mathbf{B}$, and $\mathbf{L}$ are infinite and pair-wise disjoint; collectively we call them *RDF terms*. An *RDF triple* $t := (s, p, o)$ is a member of the set $(\mathbf{I} \cup \mathbf{B}) \times \mathbf{I} \times (\mathbf{I} \cup \mathbf{B} \cup \mathbf{L})$; s is called the *subject*, p the *predicate* and o the *object* of t. An *RDF graph* G is a finite set of RDF triples. We denote by $\text{terms}(G)$ the set of all RDF terms in some triple of G. An RDF graph G is *ground* if $\text{terms}(G)$ contains no blank nodes.

In order to understand appropriate solutions to get rid of blank nodes from RDF graphs, we need to (perhaps unfortunately) first understand them in some detail. We first define two operations on the blank nodes in an RDF graph.

Definition 2. Let μ denote a mapping from RDF terms to RDF terms that is the identity on IRIs and literals. We call μ a *blank node mapping.* Slightly abusing notation, given a triple $t := (s, p, o)$, we let $\mu(t) := (\mu(s), \mu(p), \mu(o))$, and given an RDF graph G, we let $\mu(G) := \{\mu(t) \mid t \in G\}$. If μ is restricted to map from blank nodes to blank nodes in a one-to-one manner, we call it a *blank node bijection.*

Definition 3. Two RDF graphs G and H are *isomorphic*, denoted $G \cong H$, if and only if there exists a blank node bijection μ such that $\mu(G) = H$, in which case we call μ an *isomorphism* between G and H.

Two isomorphic RDF graphs can be intuitively considered as containing the same "structure" [5]. If G and H are both ground, then $G \cong H$ if and only if $G = H$. Next, an *RDF merge*, denoted $G + H$, uses isomorphism to avoid blank node clashes when combining RDF graphs.

Definition 4. An *RDF merge* of two RDF graphs, denoted $G + H$, is defined as $G' \cup H'$ where G' and H' are isomorphic copies of G and H resp. that share no blank nodes.

The RDF 1.1 Semantics recommendation [8] formalises the meaning of RDF graphs in terms of their *interpretations* (the possible worlds that they describe). The notion of *simple interpretations*, which consider existential blank nodes but no built-in vocabulary, gives rise to semantic concepts such as simple entailment, simple equivalence and leanness.

It is known from works by, e.g., Gutierrez et al. [6], that the problem of checking simple entailment/equivalence of RDF graphs is NP-COMPLETE, and that the problem of checking if G is lean is coNP-COMPLETE.

Regarding isomorphism, it is folklore – hinted at by, e.g., Carroll [4] – that deciding RDF isomorphism is of the same complexity as the standard GRAPH ISOMORPHISM problem[4] and hence is GI-COMPLETE.[5] However, to the best of our knowledge, the result has not been formally stated or proved. Though the GI-HARDNESS of RDF isomorphism is quite trivial to prove (since one can easily encode the structure of a

[4]Herein we call "simple graphs" "standard graphs" to avoid a naming clash with the simple semantics of RDF.

[5]GRAPH-ISOMORPHISM–COMPLETE: problems in this class are in NP but are not known to be NP-COMPLETE nor to have polynomial-time algorithms.

standard graph in RDF by representing each undirected edge $a - b$ as two RDF triples $a \overset{p}{\leftrightarrows} b$ for some arbitrary p), showing that it is *within* GI appears non-immediate. Hence we now formalise this result.

THEOREM 1. *Deciding if $G \cong H$ is GI-complete.*

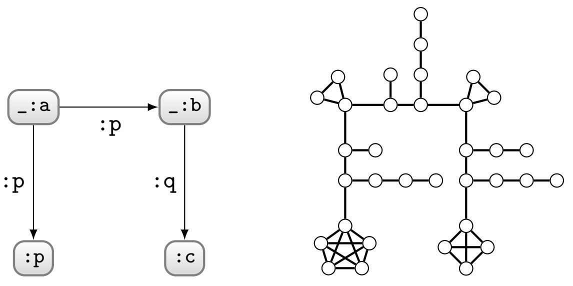
PROOF. The proof relies on a polynomial-time many-one reduction both to and from the GI-COMPLETE problem of deciding GRAPH ISOMORPHISM for standard graphs.

The reduction from standard graph isomorphism to RDF graph isomorphism can be done straightforwardly by encoding the standard graph as an RDF graph as previously discussed. Hence RDF graph isomorphism is GI-HARD.

The reduction from RDF graph isomorphism to standard graph isomorphism is more tricky. We need a polynomial-time method to encode two RDF graphs G and H as standard graphs enc(G) and enc(H) such that $G \cong H$ (under RDF isomorphism) if and only if enc(G) $\cong$ enc(H) (under standard isomorphism). We now provide an example of one such encoding mechanism that satisfies the proof.

First, let SO denote the set of literals and IRIs that appear in the subject and/or object position of some triple in G and let P denote the set of IRIs appearing as a predicate in G. We assume that the sets SO and P correspond for both G and H, otherwise isomorphism can be trivially rejected.

We first give an example input and output. The encoding scheme we propose would encode the RDF graph on the left as the standard graph on the right (we keep the intuitive "shape" of both graphs intact for reference).



We start with an empty standard graph G into which we will encode the RDF graph G. The encoding adds cliques and path graphs to G to represent nodes and edges in the original RDF graph. Each clique and path graph has a fixed *external node* chosen to connect it to other parts of G; for path graphs, the external node is a terminal node.

First add a fresh 3-clique to G for every blank node in G. Define a total ordering $\leq$ over SO (e.g., lexical) such that for all $x \in SO$, we can compute sorank(x) := card$\{y \mid y \in SO$ and $y \leq x\}$. In the example above, taking a lexical ordering, sorank(`:c`) = 1 and sorank(`:p`) = 2. Add a fresh (sorank(x) + 3)-clique to G for each term $x \in SO$.

Next we must encode the RDF edges, capturing labels and directionality. Again define a total ordering over P and let prank(p) denote the rank of p in P such that prank(p) := card$\{q \mid q \in P$ and $q \leq p\}$ (e.g., with a lexical ordering, prank(`:p`) = 1, prank(`:q`) = 2). For each $(s, p, o) \in G$, add two fresh path graphs of length prank(p) + 1 and $|P| + 1$ to G and connect their external nodes. Connect the external nodes of the clique of s and the short path, and connect the external nodes for the clique of o and the long path.

Once all triples are processed, the encoding is completed in time polynomial to the size of G.[6] Letting G and H denote the graphs encoded from G and H – where we again assume that SO and P are fixed for G and H – we now argue why $G \cong H$ (under RDF isomorphism) if and only if $\mathsf{G} \cong \mathsf{H}$ (under standard isomorphism).

$G \cong H$ implies $\mathsf{G} \cong \mathsf{H}$ since if $G \cong H$, then G and H differ only in blank node labels, which the encoding ignores.

To show that $\mathsf{G} \cong \mathsf{H}$ implies $G \cong H$, we show that each RDF graph has an encoding that's unique up to homomorphism. In particular, G can be decoded back to a G' such that $G \cong G'$. The decoding relies on two main observations:

1. No (≥ 3)-cliques can be unintentionally created in the encoding other than as part of a bigger clique for another vocabulary term. Nor can such a clique grow. Maximal cliques can thus be detected in G and mapped back to fresh blank nodes or to terms in SO.

2. No nodes with a degree of 1 can be introduced other than as the terminals of the paths used to encode RDF edges. Such nodes can thus be detected and walked back (to the node with degree 3) to decode the predicate in P and the direction.

This unambiguous decoding shows that the encoding (w.r.t. P and SO) is one-to-one modulo isomorphism. Hence two non-isomorphic RDF graphs cannot be encoded to isomorphic standard graphs under fixed vocabulary.[7]

This encoding thus completes the proof of Theorem 1. $\square$

Finding a polynomial-time algorithm for RDF isomorphism is unlikely: it would imply GI = P, where all known algorithms for GRAPH ISOMORPHISM are exponential. However, isomorphism can be efficiently computed for many graphs; e.g., Babai showed that isomorphism for the vast majority of randomly generated graphs can be performed efficiently using a naïve algorithm [1]. Hence, despite exponential worst cases, many practical algorithm exist.

One of the most famous graph isomorphism algorithms – due to McKay [17] – is called NAUTY. This algorithm derives a canonical labelling Can such that for two standard graphs G and H, Can(G) = Can(H) if and only if $\mathsf{G} \cong \mathsf{H}$. In fact, this is quite close to our own use-case, where we wish to find a canonical mapping from blank nodes to IRIs.

Definition 5. We call a blank node mapping μ a *blank node grounding* if it maps blank nodes exclusively to IRIs.

One of our main goals herein is to find a blank node grounding μ such that $\mu(G) = \mu(H)$ if and only if $G \cong H$. When computing $\mu(G)$, only information from G is used; nothing is known from H. Our problem is similar to that of canonically labelling standard graphs, where our proposals are inspired by NAUTY. However, the presence of directed edges and ground labels (IRIs and literals) on both vertexes and edges in RDF suggests the need for custom methods.

[6]Expressing the exact size of G w.r.t. G is trivial but long. It suffices to observe that the number of nodes added to G has a (loose) upper-bound of $(so+3)^2 + 3b + g(2p+3)$ where $so := |SO|$, $b := |\text{terms}(G) \cap \mathbf{B}|$, $g := |G|$ and $p := |P|$. The number of edges added to G is bounded by the square of this number, and is thus bounded polynomially by $|G|$.

[7]Note: the decoding is not part of the reduction. In particular, listing all maximal cliques may not be possible in polynomial time. However, our goal is to merely demonstrate the *existence* of an unambiguous decoding.

3. NAIVE ALGORITHM

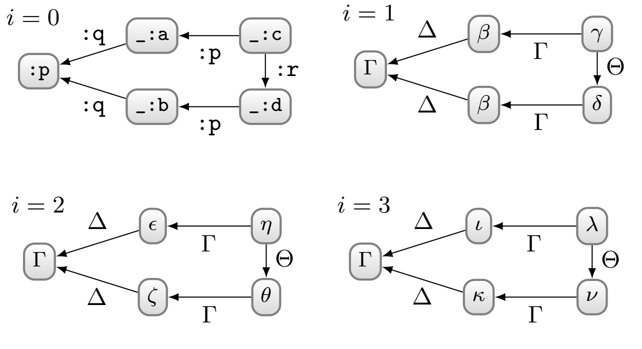
To generate a canonical labelling of a standard graph, Nauty depends on *invariants* that must be preserved by isomorphism. An example is node degree: an isomorphism can only map two nodes with the same degree. Such invariants thus help to narrow the search. In RDF, ground terms offer a rich invariant. Our first step is thus to *colour* the blank nodes in an RDF graph with the ground information surrounding them such that blank nodes of different colours cannot be mapped to one another by an isomorphism.

In Algorithm 1, we propose a hashing scheme for colouring the blank nodes. A colour map is initialised in lines 2–5; IRIs and literals are assigned static hash-based colours, whereas blank nodes hashes are computed iteratively later. For now, we assume one-to-one perfect hashing (practical hashing issues will be discussed later). Lines 9–18 iteratively compute the blank node colours, where $hashTuple(\cdot)$ is order dependant and $hashBag(\cdot)$ is commutative and associative. The `‘+’` and `‘-’` symbols distinguish edge direction. The computed colours form a partition of blank nodes. The loop terminates when the partition does not change in an iteration (though the colour values may have changed).

Algorithm 1 Colouring blank nodes

```
 1: function COLOUR($G$)                          ▷ $G$ an RDF graph
 2:     initialise $\mathsf{clr}_0[]$              ▷ a map from terms to colours
 3:     for $x \in \text{terms}(G)$ do            ▷ all terms in $G$
 4:         if $x \in \mathbf{B}$ then
 5:             $\mathsf{clr}_0[x] \leftarrow 0$   ▷ initial colour
 6:         else
 7:             $\mathsf{clr}_0[x] \leftarrow hashTerm(x)$   ▷ static colour
 8:     $i \leftarrow 0$
 9:     repeat
10:         $i$++
11:         initialise $\mathsf{clr}_i[]$ with $\mathsf{clr}_{i-1}[]$   ▷ copy map
12:         for $(b,p,o) \in G : b \in \mathbf{B}$ do   ▷ $o \in \mathbf{IBL}$
13:             $c \leftarrow hashTuple(\mathsf{clr}_{i-1}[o], \mathsf{clr}_{i-1}[p], \text{‘+’})$
14:             $\mathsf{clr}_i[b] \leftarrow hashBag(c, \mathsf{clr}_i[b])$
15:         for $(s,p,b) \in G : b \in \mathbf{B}$ do   ▷ $s \in \mathbf{IB}$
16:             $c \leftarrow hashTuple(\mathsf{clr}_{i-1}[s], \mathsf{clr}_{i-1}[p], \text{‘-’})$
17:             $\mathsf{clr}_i[b] \leftarrow hashBag(c, \mathsf{clr}_i[b])$
18:     until $\forall x,y : \mathsf{clr}_i[x] = \mathsf{clr}_i[y] \Leftrightarrow \mathsf{clr}_{i-1}[x] = \mathsf{clr}_{i-1}[y]$
19:     return $\mathsf{clr}_i[]$ as Clr          ▷ final map of terms to colours
```

Example 1. *We now exemplify how Algorithm 1 works. Blank nodes are "coloured" with hashes, but here we illustrate the process using Greek letters (upper-case for static hashes, lower-case for dynamic hashes). The iteration is given by i.*



In the initial state ($i = 0$), assume that an initial hash α is assigned to all blank nodes and static hashes to all IRIs and literals. In the iterations that follow, blank nodes are coloured according to their neighbourhood; e.g., for `_:d` at $i = 1$, the hash is computed as:

$$\begin{aligned}\mathsf{clr}_1[\texttt{\_:d}] \leftarrow hashBag\big(&\mathsf{clr}_0[\texttt{\_:d}],\\ &hashTuple(\mathsf{clr}_0[\texttt{\_:b}], \mathsf{clr}_0[\texttt{:p}], \text{‘+’}),\\ &hashTuple(\mathsf{clr}_0[\texttt{\_:c}], \mathsf{clr}_0[\texttt{:r}], \text{‘-’})\big)\end{aligned}$$

All blank nodes are distinguished by $i = 2$. At $i = 3$, we see that for each pair of blank nodes (x, y), $\mathsf{clr}_2[x] = \mathsf{clr}_2[y]$ if and only if $\mathsf{clr}_3[x] = \mathsf{clr}_3[y]$; the process thus terminates.

In fact, in the case of Example 1, the last iteration is unnecessary since once the unit partition is reached (all blank nodes are distinguished), we can end the process knowing no new blank nodes can be distinguished thereafter. Since this case is quite common, we apply this early termination heuristic in the implementation of the colouring method.

Algorithmic characteristics.

We now show that in the general case, the algorithm terminates in a bounded number of iterations, and that the computed colours preserve isomorphism.

Lemma 1. *Assuming a perfect hashing scheme, in Algorithm 1, if $\mathsf{clr}_i[x] \neq \mathsf{clr}_i[y]$, then $\mathsf{clr}_j[x] \neq \mathsf{clr}_j[y]$ for $j \geq i$.*

Proof. (*Sketch*) Can be proven by induction observing that if $i < j$, then $\mathsf{clr}_i[x]$ is encoded in the hash of $\mathsf{clr}_j[x]$. □

Theorem 2. *Let $B := \mathbf{B} \cap \text{terms}(G)$ denote the set of blank nodes mentioned in an RDF graph G. Algorithm 1 terminates in $\Theta(|B|)$ iterations for G in the worst case.*

Proof. Colours form a partition of B. Algorithm 1 terminates if the partition doesn't change. Per Lemma 1, partitions can only split. Hence only $|B| - 1$ splits can occur before the unit partition is reached. The tight asymptotic bound is given, for example, by the $\lfloor \frac{1}{2} \cdot |B| \rfloor + 1$ iterations needed for a regular path graph. □

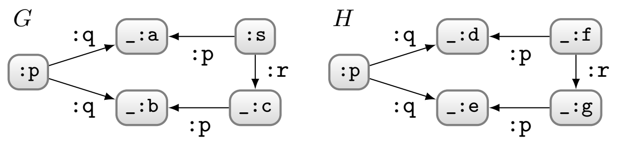
Letting T denote terms(G), and assuming for simplicity that clr has constant insert/lookup and linear copy performance, then Algorithm 1 runs in $\Theta(|B| \cdot (|T| + |G|))$ in a worst-case analysis. In terms of space, the input graph and two colour maps of size T are stored. Ground triples in G can be pre-filtered to reduce space and time.

Theorem 3. *Let Clr_G denote the map from RDF terms to colours produced by Algorithm 1 for G. Let $G \cong H$ be two isomorphic RDF graphs. Let b be a blank node of G and c be a blank node of H. If there exists a blank node bijection μ such that $\mu(G) = H$ and $\mu(b) = c$, then $\text{Clr}_G(b) = \text{Clr}_H(c)$.*

Proof. (*Sketch*) The main observations are that (i) Algorithm 1 is agnostic to lexical blank node labels, (ii) the commutativity and associativity of $hashBag(\cdot)$ ensures order-independent hashing over sets of triples, and (iii) the number of iterations is deterministic modulo isomorphism. □

A nuance of the algorithm is that when there are disconnected sub-graphs of blank nodes in the input RDF graph, the colouring of some of those sub-graphs may continue longer than necessary. Specifically, let's say we use Clr_G as the basis of a blank node grounding μ_G, where output colours would be used to mint fresh IRIs. Given two RDF graphs G and H, then $\mu_G(G)$ may not equal $\mu_{G+H}(G)$; alternatively, $\mu_G(G)$ may not be a subset of $\mu_{G+H}(G + H)$.

Example 2. *In the following, G will require one fewer iteration than H or $G+H$ to terminate. Hence the colourings of graph G would no longer correspond with the colourings of the corresponding sub-graph of $G+H$.*



Since this may be undesirable in certain applications, we propose an optional step prior to Algorithm 1 that breaks up the input RDF graph according to its blank nodes.

Definition 6. For an RDF graph G, let $\mathcal{G} := (\mathcal{V}, \mathcal{E})$ denote a standard undirected graph where $\mathcal{V} = G$ and $\mathcal{E}$ is a set of unordered pairs of triples from G such that $\{t, u\}$ is in $\mathcal{E}$ if and only if $\{t, u\} \subseteq G$ and $\text{terms}(\{t\}) \cap \text{terms}(\{u\}) \cap \mathbf{B} \neq \emptyset$. Let $t \sim_{\mathcal{G}} v$ denote that t and v are reachable from each other in $\mathcal{G}$, and let $G/{\sim_{\mathcal{G}}}$ denote the partition of G based on reachability in $\mathcal{G}$. We define a *blank node split* of G as $\text{split}(G) := \{G' \in G/{\sim_{\mathcal{G}}} \mid G' \text{ is not ground}\}$.

The *blank node split* of G contains a set of non-overlapping subgraphs of G, where each subgraph G' contains all and only the triples for a given group of "connected blank-nodes" in G; e.g., in Example 2, $\text{split}(G+H) = \{G, H\}$. Now, instead of passing a raw RDF graph G to Algorithm 1, to ensure that $\mu_G(G)$ is a subset of $\mu_{G+H}(G+H)$, we can pass the individual split sub-graphs to Algorithm 1. This process is outlined in Algorithm 2. On line 2, we compress the details of computing $\text{split}(G)$: a standard Union–Find algorithm with $O(n \cdot \log n)$ runtime suffices. The results of each split are computed and joined; a regular join operation suffices since two splits cannot disagree on the colour of a given term: colours for IRIs and literals are static and no blank node can appear in two splits. Versus Algorithm 1, Algorithm 2 may produce different colours for blank nodes due to running fewer iterations over split graphs (per Example 2) and may miss colours for ground terms (we are only interested in blank node colours). The added cost of the split computation is offset by the potential to parallelise and reduce iterations when colouring individual sub-graphs.

Algorithm 2 Colouring split graphs

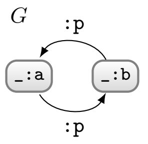
```
1: function COLOUR'($G$)                       ▷ $G$ a non-ground RDF graph
2:     $\{G_1, \ldots, G_n\} \leftarrow \text{split}(G)$         ▷ use, e.g., Union–Find
3:     Clr ← COLOUR($G_1$)                       ▷ calls Algorithm 1
4:     for $1 < i \leq n$ do
5:         Clr ← Clr ⋈ COLOUR($G_2$)             ▷ ⋈ indicates a join
6:     return Clr                                ▷ final colours
```

4. COMPLETE ALGORITHM

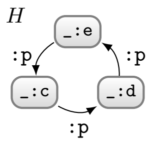
For many real-world RDF graphs, the algorithm described in the previous section would probably suffice to compute distinct colours/labels for all blank nodes. However, the naive algorithm is tractable and we would expect a complete algorithm to be exponential (since it could solve a GI-complete problem per Theorem 1). Indeed, there are various cases in which the naive approach fails; e.g., non-trivial *automorphisms* cause problems for the naive approach.

Definition 7. An *automorphism* of an RDF graph G is an isomorphism from G to itself; i.e., μ is an automorphism of G if $\mu(G) = G$. If μ is not the identity mapping on blank nodes in G then μ is a *non-trivial automorphism.* We denote the set of all automorphisms of G by $\text{Aut}(G)$.

Example 3. *We give the automorphisms for two RDF graphs. Trivial automorphisms are in grey. Applying any of the automorphisms shown for the graph in question would lead to the same graph (and not just an isomorphic copy).*



Aut(G)		
$\mu(\cdot)$	_:a	_:b
=	_:a	_:b
	_:b	_:a



Aut(H)			
$\mu(\cdot)$	_:c	_:d	_:e
=	_:c	_:d	_:e
	_:d	_:e	_:c
	_:e	_:c	_:d

If we input G into our naive colouring algorithm, the process would terminate after two iterations and no blank nodes will be distinguished by colour. The same holds for H. This is due to the presence of automorphisms that make G and H vertex transitive *(in an RDF sense): for any two vertexes u and v, there exists an automorphism μ such that $\mu(u) = v$.*

In fact, the blank nodes of G and H would both have the same global colour. To see this, imagine that a blank node is coloured blue in $i = 1$ if it is grey in $i = 0$ and has both an inlink and an outlink with label `:p` *to a grey blank node in $i = 0$; likewise, a blank node is coloured green in $i = 2$ if it is blue in $i = 1$ and has both an inlink and an outlink with label* `:p` *to a blue blank node in $i = 1$. All blank nodes in G and H – or any directed cycle with that predicate – would be green in the result. If we were to use the naive colouring to ground the triples in $G+H$, the result would be single triple: $(g, \texttt{:p}, g)$ for g the IRI generated from green.*

Thus the naive approach can be problematic in certain (not-so-exotic) cases. To overcome these problems, we first define a sound canonical version of an RDF graph.

Canonicalising RDF graphs.

Assume a total ordering of all RDF terms (e.g., lexical), of all RDF triples (e.g., lexicographic) and of all RDF graphs (e.g., $G < H$ if and only if $G \subset H$ or there exists a triple $t \in G \setminus H$ such that no triple $t' \in H \setminus G$ exists where $t' < t$). Assume that κ is a blank node bijection that labels all k blank nodes in a graph G from `:b1` to `:b`k.[8] We denote by $\mathcal{K}$ the $k!$ possible κ-mappings for G. Since we have a total ordering of graphs, there is a unique graph $\min\{\kappa(G) \mid \kappa \in \mathcal{K}\}$, which we denote by $\lfloor G \rfloor$. We now show that $\lfloor G \rfloor$ is a canonical graph with respect to isomorphism.

Lemma 2. $G \cong H$ if *and* only if $\lfloor G \rfloor = \lfloor H \rfloor$.

Proof. (if) Observe that $G \cong \lfloor G \rfloor$ and $H \cong \lfloor H \rfloor$ since κ-mappings are blank node bijections. Given the premise $\lfloor G \rfloor = \lfloor H \rfloor$, then $G \cong \lfloor G \rfloor = \lfloor H \rfloor \cong H$, which gives $G \cong H$.

(only if) Suppose the result does not hold: suppose (without loss of generality) that $\lfloor G \rfloor > \lfloor H \rfloor$ may hold if $G \cong H$. Since $G \cong H$, there exists a blank node bijection μ such that $\mu(G) = H$. Let κ be a mapping such that $\kappa(H) = \lfloor H \rfloor$. Now $\kappa \circ \mu(G) = \lfloor H \rfloor$. Since $\kappa \circ \mu$ is a valid κ-mapping for G and $\lfloor G \rfloor > \kappa \circ \mu(G)$, we arrive at a contradiction. ☐

[8] Any such scheme would do. We use an instance for brevity.

Example 4. *Take graph H from Example 7. We have* 3! *possible κ-mappings as follows.*

$\kappa(\cdot)$	_:c	_:d	_:e	H {(_:c, :p, :d), (_:d, :p, :e), (_:e, :p, :c)}
	_:b1	_:b2	_:b3	{(_:b1, :p, _:b2), (_:b2, :p, _:b3), (_:b3, :p, _:b1)}
	_:b1	_:b3	_:b2	{(_:b1, :p, _:b3), (_:b3, :p, _:b2), (_:b2, :p, _:b1)}
=	_:b2	_:b1	_:b3	{(_:b2, :p, _:b1), (_:b1, :p, _:b3), (_:b3, :p, _:b2)}
	_:b2	_:b3	_:b1	{(_:b2, :p, _:b3), (_:b3, :p, _:b1), (_:b1, :p, _:b2)}
	_:b3	_:b1	_:b2	{(_:b3, :p, _:b1), (_:b1, :p, _:b2), (_:b2, :p, _:b3)}
	_:b3	_:b2	_:b1	{(_:b3, :p, _:b2), (_:b2, :p, _:b1), (_:b1, :p, _:b3)}

The six mappings produce two distinct graphs. Assuming a typical lexical ordering, the first, fourth and fifth mappings produce $\lfloor H \rfloor$. One could (bijectively) relabel `_:c`, `_:d`, `_:e` *in the original graph without affecting the result: the result would be the same for any $\lfloor H' \rfloor$ such that $H \cong H'$.*

This suggests a correct and complete brute force algorithm for canonicalising RDF graphs: search all κ-mapping of G for one that gives $\lfloor G \rfloor$. However, optimisations to improve upon trying all $k!$ possible mappings are not obvious.

Instead we use similar notions of ordering to refine the results of the naive algorithm, distinguishing blank nodes that would otherwise have the same colour, similar in principle to standard graph isomorphism methods like Nauty [17].

Ordering colour partitions.

We assume a set of totally ordered colours $\mathbf{C}$ (in our case hashes). Let Clr be a map from blank nodes to colours computed, e.g., by the naive methods described previously. If Clr maps all blank nodes in G to a unique colour, we are done. However, as discussed, this may not always be the case. Initial colourings may assign the same colour to different blank nodes, forming a partition of blank nodes.

Definition 8. Let $\backsim$ denote an equivalence relation between two blank nodes $b_1, b_2 \in B$ such that $b_1 \backsim b_2$ if and only if $\mathrm{Clr}(b_1) = \mathrm{Clr}(b_2)$. We define a *coloured partition* P of a set of blank nodes B with respect to Clr as the quotient set of B with respect to $\backsim$, i.e., $P := B/\backsim$. We call $B' \in P$ a *part* of P. We call B' *trivial* if $|B'| = 1$; otherwise we call it *non-trivial*. We call P *fine* if it contains only trivial parts and *coarse* if it contains only one part.

Our goal is to thus use a deterministic process – avoiding arbitrary choices and use of blank node labels – to compute a colouring for the RDF graph G that results in a fine partition of blank nodes. The general idea is to manually distinguish individual blank nodes in non-trivial partitions. Since there is no deterministic way to choose one such blank node, we must try all equal choices. However, we can use an ordering of the partition to narrow the choices insofar as possible.

Definition 9. Given $P = \{B_1, \ldots B_n\}$, a partition of B w.r.t. the colouring Clr, we call a sequence of sets of blank nodes $\mathcal{P} := (B_1, \ldots B_n)$ an *ordered partition* of B w.r.t. Clr.

Thus given P and an associated colouring Clr, $\mathcal{P}$ encodes an ordering of the parts of P. To deterministically compute an initial $\mathcal{P}$ from an input P and Clr, we use a total ordering $\leq$ of parts such that $B' < B''$ if $|B'| < |B''|$, or in the case that $|B'| = |B''|$, then $B' < B''$ if and only if $\mathrm{Clr}(b') < \mathrm{Clr}(b'')$ for $b' \in B'$ and $b'' \in B''$ (recall that all elements of B' have the same colour; likewise for B'').

We can then *refine* $\mathcal{P}$ by recursively distinguishing blank nodes in its lowest non-trivial part, knowing the same part will likewise be selected for all isomorphic graphs.

Definition 10. Let $\mathcal{P} := (B_1, \ldots, B_n), \mathcal{P}' := (B'_1, \ldots, B'_m)$ be two ordered partitions. We say that $\mathcal{P}'$ is *finer* than $\mathcal{P}$, or equivalently that $\mathcal{P}$ is *coarser* than $\mathcal{P}'$, if and only if:

- for every part $B'_i \in \mathcal{P}$ there exists a $B_j \in \mathcal{P}$ such that $B'_i \subseteq B_j$; *and*
- if B'_i, $B'_j \in \mathcal{P}'$ such that $i \leq j$ and such that $B'_i \subseteq B_k$ and $B'_j \subseteq B_l$ for $B_k, B_l \in \mathcal{P}$, then $k < l$.

In other words, $\mathcal{P}'$ is finer than $\mathcal{P}$ if $\mathcal{P}'$ splits some of the parts of $\mathcal{P}$ and leaves those splits "in place". We call any function that takes as input an ordered partition and provides as output a finer partition a *refinement*.

Example 5. *Take these four ordered partitions:*

- $\mathcal{P} := (\{b_1, b_2\}, \{b_3, b_4\})$
- $\mathcal{P}_1 := (\{b_1\}, \{b_3\}, \{b_2, b_4\})$
- $\mathcal{P}_2 := (\{b_1\}, \{b_3\}, \{b_2\}, \{b_4\})$
- $\mathcal{P}_3 := (\{b_1, b_2\}, \{b_4\}, \{b_3\})$.

The last part of $\mathcal{P}_1$ is not a subset of any part of $\mathcal{P}$; hence it is neither finer nor coarser than $\mathcal{P}$. Likewise, since $\mathcal{P}_2$ "swaps" the order of b_2 and b_3, it is neither finer nor coarser than $\mathcal{P}$ (it is finer than $\mathcal{P}_1$). Only $\mathcal{P}_3$ is finer than $\mathcal{P}$.

Exploring partition refinements.

Algorithm 3 thus takes as input an initial ordered partition $\mathcal{P}$ with respect to a naive colouring Clr and deterministically explores possible refinements. For now we assume that the algorithm accepts an RDF graph with one connected component of blank nodes and no ground triples (i.e., a single blank node split). As discussed, the algorithm first computes an initial ordered partition in the entry function canonicalise. Next in distinguish, the algorithm recursively tries distinguishing, in turn, each blank node from the lowest non-trivial part of the current ordered partition. The colour of the distinguished blank node is marked and the result is fed as input along with G into the naive graph colouring method, which is run until fixpoint: this propagates the changes caused by distinguishing the blank node through the graph. The fixpoint colouring returned is passed to the refine function to compute a refinement of the previous ordered partition, dividing parts according to the new colouring while maintaining the prior precedence of parts. If the resulting refinement distinguishes all blank nodes, a labelling function uses the colours as a basis to label the blank nodes in G. Throughout the recursive process of distinguishing blank nodes, the lowest labelled graph seen thus far – $\underline{G}$ – is tracked. When all possibilities have been explored, $\underline{G}$ serves as a canonical version of G: the lowest graph found during the deterministic exploration of refinements.

The algorithm thus explores a directed labelled search-tree $T = (V, E, L)$, where the set of vertices V are ordered partitions, edges E connect ordered partitions to their direct refinements, and L labels edges with an ordered list of blank nodes manually distinguished for that refinement. More specifically, let $\mathcal{P} \in V$ be a node in the tree and $\mathrm{lin}(\mathcal{P})$ denote the label of its inlink (or an empty list for the root

Algorithm 3 Finding a canonical version of G

```
1:  function CANONICALISE($G$)                          ▷ $G$ an RDF graph
2:      Clr ← COLOUR($G$)
3:      $B \leftarrow \mathrm{terms}(G) \cap \mathbf{B}$
4:      compute partition $P$ of $B$ w.r.t. Clr
5:      $\mathcal{P} \leftarrow$ order $P$ by $\leq$
6:      $\underline{G} \leftarrow$ DISTINGUISH($G$, Clr, $\mathcal{P}$, $\emptyset$, $B$)
7:      return $\underline{G}$                              ▷ canonical graph
8:  function DISTINGUISH($G$, Clr, $\mathcal{P}$, $\underline{G}$, $B$)  ▷ $\underline{G}$: min. graph
9:      $B_i \leftarrow$ lowest non-trivial part of $\mathcal{P}$
10:     for $b \in B_i$ do
11:         Clr($b$) ← hashTuple(Clr($b$), '@')          ▷ '@' a marker
12:         Clr′ ← COLOUR($G$, Clr)                  ▷ initialise clr$_0$ with Clr
13:         $\mathcal{P}_{[b]} \leftarrow$ REFINE($\mathcal{P}$, Clr′, $b$, $B$)
14:         if $|\mathcal{P}_{[b]}| = |B|$ then              ▷ all nodes distinguished
15:             $G_C \leftarrow$ label($G$, Clr)             ▷ gen. bnodes from Clr
16:             if $\underline{G} = \emptyset$ or $G_C < \underline{G}$ then $\underline{G} \leftarrow G_C$
17:         else $\underline{G} \leftarrow$ DISTINGUISH($G$, Clr′, $\mathcal{P}_{[b]}$, $\underline{G}$, $B$)
18:     return $\underline{G}$                              ▷ canonical graph
19: function REFINE($\mathcal{P}$,Clr,$b$,$B$)
20:     denote $\mathcal{P} = (B_1, \ldots, B_i, \ldots, B_n)$ s.t. $b \in B_i$
21:     $\mathcal{P}_{[b]} \leftarrow (B_1, \ldots, B_{i-1}, \{b\})$        ▷ init. refined partition
22:     compute partition $P'$ of $B$ w.r.t. Clr
23:     $\mathcal{P}' \leftarrow$ order $P'$ by $\leq$
24:     for $B^+ \in (B_i \setminus \{b\}, B_{i+1}, \ldots, B_n)$ do
25:         $\mathcal{P}^+ := ()$                               ▷ empty list
26:         for $B' \in \mathcal{P}'$ do
27:             if $B^+ \cap B' \neq \emptyset$ then
28:                 $\mathcal{P}^+ \leftarrow \mathcal{P}^+ \parallel (B^+ \cap B')$     ▷ $\parallel$: concatenate
29:         $\mathcal{P}_{[b]} \leftarrow \mathcal{P}_{[b]} \parallel \mathcal{P}^+$
30:     return $\mathcal{P}_{[b]}$                          ▷ refined partition
```

node). Let B be the first non-trivial part of $\mathcal{P}$ (if $\mathcal{P}$ is fine, it has no children). Then, for every $b \in B$, an edge extends from $\mathcal{P}$ with label $\mathrm{lin}(\mathcal{P}) \parallel [b]$ (where $\parallel$ denotes concatenation) to the refinement of $\mathcal{P}$ computed by distinguishing b and rerunning the colouring to a fixpoint. Algorithm 3 explores this refinement tree in a depth-first manner looking for a leaf that corresponds to the lowest labelled graph.

EXAMPLE 6. *We adapt the canonical example of the canonical labelling of a 2D-grid-like graph – used for example by McKay and Piperno [18] – to an RDF version as follows:*

$$\begin{tikzcd}
\texttt{\_:a} \arrow[r, ":p"'] \arrow[d, ":p"'] & \texttt{\_:b} \arrow[d, ":q"'] & \texttt{\_:c} \arrow[l, ":p"] \arrow[d, ":p"'] \\
\texttt{\_:d} \arrow[r, ":q"'] & \texttt{\_:e} & \texttt{\_:f} \arrow[l, ":q"] \\
\texttt{\_:g} \arrow[u, ":p"] \arrow[r, ":p"'] & \texttt{\_:h} \arrow[u, ":q"] & \texttt{\_:i} \arrow[l, ":p"] \arrow[u, ":p"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\alpha \arrow[r, "\Gamma"'] \arrow[d, "\Gamma"'] & \beta \arrow[d, "\Delta"'] & \alpha \arrow[l, "\Gamma"] \arrow[d, "\Gamma"'] \\
\beta \arrow[r, "\Delta"'] & \gamma & \beta \arrow[l, "\Delta"] \\
\alpha \arrow[u, "\Gamma"] \arrow[r, "\Gamma"'] & \beta \arrow[u, "\Delta"] & \alpha \arrow[l, "\Gamma"] \arrow[u, "\Gamma"]
\end{tikzcd}$$

The graph on the right depicts the fixpoint of the naive colouring. The ordered partition resulting from this would be $\mathcal{P} = (\{\texttt{\_:e}\}, \{\texttt{\_:a}, \texttt{\_:c}, \texttt{\_:g}, \texttt{\_:i}\}, \{\texttt{\_:b}, \texttt{\_:d}, \texttt{\_:f}, \texttt{\_:h}\})$ *assuming* $\alpha < \beta$. *So now we distinguish blank nodes in the first non-trivial part. The following graph to the left shows the result of distinguishing* `_:a` *and colouring to fixpoint:* $\mathcal{P}_{[\texttt{\_:a}]} = (\{\texttt{\_:e}\}, \{\texttt{\_:a}\}, \{\texttt{\_:i}\}, \{\texttt{\_:c}, \texttt{\_:g}\}, \{\texttt{\_:b}, \texttt{\_:d}\}, \{\texttt{\_:f}, \texttt{\_:h}\})$.

$$\begin{tikzcd}
\delta \arrow[r, "\Gamma"'] \arrow[d, "\Gamma"'] & \epsilon \arrow[d, "\Delta"'] & \zeta \arrow[l, "\Gamma"] \arrow[d, "\Gamma"'] \\
\epsilon \arrow[r, "\Delta"'] & \eta & \theta \arrow[l, "\Delta"] \\
\zeta \arrow[u, "\Gamma"] \arrow[r, "\Gamma"'] & \theta \arrow[u, "\Delta"] & \iota \arrow[l, "\Gamma"] \arrow[u, "\Gamma"]
\end{tikzcd}
\qquad
\begin{tikzcd}
\kappa \arrow[r, "\Gamma"'] \arrow[d, "\Gamma"'] & \lambda \arrow[d, "\Delta"'] & \nu \arrow[l, "\Gamma"] \arrow[d, "\Gamma"'] \\
\xi \arrow[r, "\Delta"'] & o & \pi \arrow[l, "\Delta"] \\
\rho \arrow[u, "\Gamma"] \arrow[r, "\Gamma"'] & \sigma \arrow[u, "\Delta"] & \tau \arrow[l, "\Gamma"] \arrow[u, "\Gamma"]
\end{tikzcd}$$

$\mathcal{P}_{[\texttt{\_:a}]}$ *is still not a fine partition. Hence we proceed by distinguishing one of the blank nodes in the first non-trivial part* $\{\texttt{\_:c}, \texttt{\_:g}\}$. *If we distinguish, e.g.,* `_:c`*, then* $\mathcal{P}_{[\texttt{\_:a},\texttt{\_:c}]}$ *would be a fine ordered partition, as shown on the right. We can then imagine a deterministic labelling for the blank nodes of the graph based on the colours computed (e.g.,* `_:kappa`*, etc.) where only the minimum such graph is kept.*[9]

This example illustrates traversing one path down the refinement tree to a leaf: $[\texttt{\_:a}, \texttt{\_:c}]$. *Next the algorithm would try* $[\texttt{\_:a}, \texttt{\_:g}]$*, then* $[\texttt{\_:c}, \texttt{\_:a}]$, $[\texttt{\_:c}, \texttt{\_:i}]$*, and so on until all eight leaves of the tree are checked. Figure 1 depicts the refinement tree with the ordered partition resulting at every step where the first non-trivial part of each partition is shaded (other greyed-out parts of the tree will be discussed later). For every leaf, the corresponding labelled graph is computed and the minimum such graph is kept.*

In this particular example, every leaf of the tree will result in the same labelled graph: consider, e.g., the path $[\texttt{\_:c}, \texttt{\_:a}]$. *The result would be the same as for* $[\texttt{\_:a}, \texttt{\_:c}]$ *(but mirrored on the vertical axis): all edges, like* (ρ, Γ, σ)*, will appear in every such graph produced by a leaf (in this example).*

Pruning by automorphisms.

As observed in the previous example, traversing the entire tree may involve unnecessary repetitions of orientations due to automorphisms. In the given grid graph, there are two large *orbits* – sets of nodes mappable by an automorphism – namely $\{\texttt{\_:a}, \texttt{\_:c}, \texttt{\_:g}, \texttt{\_:i}\}$ and $\{\texttt{\_:b}, \texttt{\_:d}, \texttt{\_:f}, \texttt{\_:h}\}$. As such, exploring each leaf is redundant. The NAUTY algorithm – and similar canonical labelling methods such as TRACES [20] or BLISS [12] – keep track of automorphisms found while exploring the search tree. If two leaves produce the same labelled graph, then the mapping between the generating ordered partitions represents an automorphism.

EXAMPLE 7. *Taking Example 6, consider the two partitions* $\mathcal{P}_{[a,c]} = (\{\texttt{e}\}, \{\texttt{a}\}, \{\texttt{i}\}, \{\texttt{c}\}, \{\texttt{g}\}, \{\texttt{b}\}, \{\texttt{d}\}, \{\texttt{f}\}, \{\texttt{h}\})$ *and* $\mathcal{P}_{[a,g]} = (\{\texttt{e}\}, \{\texttt{a}\}, \{\texttt{i}\}, \{\texttt{g}\}, \{\texttt{c}\}, \{\texttt{d}\}, \{\texttt{b}\}, \{\texttt{h}\}, \{\texttt{f}\})$.[10] *Since both partitions generate the same labelled graph, we can assert the following bidirectional automorphism:* $\{\texttt{e}\} \leftrightarrow \{\texttt{e}\}$, $\{\texttt{a}\} \leftrightarrow \{\texttt{a}\}$, $\{\texttt{i}\} \leftrightarrow \{\texttt{i}\}$, $\{\texttt{c}\} \leftrightarrow \{\texttt{g}\}$, $\{\texttt{b}\} \leftrightarrow \{\texttt{d}\}$, $\{\texttt{f}\} \leftrightarrow \{\texttt{h}\}$.

Automorphisms can thus be used to prune the refinement tree [18] where we employ one such strategy. Let $\mathcal{P}$ be a node in the tree with children $\mathcal{P}'$ and $\mathcal{P}''$ derived by distinguishing b' and b'' respectively. Assume that $\mathcal{P}'$ has been visited and we are considering visiting $\mathcal{P}''$ next. If we can find an

[9]We could equivalently (modulo isomorphism) consider a labelling based on the index of the part containing the blank node in the ordered partition.

[10]Dropping underscores for formatting reasons.

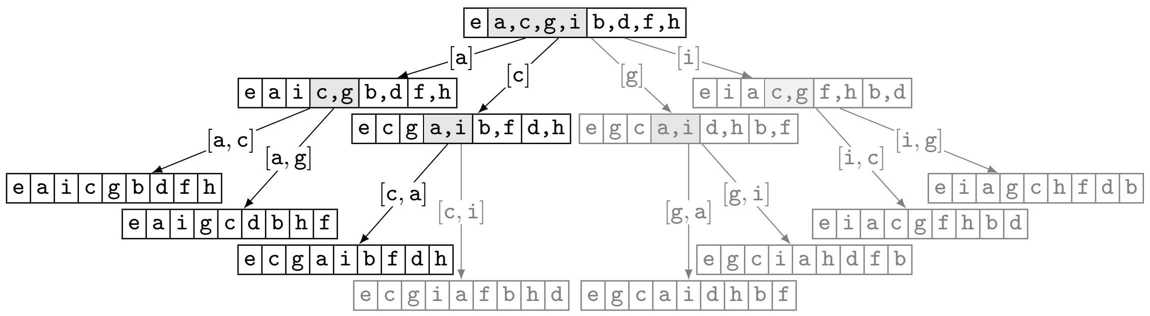


Figure 1: Refinement tree for Example 6

automorphism μ that is the *pointwise stabiliser* for all the blank nodes in $\text{lin}(\mathcal{P})$ (i.e., $\mu(b) = b$ for all $b \in \text{lin}(\mathcal{P})$) and that maps $\mu(b') \to \mu(b'')$, then we need not visit $\mathcal{P}''$ [18].

EXAMPLE 8. *In Figure 1, the greyed-out sub-tree need not be explored if automorphisms are tracked and used to prune branches (for now, we include* [c, i]*). After discovering that the* [a, c]*,* [a, g]*,* [c, a] *and* [c, i] *leaves form the same graph, automorphisms can be formed, per Example 7, by mapping the leaf nodes (in the same column) from the following table:*

[a, c]	e	a	i	c	g	b	d	f	h
[a, g]	e	a	i	g	c	d	b	h	f
[c, a]	e	c	g	a	i	b	f	d	h
[c, i]	e	c	g	i	a	f	b	h	d

Assume we are now considering visiting [g] *from the root. No nodes need be stabilised at the root, so we need not restrict the automorphisms considered. Take the automorphism derived from, e.g.,* [c, a] $\to$ [a, g]*, which gives* e $\to$ e*,* c $\to$ a*,* g $\to$ i*,* a $\to$ g*, and so on. We can use this automorphism to map to* [g] *from its sibling* [a]*, which has already been visited. We can now compute the sub-tree below* [g] *by applying the same automorphism to the sub-tree of* [a] *where we would ultimately end up with the same leaf graphs. Hence we know we can prune* [g]*. We need not visit* [i] *along the same lines.*

In fact, going back a little, as hinted at by Figure 1, in theory we need not have visited [c, i] *either. When considering visiting* [c, i]*, we must look for automorphisms that root* c*, but no such automorphism is computable from the first three leaves. However, if we look at a higher level, after visiting the* [c, a] *leaf, we have found an automorphism that makes visiting the higher branch at* [c] *redundant. Thus we need not continue with the* [c] *branch any further.*

A major challenge associated with the pruning phase is that naively materialising and indexing the entire automorphism group as it is discovered can consume lots of space. Our current implementation thus computes automorphisms on-the-fly as needed, lazily generating and caching orbits with pointwise stabilisers relevant for a given level of the tree. For this reason, in the previous example we would not prune [c, i]: instead of checking pruning possibilities at every level for all steps, we only check on the current level of the depth-first search. Thus we would run [c, i] without checking at the [c] level. When the depth-first-search returns to the higher level, we would prune at [g] and [i].

In general, a variety of pruning and search strategies have been explored in the graph isomorphism literature that are not considered in this current work (see, e.g., [18] for more details, explaining how different strategies may work better for different types of graphs). However, such strategies only become crucial when considering larger instances of difficult cases which, as we will put forward later, are unlikely to be of concern when dealing with real-world RDF data.

Algorithmic characteristics.

We briefly remark on two properties of the algorithm.

THEOREM 4. *Algorithm 3 terminates.*

PROOF. The search tree, though exponential, is finite. The search follows standard depth-first recursion. $\square$

THEOREM 5. *The graph produced by Algorithm 3 is a canonical version of G with respect to isomorphism.*

PROOF. (*Sketch*) Follows from the fact that the output is isomorphic with G and, coupled with Lemma 2, that the process provides a deterministic ordering of the isomorphs of G without considering blank node labels. $\square$

With respect to RDF graphs containing a blank node split with multiple graphs, we can perform the split per Algorithm 2 and then run Algorithm 3 over each split. We can then compare the resulting canonical graphs. If two or more graphs contain the same blank node label, we can distinguish all blank nodes in said graphs by hashing with an arbitrary fresh symbol to separate them, taking the union with the ground triples for output. (Another option would be to run Algorithm 3 directly over the full RDF graph, but this would lead to products in the orbits and thus a larger search tree.)

5. SKOLEMISATION: GENERATING IRIS

Thus far we've discussed the general principles of canonically labelling an RDF graph to preserve isomorphism. In this context, it is only important that the labels produced from the colouring are locally unique. However, if we were to use such a scheme to Skolemise the blank nodes and produce IRIs, these IRIs would have to be globally unique. This raises two important aspects that we now discuss.

Hashing.

Table 1 presents the estimated risk of collisions for hypothetical hashing schemes of various lengths (represented in bits, hexadecimal strings and Base64, rounding up the number of characters where necessary). In particular, we present the approximate number of inputs needed to reach the given

Table 1: Approximate number of elements needed to reach the given probability of hash collision for the given length hash (assuming perfect uniformity)

Bits	Hex	B64	Probability			
			2^{-1}	2^{-4}	2^{-16}	2^{-64}
32	8	6	77,163	22,820	362	< 2
64	16	11	5.1×10^{09}	1.5×10^{09}	2.4×10^{07}	< 2
128	32	22	2.2×10^{19}	6.4×10^{18}	1.0×10^{17}	6.1×10^{09}
160	40	27	1.1×10^{24}	4.2×10^{23}	6.7×10^{21}	4.0×10^{14}
256	64	43	4.0×10^{38}	1.1×10^{38}	1.9×10^{36}	1.1×10^{29}
512	128	86	1.4×10^{77}	4.0×10^{76}	6.4×10^{74}	3.8×10^{67}

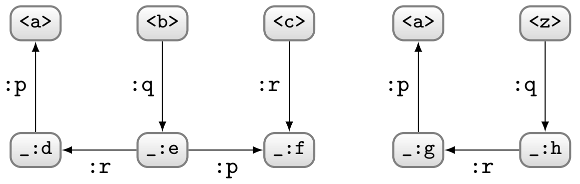
probability of collision, where 2^{-1} indicates a $\frac{1}{2}$ probability, 2^{-4} a $\frac{1}{16}$ probability, etc. We assume hashing schemes with perfect uniformity, i.e., we assume that the schemes produce an even spread of hashes across different inputs.

We see a trade-off: longer hashes require longer string labels but increase tolerance to collisions. When considering the Web, we could be talking about billions or trillions of inputs to the scheme. As such, we can rule out hashes of 32 or 64 bits, where even relatively modest inputs cause a 50% or greater chance of a collision. However, if we use a very long labelling scheme, the resulting labels would be cumbersome and slow down transmission times, clog up storage, etc. For reference, we previously found that the average length of IRIs found in a large RDF crawl was about 52 characters [11]. Even in Base64, a 512- or 256-bit hash would produce a relatively cumbersome IRI. Hence we propose that the sweet-spot is around 128-bit (MD5 or Murmur3_128) or 160-bits (SHA1): in this range, the likelihood of collisions – even assuming very large inputs in the trillions – are negligible when compared with the risk of, say, a comet wiping out life as we know it in the meantime.

Global colours.

When generating Skolem IRIs, we would like to guarantee that these IRIs are globally unique with respect to the graph they originate from. This requires an additional step.

Example 9. *Consider the following two RDF graphs:*



Both graphs would have distinguished colours after one iteration of Algorithm 1 but nodes `_:d` *and* `_:g` *would have the same colour:* `_:d` *would not yet have "encoded" the information from* `<b>` *and* `_:f`*. The Skolem IRIs produced for* `_:d` *and* `_:g` *would (problematically) thus be the same.*

As such, we need an additional step to ensure that Skolem IRIs are unique with respect to a given graph, modulo isomorphism. Our solution is to compute a hash of the entire canonicalised graph – which assuming perfect hashing, is unique to that graph modulo isomorphism – and combine that hash with the hash of each blank node. The hash of each blank node then includes a hash signature unique (modulo hash collisions) to the structure of the entire graph.

Table 2: BTC–14 runtimes for three hash functions

Hash	Bits	Runtime (h)	Adjusted (h)
MD5	128	16.4	12.4
Murmur3_128	128	13.6	9.6
SHA1	160	16.5	12.5

6. EVALUATION

We implemented our methods as a Java package we call BLabel.[11] We now present evaluation in two main parts. First we look at real-world graphs: given that our complete algorithm is indeed exponential (as expected from the outset), we want to see if bad cases occur in reality; we also wish to compare the performance of different possible hashing schemes in the 128/160-bit range as previously justified. Second we stress-test our algorithm for some nasty cases known from the graph isomorphism literature to illustrate the types of graphs for which our algorithm can struggle.

Real-world RDF graphs.

We ran our complete algorithm for the BTC–14 dataset: a collection of 43.6 million RDF graphs crawled from Web documents spanning 47,560 pay-level-domains [13].[12] The dataset contains around 4 billion quadruples, taking up about 1.1 TB uncompressed in N-Quads format [3]. Experiments were run in a single-threaded manner on an Intel® E5-2407 Quad-Core 2.2GHz machine with 30 GB of heap space.

We first sorted the data according to context to group the triples of documents together (this took 20.8 hours). We then scanned the sorted file, loading each RDF graph individually into memory and computing a canonical labelling before moving onto the next graph. We kept a hash for every graph to identify isomorphic duplicates. We also ran a control that just parsed the data and loaded the graphs without canonicalisation; this took almost precisely 4 hours.

We then ran canonicalisation tests with three suitable hashing schemes: results are provided in Table 2. We present the arity of the hash scheme, the total runtime and the runtime adjusted to subtract the control overhead. We see that the total time taken to canonicalise all of the graphs, minus the control, was between 9.6–12.5 hours, where Murmur3_128 was the fastest. Given that there were 9.9 million graphs containing blank nodes, this averages to 3.5–4.5 ms per graph. The slowest graph took 31–40 seconds, mainly due to its size: 7.3 million triples with 254 thousand blank nodes.[13] Of the 9.9 million graphs with blank nodes, 9.4 million (95%) were left after removing isomorphic duplicates.

These results show that our canonicalisation scheme is indeed practical for a large collection of real-world graphs.

Synthetic cases.

We also tried some difficult synthetic cases from the Bliss benchmark [12] for standard graph isomorphism.[14] We took five well-known classes of standard graphs at various sizes and represented them as RDF graphs (using a single predicate with edges in both directions). We also took a set of

[11]See http://aidanhogan.com/skolem/ for code/data.

[12]Although many such domains only provide "vacuous" RDFa metadata extracted from HTML pages.

[13]http://www.berkeleybop.org/ontologies/ncbitaxon.owl

[14]http://www.tcs.hut.fi/Software/bliss/benchmarks/index.shtml

Table 3: Results for synthetic graphs

Class	k	Triples	BNodes	Time (ms)
Grid 2D	15	840	225	955
	100	39,600	10,000	58,482
Grid 3D	7	1,764	343	938
	19	38,988	6,589	37,402
Clique	16	240	16	719
	55	2,970	55	578,830
	56	3,080	56	—
Lattice	6	360	36	532
	18	11,016	324	371,073
	19	12,996	361	—
Triangle	9	504	36	656
	29	21,924	406	507,036
	30	24,360	435	—
Miyazaki	2	120	40	140
	8	480	160	195,770
	10	600	200	—

Miyazaki graphs known to be a particularly tough case for graph isomorphism [19]. A timeout of ten minutes was set. The experiments were run on a laptop with 1GB of heap-space and an Intel® i3 Dual-Core 2.4GHz processor. We use Murmur3_128: the fastest hashing method per Table 2.

Table 3 summarises the results. For each class, we present the largest graph run under one second, the largest graph under the ten minute timeout, and for classes with graphs that did not succeed, the smallest graph that hit the timeout ("—" indicates a timeout). We see that our algorithm struggles for even modest sizes of certain synthetic graphs, with Miyazaki graphs being the hardest tested. However, we argue that such structures would be rare in real-world graphs: we speculate these cases would require an "adversary" deliberately creating them to be problematic in reality. Likewise we take encouragement from the fact that, e.g., cliques of size 16 can be processed in under a second.

7. RELATED WORK

In 2003, Carroll [4] proposed methods to canonicalise RDF graphs with blank nodes in such a manner that they could be signed. Carroll had quite similar aims to this work, albeit with a different use-case in mind: Carroll's aim was to generate a digital signature of the entire graph rather than to label individual blank nodes. The method he proposes for signing the graph is based on writing it to N-Triples, temporarily mapping all blank nodes to a global blank node, sorting the triples lexically, and then relabelling the blank nodes sequentially as the sorted file is scanned, preserving a bijection between the original input and output blank nodes. In cases where blank nodes are not distinguished by this method, Carroll proposes to inject new triples on such blank nodes that uniquely "mark" them but in such a way that the semantics ignores these triples. This is a weakness of the approach: it modifies the signed graph in an ad hoc way.

Jena [16] offers a method for checking isomorphism between two RDF graphs. However, the method is designed for pairwise isomorphism-checks rather than for producing a (globally-unique) canonical labelling.

Tzitzikas at al. [21] compute minimal deltas between RDF graphs based on an edit distance metric. They propose two algorithms: one views the computation as a combinatorial optimisation problem to which the Hungarian method can be applied; the other is based on computing a signature for blank nodes based on the constant terms in their direct neighbourhood. Although the goals of Tzitzikas at al. and our goals differ somewhat, the signature method that they propose for blank nodes is similar to a non-recursive version of our naive colouring algorithm.

We recently conducted a survey of blank nodes [15, 10], covering their theory and practice, their semantics and complexity, how they are used in the standards and in published data, etc. Some of the main conclusions were that blank nodes are common in RDF data published on the Web (as previously discussed), and that although RDF graphs can sometimes contain connected blank nodes with treewidths as high as 6, most graphs do not contain cycles of blank nodes, which makes various tasks involving them easier. We also found that in a merge of the BTC–12 corpus, 6% of the blank nodes were redundant under simple entailment [10]; for this, we used a signature method similar to the naive algorithm proposed earlier, but only for a fixed depth: we used the method to initially cluster potentially isomorphic graphs rather than to generate a full canonical labelling.

Popular RDF syntaxes like Turtle or RDF/XML are tree-based and require explicit blank node labels for blank nodes to form cycles. The observation that many RDF graphs have acyclical blank nodes and that the complexity of (implementing) various operations over such graphs drops significantly has led to calls for defining a profile of RDF that disallows cyclical blank nodes [15, 2]. Booth calls this profile "Well Behaved RDF" [2]. Recent discussion on the public mailing lists have centred around "deterministic naming" of blank nodes (which is what we tackle here) for such "well-behaved graphs". Although removing consideration of blank node cycles would simplify matters, in this paper we show that in terms of deterministically labelling blank nodes, many real-world graphs, even with cycles, can be labelled cheaply. Our results support the conjecture that exponential cases are unlikely to be found in "real-world" RDF graphs.

8. CONCLUSION

In this paper, we presented methods to compute a canonical labelling of RDF graphs. First we proved that RDF isomorphism is GI-complete: hence we would expect such an algorithm to be exponential. We then presented an efficient naive colouring method that we would expect to cover many common cases, and showed why it could fail to distinguish blank nodes. Inspired by graph isomorphism algorithms like Nauty, we then presented a sound algorithm for producing a canonical labelling, which can be used to check isomorphism between RDF graphs or to cluster isomorphic RDF graphs within a collection. We also addressed issues regarding use of the algorithm to compute global Skolem IRIs that deterministically replace blank nodes. Though our algorithm is exponential, we found that we could process 9.9 million real-world RDF graphs (with blank nodes) in approximately 9.6 hours: on average, about 3.5 ms a graph. We showed that some synthetic cases can cause performance problems but argued that such cases seem unlikely to occur "naturally". We thus see our proposal as being both well-founded in theory and (hopefully) useful in practice.

9. REFERENCES

[1] L. Babai, P. Erdös, and S. M. Selkow. Random graph isomorphism. *SIAM J. Comput.*, 9(3):628–635, 1980.

[2] D. Booth. Well Behaved RDF: A Straw-Man Proposal for Taming Blank Nodes, 2012. http://dbooth.org/2013/well-behaved-rdf/Booth-well-behaved-rdf.pdf.

[3] G. Carothers. RDF 1.1 N-Quads. W3C Recommendation, 2014. http://www.w3.org/TR/n-quads/.

[4] J. J. Carroll. Signing RDF graphs. In *International Semantic Web Conference*, pages 369–384, 2003.

[5] R. Cyganiak, D. Wood, and M. Lanthaler. RDF 1.1 Concepts & Abstract Syntax. W3C Recommendation, Feb. 2014. http://www.w3.org/TR/rdf11-concepts/.

[6] C. Gutierrez, C. A. Hurtado, A. O. Mendelzon, and J. Pérez. Foundations of Semantic Web databases. *J. Comput. Syst. Sci.*, 77(3):520–541, 2011.

[7] P. Hayes. RDF Semantics. W3C Recommendation, 2004. http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

[8] P. Hayes and P. F. Patel-Schneider. RDF 1.1 Semantics. W3C Recommendation, 2014. http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

[9] T. Heath and C. Bizer. *Linked Data: Evolving the Web into a Global Data Space*, volume 1. Morgan & Claypool, 2011.

[10] A. Hogan, M. Arenas, A. Mallea, and A. Polleres. Everything you always wanted to know about blank nodes. *J. Web Sem.*, 27:42–69, 2014.

[11] A. Hogan, J. Umbrich, A. Harth, R. Cyganiak, A. Polleres, and S. Decker. An empirical survey of Linked Data conformance. *J. Web Sem.*, 14:14–44, 2012.

[12] T. A. Junttila and P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007.

[13] T. Käfer and A. Harth. Billion Triples Challenge data set. http://km.aifb.kit.edu/projects/btc-2014/, 2014.

[14] O. Lassila and R. R. Swick. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation, 1999. http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.

[15] A. Mallea, M. Arenas, A. Hogan, and A. Polleres. On blank nodes. In *International Semantic Web Conference*, pages 421–437, 2011.

[16] B. McBride. Jena: A Semantic Web Toolkit. *IEEE Internet Computing*, 6(6):55–59, 2002.

[17] B. McKay. Practical Graph Isomorphism. In *Congressum Numerantium*, volume 30, pages 45–87, 1980.

[18] B. D. McKay and A. Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014.

[19] T. Miyazaki. The Complexity of McKay's Canonical Labeling Algorithm. In *Groups and Computation, II*, pages 239–256, 1997.

[20] A. Piperno. Search Space Contraction in Canonical Labeling of Graphs (Preliminary Version). *CoRR*, abs/0804.4881, 2008.

[21] Y. Tzitzikas, C. Lantzaki, and D. Zeginis. Blank Node Matching and RDF/S Comparison Functions. In *International Semantic Web Conference*, pages 591–607, 2012.