Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes

AIDAN HOGAN, Center for Semantic Web Research, DCC, University of Chile, Chile

Existential blank nodes greatly complicate a number of fundamental operations on RDF graphs. In particular, the problems of determining if two RDF graphs have the same structure modulo blank node labels (i.e. if they are *isomorphic*), or determining if two RDF graphs have the same meaning under simple semantics (i.e., if they are *simple-equivalent*), have no known polynomial-time algorithms. In this paper, we propose methods that can produce two canonical forms of an RDF graph. The first canonical form preserves isomorphism such that any two isomorphic RDF graphs will produce the same canonical form; this *iso-canonical* form is produced by modifying the well-known canonical labelling algorithm NAUTY for application to RDF graphs. The second canonical form additionally preserves simple-equivalence such that any two simple-equivalent RDF graphs will produce the same canonical form; this *equi-canonical* form is produced by, in a preliminary step, leaning the RDF graph, and then computing the iso-canonical form. These algorithms have a number of practical applications, such as for identifying isomorphic or equivalent RDF graphs in a large collection without requiring pair-wise comparison, for computing checksums or signing RDF graphs, for applying consistent Skolemisation schemes where blank nodes are mapped in a canonical manner to IRIs, and so forth. Likewise a variety of algorithms can be simplified by presupposing RDF graphs in one of these canonical forms. Both algorithms require exponential steps in the worst case; in our evaluation we demonstrate that there indeed exist difficult synthetic cases, but we also provide results over 9.9 million RDF graphs that suggest such cases occur infrequently in the real world, and that both canonical forms can be efficiently computed in all but a handful of such cases.

CCS Concepts: • **Information systems → Resource Description Framework (RDF)**; • **Mathematics of computing → Graph algorithms**.

Additional Key Words and Phrases: Semantic Web, Linked Data, Skolemisation, Isomorphism, Signing

ACM Reference Format:
Aidan Hogan. 2017. Canonical Forms for Isomorphic and Equivalent RDF Graphs: Algorithms for Leaning and Labelling Blank Nodes. *ACM Trans. Web* 0, 0, Article 00 (January 2017), 61 pages. https://doi.org/0000001.0000001

1 INTRODUCTION

At the very core of the Semantic Web is the Resource Description Framework (RDF): a standard for publishing graph-structured data that uses *IRIs* as global identifiers such that graphs in remote locations on the Web can collaborate to contribute information about the same resources using consistent terminology in an interoperable manner. The adoption of RDF on the Web has been continuously growing, where we can point to the hundreds of datasets published as RDF using Linked Data principles [22] spanning a variety of domains, including collections from governmental

This work was supported by the Millennium Nucleus Center for Semantic Web Research under Grant No. NC120004 and by Fondecyt Grant No. 11140900.
Author's address: A. Hogan, DCC, Universidad de Chile, Avenida Beauchef 851, Santiago, Chile.
Author's address: Aidan Hogan, Center for Semantic Web Research, DCC, University of Chile, Santiago, Chile.

1559-1131/2017/1-ART00 $15.00
https://doi.org/0000001.0000001

organisations, scientific communities, social web sites, media outlets, online encyclopaedias, and so forth [52]. Furthermore, hundreds of thousands of web-sites and hundreds of millions of web-pages now contain embedded RDFa [24] – incentivised by initiatives such as Schema.org (promoted by Google, Microsoft, Yahoo! and Yandex), and the Open Graph Protocol (promoted by Facebook) – with three of the largest providers being, for example, `tripadvisor.com`, `yahoo.com` and `hotels.com` [44].

Despite this trend of RDF playing an increasingly important role as a format for structured-data exchange on the Web, there are a number of fundamental operations over RDF graphs for which we lack practical algorithms. In fact, RDF does not consist purely of statements containing IRIs, but also supports *literals* that represent datatyped-values such as strings or numbers, and, more pertinently for the current scope, *blank nodes* that represent a resource without an explicit identifier. It is the presence of blank nodes in RDF graphs that particularly complicates matters.

In the original W3C Recommendation for RDF published in 1999 [35], *anonymous nodes* were introduced as a means of describing a resource without an explicit identifier, quoting use-cases such as the representation of bags of resources in RDF, the use of reification to describe RDF statements as if they were themselves resources, or simply to describe resources that did not have a native URI/IRI associated with them. When the W3C Recommendation for RDF was revised in 2004 [20], the serialisation of RDF graphs as triples was supported through the introduction of the modern notion of *blank nodes* to represent resources without explicit identifiers; these blank nodes were defined as existential variables that are locally-scoped. Intuitively speaking, this existential semantics captures the idea that one can relabel the blank nodes of an RDF graph in a one-to-one manner without affecting the structure [11] nor the semantics [21] of the RDF graph, nor without having to worry if those same labels already exist in another graph elsewhere on the Web.

Practically speaking, blank nodes are used for two main reasons [28]:

- Blank nodes allow publishers to avoid having to explicitly identify specific resources, where RDF syntaxes such as Turtle [5] use this property to enable various convenient shortcuts for specifying ordered lists, n-ary relations, etc.; tools parsing these syntaxes can invent blank nodes to represent these implicit nodes.
- In other cases, publishers may use blank nodes to represent true existential variables, where a value is known to exist, but the exact value is not known.

In a recent questionnaire we conducted with the Semantic Web community, we found that publishers may (hypothetically) use blank nodes sometimes in one case, or the other, or both [28]. In any case, blank nodes have become widely used on the Web, where in previous work we found that in a survey of 8.4 million Web documents containing RDF crawled from 829 pay-level domains[1], 66% of domains and 45% of documents used blank nodes [28].

Unfortunately, the presence of blank nodes in RDF complicates some fundamental operations on RDF graphs. For example, imagine two different tools parsing the same RDF graph – say, for example, retrieved from the same location on the Web in the same syntax – into a set of triples, labelling blank nodes in an arbitrary manner. Now take the two resulting sets of triples and say we wish to determine if the two RDF graphs are the same modulo blank node labels; i.e., to determine if they are *isomorphic* [11]. If the original RDF graph did not contain blank nodes, this process is trivially possible in polynomial time by checking if both sets of triples are equal, for example, by sorting both sets of triples and then comparing them sequentially. However, if the original RDF graph contains blank nodes, then the problem of deciding RDF isomorphism has the same computational complexity class as *graph isomorphism* (GI-complete), for which there are no known polynomial-time algorithms (if such an algorithm were found, it would establish GI = P).

[1]Domains such as `bbc.co.uk` or `facebook.com`, but not `news.bbc.co.uk` or `co.uk`

While isomorphism refers to a structural comparison of RDF graphs, it is also possible to consider a semantic comparison of such graphs. The RDF semantics [21] defines a notion of entailment between RDF graphs such that one RDF graph entails another, loosely speaking, if the entailed graph adds no new information over the entailing graph; in other words, if the entailing graph is considered to be true under the model theory of the semantics, then the entailed graph must likewise be considered true. Two RDF graphs that entail each other are thus considered *equivalent*: as having the same meaning under a particular semantics. The foundational semantics for RDF, called *simple semantics*, does not consider any special vocabulary nor the interpretation of datatype values; rather, it considers the meaning of RDF graphs considering blank nodes as existential variables and IRIs and literals as *ground* terms denoting a particular resource. Given an RDF graph G and H without blank nodes, then asking if G entails H is the same as asking if H contains a subset of the triples of G, which again is possible in polynomial time by, for example, sorting both sets of triples and comparing them sequentially to see if every triple of H is in G. However, as we discuss later, if both RDF graphs contain blank nodes, the problem is in the same complexity class as the problem of *graph homomorphism* (namely NP-complete), implying that there is no known polynomial-time solution. Furthermore, it is known that determining if two RDF graphs are simple-equivalent – i.e. if they simple-entail each other – falls into the same complexity class [18].

In summary then, there are no known polynomial-time algorithms for these two fundamental operations of determining if two RDF graphs are structurally the same (per isomorphism) or semantically the same (per simple equivalence).[2]

In this paper, we propose two different canonical forms for RDF graphs. First we must define two RDF graphs as *equal* (or we may sometimes say *the same*) if and only if they are equal as sets of RDF triples considering blank node labels as fixed in the same manner as IRIs and literals. The first canonical form, which we call *iso-canonical*, is an RDF graph unique for each set of isomorphic RDF graphs; in other words, it is a form that is canonical with respect to the structure of RDF graphs. The second canonical form, which we call *equi-canonical*, is an RDF graph unique for each set of simple-equivalent RDF graphs; in other words, it is a form that is canonical with respect to the (simple) semantics of RDF graphs. More specifically, two RDF graphs are isomorphic if and only if their iso-canonical forms are the same; two RDF graphs are simple-equivalent if and only if their equi-canonical forms are the same.

These canonical forms have a number of use-cases, including:

- given a large set of RDF graphs, detect/remove graphs that are duplicates;
- given an RDF graph, compute a hash of that RDF graph, which can be used for computing and verifying checksums, signatures, etc.;
- given an RDF graph, Skolemise the blank nodes in the RDF graph – replacing them with fresh IRIs – in a deterministic manner based on the content of the graph.

Our methods do not rely on the syntax of the RDF documents in question, but rather operate on the abstract representation of an RDF graph as a set of triples, where the user can decide whether they wish to consider duplicates, signatures, Skolem constants, etc., to be consistent with respect to either isomorphism or (simple) equivalence.

Given an input RDF graph, we propose algorithms for computing both its iso-canonical form and equi-canonical form. As expected from earlier discussion, neither of these algorithms is in polynomial-time: both have exponential-time worst-case performance in the general case. However, unlike the more general problems of graph isomorphism and graph homomorphism, in the case of

[2]Although polynomial-time algorithms have been proposed in the literature for computing canonical forms of RDF graphs with respect to isomorphism (e.g., [2, 9, 32]), these may not always yield correct results. We will discuss such works in more detail in Section 7.

real-world RDF graphs, we often have ground information that helps to distinguish blank nodes. Hence in our evaluation, we first present results for computing both canonical forms over the BTC−14 dataset [31] – a collection of 43.6 million RDF graphs of which 9.9 million contain blank nodes – where said results suggest that computing these forms is efficient in all but a handful of cases. To stress-test our algorithms, we also present results for canonicalising a collection of synthetic graphs at various sizes, which give some idea of the type of RDF graph required to invoke exponential runtime, arguing that such graphs are unlikely to occur naturally in practice.

This paper is an extension of earlier work [27] where we first introduced methods to canonically label blank nodes, computing an iso-canonical form of RDF graphs for the purposes of Skolemisation. Aside from extended discussion throughout, the main novel contribution of this paper is to discuss an algorithm for leaning RDF graphs and for computing their equi-canonical form in a manner that takes into account not only structural but also semantic identity. In this latter contribution, we take some of the ideas and experiences learnt from another previous work where we presented some initial ideas on leaning RDF graphs [28]; however, the methods we present in this paper focus on leaning potentially complex RDF graphs in memory where, in particular, we present a novel depth-first search algorithm that is shown to perform better than a variety of baseline methods.

We begin by presenting some preliminaries relating to the structure and semantics of RDF graphs (Section 2). We then present a theoretical analysis with a mix of new and existing results that help establish both the hardness of the canonicalisation problems we propose to tackle, as well as high-level approaches by which they can be computed (Section 3). Afterwards, we present in detail our algorithms for computing the iso-canonical version of an RDF graph, which relies on a canonical labelling of blank nodes (Section 4); and the equi-canonical version of an RDF graph (Section 5), which relies on a pre-processing step that leans the RDF graph. We then present evaluation results over collections of both real-world and synthetic RDF graphs (Section 6). We then discuss related works before concluding the paper (Sections 7 and 8).

2 PRELIMINARIES

We now present some formal preliminaries with respect to RDF graphs, isomorphism, and the simple semantics of RDF.

2.1 RDF terms, triples and graphs

RDF graphs are sets of triples containing RDF terms, with certain restrictions on which terms can appear in which positions of a triple.

Definition 2.1 (RDF term). Let $\mathbf{I}$, $\mathbf{L}$ and $\mathbf{B}$ denote the infinite sets of IRIs, literals and blank nodes respectively. These sets are pair-wise disjoint. We refer generically to an element of one of these sets as an *RDF term*. We refer to elements of the set $\mathbf{IL}$ (i.e., $\mathbf{I} \cup \mathbf{L}$) as *ground RDF terms*.

Definition 2.2 (RDF triple). We call a triple $(s, p, o) \in \mathbf{IB} \times \mathbf{I} \times \mathbf{ILB}$ an RDF triple, where the first element, called the *subject*, must be an IRI or a blank node; the second element, called the *predicate*, must be an IRI; and the third element, called the *object*, can be any RDF term.

Definition 2.3 (RDF graph). An RDF graph $G \subset \mathbf{IB} \times \mathbf{I} \times \mathbf{ILB}$ is a finite set of RDF triples. We denote by terms(G) the set of all RDF terms appearing in G, and bnodes(G) the set of all blank nodes appearing in G.

Remark 1. *We say that two RDF graphs G and H are* equal*, or* the same*, if and only if $G = H$ in terms of set equality.*

2.2 RDF isomorphism

Two RDF graphs that are the same modulo blank nodes labels – i.e., where one can be obtained from the other through a one-to-one mapping of blank nodes – are called isomorphic. We first give a preliminary definition to capture the idea of mapping blank nodes to other RDF terms:

Definition 2.4 (Blank node mapping and bijection). Let $\mu : \mathbf{ILB} \longrightarrow \mathbf{ILB}$ be a partial mapping of RDF terms to RDF terms, where we denote by $\mathrm{dom}(\mu)$ the domain of μ and by $\mathrm{codom}(\mu)$ the codomain of μ. If μ is the identity on **IL**, we call it a *blank node mapping*. If μ is a blank node mapping that maps blank nodes in $\mathrm{dom}(\mu)$ to blank nodes in $\mathrm{codom}(\mu)$ in a bijective manner, we call it a *blank node bijection.*

Abusing notation, given an RDF graph G, we may use $\mu(G)$ to denote the image of G under μ (i.e., the result of applying μ to every term in G).

We are now ready to define the notion of isomorphism between RDF graphs.

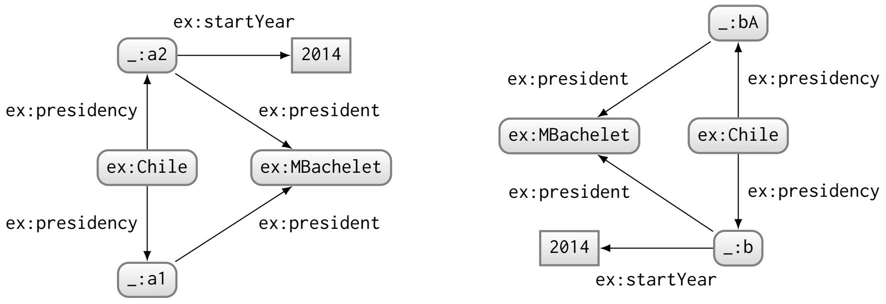
Definition 2.5 (RDF isomorphism). Two RDF graphs G and H are *isomorphic*, denoted $G \cong H$, if and only if there exists a blank node bijection μ such that $\mu(G) = H$, in which case we call μ an *isomorphism.*

Lemma 2.6. *RDF isomorphism ($\cong$) is an equivalence relation.*

Proof. First, $\cong$ is reflexive per the existence of the identity map μ on blank nodes, which is a blank node bijection. Second, $\cong$ is symmetric since if $G \cong H$, then there exists μ such that $\mu(G) = H$ and such that $\mu^{-1}(H) = G$, where μ^{-1} is also a blank node bijection. Third, $\cong$ is transitive since if $G \cong H$ and $H \cong I$, then there exist blank node bijections μ and μ' such that $\mu(G) = H$, $\mu'(H) = I$, and thus $\mu'(\mu(G)) = I$, where $\mu' \circ \mu$ is a blank node bijection that witnesses $G \cong I$. □

Remark 2. *If $G = H$, then $G \cong H$.* □

Example 2.7. Take the following two RDF graphs, with G on the left and H on the right, where the term `2014` is a literal (denoted with a square box), all terms prefixed with underscore are blank nodes, and all other terms (in the `ex:` example namespace) are IRIs. Now we consider: are these RDF graphs isomorphic?

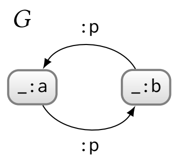


In fact they are: there is a blank node bijection μ such that $\mu(\texttt{\_:a1}) = \texttt{\_:bA}$ and $\mu(\texttt{\_:a2}) = \texttt{\_:b}$ where $\mu(G) = H$. We could also take the inverse mapping μ^{-1}, where $\mu^{-1}(H) = G$ and where μ^{-1} is also a blank node bijection. Thus we conclude $G \cong H$. □

A related concept to isomorphism – and one that will play an important role later – is that of an automorphism, which is an isomorphism that maps an RDF graph to itself. Intuitively speaking, automorphisms represent a form of symmetry in the graph.

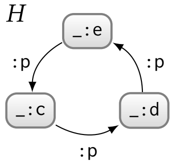
Definition 2.8 (RDF automorphism). An *automorphism* of an RDF graph G is an isomorphism that maps G to itself; i.e., a blank node mapping μ is an automorphism of G if $\mu(G) = G$. If μ is the identity mapping on blank nodes in G then μ is a *trivial automorphism*; otherwise μ is a *non-trivial automorphism*. We denote the set of all automorphisms of G by $\mathrm{Aut}(G)$.

Example 2.9. We give the automorphisms for two RDF graphs G and H. Trivial automorphisms (i.e., those that are the identity mapping) are shown in grey.



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

Applying any of the automorphisms shown for the graph in question would lead to the same graph (and not just an isomorphic copy). □

2.3 Simple semantics, interpretations, entailment and equivalence

The RDF (1.1) Semantics recommendation [21] defines four model-theoretic regimes that, loosely speaking, provide a mathematical basis for assigning truth to RDF graphs and, subsequently, for formally defining when one RDF graph *entails* another: in other words, if one assigns truth to a particular RDF graph, entailment defines which RDF graphs must also hold true as a logical consequence. Thus, unlike RDF isomorphism which is, in some sense, a structural comparison of RDF graphs [11], entailment offers a semantic comparison of RDF graphs in terms of their underlying meaning [21]. The four regimes are: simple semantics, datatype semantics, RDF semantics and RDFS semantics. In this paper, we are interested in the simple semantics, which codifies a meaning for RDF graphs without considering the interpretation of datatype values or special vocabulary terms (such as `rdf:type` or `rdfs:subClassOf`).

Each regime is based on the notion of an interpretation, which maps the terms in an RDF graph to a set, and then defines some set-theoretical conditions on the set. The intuition is that RDF describes resources and relationships between them, where interpretations form a bridge from syntactic terms to the resources and relationship they denote. We now define a simple interpretation.

Definition 2.10 (Simple interpretation). A *simple interpretation* is a 4-tuple $\mathcal{I} = (Res, Prop, Ext, Int)$ where Res is a set of resources; $Prop$ is a set of properties that represent types of binary relations between resources (not necessarily disjoint from Res); Ext maps properties to a set of pairs of resources, thus denoting the extension of the relations; and Int maps terms in $\mathbf{IL}$ to $Res \cup Prop$, i.e., maps terms in the RDF graph to the resources and properties they describe. With respect to blank nodes, let $A : \mathbf{B} \to Res$ be a function that maps blank nodes to resources, and let Int_A denote a version of Int that maps terms in $\mathbf{ILB}$ to $Res \cup Prop$ using A for blank nodes. We say that $\mathcal{I}$ is a *model* of an RDF graph G if and only if there exists a mapping A such that for each $(s, p, o) \in G$, it holds that $Int(p) \in Prop$ and $(Int_A(s), Int_A(o)) \in Ext(Int(p))$.

Here, the existential semantics of blank nodes is covered by the existence of an auxiliary function A, which is not part of the actual interpretation.

Simple entailment of RDF graphs can then be defined in terms of the models of each graph.

Definition 2.11 (Simple entailment). An RDF graph G simple-entails an RDF graph H, denoted $G \models H$, if and only if every model of G is also a model of H.

Intuitively speaking, if $G \models H$, then H says nothing new over G, or in other words if we hold G to be true, then we must hold H to be true as a logical consequence. If two RDF graphs entail each other, then we state that they are simple-equivalent. In other words, semantically speaking, both graphs contain the same information.

Definition 2.12 (Simple equivalence). An RDF graph G is simple-equivalent with an RDF graph H, denoted $G \equiv H$, if and only if every model of G is a model of H and every model of H is a model of G (in other words, $G \models H$ and $H \models G$).

LEMMA 2.13. *Simple equivalence ($\equiv$) is an equivalence relation.*

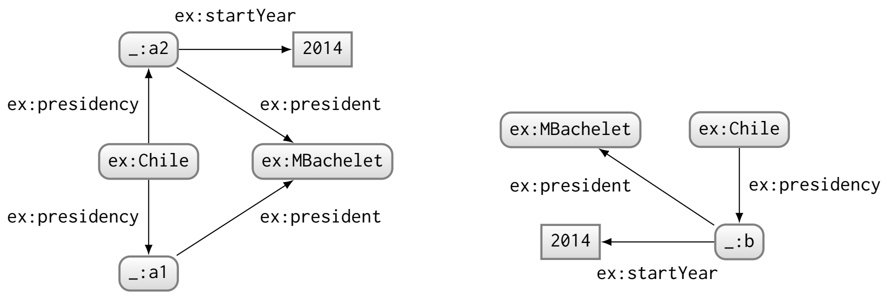
PROOF. $G \equiv H$ if and only if the sets of models of G and H are equal. Since set-equality is an equivalence relation, so is $\equiv$. □

REMARK 3. *If $G \cong H$, then $G \equiv H$.* □

Given that we currently deal exclusively with the simple semantics of RDF, for brevity, henceforth, we may refer to interpretations, models, entailment, equivalence, etc., without qualification, where we implicitly refer to the simple semantics.

An important question is how isomorphism and equivalence are different for RDF graphs. This is perhaps best illustrated with an example.

Example 2.14. Take the following two RDF graphs with G on the left and H on the right. First of all, we ask does $G \models H$ hold (i.e., is every model of G also a model of H)?



Let's say $\mathcal{I} = (Res, Prop, Ext, Int)$ is a model of G, where for the purposes of generality we give no further details. Since the set of ground terms in H is a subset of G, then Int maps ground terms in H to $Res \cup Prop$ in the same manner as for G. Let A denote an auxiliary mapping of blank nodes for G such that for each $(s, p, o) \in G$ it holds that $Int(p) \in Prop$ and $(Int_A(s), Int_A(o)) \in Ext(Int(p))$; in other words, A witnesses that $\mathcal{I}$ is a model of G. Now let μ denote a blank node mapping such that $\mu(\texttt{\_:b}) = \texttt{\_:a2}$. Then, for each $(s, p, o) \in H$, it holds that $Int(p) \in Prop$ and $(Int_{A\circ\mu}(s), Int_{A\circ\mu}(o)) \in Ext(Int(p))$, where $A \circ \mu$ is a valid auxiliary mapping that satisfies the condition for $\mathcal{I}$ to be a model of H. Hence, any model of G is also a model of H, or in other words, $G \models H$. Intuitively speaking, if we first map _:b in H to _:a2 in G, then we see that H contains a subset of the information of G.

Now let us ask: does $H \models G$ hold? This time consider a blank node mapping μ such that $\mu(\texttt{\_:a1}) = \texttt{\_:b}$ and $\mu(\texttt{\_:a2}) = \texttt{\_:b}$. Using a similar argument as above but in the opposite direction, we can now see that any model of H must be a model of G: the ground terms of G are a subset of H, and if A is an auxiliary mapping that witnesses that $\mathcal{I}$ is a model of H, then $A \circ \mu$ witnesses that $\mathcal{I}$ is a model of G. Put another way, looking just at G, if we map _:a1 to _:a2, we do not change the meaning of G, and we end up with an isomorphic copy of H, and hence we may see that $H \models G$.

Given that $G \models H$ and $H \models G$, we have that $G \equiv H$, even though $G \ncong H$. Though both graphs are structurally different, from a semantic perspective both graphs have the same set of models under RDF's simple semantics: they imply each other. □

In this example, we saw that an RDF graph can be equivalent to a smaller graph: in other words, an RDF graph can contain redundant triples that add no new information in semantic terms. This gives rise to the notion of a *lean RDF graph* [21], which is one that does not contain such redundancy.

Definition 2.15 (Lean RDF graph). An RDF graph G is considered *lean* if and only if there does not exist a proper subgraph $G' \subset G$ such that $G' \models G$. Otherwise we call G *non-lean.*

Example 2.16. Referring back to Example 2.14, H is lean. However, let G' denote the set of triples of G that do not mention `_:a1`. We can see that $G' \models G$ (with the same line of reasoning as to why $H \models G$). In terms of explaining why G is non-lean, intuitively the graph can be read as making the following claims:

- Chile has a presidency with MBachelet as president and startYear 2014.
- Chile has a presidency with MBachelet as president.

Under simple semantics, the second claim is considered redundant. □

3 THEORETICAL SETTING

The goal of this paper is to propose and develop algorithms to compute two canonical forms for RDF: a canonical form with respect to isomorphism and a canonical form with respect to equivalence. Having defined these notions in the previous section, we now present some theoretical results that show these to be hard problems in general; we also present high-level strategies as to how these canonical forms could be computed. We first focus on isomorphism and later discuss equivalence.

3.1 Isomorphism

To begin, we wish to establish that RDF isomorphism is in the same complexity class as the related and more well-established problem of graph isomorphism for undirected graphs. In fact, this result is folklore and was hinted at previously by other authors, such as Carroll [9], but to the best of our knowledge, no formal proof of this result was given. First we need some preliminary definitions.

Definition 3.1 (Undirected graph). An *undirected graph* $\mathsf{G} = (V, E)$ is a graph were V is the set of *vertexes*, $E \subseteq V \times V$ is the set of *edges*, and $(v, v') \in E$ if and only if $(v', v) \in E$ (or in other words, the edges are unordered pairs).

Definition 3.2 (Graph isomorphism). Given two undirected graphs $\mathsf{G} = (V_\mathsf{G}, E_\mathsf{G})$ and $\mathsf{H} = (V_\mathsf{H}, E_\mathsf{H})$, these graphs are *isomorphic*, denoted $\mathsf{G} \cong \mathsf{H}$, if and only if there exists a bijection $\beta : V_\mathsf{G} \to V_\mathsf{H}$ such that $(v, v') \in E_\mathsf{G}$ if and only if $(\beta(v), \beta(v')) \in E_\mathsf{H}$. In this case, we call β an *isomorphism.*

The graph isomorphism problem – of deciding if $\mathsf{G} \cong \mathsf{H}$ – is GI-complete: a class that belongs to NP but is not known to be equivalent to NP-complete nor to permit polynomial-time solutions. We now give a result stating that the RDF isomorphism problem – of deciding for two RDF graphs if $G \cong H$ – is in the same complexity class as graph isomorphism.

Theorem 3.3. *Given two RDF graphs G and H, determining if $G \cong H$ is GI-complete.*

A proof for Theorem 3.3 – which originally appeared in the conference version of this work [27] – is provided in Appendix A. From Theorem 3.3, we can conclude that it is not known if there exists a polynomial-time algorithm for RDF isomorphism: the existence of such an algorithm would imply GI = P, solving a long-open problem in Computer Science. In a recent result, Babai [3] proved that the graph isomorphism problem can be solved in quasi-polynomial time[3], where although a polynomial-time algorithm has not yet been found, an algorithm better than exponential is now known; hence GI is contained within QP: the class of problems solvable in quasi-polynomial time. These results extend to RDF isomorphism per Theorem 3.3. But these theoretical results refer to worst-case analyses, where a previous result by Babai et al. [4] showed that isomorphism for randomly generated graphs can be performed efficiently using a naive algorithm. Hence, despite the possibility of non-polynomial worst cases, many practical algorithms exist to solve graph isomorphism quickly for many cases.

In fact, our goal here is not the solve the isomorphism decision problem but rather to tackle the harder problem of computing an iso-canonical version of an RDF graph.

Definition 3.4 (Iso-canonical RDF mapping). Let M be a mapping from an RDF graph to an RDF graph. M is *iso-canonical* if and only if $M(G) \cong G$ for any RDF graph G and $M(G) = M(H)$ if and only if $G \cong H$ for any two RDF graphs G and H.

We know that computing an iso-canonical form of an RDF graph is GI-hard since it can be used to solve the RDF isomorphism problem: we can compute the iso-canonical form of both graphs and check if the results are equal. Hence we are unlikely to find polynomial-time algorithms.

Before we continue, let us establish an initial iso-canonical mapping that is quite naive and impractical, but establishes the idea of how such a form can be achieved: define a total ordering of RDF graphs and for a set of pairwise isomorphic RDF graphs, define the lowest such graph (following certain fixed criteria) to be canonical.

Definition 3.5 (κ-mapping). Assume a total ordering of all RDF terms, triples and graphs.[4] Assume that κ is a blank node bijection that labels all k blank nodes in a graph G from `_:b1` to `_:b`k, and let $\kappa(G)$ denote the image of G under κ. Furthermore, let $\mathcal{K}$ denote the set of all such κ-mappings valid for G. We then define the minimal κ-mapping of an RDF graph G as $M_\kappa(G) = \min\{\kappa(G) \mid \kappa \in \mathcal{K}\}$.

Proposition 3.6. *M_κ is an iso-canonical mapping.*

Proof. First, since we have a total ordering of graphs, there is a unique graph $\min\{\kappa(G) \mid \kappa \in \mathcal{K}\}$, and hence M_κ is a mapping from RDF graphs to RDF graphs.

Second, $M_\kappa(G) \cong G$ since we only relabel blank nodes: κ is a blank node bijection.

We are now left to prove that $M_\kappa(G) = M_\kappa(H)$ if and only if $G \cong H$.

($M_\kappa(G) = M_\kappa(H)$ *implies* $G \cong H$) We know that $M_\kappa(G) \cong G$ and $M_\kappa(H) \cong H$. If we are given $M_\kappa(G) = M_\kappa(H)$, then we have that $G \cong M_\kappa(G) \cong M_\kappa(H) \cong H$, and since $\cong$ is an equivalence relation, if follows that $G \cong H$.

($G \cong H$ *implies* $M_\kappa(G) = M_\kappa(H)$) Suppose the result does not hold and there exist G and H such that $G \cong H$ and (without loss of generality) $M_\kappa(G) > M_\kappa(H)$. Since $G \cong H$, there exists a blank node bijection μ such that $\mu(G) = H$. Let κ be the mapping such that $\kappa(H) = M_\kappa(H)$. Now $\kappa \circ \mu(G) = M_\kappa(H)$. Since $\kappa \circ \mu$ is a κ-mapping for G and $M_\kappa(G) > \kappa \circ \mu(G)$, we arrive at a contradiction per the definition of M_κ since it does not use the minimum κ-mapping for G. □

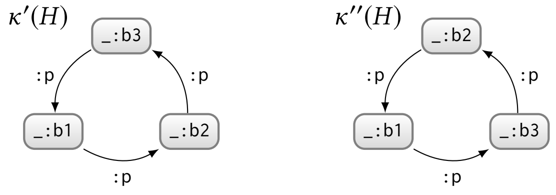
Example 3.7. Take graph H from Example 2.9. We have a total of $3! = 6$ possible κ-mappings (with only blank nodes from bnodes(H) in their domain and blank nodes from {`_:b1`, `_:b2`, `_:b3`} in their codomain) as follows.

[3]On January 9, 2017, the result was briefly retracted as a bug was found in the proof. The result was reasserted a few days later when a fix was found. See http://people.cs.uchicago.edu/~laci/update.html.

[4]For example, we can consider terms ordered syntactically, triples ordered lexicographically, and graphs ordered such that $G < H$ 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 $t' < t$.

$\kappa(\cdot)$	`_: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 κ-mappings only produce two distinct graphs, where the first, fourth and fifth mappings correspond to $\kappa'(H)$ and the rest to $\kappa''(H)$, as follows:



Assuming a typical lexical ordering (as per Footnote 4), $M_\kappa(H) = \kappa'(H)$. Importantly, one could (bijectively) relabel `_:c`, `_:d`, `_:e` in the original graph without affecting the result: the output would be the same for any $M_\kappa(H')$ such that $H \cong H'$. □

This discussion suggests a correct and complete brute force algorithm to compute an iso-canonical form for any RDF graph G: search all κ-mappings of G for one that gives the minimum such graph. However, such a brute-force process is unnecessary and naive: by applying a more fine-grained total ordering on RDF graphs, we can use a similar principle to find an iso-canonical form in a much more efficient way. Such an algorithm will be presented later in Section 4.

3.2 Equivalence

As previous discussed in Section 2.3, two RDF graphs are equivalent if they entail each other. Towards an initial procedure for deciding if two RDF graphs entail each other, we have the following result:

Theorem 3.8. *$G \models H$ if and only a blank node mapping μ exists such that $\mu(H) \subseteq G$ [18, 21].* □

Example 3.9. Referring back to Example 2.14, as a witness that $G \models H$ holds, we have a mapping μ such that $\mu($`_:b`$) =$ `_:a2` and $\mu(H) \subset G$. As a witness that $H \models G$ holds, we have a mapping μ' such that $\mu($`_:a1`$) = \mu($`_:a2`$) =$ `_:b` and $\mu(G) = H$.

For argument's sake, let us consider an RDF graph H' derived from H by replacing `_:b` with an IRI `:I`. We no longer have a mapping μ that witnesses $G \models H'$; in fact, $G \not\models H'$. However, for $H' \models G$, we have the mapping $\mu($`_:a1`$) = \mu($`_:a2`$) =$ `:I`. □

In traditional graph terms, finding a blank node mapping μ that witnesses such an entailment relates closely with the notion of graph homomorphism.

Definition 3.10 (Graph homomorphism). Given two undirected graphs $\mathsf{G} = (V_\mathsf{G}, E_\mathsf{G})$ and $\mathsf{H} = (V_\mathsf{H}, E_\mathsf{H})$, a mapping $\beta : V_\mathsf{G} \to V_\mathsf{H}$ is a *homomorphism* from G to H if and only if $(v, v') \in E_\mathsf{G}$ implies $(\beta(v), \beta(v')) \in E_\mathsf{H}$.

The problem of deciding simple entailment between RDF graphs – i.e., given two RDF graphs G and H, to decide if $G \models H$ – falls into the same complexity class as the problem of deciding graph homomorphism between undirected graphs – i.e., given two undirected graphs G and H, deciding if there exists a homomorphism from G to H. Furthermore, it is known that deciding whether or not $G \equiv H$ is also NP-complete.

Theorem 3.11. *Deciding if* $G \models H$ *is NP-complete [18].* □

Theorem 3.12. *Deciding if* $G \equiv H$ *is NP-complete [18].* □

However, much like the case for isomorphism, although the worst-case analysis suggests that it is unlikely we will find a general polynomial-time algorithm for determining equivalence, for many real-world RDF graphs, efficient procedures are possible. From work by Pichler et al. [47], for example, we know that the entailment problem is fixed-parameter tractable; more specifically, they demonstrated the following result.

Definition 3.13 (Blank node graph). Let B denote a mapping from RDF graphs to undirected graphs $\mathrm{B}(G) = (V, E)$ such that $V = \mathrm{bnodes}(G)$, and such that $(b_1, b_2) \in E$ (and thus $(b_2, b_1) \in E$) if and only if $b_1 \in V$, $b_2 \in V$ and there exists a $p \in \mathbf{I}$ such that $(b_1, p, b_2) \in G$. We call $\mathrm{B}(G)$ the *blank node graph* of G.

Theorem 3.14. $G \models H$ *can be decided in time* $O(m^2 + mn^{2k})$, *where* $n = |G|$, $m = |H|$ *and* $k = \mathrm{tw}(H) + 1$, *where* $\mathrm{tw}(H)$ *is the treewidth of* $\mathrm{B}(H)$. *[47].* □

Intuitively speaking, treewidth is a measure of how cyclical an undirected graph is. For example, an acyclical graph has a treewidth of 1, any graph with cycles has a treewidth of at least 2, an n-clique has a treewidth of $n - 1$, etc. Thus, for example, if an RDF graph G does not contain blank nodes cycles (i.e., if $\mathrm{B}(G)$ is acyclic), then entailment can be checked in time $O(m^2 + mn^2)$. Likewise, for RDF graphs with blank node graphs that have bounded treewidth, entailment can be decided in polynomial time. In previous work, in a survey of 3.8 million Web documents containing RDF with blank nodes, we found 3.3 million connected components of blank nodes, of which 62.3% were acyclical (treewidth of 1), where 37.7% had a treewidth of 2; cases with higher treewidth were very rare, with the highest treewidth encountered being 6. The conclusion here is that although we would expect a general algorithm to have an exponential running time in the worst-case, entailment – and by extension equivalence – should be efficiently computable for most real-world cases.

Again however, our goal is not to decide entailment or equivalence, but rather to address the harder problem of computing an equi-canonical form for any RDF graph.

Definition 3.15 (Naive equi-canonical RDF mapping). Let M be a mapping from RDF graphs to RDF graphs. M is a *naive equi-canonical* mapping if an only if, for any two RDF graphs G and H, $M(G) \equiv G$, and $M(G) = M(H)$ if and only if $G \equiv H$.

We know that computing a naive equi-canonical form of an RDF graph is NP-hard since it can be used to solve equivalence. However, per the previous arguments, we hope to be able to find a general algorithm that is efficient for most real-world cases.

Our general approach is to *lean* an RDF graph – i.e., given an RDF graph G, compute G' such that G' is lean and $G' \models H$ – and then compute an iso-canonical form of the leaned graph. As such, we aim for a more specific form of mapping.[5]

Definition 3.16 (Equi-canonical RDF mapping). A naive equi-canonical RDF mapping M is a *lean equi-canonical mapping* (or simply an *equi-canonical mapping*) if $M(G)$ is lean for any RDF graph G.

[5]We call the original version of the mapping "naive" as it appears that any practical such mapping would produce a lean graph.

With respect to leaning RDF graphs, there are various problems one could consider that relate to the more well-established notion of cores in undirected graphs.[6]

Definition 3.17 (Graph core). An undirected graph G is a *core* if each homomorphism from G to itself is an isomorphism.

In other words, a graph is a core if there is no homomorphism from G to a sub-graph of itself. There is thus a direct correspondence between the notion of leanness for RDF graphs, and the notion of cores in undirected graphs [23]. Hence, following the precedent of Gutierrez et al. [18], we can define the core of an RDF graph analogously:

Definition 3.18 (RDF core). Let G be an RDF graph. Let G' be a lean RDF graph such that $G' \subseteq G$ and $G' \models G$. We call G' a *core* of G. We denote by $\mathrm{core}(G)$ a core of G.

In fact, here we are slightly abusing notation since $\mathrm{core}(G)$ is not uniquely defined: a non-lean graph may have multiple cores. However, there is a unique core modulo isomorphism, where we assume $\mathrm{core}(G)$ selects any such graph.

THEOREM 3.19. $\mathrm{core}(G)$ *is unique up to isomorphism [18].* □

THEOREM 3.20. $G \equiv H$ *if and only if* $\mathrm{core}(G) \cong \mathrm{core}(H)$ *[18].* □

These results lead us almost directly to the following result, which is important to establish the correctness of our approach for computing equi-canonical forms:

THEOREM 3.21. *Let M be an iso-canonical mapping. Then $M \circ \mathrm{core}$ is an equi-canonical mapping.*

PROOF. We need to check that $M \circ \mathrm{core}$ meets all of the following conditions for an equi-canonical mapping:

- $M(\mathrm{core}(G)) \equiv G$ for any RDF graph G:
  $G \equiv \mathrm{core}(G) \cong M(\mathrm{core}(G))$, and hence $G \equiv M(\mathrm{core}(G))$ (from Lemma 2.13).
- $M(\mathrm{core}(G)) = M(\mathrm{core}(H))$ IF and ONLY IF $G \equiv H$ for any RDF graphs G and H:
  (IF) If $M(\mathrm{core}(G)) = M(\mathrm{core}(H))$, then $G \equiv \mathrm{core}(G) \cong M(\mathrm{core}(G)) = M(\mathrm{core}(H)) \cong \mathrm{core}(H) \equiv H$, and hence $G \equiv H$ (from Lemma 2.13).
  (ONLY IF) If $G \equiv H$, then from Theorem 3.20, $\mathrm{core}(G) \cong \mathrm{core}(H)$, and since M is iso-canonical, $M(\mathrm{core}(G)) = M(\mathrm{core}(H))$.
- $M(\mathrm{core}(G))$ is lean for any RDF graph G:
  $\mathrm{core}(G)$ is lean by definition and given that $M(\mathrm{core}(G)) \cong \mathrm{core}(G)$, it holds that $M(\mathrm{core}(G))$ is also lean.

Thus $M \circ \mathrm{core}$ satisfies all conditions for an equi-canonical mapping. □

In other words, once we have a practical algorithm for computing an iso-canonical form for RDF graphs, we can lean the RDF graph and then apply the iso-canonical form to arrive at an equi-canonical form. The question then is how much harder the problem becomes when we add leaning as a pre-processing step. We have the following complexity results from the literature:

THEOREM 3.22. *Deciding if an RDF graph G is lean is coNP-complete [18].* □

A homomorphism μ, such that $\mu(G) \subset G$, acts as an efficiently-verifiable witness that G is *not* lean, which, along with a reduction from the analogous coNP-complete procedure for undirected graphs [23] to the RDF case [18], means that deciding if an RDF graph is lean is coNP-complete.

On the other hand, deciding if one graph is the lean version of another has different complexity:

[6]This is not to be confused with the notion of k-cores proposed by Seidman [53].

Theorem 3.23. *Given two RDF graphs G and H, deciding if* $\mathrm{core}(G) \cong H$ *(i.e., if H is a core of G) is DP-complete [18].* □

The DP-complete class, loosely speaking, refers to when separate calls to an NP-complete procedure and a coNP-complete procedure are required. To determine if H is a core of G, we can check if H is lean (coNP-complete) and in parallel check if there is a homomorphism from G to H (NP-complete); this establishes membership in DP. Conversely, DP-hardness can be shown by reduction from the analogous DP-complete problem for undirected graphs [12] to the RDF case [18].

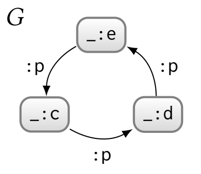
From these results, we know that computing the equi-canonical form of an RDF graph is coNP-hard, since we can use such a procedure to efficiently decide if G is lean: compute the equi-canonical form of G and and test if it has fewer triples than G.

The question we now must ask is what procedure we can apply to lean an RDF graph. With respect to the computation of cores, in the setting of incomplete databases, papers by Fagin et al. [12], Gottlob and Nash [16], Pichler and Savenkov [48], Marnette et al. [39], Mecca et al. [43], etc., propose polynomial-time algorithms for computing *core solutions* under restricted data exchange settings. However, in a survey of such approaches by Savenkov [50], it appears that the most practical approach is to compute a core solution directly rather than computing a solution and then computing its core, where the most practical post-processing algorithm (one that computes a solution and then the core) was proposed by Gottlob [15] and is based on hypertree decompositions. However, it is unclear how such approaches could be applied to our scenario and how practical they might be (especially since we are not aware of any implementations thereof). This might be an interesting question for future work.

Instead, we propose a much more "direct" algorithm for computing the lean version of an RDF graph, which exploits the relation between conjunctive query answering (aka. basic graph pattern matching) over RDF graphs and computing cores [28]. On a high level, conjunctive query answering involves computing all homomorphisms from a query (a graph pattern similar in structure to an RDF graph) to a target RDF graph. Each such homomorphism constitutes a solution to the conjunctive query. If we use well-known conjunctive query answering techniques to – in a manner we will define later where we consider blank nodes as query variables – match an RDF graph G against itself, we can compute all *endomorphisms* within G: all homomorphisms from G to itself. At least one such endomorphism must thus be a homomorphism from G to a core of G; we call such an endomorphism a *core endomorphism*. In fact, given the set of all endomorphisms of G, it is trivial to identify the core endomorphisms: as we will prove momentarily, the core endomorphisms of G are the endomorphisms that map to the fewest unique blank nodes in G.

Definition 3.24 ((Core) Endomorphism). Given an RDF graph G, we denote by $\mathrm{End}(G)$ all blank node mappings with domain $\mathrm{terms}(G)$ that map G to a subgraph of itself: $\mathrm{End}(G) = \{\mu \mid \mu(G) \subseteq G$ and $\mathrm{dom}(\mu) = \mathrm{terms}(G)\}$. We call $\mathrm{End}(G)$ the *endomorphisms* of G. If an endomorphism μ is not an automorphism – i.e., if $\mu(G) \subset G$ – we call it a *proper endomorphism*. We denote by $\mathrm{CEnd}(G)$ the set of all mappings in $\mathrm{End}(G)$ that map to the fewest unique blank nodes, which we call the *core endomorphisms* of G; more formally, $\mathrm{CEnd}(G)$ is the set of all $\mu \in \mathrm{End}(G)$ such that there does not exist $\mu' \in \mathrm{End}(G)$ such that $|\mathrm{codom}(\mu') \cap \mathbf{B}| < |\mathrm{codom}(\mu) \cap \mathbf{B}|$.

Example 3.25. We look at the endomorphisms for three RDF graphs: G, H and I. Automorphisms are shown in grey and core endomorphisms are shaded.

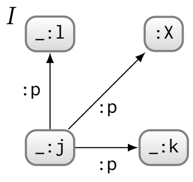


End(G)

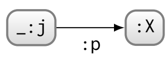
$\mu(\cdot)$	_:c	_:d	_:e
	_:c	_:d	_:e
=	_:d	_:e	_:c
	_:e	_:c	_:d

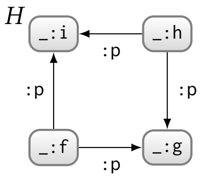
End(I)

$\mu(\cdot)$	_:j	_:k	_:l
	_:j	_:k	_:l
	_:j	_:l	_:k
	_:j	_:k	_:k
	_:j	_:l	_:l
=	_:j	_:k	:X
	_:j	_:l	:X
	_:j	:X	_:k
	_:j	:X	_:l
	_:j	:X	:X

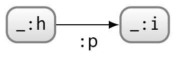


$\mu'(I)$





$\mu'(H)$



End(H)

$\mu(\cdot)$	_:f	_:g	_:h	_:i
	_:f	_:g	_:h	_:i
	_:f	_:i	_:h	_:g
	_:h	_:g	_:f	_:i
	_:h	_:i	_:f	_:g
	_:f	_:g	_:h	_:g
	_:f	_:g	_:f	_:i
	_:f	_:i	_:h	_:i
=	_:f	_:i	_:f	_:g
	_:h	_:g	_:f	_:g
	_:h	_:g	_:h	_:i
	_:h	_:i	_:f	_:i
	_:h	_:i	_:h	_:g
	_:f	_:g	_:f	_:g
	_:f	_:i	_:f	_:i
	_:h	_:g	_:h	_:g
	_:h	_:i	_:h	_:i

We see that all of G's endomorphisms are automorphisms (note: $\mu(G) = G$ if and only if $\text{dom}(\mu) = \text{codom}(\mu)$); as will prove momentarily, this implies that G is lean.

In the case of H, we see 16 endomorphisms returned, 4 of which are automorphisms (in grey) that map four blank nodes bijectively to four blank nodes; 8 of which are not automorphisms, but map to three unique blank nodes; and 4 of which map to only two unique blank nodes, and are thus the core endomorphisms (CEnd(H)). Taking μ' to be the last (core) homomorphism listed for End(H), we show the results of applying $\mu'(G)$; we will prove momentarily that since $\mu' \in \text{CEnd}(H)$, we can conclude that $\mu'(H)$ is a core of H.

Finally in the case of I, we consider a node with an IRI `:X`. In this case, we have 9 endomorphisms in total, including 2 automorphisms and 1 core endomorphism. Letting μ' be the unique core endomorphism in this case, $\mu'(I)$ shows the result of sending both `_:j` and `_:k` to `:X` in I.

With respect to the relation to query answering, note that each set End(G), End(H) and End(I) is just the set of solutions considering that graph as a query against itself, where blank nodes are considered query variables. To compute the set End(I), for example, in SPARQL [19] we could consider the following query:

```
SELECT DISTINCT ?j ?k ?l WHERE { ?j :p ?k , ?l , :X . }
```

The solutions returned for this query would effectively be as seen in End(I). In any case, we do not use SPARQL nor SPARQL engines for various reasons (e.g., SPARQL does not preserve blank nodes labels in solutions), but the idea of querying an RDF graph G against itself and finding the solution with the fewest unique blank nodes may help to conceptualise the approach we will take to compute the core of G.[7] □

[7]In SPARQL 1.1, it would even be possible to compute a "core solution" directly by finding the solution(s) with the fewest blank nodes using a combination of `BIND` and `IF` to increment a chain of fresh variables by 1 if a binding for a given variable is a blank node, and then applying an `ORDER BY` with `LIMIT 1`.

Having given the intuition in Example 3.25 of the relation between endomorphisms and leanness/cores, we first give some remarks that follow generally for endomorphisms and automorphisms before formalising this relationship.

Remark 4. $\mathrm{Aut}(G) \subseteq \mathrm{End}(G)$ *holds for any RDF graph* G. □

Remark 5. *If* $\mu \in \mathrm{End}(G)$ *and* $\mu' \in \mathrm{End}(G)$*, then* $\mu \circ \mu' \in \mathrm{End}(G)$. □

Now we establish the relationship between endomorphisms and leanness.

Lemma 3.26. *G is lean if and only if* $\mathrm{End}(G) = \mathrm{Aut}(G)$.

Proof. (G is lean *implies* $\mathrm{End}(G) = \mathrm{Aut}(G)$) If G is lean, it implies that there is no subgraph $G' \subset G$ such that $G' \models G$, which implies that there is no μ such that $\mu(G) \subseteq G'$ (or $\mu(G) \subset G$). Hence, if G is lean, for any $\mu \in \mathrm{End}(G)$, it holds that $\mu(G) = G$, and hence $\mathrm{End}(G) = \mathrm{Aut}(G)$.

($\mathrm{End}(G) = \mathrm{Aut}(G)$ *implies* G is lean) Likewise, if $\mathrm{End}(G) = \mathrm{Aut}(G)$, then there can be no subgraph $G' \subset G$ such that $\mu(G) \subseteq G'$ (since μ would be in $\mathrm{End}(G) \setminus \mathrm{Aut}(G)$), and hence there is no subgraph G' such that $G' \models G$, and hence G is lean. □

We can now show that any core endomorphism of G maps G to a core of G.

Theorem 3.27. *If* $\mu \in \mathrm{CEnd}(G)$*, then* $\mu(G)$ *is a core of* G.

Proof. Recall that G' is a core of G if and only if $G' \models G$ and G' is lean. Given any blank node mapping μ, then $\mu(G) \models G$ trivially holds with μ as a witness (see Theorem 3.8). We are thus left to prove that if $\mu \in \mathrm{CEnd}(G)$, then $\mu(G)$ is lean.

Suppose a case where $\mu \in \mathrm{CEnd}(G)$ but $\mu(G)$ is not lean. Given that $\mu(G)$ is not lean, there must exist a mapping $\mu' \in \mathrm{End}(\mu(G)) \setminus \mathrm{Aut}(\mu(G))$ (per Lemma 3.26); in other words, there must exist a proper endomorphism μ'. However, in that case, $\mu' \circ \mu$ must also be in $\mathrm{End}(G)$ (per Remark 5), and $\mu' \circ \mu$ must map to strictly fewer unique blank nodes in its codomain than μ, which contradicts with $\mu \in \mathrm{CEnd}(G)$. Hence the theorem holds. □

Finally we provide a lemma that will be useful later, indicating that one can apply proper endomorphisms to G in an iterative manner to get to the core.

Lemma 3.28. *If* $\mu \in \mathrm{End}(G)$*, then for any* $\mu' \in \mathrm{CEnd}(\mu(G))$*, it holds that* $\mu'(\mu(G))$ *is a core of* G.

Proof. We need to prove that $\mu'(\mu(G))$ is lean and $\mu'(\mu(G)) \models G$. The premise $\mu' \in \mathrm{CEnd}(\mu(G))$ combined with Theorem 3.27, implies that $\mu'(\mu(G))$ is a core of $\mu(G)$ and must thus be lean. With respect to showing $\mu'(\mu(G)) \models G$, since $\mu \in \mathrm{End}(G)$, then $\mu(G) \models G$, and likewise since $\mu' \in \mathrm{End}(\mu(G))$, then $\mu'(\mu(G)) \models \mu(G)$, and thus $\mu'(\mu(G)) \models G$. Hence the lemma holds. □

In summary, we have shown that to compute a core of a graph, it suffices to find an endomorphism that maps to the fewest blank nodes (which we call a core endomorphism). The endomorphism can be considered as any solution of posing the RDF graph as a query against itself considering its blank nodes as variables. We have also shown that we can apply the process iteratively, meaning that we can compute the core of an RDF graph by iteratively finding and simplifying the graph according to any sequence of proper endomorphisms (an endomorphism that removes some blank nodes). These results serve as the basis for our algorithm for leaning – and for computing the equi-canonical form of an RDF graph – discussed in Section 5.

4 ISO-CANONICAL ALGORITHM: LABELLING BLANK NODES

We will now propose an algorithm for computing the iso-canonical form of an RDF graph, which implements a function M from RDF graphs to RDF graphs such that $M(G) = M(G')$ if and only if G is isomorphic with G'. This has a number of applications, including hashing or signing graphs, detecting duplicate (more specifically, isomorphic) RDF graphs from a large collection without needing pairwise comparison, as well as a form of Skolemisation [11] where blank nodes are mapped to IRIs in a deterministic (more specifically, isomorphism-preserving) manner.

As discussed in Section 3, this problem is GI-hard, and thus according to recent results for graph isomorphism, we can expect worst-cases with at least quasi-polynomial behaviour [3] (unless we resolve the open problem of GI = P). In fact, our algorithm will have exponential worst-case behaviour. However, our conjecture is that the worst-cases predicted by theory are not likely to occur often in practice, and that it is possible to derive algorithms that are efficient in practice.

The algorithm we propose is based on the idea of considering a total ordering of RDF graphs, where we can then choose the lowest RDF graph in every isomorphic partition of RDF graphs to be the canonical graph for that partition. We previously illustrated this idea in Example 3.7 using κ-mappings to define the ordering; however, this particular ordering of RDF graphs is very generically defined and thus the search space of κ-graphs is quite broad—this was only intended to illustrate the high-level idea. In practice, we can leverage the ground information in an RDF graph – i.e., IRIs and literals – to specify a much more constrained blank node labelling that greatly reduces the search space in many cases. In cases where ground information is not enough to provide distinguishing labels to blank nodes, we can adapt ideas from well-known algorithms for graph isomorphism to complete the labelling of blank nodes.

4.1 Hash-based labelling of blank nodes

In the graph isomorphism literature [41, 42, 49], an *invariant* is a property of a node in a graph that is preserved in an isomorphism. For example, given an isomorphism between two undirected graphs G and G′, a node of degree d in G must map to a node with degree d in G′, and hence node degree is invariant under isomorphism. Such invariants help narrow the search space of possible isomorphisms. In the case of RDF, ground terms attached to blank nodes additionally offer a rich invariant that can often be highly selective. Thus, rather than in the naive case of κ-mappings where any blank node can be given any of the available labels, we can instead begin by assigning each blank node a deterministic hash based on the ground terms surrounding it.

We highlight that similar hashing schemes have been proposed in the literature by, for example, Arias-Fisteus et al. [2], Kasten et al. [32] and Lantzaki et al. [34] for similar purposes; however, to the best of our knowledge, these approaches are not sound and complete with respect to isomorphism. On the other hand, while the earlier work of Longley and Sporny [37] on canonically labelling blank nodes indeed appears to be sound and complete with respect to isomorphism, a formal proof of such has yet to be provided. We will discuss such works in more detail in Section 7.

In Algorithm 1, we propose such an iterative hashing scheme for initially labelling blank nodes based on the ground terms that surround them in the RDF graph.

Lines 2–7 A map of terms to hashes is initialised. IRIs and literals are assigned unique static hashes. Blank nodes hashes are initialised to zero and will be computed iteratively in subsequent steps. For now, we assume perfect hashing without clashes, where practical hashing issues will be discussed later in Section 4.4.

Lines 9–17 A hash for each blank node is computed iteratively per its inward and outward edges in the RDF graph. The function $hashTuple(\cdot)$ will compute an order-dependant hash of its inputs, and is used to compute the hash of an edge based on the hash of the predicate, value (the subject for inward edges or the object for outward edges) and the direction. The symbols '+' (outward) and '-' (inward) are used to distinguish edge directions. The hash of the value is static in the case of IRIs or literals; otherwise it is the hash of the blank node from the previous iteration. The function *hashBag*(·) computes hashes in a commutative and associative way over its inputs and is used to aggregate the hash across all edges.

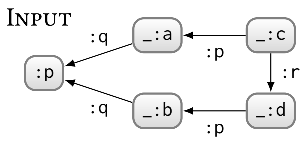
Line 18 The computed hashes form a partition of blank nodes. The hash of each blank node changes in every iteration. The loop terminates when either (i) the hash-based partition of terms does not change in an iteration, or (ii) no two terms share a hash.

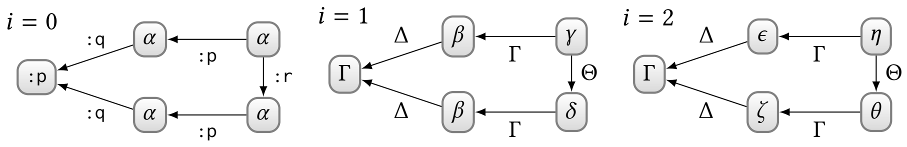
Algorithm 1 Deterministically hashing blank nodes

```
1:  function HASHBNODES($G$)                                  ▷ $G$ is any RDF graph
2:      initialise $\mathrm{hash}_0[]$                         ▷ a map from terms to hashes
3:      for $x \in \mathrm{terms}(G)$ do                        ▷ all terms in $G$
4:          if $x \in \mathbf{B}$ then
5:              $\mathrm{hash}_0[x] \leftarrow 0$               ▷ an initial hash
6:          else
7:              $\mathrm{hash}_0[x] \leftarrow hashTerm(x)$     ▷ a static hash based on the string of the term
8:      $i \leftarrow 0$
9:      repeat
10:         $i$++
11:         initialise $\mathrm{hash}_i[]$ with $\mathrm{hash}_{i-1}[]$          ▷ copy map
12:         for $(b, p, o) \in G : b \in \mathbf{B}$ do                         ▷ $o \in \mathbf{IBL}$
13:             $c \leftarrow hashTuple(\mathrm{hash}_{i-1}[o], \mathrm{hash}_{i-1}[p], +)$   ▷ $hashTuple(\cdot)$ is order-dependent
14:             $\mathrm{hash}_i[b] \leftarrow hashBag(c, \mathrm{hash}_i[b])$    ▷ $hashBag(\cdot)$ is commutative and associative
15:         for $(s, p, b) \in G : b \in \mathbf{B}$ do                         ▷ $s \in \mathbf{IB}$
16:             $c \leftarrow hashTuple(\mathrm{hash}_{i-1}[s], \mathrm{hash}_{i-1}[p], -)$
17:             $\mathrm{hash}_i[b] \leftarrow hashBag(c, \mathrm{hash}_i[b])$
18:     until $(\forall x, y : \mathrm{hash}_i[x] = \mathrm{hash}_i[y]$ iff $\mathrm{hash}_{i-1}[x]=\mathrm{hash}_{i-1}[y])$ or $(\forall x, y : \mathrm{hash}_i[x]=\mathrm{hash}_i[y]$ iff $x = y)$
19:     return $\mathrm{hash}_i[]$                                  ▷ final map of terms to hashes
```

We now give an example of how Algorithm 1 works.

Example 4.1. In practice, blank nodes are assigned numeric hashes, but here we will use Greek letters to represent hashes, where upper-case denotes static hashes (for IRIs and literals) and lower-case denotes dynamic hashes (for blank nodes). The iteration is given by i.





In the initial state ($i = 0$), an initial hash α is assigned to all blank nodes and static hashes to all IRIs and literals. In the iterations that follow, hashes are computed for blank nodes according to their neighbourhood; e.g., for `_:d` at $i = 1$, the hash will be computed based on its previous hash and its inward and outward edges as follows:

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

The process continues until either each blank node is assigned a distinct hash, or the partition of blank nodes under hashes does not change. In this example, all blank nodes have a distinct hash for $i = 2$, and thus the process terminates. □

Algorithmic characteristics. We now show that assuming a perfect hashing scheme, the algorithm terminates in a bounded number of iterations, and that the computed hashes preserve isomorphism. We start with a lemma which states that if two blank nodes are assigned different hashes in a given iteration, they will remain distinguished in subsequent iterations.

Lemma 4.2. *Assuming a perfect hashing scheme, in Algorithm 1, if* $\mathsf{hash}_i[x] \neq \mathsf{hash}_i[y]$, *then* $\mathsf{hash}_j[x] \neq \mathsf{hash}_j[y]$ *for* $j \geq i$.

Proof. A perfect hashing scheme implies that given distinct inputs, a distinct output will be produced: in other words, clashes do not occur. We can then prove the lemma by induction, with the base case that $\mathsf{hash}_i[x] \neq \mathsf{hash}_i[y]$. For $i + 1$, we know that $\mathsf{hash}_{i+1}[x]$ takes $\mathsf{hash}_i[x]$ as input whereas $\mathsf{hash}_{i+1}[y]$ does not (and that no other hash in the computation of $\mathsf{hash}_{i+1}[y]$ can clash with $\mathsf{hash}_i[x]$). Hence we have that $\mathsf{hash}_{i+1}[x] \neq \mathsf{hash}_{i+1}[y]$. The lemma is thus true under the induction hypothesis. □

Lemma 4.3. *Algorithm 1 terminates in* $\Theta(\beta)$ *iterations in the worst case for an RDF graph* G *where* $\beta := |\mathrm{bnodes}(G)|$.

Proof. Hashes form a partition of bnodes(G). Algorithm 1 terminates if the partition does not change. Per Lemma 4.2, partitions can only split. Hence only $\beta - 1$ splits can occur before the unit partition is reached. The tight asymptotic bound is given, for example, by the path graph of the form $x_1 \xrightarrow{p} x_2 \xrightarrow{p} \ldots \xrightarrow{p} x_n$ ($x_i \in \mathbf{B}$ for $1 \leq i \leq n$), where $\lceil \frac{n-1}{2} \rceil$ iterations are needed to terminate. □

Assuming for simplicity that $\mathsf{hash}_i[\cdot]$ has constant insert/lookup and linear copy performance, then Algorithm 1 runs in $\Theta(\beta \cdot (\tau + \gamma))$ in a worst-case analysis, where $\beta := |\mathrm{bnodes}(G)|$, $\tau := |\mathrm{terms}(G)|$ and $\gamma := |G|$. In terms of space, the graph and two maps of size τ are stored. Ground triples can be pre-filtered to reduce space and time.

Next we show a specific correctness property of the algorithm, which intuitively states that two blank nodes that could be mapped to each other through an isomorphism will be assigned the same hash by the algorithm. In fact, this is quite a weak result (since, e.g., it would trivially hold if we gave all blank nodes the same hash), but it will be extended upon later for the complete algorithm.

Lemma 4.4. *Let* $G \cong H$ *be two isomorphic RDF graphs and let* hash_G *denote the map from RDF terms to hashes produced by Algorithm 1 for* G*, and likewise* hash_H *for* H*. 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* $\mathsf{hash}_G(b) = \mathsf{hash}_H(c)$.

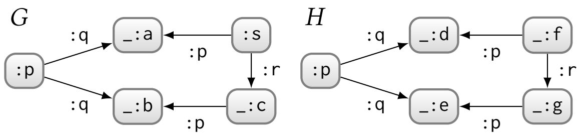
Proof. If G and H are isomorphic, then they differ only in the strings that are used to label blank nodes. With respect to blank node label strings, no part of Algorithm 1 performs computation based on these label strings (other than to identify the blank node, which is preserved under isomorphism). Thus, relabelling blank nodes in the input graph in a one-to-one manner cannot affect the output.

Although an RDF graph is unordered, Algorithm 1 must read the triples in *some* order, where the order for G and H may be different. However, this cannot affect the output since the commutativity and associativity of $hashBag(\cdot)$ ensures order-independent hashing over sets of triples.

Hence neither the strings on the blank node labels nor the implicit order in which triples are read can affect the output of Algorithm 1, and we conclude it will give consistent hashes for isomorphic graphs, and that the lemma holds. □

Splitting the graph. When there are disconnected sub-graphs of blank nodes in the input RDF graph, the hashing of terms in those sub-graphs may continue longer than necessary. Specifically, let's say we use hash_G as the basis of a Skolemisation scheme to generate a blank node mapping μ_G, where output hashes are 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 4.5. In the following, the hashing of blank nodes in G will require one fewer iteration than H or $G + H$ to terminate.



Hence the hashes of graph G would no longer correspond with the hashes of the corresponding sub-graph of $G + H$ since blank node hashes change in every iteration. □

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

Definition 4.6 (Blank node split). For an RDF graph G, let $\mathrm{G} := (V, E)$ denote an undirected graph where $V := G$ and a pair of triples (t, u) is in E if and only if $\{t, u\} \subseteq G$ and $\mathrm{bnodes}(\{t\}) \cap \mathrm{bnodes}(\{u\}) \neq \emptyset$. Let $t \sim_{\mathrm{G}} v$ denote that t and v are reachable from each other in G, and let $G/{\sim_{\mathrm{G}}}$ denote the partition of G based on reachability in G. We define a *blank node split* of G as $\mathrm{split}(G) := \{G' \in G/{\sim_{\mathrm{G}}} \mid G' \text{ is not ground}\}$. We define the *ground split* of G as the graph that contains all (and only the) ground triples of G.

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. For instance, in Example 4.5, split(G) results in two subgraphs: one with all triples mentioning either _:b or _:c (or both), and another with the two other triples (mentioning _:a). On the other hand, split(H) = H.

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.

Line 2 We compute split(G) using a standard Union–Find algorithm, which runs in $O(n \log n)$ worst-case for n the number of triples in G.

Lines 3–5 The results of each split are computed by calling Algorithm 1 and unioned: two splits cannot disagree on the hash of a given term since hashes for IRIs and literals are static and no blank node can appear in two splits.

Line 6 The union of hashes for blank node splits are returned. If needed, we can also add the hashes of constants in the ground split not appearing elsewhere to align the results with Algorithm 1; however, most of the time we are only interested in the hashes of blank nodes.

Algorithm 2 Hashing blank nodes splits independently

```
1: function hashBnodesPerSplit($G$)                   ▷ $G$ a non-ground RDF graph
2:     $\{G_1, \ldots, G_n\} \leftarrow$ split($G$)    ▷ use, e.g., a Union–Find algorithm
3:     initialise hash
4:     for $1 \leq i \leq n$ do
5:         hash ← hash ∪ hashBnodes($G_i$)            ▷ hashBnodes(·) calls Algorithm 1
6:     return hash                                    ▷ final hashes
```

When compared with Algorithm 1, Algorithm 2 may produce different hashes for blank nodes due to running fewer iterations over split graphs (per Example 4.5). The added cost of the split computation is offset by the potential to reduce iterations when hashing the blank nodes of individual splits; likewise splitting the graph in such a manner offers the potential to parallelise the processing of each split. In general, however, we provide Algorithm 2 not for performance reasons, but rather to offer users a choice in how individual splits should be labelled, as per Example 4.5.

4.2 Distinguishing the remaining blank nodes

Algorithm 1 described in the previous section would suffice to compute distinct hashes for each blank node in many real-world RDF graphs. However, the algorithm does not guarantee to produce unique hashes. For example, in the case of non-trivial automorphisms, Algorithm 1 would give sets of blank nodes with the same hash.

Example 4.7. Referring back to graphs G and H from Example 2.9, if we input G into Algorithm 1, after each iteration, all blank nodes will have the same hash and the process will terminate after two iterations since the partition does not change: there is no way that Algorithm 1 can distinguish the two blank nodes. The exact same behaviour will be observed for H: the non-trivial automorphisms in G and H make it impossible for Algorithm 1 to distinguish those blank nodes in the graph. □

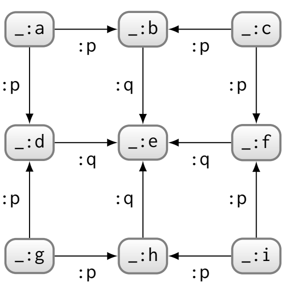
Instead, when Algorithm 1 fails to distinguish all blank nodes, we need to revert back to using some sort of total ordering over RDF graphs such that, for each partition of isomorphic RDF graphs, we can define the lowest such graph in each partition as a canonical version. Instead of using the naive κ-method outlined previously, we can use a more specific style of ordering – inspired by standard graph isomorphism methods like Nauty [41] – to narrow the search space in a deterministic manner.

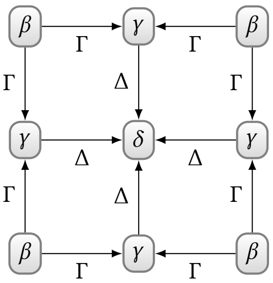
First we assume a set of totally ordered hashes $\mathbf{H}$. Let hash be a map from blank nodes to hashes computed by Algorithm 1; these hashes form a partition of blank nodes where blank nodes with the same hash are in the same set of the partition.

Definition 4.8 (Hash partition). Given a set of blank nodes B and a mapping hash from B to hashes, let $\backsim$ denote an equivalence relation between blank nodes such that $b_1 \backsim b_2$ if and only if $b_1, b_2 \in B$ and hash$[b_1]$ = hash$[b_2]$. We define a *hash partition* P of a set of blank nodes B with respect to hash 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 a part B' *trivial* if $|B'| = 1$; otherwise we call it *non-trivial.* We call a partition P *fine* if it has only trivial parts, *coarse* if it has only one part, and *intermediate* if it is neither coarse nor fine.

In Example 4.1, we gave an example where Algorithm 1 produces a fine hash partition, and in Example 4.7 we discussed two cases where the hash partitions will be coarse. We now introduce a running example where Algorithm 1 rather produces an intermediate hash partition.

Example 4.9. On the left hand side, we have an RDF graph that is similar to a 2D-grid.[8] On the right hand side, we have the result of applying Algorithm 1 over this graph, using the same conventions as in Example 4.1 (i.e., representing static hashes with upper-case Greek letters and blank node hashes with lower-case Greek letters).





The hash partition of the graph is $P = \{\{$`_:a`, `_:c`, `_:g`, `_:i`$\}, \{$`_:e`$\}, \{$`_:b`, `_:d`, `_:f`, `_:h`$\}\}$. The part $\{$`_:e`$\}$ is trivial. The partition is neither fine (since it contains non-trivial parts) nor coarse (since it contains more than one part); hence it is intermediate. □

A fine partition corresponds to having a unique hash for each blank node. These hashes can be used to compute iso-canonical blank node labels and ultimately an iso-canonical version of the RDF graph. Our goal is to thus use a deterministic process – avoiding arbitrary choices and use of blank node label strings – to compute a set of hashes that correspond to a fine partition of blank nodes. However, since there may be no obvious way to deterministically choose from the blank nodes in a non-trivial partition, we must try all equal choices. Exploring different alternatives will result in a *set* of fine partitions, where we can define the partition that, intuitively speaking, produces the lowest possible graph G (after mapping hashes to blank nodes in a fixed manner) as providing the canonical labelling for the blank nodes in G.

To start with, if we have an intermediate partition, we must choose a non-trivial part to begin distinguishing blank nodes. By defining an order over partitions, we can do so in a deterministic manner that helps narrow the search space.

Definition 4.10 (Ordered partition). Given $P = \{B_1, \ldots, B_n\}$, a hash partition of B with respect to hash, we call a sequence of sets of blank nodes $\mathsf{P} := (B_1, \ldots, B_n)$ an *ordered partition* of B with respect to hash.

To deterministically compute an initial P from an input P and hash, 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 $\mathsf{hash}(b') < \mathsf{hash}(b'')$ for $b' \in B'$ and $b'' \in B''$ (recall that all elements of B' have the same hash and likewise for B'', and that B' and B'' are disjoint). In other words, we start with the smallest parts first, breaking ties by selecting the part with the lower hash. We will then start distinguishing blank nodes in the first non-trivial part of P. It is worth noting that different graph isomorphism algorithms apply similar ideas but try different orderings of partitions, where some orderings work well for certain classes of graph whereas other orderings work better for other classes of graph [42]. For the moment, we explore smaller partitions first; however, it may be interesting to investigate the effect of varying the ordering of partitions, for example to start with the largest partitions first.

Algorithm 3 presents the method we use to compute a fine partition from a non-fine partition by recursively distinguishing blank nodes in its lowest non-trivial part, knowing the same part will

[8]This is an RDF version of similar examples used in the graph isomorphism literature, for example, by McKay and Piperno [42].

likewise be selected for all isomorphic graphs.[9] The lowest such graph found in this process – by labelling blank nodes according to the computed hashes – represents the iso-canonical version of the RDF graph G.

Lines 2–3 The algorithm first calls Algorithm 1 to compute an initial set of hashes for blank nodes (note that depending on the user's preference, we could equivalently call Algorithm 2). The results are used to compute an initial hash partition.

Line 4–5 If the initial hashes produce a fine partition, we can just label the graph according to the initial hashes and return the result as the iso-canonical version.

Line 6 If the initial hashes produce a non-fine partition, we make the first call to the function that will recursively distinguish blank nodes.

Lines 9–15 The algorithm first orders the partition and then selects the lowest non-trivial part. For each blank node in this part, the algorithm first marks the blank node with a new hash and then calls Algorithm 1 initialised with the intermediate hashes. Based on the results of this call, a new partition is computed and ordered.

Lines 16–18 If the new partition is fine, then the algorithm labels the blank nodes in G according to the hashes, and checks to see if the resulting graph is lower than the lowest that has been found before; if so, it is set as the lowest.

Line 19 If the new partition is not fine, blank nodes in the lowest non-trivial part will be distinguished recursively.

Line 20 Once all of the alternatives have been explored, the lowest graph found is returned as the iso-canonical version of G.

Algorithm 3 Computing an iso-canonical version of an RDF graph

```
1:  function isoCanonicalise($G$)                                   ▷ $G$ an RDF graph
2:      hash ← hashBnodes($G$)                                      ▷ calls Algorithm 1
3:      compute hash partition $P$ of bnodes($G$) w.r.t. hash
4:      if $P$ is fine then
5:          $G_\perp \leftarrow label(G, \text{hash})$               ▷ we are done: generate blank node labels from hash
6:      else $G_\perp \leftarrow$ distinguish($G$, hash, $P$, $\emptyset$, bnodes($G$))   ▷ start recursively distinguishing blank nodes
7:      return $G_\perp$                                            ▷ canonical graph is the lowest graph found
8:  function distinguish($G$, hash, $P$, $G_\perp$, $B$)            ▷ $G_\perp$: smallest hash-labelled graph found thus far
9:      P ← order $P$ by $\leq$                                     ▷ order by smallest parts first; use hash to break ties
10:     $B' \leftarrow$ lowest non-trivial part of P
11:     for $b \in B'$ do
12:         hash′ ← $clone$(hash)
13:         hash′[$b$] ← $hashTuple$(hash′[$b$], ‘@’)               ▷ ‘@’: an arbitrary distinguishing marker
14:         hash″ ← hashBnodes($G$, hash′)                          ▷ abusing notation: initialise $\text{hash}_0$ in Algorithm 1 with hash′
15:         compute partition $P'$ of $B$ w.r.t. hash″
16:         if $P'$ is fine then
17:             $G_C \leftarrow label(G, \text{hash}'')$             ▷ generate blank node labels from hash″
18:             if $G_\perp = \emptyset$ or $G_C < G_\perp$ then $G_\perp \leftarrow G_C$   ▷ keep track of lowest graph generated
19:         else $G_\perp \leftarrow$ distinguish($G$, hash″, $P'$, $G_\perp$, $B$)   ▷ recursively refine next non-trivial part
20:     return $G_\perp$                                            ▷ the lowest graph found on this branch of recursion
```

[9] Here we simplify the presentation of the conference version [27], which spoke about refinements of partitions similar in principle to Nauty [42], where the idea is that when computing a fine partition, the order of elements from the input partition is somehow preserved and parts are split "in situ"; the order of elements in the output partition then represents the identifier for the node. In our case, however, we can simply use our hashes as identifiers, which means we have no need for this (rather awkward) notion of in-situ partition refinements.

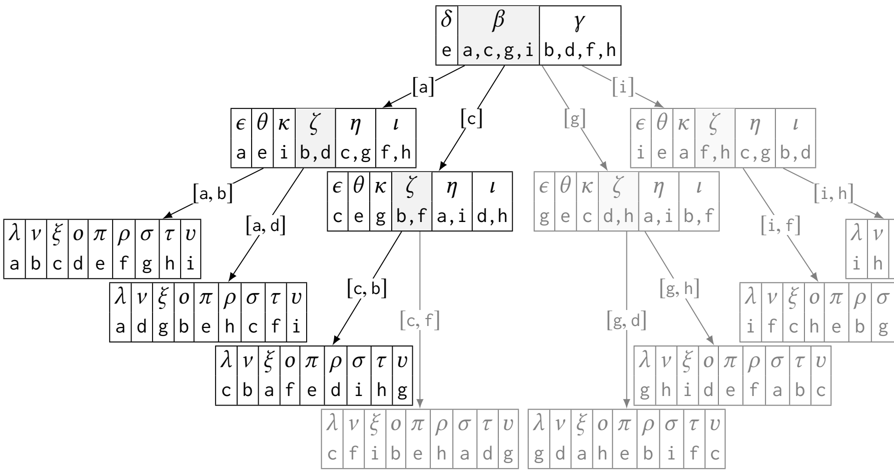


Fig. 1. Search tree for Example 4.9

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 $\mathrm{P} \in V$ be a node in the tree and $L(\cdot, \mathrm{P})$ denote the label of its inlink (or an empty list for the root node). Let B be the first non-trivial part of P (if P is fine, it has no children). Then, for every $b \in B$, an edge extends from P with label $L(\cdot, \mathrm{P}) \parallel [b]$ (where $\parallel$ denotes concatenation) to the refinement of P computed by distinguishing b and rerunning the hashing 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 4.11. In the following, to ease presentation, we will assume that blank nodes with lower labels happen to be hashed with lower values, but this should be considered a coincidence: in reality, hashes have nothing to do with blank node labels.

Referring back to Example 4.9, the ordered partition resulting from the initial hashing would be $\mathrm{P} = (\{\texttt{\_:e}\}, \{\texttt{\_:a}, \texttt{\_:c}, \texttt{\_:g}, \texttt{\_:i}\}, \{\texttt{\_:b}, \texttt{\_:d}, \texttt{\_:f}, \texttt{\_:h}\})$ assuming $\beta < \gamma$. So now we distinguish blank nodes in the first non-trivial part. Let us denote by $\mathrm{P}_{[b_1,\ldots,b_n]}$ the ordered partition derived by first distinguishing b_1, then b_2, etc. The following left-hand-side graph shows the result of distinguishing `_:a` and running Algorithm 1 to fixpoint: $\mathrm{P}_{[\texttt{\_:a}]} = (\{\texttt{\_:a}\}, \{\texttt{\_:e}\}, \{\texttt{\_:i}\}, \{\texttt{\_:b}, \texttt{\_:d}\}, \{\texttt{\_:c}, \texttt{\_:g}\}, \{\texttt{\_:f}, \texttt{\_:h}\})$.

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

$P_{[\_:a]}$ is still not a fine partition. Hence we proceed by distinguishing one of the blank nodes in the first non-trivial part {_:b, _:d} of $P_{[\_:a]}$. If we distinguish, e.g., _:b, then $P_{[\_:a,\_:b]}$ = ({_:a}, {_:b}, {_:c}, {_:d}, {_:e}, {_:f}, {_:g}, {_:h}, {_:i}}) would be a fine partition, as shown to the right.

However, when we chose _:a from the part {_:a, _:c} and later when we chose _:b from the part {_:b, _:d}, we did so arbitrarily: the hashes for the blank nodes within each part are the same and we cannot use blank node labels to choose one since that choice would not be deterministic with respect to different isomorphic graphs. Hence, we must try all alternatives. Having computed $P_{[\_:a,\_:b]}$, next the algorithm would try $P_{[\_:a,\_:d]}$, then $P_{[\_:c,\_:b]}$, $P_{[\_:c,\_:f]}$, and so on. For example, below on the left we give the graph resulting from $P_{[\_:c]}$ and on the right the graph resulting from $P_{[\_:c,\_:b]}$ = ({_:c}, {_:b}, {_:a}, {_:f}, {_:e}, {_:d}, {_:i}, {_:h}, {_:g}}), where the parts are ordered by hash as before; e.g., _:c in $P_{[\_:c,\_:b]}$ has the same hash as _:a in $P_{[\_:a,\_:b]}$.

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

Figure 1 depicts the search tree with the ordered partition resulting at every step, where the first non-trivial part for each intermediate partition – from which a blank node will be distinguished – is shaded (we omit blank node underscores for formatting purposes; other greyed-out parts of the tree will be discussed later). This tree is traversed by the algorithm in a depth-first manner and 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 again $P_{[\_:a,\_:b]}$ and $P_{[\_:c,\_:b]}$ above. The latter graph is the same as the former but read right-to-left instead of left-to-right. A closer examination of Figure 1 will reveal a similar pattern for all leafs: some read left-to-right, some right-to-left, some read along rows, some read along columns, but once we map the hashes to blank nodes, all edges in the resulting RDF graphs, like (_:ρ, :p, _:σ), will appear for every leaf in this particular example. Hence the labelling given by any leaf will serve as an iso-canonical labelling of the RDF graph (though we do not know in advance). □

In the previous example, we saw that all leaves produce the same labelled graph. It is then interesting to consider in what cases such leaves would produce different labelled graphs; in fact, finding such examples is non-trivial. While some such examples have been found [7], the constructions involved are quite intricate. We expect in most such cases that only one distinct labelled graph is found [7].

Algorithmic characteristics. We briefly remark on two properties of the algorithm.

Theorem 4.12. *For any RDF graph G, Algorithm 3 terminates.*

Proof. The search tree, though exponential in the number of blank nodes in the graph, is finite. The search follows a standard depth-first recursion on the search tree. □

The algorithm is indeed exponential, but as we will see momentarily, we can implement some optimisations to avoid exploring the entire search tree in every case (though the worst case behaviour still remains exponential [45]).

Theorem 4.13. *Given an RDF graph G, the graph $G_\perp$ produced by Algorithm 3 is a canonical version of G with respect to isomorphism.*

Proof. Algorithm 3 only terminates once all blank nodes in G are distinguished in the output and blank nodes are generated from hashes in a one-to-one manner: hence the algorithm produces a blank node bijection and thus G and $G_\perp$ are isomorphic.

Using a similar argument as Lemma 3.6, we can see that Algorithm 3 provides a deterministic total ordering of the isomorphic partition in which G is contained that does not consider blank node labels. Hence $G_\perp$ is also canonical. □

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 such graphs are themselves isomorphic, we can distinguish all blank nodes in said graphs by hashing with a fixed fresh symbol to separate them, taking the union with the ground triples for output. Another simpler option would be to run Algorithm 3 directly over the full RDF graph, though this may lead to a larger search tree.

4.3 Pruning the search tree using automorphisms

In Example 4.11, we saw that exploring the eight leaves of the search tree was redundant since they all resulted in equal graphs: this was due to symmetries in the graph caused by automorphisms, where there are two large *orbits* – sets of nodes mappable by an automorphism – namely {_:a, _:c, _:g, _:i} and {_:b, _:d, _:f, _:h}.

The Nauty algorithm – and similar labelling methods such as Traces [49] or Bliss [30] – track 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 4.14. Taking Example 4.11, consider the two ordered partitions

$$\mathsf{P}_{[a,b]} = (\{\texttt{\_:a}\}, \{\texttt{\_:b}\}, \{\texttt{\_:c}\}, \{\texttt{\_:d}\}, \{\texttt{\_:e}\}, \{\texttt{\_:f}\}, \{\texttt{\_:g}\}, \{\texttt{\_:h}\}, \{\texttt{\_:i}\})$$
$$\mathsf{P}_{[a,d]} = (\{\texttt{\_:a}\}, \{\texttt{\_:d}\}, \{\texttt{\_:g}\}, \{\texttt{\_:b}\}, \{\texttt{\_:e}\}, \{\texttt{\_:h}\}, \{\texttt{\_:c}\}, \{\texttt{\_:f}\}, \{\texttt{\_:i}\})$$

Note that one partition is reading across rows and the other down columns; they both produce the same graph. Since both partitions generate the same labelled graph, we can assert the following bidirectional automorphism:

$$\texttt{\_:a} \leftrightarrow \texttt{\_:a}, \texttt{\_:b} \leftrightarrow \texttt{\_:d}, \texttt{\_:c} \leftrightarrow \texttt{\_:g}, \texttt{\_:d} \leftrightarrow \texttt{\_:b}, \texttt{\_:e} \leftrightarrow \texttt{\_:e}$$
$$\texttt{\_:f} \leftrightarrow \texttt{\_:h}, \texttt{\_:g} \leftrightarrow \texttt{\_:c}, \texttt{\_:h} \leftrightarrow \texttt{\_:f}, \texttt{\_:i} \leftrightarrow \texttt{\_:i}$$

If we apply this blank node mapping (from left-to-right or right-to-left) to the input RDF graph, we will end up with the same RDF graph. □

Automorphisms can thus be used to prune the search tree [42] where we employ one such strategy. Let $\mathsf{P}_{[b_1,\ldots,b_n]}$ be a node in the tree with the children $\mathsf{P}_{[b_1,\ldots,b_n,b']}$ (which we shall denote by P') and $\mathsf{P}_{[b_1,\ldots,b_n,b'']}$ (which we shall denote P'') derived by subsequently distinguishing b' and b'', respectively. Assume that P' has been visited and we are considering visiting P'' next. If we can find an automorphism μ that is the *pointwise stabiliser* for all the blank nodes in $\{b_1, \ldots, b_n\}$ (i.e., $\mu(b) = b$ for all $b \in \{b_1, \ldots, b_n\}$) such that $\mu(b') = \mu(b'')$, then we need not visit P'', since we can map the explored path $[b_1, \ldots, b_n, b']$ to the unexplored path $[b_1, \ldots, b_n, b'']$ by a known automorphism, meaning that we are exploring a different symmetry of the same graph and we will find the same leaves under P'' as we did for P' [42].

Example 4.15. 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, _:f]). Each such leaf in the search tree proposes a distinct labelling for the graph where each node is assigned a unique label based on its corresponding hash; but if we consider the labelled graphs that would be produced by [_:a, _:b] and [_:a, _:d], these would lead to precisely the same graphs (the same set of triples), which suggests that there is an automorphism mapping _:d ↔ _:b, _:g ↔ _:c, and so forth, based on having the same hash. After discovering that the [_:a, _:b], [_:a, _:d], [_:c, _:b] and [_:c, _:f] leaves form the same labelled graph in this manner, a variety of automorphisms are thus known, per Example 4.14, by mapping the leaf nodes (in the same column) from the following:

	λ	ν	ξ	o	π	ρ	σ	τ	υ
[_:a, _:b]	_:a	_:b	_:c	_:d	_:e	_:f	_:g	_:h	_:i
[_:a, _:d]	_:a	_:d	_:g	_:b	_:e	_:h	_:c	_:f	_:i
[_:c, _:b]	_:c	_:b	_:a	_:f	_:e	_:d	_:i	_:h	_:g
[_:c, _:f]	_:c	_:f	_:i	_:b	_:e	_:h	_:a	_:d	_:g

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, _:b] ⟶ [_:a, _:d], which gives _:c ⟶ _:a, _:b ⟶ _:d, _:a ⟶ _:g, _:f ⟶ _:b, 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, and so we can terminate the process.

In fact, going back a little, as hinted at by Figure 1, in theory we need not have visited [_:c, _:f] either. When considering visiting [_:c, _:f], 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. □

However, naively materialising and indexing the entire automorphism group as it is discovered would consume prohibitive amounts of space for certain graphs: given a graph with β blank nodes, the number of automorphisms may approach $\beta!$. 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, _:f]: 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, _:f] 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., [42] 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.

4.4 Hashing and globally-unique labels

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, we may wish to avoid collisions. For this reason, we may wish to compute hashes for blank nodes that, instead of being locally unique in the graph (like for example the κ-mapping example

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}

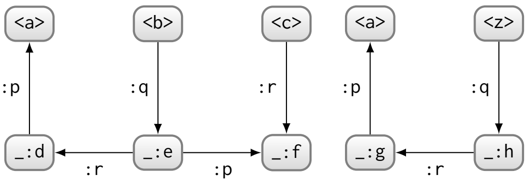
earlier), are rather globally unique, meaning that a given label will only appear in an isomorphic graph. Unfortunately, for reasons we discuss presently, such a guarantee is practically speaking impossible; however, with an appropriate choice of hash function, we can guarantee that such clashes are extremely unlikely in practice.

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 to the nearest whole number where necessary); in particular, we present the approximate number of inputs needed to reach the given 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 [29]. Even in Base64, a 512- or 256-bit hash would produce a 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 inputs in the trillions – are negligible.[10]

Aside from hash lengths, when we wish to compute globally unique labels we require an additional step, as the following example demonstrates.

Example 4.16. Consider the following two RDF graphs:



[10]In certain applications, it may also be important to have cryptographically secure hashes, in which case stronger functions than MD5 or SHA1 may be required to ensure, e.g., pre-image and collision resistance.

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. □

In fact, the only way to ensure globally unique blank nodes would be if each blank node encodes the information from the input graph itself in a lossless manner; otherwise, given that graphs are composed of terms from infinite sets, some other (non-isomorphic) graph must exist that would have to contain that term. Obviously such a lossless encoding would lead to infeasibly long labels. Our solution, instead, is to compute a hash of the entire canonicalised graph – which as per Theorem 4.13, 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 incorporates a hash signature unique (modulo hash collisions) to the structure of the entire graph.

5 EQUI-CANONICAL FORM

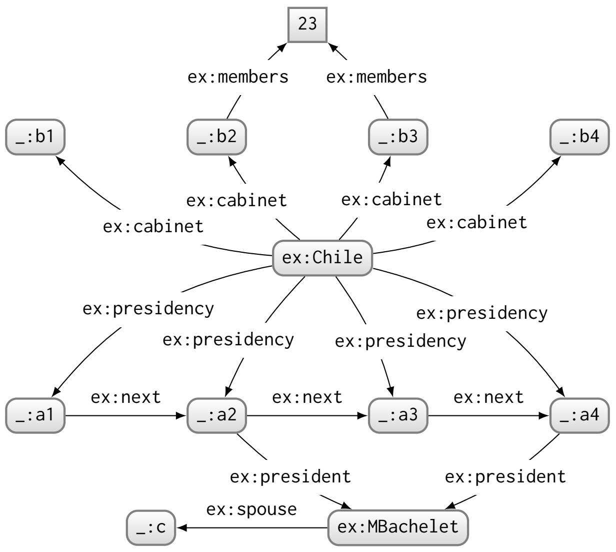
We will now present algorithms for computing the equi-canonical form of an RDF graph, which gives a function M such that $M(G) = M(G')$ if and only if $G \equiv G'$. This allows to canonicalise graphs not in terms of structure, but in terms of semantics.

As per the discussion in Section 3, the problem we face is coNP-hard: thus we can (probably) expect worst-cases with non-polynomial behaviour. However, we again conjecture that it is feasible to develop an algorithm that is efficient for the vast majority of real-world cases. Our strategy, as mentioned before, is to compute the lean version of an RDF graph and then apply the iso-canonical procedure discussed in the previous section. Thus, in this section, we focus on designing efficient algorithms for leaning RDF graphs. As likewise discussed in Section 3, we conceptualise this problem as looking for a core endomorphism: a mapping of the blank nodes in an RDF graph to the graph itself that has the fewest possible unique blank nodes in its codomain.

5.1 Removing initially redundant blank nodes

In the first step, we wish to remove blank nodes that are obviously causing non-leanness in the graph. These are blank nodes whose edges are a subset of another term in the RDF graph and are thus relatively trivial to identify, where all triples that mention such blank nodes are candidates for removal from the graph. However, we must be careful if two or more blank nodes have the same set of edges, in which case we may need to preserve one such blank node and its triples.

Example 5.1. Let us consider the following spider-like RDF graph:

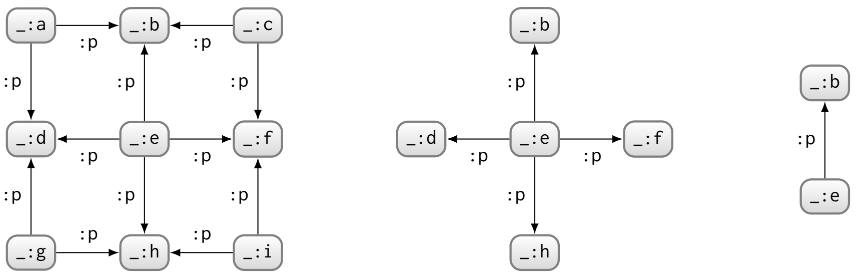


In terms of edges, the blank nodes `_:b1` and `_:b4` are covered by `_:b2` and `_:b3`. Likewise `_:b2` covers `_:b3` and `_:b3` covers `_:b2`. Hence we can consider either {`_:b1`, `_:b2`, `_:b4`} or {`_:b1`, `_:b3`, `_:b4`} as a set of redundant blank nodes; note we cannot remove both `_:b2` and `_:b3` since their edges are equal and they are not covered by the edges for another term; thus we must choose one such blank node to keep. Assuming we consider {`_:b1`, `_:b2`, `_:b4`} redundant, we can remove all triples mentioning these blank nodes from G and produce a graph closer to the core of G.

Looking at the other blank nodes, none of their edges are covered by another term; hence they will not be removed from the graph. □

In fact, in some cases, this process can be iterative.

Example 5.2. Let us consider a similar graph as presented in Example 4.9, but where all edge labels are the same and some edges are reversed.



In the leftmost graph, the edges of nodes `_:a`, `_:c`, `_:g` and `_:i` are subsets of the edges of the node `_:e`; for example, `_:a` connects to `_:b` and `_:d` with label `:p`, as does `_:e`. Hence we can remove the four corner blank nodes to arrive at the middle graph. But now the edges of `_:b`, `_:d`, `_:f` and `_:h` are covered by each other (which they were not before), so we can remove all but one such blank node, as per the rightmost graph. □

We detail the process of identifying and removing redundant blank nodes in Algorithm 4:

Lines 2–3 A copy of the input graph is made that will be refined in subsequent steps; the iterative process then begins.

Lines 4–6 The algorithm loads the set of all edges for all terms. Note that '+' is again a fixed symbol used to denote an outward edge and '−' the analogue for an inward edge. This representation makes it easier to compare sets of edges.

Lines 7–8 We initialise a set X that will store all blank nodes that have already been seen as well as all ground terms; we use this to ensure that whenever a blank node b has the same set of edges as another term x, we will only remove b if x is a ground term or a blank node that has already been seen. We also allocate R: an initially empty set of redundant blank nodes.

Lines 9–13 For each blank node, we check each term in the graph. If the edges of the blank node are a proper subset of the edges for another term, we can consider the blank node redundant. Otherwise if the set of edges of the blank node is equal to a ground term or a blank node that was previously seen, we can also remove it. Although not shown in the algorithm, in practice, we do not check each term for each blank node, but rather only check candidate terms that share the most selective edge of that blank node (using an index on edges).

Lines 14–15 We make a note of the size of the current RDF graph and then remove any redundant blank nodes identified.

Lines 16–17 If any triples are removed, the above process is repeated (see Example 5 for a case requiring two iterations); otherwise the current graph is returned.

Algorithm 4 Removing initial redundant blank nodes

```
1:  function REMOVEREDUNDANTBNODES($G$)                                   ▷ $G$ an RDF graph
2:      $G' \leftarrow G$
3:      do
4:          initialise edges
5:          for $x \in \text{terms}(G')$ do
6:              $\text{edges}[x] \leftarrow \{(p, o, +) \mid (x, p, o) \in G'\} \cup \{(p, s, -) \mid (s, p, x) \in G'\}$    ▷ store edges of $x$
7:          $X \leftarrow \text{terms}(G') \cap \mathbf{IL}$                 ▷ will store ground terms and blank nodes seen
8:          $R \leftarrow \emptyset$                                     ▷ will store redundant blank nodes
9:          for $x \in \text{bnodes}(G')$ do
10:             for $x' \in \text{terms}(G')$ do
11:                 if $\text{edges}[x] \subset \text{edges}[x']$ or ($\text{edges}[x] = \text{edges}[x']$ and $x' \in X$) then
12:                     $R \leftarrow R \cup \{x\}$ (break)                 ▷ $x$ is a redundant blank node
13:             $X \leftarrow X \cup \{x\}$                              ▷ add $x$ as a seen blank node
14:         $n \leftarrow |G'|$                                          ▷ used later to see if something changed
15:         $G' \leftarrow G' \setminus \{(s, p, o) \in G' \mid s \in R \text{ or } o \in R\}$   ▷ remove triples from $G$ with a redundant blank node
16:     while $n \neq |G'|$
17:     return $G'$
```

Algorithm characteristics. We present some performance and correctness results:

LEMMA 5.3. *Algorithm 4 terminates in $O(\beta)$ iterations (Line 17) in the worst case for $\beta = |\text{bnodes}(G)|$.*

PROOF. The process will iterate at Line 16 if and only if the graph is changed, which can only happen if a blank node is removed, which can only happen at most β times. □

Assuming linear lookups and insertions to edges, as a loose worst-case analysis, the algorithm runs in the order of $O(\beta^2\tau\gamma^2)$ (where $\beta := |\text{bnodes}(G)|$, $\tau := |\text{terms}(G)|$ and $\gamma := |G|$). However, it is important to note that this bound may not be tight and this analysis excludes the indexing and selectivity optimisations we apply.

LEMMA 5.4. *Let G be an RDF graph and G' be the result of applying Algorithm 4 to G. Then it holds that $G' \equiv G$.*

Proof. We need to prove that $G' \models G$ and $G \models G'$. Since $G' \subseteq G$, we have that $G \models G'$. In order to prove $G' \models G$, following Theorem 3.8, we can show that there exists a blank node mapping μ' such that $\mu'(G) \subseteq G'$. We will do so for the first iteration of the algorithm, which is sufficient for an induction argument.

Let μ map each x that passes the condition on Line 11 to the term x' that satisfies the condition,[11] and let μ be the identity for all other terms in G. Since the edges of x are a subset of the edges of $\mu(x)$, we know that if $(x, p, o) \in G$, then $(\mu(x), p, o) \in G$, and that if $(s, p, y) \in G$, then $(s, p, \mu(y)) \in G$, and hence if $(x, p, y) \in G$, then $(\mu(x), p, \mu(y)) \in G$. Hence we have that $\mu(G) \subseteq G$, and thus that μ is an endomorphism.

As a slight complication, note that with μ, x may map to x' such that x' maps to x'', where in G' we will remove x and x'. Thus we need to apply μ to a fixpoint to "skip" x' and map both x and x'' directly to x'' (assuming of course x'' maps to itself). Letting $\mu_1 := \mu$ and $\mu_{i+1} = \mu \circ \mu_i$, further let μ' be defined as the fixpoint μ_n for the least value of n such that $\mu_n = \mu_{n+1}$. Importantly, μ does not contain cycles other than identity loops – $\mu(x) = x'$ if and only if the edges of x are a subset of x', or the set of edges of x' are equal to x and either x' is a ground term or all other blank nodes with the same set of edges map to x', where proper set containment cannot contain cycles and only one blank node can be in the domain of μ with an equal set of edges – and hence the fixpoint is well-defined. Since μ' is the composition of endomorphisms, from Remark 5 it follow that μ' is itself an endomorphism: $\mu' \in \mathrm{End}(G)$.

Finally, we need to show that $\mu'(G) = G'$. Since μ' is, by definition, the closure of the subset relation on edges, we know that $\mu'(x) \neq x$ if and only if there exists a term x' such that x' has a proper superset of the edges of x or x' has the same set of edges but is chosen first; this is how Algorithm 4 computes the set of redundant blank nodes R (the base mapping μ of μ' was defined in terms of the same condition on Line 11 used to select R). Since μ' is an endomorphism, and since μ' is a fixpoint, removing triples mentioning blank nodes in R gives the same result as applying $\mu'(G)$.

With $\mu'(G) = G'$ for the first iteration, and applying the induction hypothesis to remaining iterations in consideration of Lemma 3.28, we complete the proof. □

We can show that if a blank node is not connected to another blank node, then Algorithm 4 will either remove the blank node, or it can be considered "fixed" in the leaning process: there exists a core endomorphism that will map that blank node to itself.

Lemma 5.5. *Let G be an RDF graph and let b be a blank node in G such that there does not exist a triple $(b, p, b') \in G$ nor a triple $(b', p, b) \in G$ for any $b' \in \mathbf{B}, p \in \mathbf{I}$. Let G' be the result of applying Algorithm 4 to G. It holds that either $b \notin \mathrm{bnodes}(G')$, or there exists a core endomorphism $\mu \in \mathrm{CEnd}(G)$ such that $\mu(b) = b$.*

Proof. We know from Lemma 5.4 that there exists an endomorphism $\mu' \in \mathrm{End}(G)$ such that $\mu'(G) = G'$, and such that μ' is a fixpoint, meaning that if $x \in \mathrm{codom}(\mu')$, then $\mu'(x) = x$. Since $\mu' \in \mathrm{End}(G)$, per Lemma 3.28, for any mapping $\mu'' \in \mathrm{CEnd}(\mu'(G))$, $\mu''(\mu'(G))$ gives a core of G; in other words, letting μ''_+ denote the mapping μ'' extended by the identity for terms in $\mathrm{terms}(G) \setminus \mathrm{dom}(\mu'')$, and letting μ be the mapping $\mu''_+ \circ \mu'$, it holds that $\mu \in \mathrm{CEnd}(G)$.

Now we left to show that for an unconnected blank node b, either $\mu'(b) \neq b$, or $\mu(b) = b$; in other words, we show that for any unconnected blank node b in G', b can only be mapped to itself by an endomorphism over G'; i.e., if $\mu'' \in \mathrm{CEnd}(G')$, then $\mu''(b) = b$.

First of all, such a blank node by definition will only have ground edges.

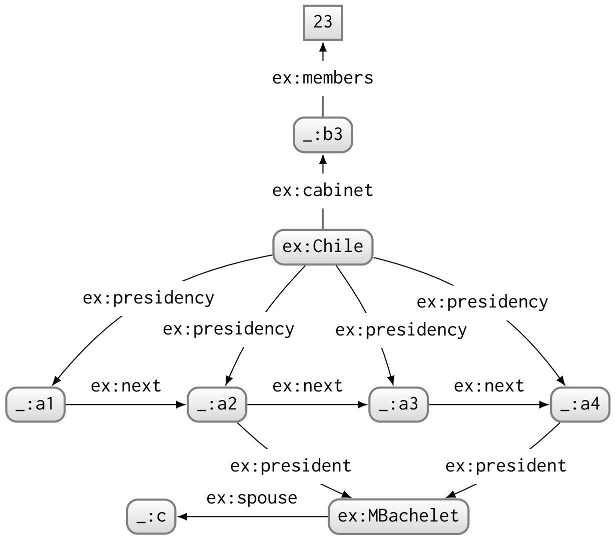
[11]As a minor detail, we must assume that X preserves insertion order to make sure a consistent blank node is picked from every equivalence class of blank nodes with the same set of edges.

If such a blank node is not removed by Algorithm 4 – i.e., if $\mu'(b) = b$ – then there can be no RDF term in G' with a superset of the (ground) edges of b (nor an equal set). Hence any mapping from b to another term would create a triple not in G, contradicting the definition of an endomorphism. Hence μ'' must map such a blank node b to itself, and hence if $\mu'(b) = b$, then $\mu(b) = b$. The existence of the core endomorphism μ then satisfies the lemma. □

5.2 Identifying candidate mappings for blank nodes

In a practical sense, Lemma 5.5 tells us that after applying Algorithm 4, all unconnected blank nodes from G will be either removed or "fixed" (meaning they cannot be redundant), where we thus now need to focus only on connected blank nodes. To begin, we can use the ground information in G to identify which terms each remaining blank node could be mapped to; we call such terms the *candidates* for that blank node, with the idea that an endomorphism can only map a blank node to one of its candidates. Each blank node can be mapped to at least itself. Blank nodes that can only be mapped to themselves are fixed, and fixed blank nodes act as ground terms; hence, when we fix a blank node, we can iterate the process to fix further blank nodes.

Example 5.6. The following is the result of Algorithm 4 applied to the graph of Example 5.1 where some redundant blank nodes have been removed:



b	cands[b]
`_:a1`	`_:a1`, `_:a2`, `_:a3`
`_:a2`	`_:a2`
`_:a3`	`_:a2`, `_:a3`
`_:a4`	`_:a2`, `_:a4`
`_:b3`	`_:b3`
`_:c`	`_:c`

To the right, we provide the initial candidates for all blank nodes. We know as a result of Lemma 5.5 that the remaining unconnected blank nodes `_:b3` and `_:c` will map to themselves in any further steps. We are thus more concerned with the connected blank nodes `_:a1`, `_:a2`, `_:a3`, `_:a4`.

If we start with blank node `_:a1`, considering its local ground information, we see that it is covered by `_:a2` and `_:a3` but not `_:a4`, which has no outgoing `ex:next` edge. Next, `_:a2` is fixed because no node has both incoming and outgoing `ex:next` edges *and* an outgoing `ex:president` edge to `ex:MBachelet`: hence `_:a2` can only map to itself. By similar arguments, we see that `_:a3` can only map to itself or `_:a2` due to having both incoming and outgoing `ex:next` edges, and that `_:a4` can only map to `_:a2` or `_:a4` due to having an outgoing `ex:president` edge to `ex:MBachelet`.

However, when we fix `_:a2`, we know it can only map to itself and that it acts like a ground term in the sense that for any endomorphism, $\mu($`_:a2`$)$ = `_:a2`. Hence `_:a1` can now be fixed since it has a unique outward `ex:next` edge to `_:a2`, and likewise, iteratively, `_:a3` can be fixed for its incoming `ex:next` edge from `_:a2`, and `_:a4` can be fixed due to its unique incoming `ex:next` edge from `_:a3`.

Hence, in this particular case, we have managed to fix all blank nodes, and the graph is thus lean, and we need to go no further. In other cases we will look at later, however, some blank nodes may maintain multiple candidates. □

In Algorithm 5, we outline a procedure for finding candidates for blank nodes based on the ground information directly surrounding terms in the graph, and likewise for finding blank nodes that are fixed (mapped to themselves by any endomorphism).

Lines 1–3 We first call Algorithm 4 to remove redundant blank nodes. Any unconnected blank nodes in the output can only map to themselves: they are fixed.

Lines 4–7 This initialises the sets of candidate terms for individual blank nodes. Note that in practice, we re-use indexes from Algorithm 4 to speed up the process if b is not fixed, where, e.g., we only initialise cands[b] with terms that have the most selective edge found for b; we do not show these optimisations for brevity.

Line 8 If all blank nodes are fixed, we can return since the graph is lean.

Lines 9–19 For each blank node b that is not yet fixed, check for each candidate of b that it has the incoming and outgoing edges to cover b; if not, remove the candidate. Again, in practice, we re-use indexes from Algorithm 4 to accelerate these checks.

Lines 20–21 Update the set of fixed blank nodes as those blank nodes whose only valid candidate is the blank node itself.

Lines 22 Apply the previous two steps iteratively until no new fixed blank nodes are found or all blank nodes have been fixed, at which point return the graph, the set of fixed blank nodes, and the candidates found.

Algorithm 5 Finding candidate mappings for blank nodes

```
1:  function findCandidates($G$)                                           ▷ $G$ an RDF graph
2:      $G' \leftarrow$ removeRedundantBnodes($G$)                          ▷ call Algorithm 4
3:      $F \leftarrow \{b \in \mathrm{bnodes}(G') \mid \nexists(p, b') \in \mathbf{I} \times \mathbf{B} : (b, p, b') \in G' \text{ or } (b', p, b) \in G'\}$   ▷ set of fixed blank nodes
4:      initialise cands                                 ▷ a map from blank nodes to sets of terms
5:      for $b \in \mathrm{bnodes}(G')$ do
6:          if $b \in F$ then cands[$b$] $\leftarrow \{b\}$                  ▷ can only map to itself
7:          else cands[$b$] $\leftarrow \mathrm{terms}(G')$     ▷ initially any blank node can map to any term
8:      if $F = \mathrm{bnodes}(G')$ then return $(G', F, \mathrm{cands})$    ▷ $G'$ is lean
9:      do
10:         $B \leftarrow \mathrm{bnodes}(G') \setminus F$
11:         for $b \in B$ do
12:             for $x \in$ cands[$b$] such that $x \neq b$ do
13:                 for $(b, p, o) \in G'$ do                               ▷ $\exists p, \exists o$
14:                     if ($o \in \mathbf{IL}F$ and $(x, p, o) \notin G'$) or ($o \in B$ and $\nexists o' : (x, p, o') \in G'$) then
15:                         cands[$b$] $\leftarrow$ cands[$b$] $\setminus \{x\}$ (break)
16:                 if $x \in$ cands[$b$] then
17:                     for $(s, p, b) \in G'$ do                           ▷ $\exists s, \exists p$
18:                         if ($s \in \mathbf{I}F$ and $(s, p, x) \notin G'$) or ($s \in B$ and $\nexists s' : (s', p, x) \in G'$) then
19:                             cands[$b$] $\leftarrow$ cands[$b$] $\setminus \{x\}$ (break)
20:         $F' \leftarrow F$                                  ▷ used to see if anything has changed
21:         $F \leftarrow F \cup \{b \in B \mid \mathrm{cands}[b] = \{b\}\}$   ▷ fixed blank nodes are those that can only map to themselves
22:     while $F' \neq F$ and $F \neq \mathrm{bnodes}(G')$
23:     return $(G', F, \mathrm{cands})$
```

If all the blank nodes are fixed (i.e., if Algorithm 5 terminates with $F = \mathrm{bnodes}(G)$), then the graph is lean and we are done. Otherwise, we must continue to further steps.

Algorithm characteristics. We now briefly discuss some properties of the algorithm.

Lemma 5.7. *Algorithm 5 terminates in $\Theta(\beta)$ iterations (starting Line 9) for G in the worst case, where $\beta := |\text{bnodes}(G)|$.*

Proof. In each iteration, a new blank node must be fixed. Once fixed, a blank node cannot be unfixed. Hence the number of iterations is bounded by the number of blank nodes. Again, the upper bound is tight, for example, due to the path graph of the form $y \xrightarrow{p} x_1 \xrightarrow{p} x_2 \ldots \xrightarrow{p} x_n$ ($x_i \in \mathbf{B}$ for $1 \leq i \leq n$, $y \in \mathbf{IL}$), where n iterations are needed to terminate, starting by fixing x_1 and iterating towards the right one step at a time. □

Each iteration has quadratic behaviour with respect to the number of triples in G in the worst case. Hence the overall algorithm (excluding the call to Algorithm 4 on Line 2) has $O(\beta^2\tau\gamma^2)$ runtime in the worst case, similar to Algorithm 4. In fact, the processes have some similarities in that we are comparing the edges of blank nodes with other terms, except that in the case of Algorithm 5, we consider blank nodes on edges as variables, whereas in the case of Algorithm 4, we do not; likewise in Algorithm 4 we are looking for blank nodes to remove, whereas in Algorithm 5 we are looking for blank nodes that are fixed or otherwise looking at what terms they could be mapped to. Hence although the processes are similar, we could not find a way to neatly combine the two processes, although it was possible to share some indexing steps.

Next we show that cands does indeed represent a set of candidate mappings for the endomorphisms of G.

Lemma 5.8. *Let* cands *be the result of applying Algorithm 5 over an RDF graph G. Then for any $b \in \text{bnodes}(b)$ and any $\mu \in \text{End}(G)$, it holds that $\mu(b) \in \text{cands}[b]$.*

Proof. For the purposes of proof by contradiction, let us assume that there exists a $b \in \text{bnodes}(G)$, an $x \in \text{terms}(G)$, and a $\mu \in \text{End}(G)$ such that $\mu(b) = x$ and $x \notin \text{cands}[b]$. If $x \notin \text{cands}[b]$, at least one of the following must hold per Algorithm 5:

- there is a triple $(b, p, y) \in G$, for some $p \in \mathbf{I}$, $y \in \mathbf{IL}$, such that (x, p, y) is not in G;
- there is a triple $(y, p, b) \in G$, for some $p \in \mathbf{I}$, $y \in \mathbf{I}$, such that (y, p, x) is not in G;
- there is a triple $(b, p, y) \in G$, for some $p \in \mathbf{I}$, $y \in \mathbf{B}$, such that (x, p, z) is not in G for any $z \in \text{terms}(G)$;
- there is a triple $(y, p, b) \in G$, for some $p \in \mathbf{I}$, $y \in \mathbf{B}$, such that (z, p, x) is not in G for any $z \in \text{terms}(G)$.

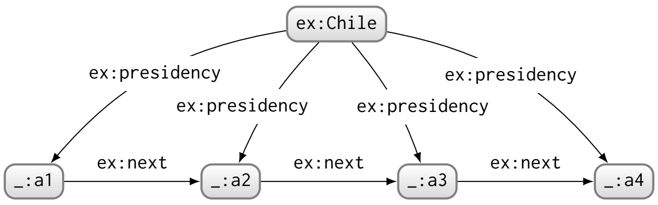
Let t denote a triple of the form (b, p, y) or (y, p, b) that does not have a corresponding triple for x as outlined above. Now, when we apply $\mu(\{t\})$, rewriting b to x, we create a new triple not in G; hence μ is not a valid endomorphism for G – we arrive at a contradiction per the definition of $\text{End}(G)$ – and thus the lemma holds. □

Algorithm 5 thus helps find a selective set of candidates to which blank nodes can be mapped by endomorphisms. In fact, in many real world cases, we would except this algorithm to fix all blank nodes, as per Example 5.6, in which case we are effectively finished. However, this will not be the case for all graphs, where we may have to begin a potentially exponential search process to identify valid endomorphisms.

5.3 Finding a core endomorphism for connected blank nodes

In the previous section, we used the ground information in the direct neighbourhood of a blank node to fix certain blank nodes and identify an initial list of candidates that blank nodes could map to. In some cases, all blank nodes will be fixed and hence the only valid endomorphism – more specifically, the only core endomorphism – will be the identity map on blank nodes; in such cases, the graph is lean. However, in other cases, some blank nodes will still have multiple candidates, and those candidates may not lead to a valid homomorphism since we have yet to consider the connectivity of blank nodes. We thus now look at methods to compute core endomorphisms taking into consideration the connectivity of blank nodes.

Example 5.9. On the left, we provide a simplified sub-graph of the RDF graph from Example 5.6.[12] On the right, we provide the candidates produced by Algorithm 5. We see that no blank nodes are fixed but that some blank nodes are deemed not to be candidates of other blank nodes.

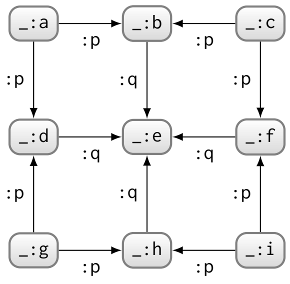


b	cands[b]
_:a1	_:a1, _:a2, _:a3
_:a2	_:a2, _:a3
_:a3	_:a2, _:a3
_:a4	_:a2, _:a3, _:a4

However, if we consider the endomorphisms of this graph – in other words if we consider this graph as a query with blank nodes as variables applied against itself – we see that each blank node can only map to itself. To illustrate this, let's say, for example, we tried to map `_:a2` to `_:a3` and other blank nodes to themselves; in this case, we create a loop that did not exist in the original graph from `_:a3` to itself, and hence we do not have a valid endomorphism. □

The previous example turns out to be lean; let's look at an example that is not and give an idea of what we wish to achieve in the next stage of the process.

Example 5.10. We return to the graph from Example 4.9 with the original edges.



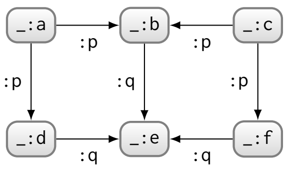
b	cands[b]
_:a	_:a, _:c, _:g, _:i
_:b	_:b, _:d, _:f, _:h
_:c	_:a, _:c, _:g, _:i
_:d	_:b, _:d, _:f, _:h
_:e	_:e
_:f	_:b, _:d, _:f, _:h
_:g	_:a, _:c, _:g, _:i
_:h	_:b, _:d, _:f, _:h
_:i	_:a, _:c, _:g, _:i

With this configuration of edges – unlike the similar graph of Example 5.2 – no blank nodes in this case are trivially redundant. From the candidates on the right, we see that one blank node (`_:e`) is fixed while the rest are not.

If we consider the endomorphisms of this graph, we could start by mapping `_:g` to `_:a`, `_:h` to `_:b`, and `_:i` to `_:c`, effectively folding the bottom half of the graph into the top half while preserving edges as shown to the right in the following; the result is a proper endomorphism as shown to the left in the following.

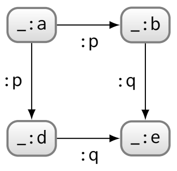
[12]We emphasise that for the purposes of illustration, this is a different sub-graph: it is not the output for the original graph of a previous algorithm.

_:a	_:b	_:c	_:d	_:e	_:f	_:g	_:h	_:i
_:a	_:b	_:c	_:d	_:e	_:f	_:a	_:b	_:c



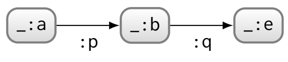
We can subsequently map `_:c` to `_:a` and `_:f` to `_:d`, effectively folding the right side of the resulting graph into the left half while preserving edges; again we have a proper endomorphism. The composition of the two endomorphisms is itself an endomorphism, as shown on the left below; the result of applying this endomorphism to the original graph is illustrated on the right below.

_:a	_:b	_:c	_:d	_:e	_:f	_:g	_:h	_:i
_:a	_:b	_:a	_:d	_:e	_:d	_:a	_:b	_:a



Finally we can, e.g., map `_:b` into `_:d`, giving the core endomorphism on the left and the resulting lean graph on the right below.

_:a	_:b	_:c	_:d	_:e	_:f	_:g	_:h	_:i
_:a	_:b	_:a	_:b	_:e	_:b	_:a	_:b	_:a



Note that in practice, it may not be necessary to apply this process step-by-step; rather, it may be possible to find a final core endomorphism directly. □

We consider two strategies for computing core endomorphisms from a graph with connected (non-fixed) blank nodes.

The first strategy, which we call breadth-first (BFS), computes all endomorphism (aka. solutions) for the connected blank nodes and then simply selects the endomorphism with the fewest unique blank nodes from its domain in its codomain.[13] The implementation applies a standard nested-loop join strategy that is often used for enumerating answers to conjunctive queries in databases. Starting with the most selective triple (pattern) in the query, evaluation continues in a "left-deep" fashion; all results for the first pattern are returned and then joined with the next pattern, and so forth, until all patterns are exhausted and all solutions are generated from which a core endomorphism can be chosen and returned.

One may observe that the first strategy enumerates all solutions but that, in reality, we are only searching for one. Hence we propose a more customised second strategy that we call depth-first (DFS), which looks at one solution at a time. The strategy again starts with the most selective triple (pattern) in the query, but this time the intermediate solutions are ranked and the first solution is sent immediately to the second pattern, which extends the solution and again ranks the results and passes the best to the third pattern, and so forth. The ranking of intermediate solutions is designed as a heuristic to try find a solution with as few blank nodes as possible as quickly as possible: it first chooses intermediate solutions with the fewest unique blank nodes (from the domain) in the codomain, and if tied, chooses those that map the fewest blank nodes to themselves. When the last

[13]There is a corner-case to be careful of here: blank nodes that are not connected act as ground terms to which blank nodes in the "query" can be matched "for free"; hence it is not sufficient to simply say the mapping with the fewest blank nodes in the codomain since blank nodes that have been fixed and are not in the domain of the query should not count as a blank node in the codomain.

pattern is reached, if any proper endomorphism is found (i.e., one with fewer unique blank nodes in the codomain than the domain), the recursive search immediately returns that endomorphism; otherwise, during the search, if a branch hits a "dead-end" where the intermediate solution cannot be extended any further, the recursion returns to the higher level which continues with the next best pattern selected by the heuristic. If no proper endomorphism is found during the search, the graph is lean and we can return whatever endomorphism (i.e., automorphism) we did find. However, if a proper endomorphism is found, we have no guarantee it is a core endomorphism; hence we rewrite the graph to remove the redundant blank nodes found thus far, and restart the process with the rewritten graph, until such a point that we have found a graph with no proper endomorphism.

The benefit of the DFS strategy is that, unlike the BFS strategy, we do not need to keep potentially exponentially many (intermediate) solutions in memory; rather we just need to track the intermediate solutions for one branch of the search space. Likewise, if there is a core endomorphism, the DFS strategy with its solution-ordering heuristics is guided towards finding it, potentially avoiding having to explore exponentially many solutions. Also, the DFS strategy allows for simplifying the graph iteratively, which may help to reduce the search space quickly. On the other hand, if there are no proper endomorphisms, in both strategies all solutions need to be explored; while DFS still benefits in terms of space versus BFS, it has a similar workload.

In Algorithm 6, we go through the two strategies for finding a core endomorphism over the connected blank nodes in an RDF graph given the candidates and rewritten graph provided previously by Algorithm 5.

Lines 1–8 This high-level function will return a core for G. It first calls Algorithm 5, which removes redundant blank nodes, tries to fix others, and finds the candidates for remaining blank nodes. If all blank nodes are fixed, or if the resulting graph does not contain any connected blank nodes, then this graph can be returned directly. Otherwise if the graph returned by Algorithm 5 contains any connected blank nodes, first a method is called to find a core endomorphism from the graph, and second the graph is rewritten per the endomorphism and returned; here σ is a configuration variable that selects either the BFS or DFS strategy.

Lines 10–12 This starts the process of finding a core endomorphism over the connected blank nodes. We extract the triples from G where both the subject and object are blank nodes – resulting in what we call the *query* – and then reorder the query using a selectivity heuristic based on a product of the number of candidates that the subject and object blank nodes have and the number of triples in G with the same predicate, placing more selective triples in Q first; we also add a constraint to ensure that blank nodes are "grouped" in the query to avoid cross-products where possible. In fact, Q need not necessarily be connected and may contain multiple blank node splits; this is a deliberate choice since it simplifies the process of identifying proper and core endomorphisms, particularly in the DFS strategy.

Lines 13–14 Here we call either the BFS or DFS strategy (indicated by σ) and return the core endomorphism that it returns.

Lines 15–16 In the BFS method, we pass a mapping to the recursive search function containing the identity map for all fixed blank nodes; this is necessary to ensure that the count of blank nodes in the codomain of the mapping is accurate for selecting a core endomorphism, prioritising mappings to a blank node that is already fixed.

Lines 17–24 Here we have the recursive BFS method, where the next pattern is popped from Q and used to extend the intermediate solutions found thus far (using the function JOIN, which we describe later). If there are more patterns left, the method is called recursively for the subsequent patterns with the extended solutions. When no patterns are left, all solutions/endomorphisms have been computed and a core endomorphism is selected arbitrarily and returned. Note that M' can never be empty since it must contain at least the identity mapping of blank nodes; hence we never need to check for empty solutions.

Lines 25–32 This is the high-level recursion for the DFS strategy. The method calls the recursive search process with an initial mapping that is the identity for fixed blank nodes, again needed to ensure that the count of blank nodes in the codomain is accurate when choosing solutions. The recursive search function will return an endomorphism. If a proper endomorphism exists, the method will return a proper endomorphism, where the original graph G will be rewritten and the algorithm – including extraction and reordering of the query – will be called again. This continues until no proper endomorphism is found, at which point the endomorphisms found are recursively merged and returned. The result of this process is then a final core endomorphism.

Lines 33–46 This is the lower-level recursive search for the DFS strategy that will return a proper homomorphism for the graph if and only if one exists; otherwise it will return an automorphism. Unlike BFS, the DFS search only passes one solution recursively. The process again pops the first pattern from Q and extends the current solution μ with the results for that pattern. The extended solutions are ordered according to the heuristic. If there are further patterns to evaluate, the search will try each solution in order of the heuristic and call the function for the next pattern; if any proper homomorphism is found, it is immediately returned to allow the graph to be rewritten before proceeding.

Lines 47–51 This auxiliary function extends the given results M by the results for the next pattern q with respect to G; it also checks the candidates to ensure that the produced results are compatible with the ground information processed previously in Algorithm 5. Although not shown in the function for reasons of brevity, we follow a standard nested-loop strategy for evaluating the joins and use in-memory maps to perform efficient lookups.

We now give an example that provides an intuition of how both strategies work.

Example 5.11. Let us briefly reconsider Example 5.10. Our query will look like the following when ordered by selectivity (based on the higher selectivity of :q and _:e):[14]

$$\mathrm{Q} = \big((\texttt{\_:b}, \texttt{:q}, \texttt{\_:e}), (\texttt{\_:d}, \texttt{:q}, \texttt{\_:e}), \ldots, (\texttt{\_:i}, \texttt{:p}, \texttt{\_:f}), (\texttt{\_:i}, \texttt{:p}, \texttt{\_:h})\big)$$

We will first generate all intermediate solutions M_1 for (_:b, :q, _:e), such that:

$$M_1 = \big\{\{(\texttt{\_:b}, \texttt{\_:b}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:f}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:h}), (\texttt{\_:e}, \texttt{\_:e})\}\big\}$$

In the BFS strategy, we will extend these mappings by the compatible solutions for the next pattern (_:d, :q, _:e), arriving at M_2 with sixteen results as follows:

$$M_2 = \big\{\{(\texttt{\_:b}, \texttt{\_:b}), (\texttt{\_:d}, \texttt{\_:b}), (\texttt{\_:e}, \texttt{\_:e})\}, \ldots, \{(\texttt{\_:b}, \texttt{\_:h}), (\texttt{\_:d}, \texttt{\_:h}), (\texttt{\_:e}, \texttt{\_:e})\}\big\}$$

This process continues through all patterns until all solutions are computed. Any solution with the fewest number of domain blank nodes appearing in the codomain is given as a core solution (see Example 5.10 for one such solution).

In the DFS strategy, we use heuristics to order solutions in M_1 such that:

$$\mathsf{M}_1 = \big(\{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:f}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:h}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:b}), (\texttt{\_:e}, \texttt{\_:e})\}\big)$$

We order by blank nodes in the codomain, but since this is equal for all initial solutions, we then order by fewest self mappings and then by arbitrary order. We take the first such solution, $\mu_{1,1} = \{(\texttt{\_:b}, \texttt{\_:d}),(\texttt{\_:e}, \texttt{\_:e})\}$, and send it to the second pattern, where we then generate:

[14]We allow a small fudge here: _:e would be fixed and thus not part of the query, but rather only part of the data; however, for the purposes of illustration, we ignore this for the moment.

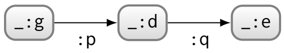
$$M_2 = \big\{\{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:b}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:d}), (\texttt{\_:e}, \texttt{\_:e})\},$$
$$\{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:f}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:h}), (\texttt{\_:e}, \texttt{\_:e})\}\big\}$$

After ordering M_2, we will get

$$\mathsf{M}_2 = \big(\{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:d}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:b}), (\texttt{\_:e}, \texttt{\_:e})\},$$
$$\{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:f}), (\texttt{\_:e}, \texttt{\_:e})\}, \{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:h}), (\texttt{\_:e}, \texttt{\_:e})\}\big)$$

The first solution $\mu_{2,1} = \{(\texttt{\_:b}, \texttt{\_:d}), (\texttt{\_:d}, \texttt{\_:d}), (\texttt{\_:e}, \texttt{\_:e})\}$ maps to the fewest unique blank nodes. This solution is passed to the next pattern, and the process continues. After the first four patterns, _:b, _:d, _:f and _:g will all be mapped to _:d, and _:e will be mapped to itself. The fifth pattern will be (_:a, :p, _:b), where since _:b already maps to _:d, _:a will map to itself or to _:g; the heuristic ordering will prefer _:g since it is not a self-mapping and neither _:a nor _:g appear in the codomain thus far. The heuristic will then guide the three remaining corners to map to _:g. This will yield the proper endomorphism on the left and the resulting graph on the right.

_:a	_:b	_:c	_:d	_:e	_:f	_:g	_:h	_:i
_:g	_:d	_:g	_:d	_:e	_:d	_:g	_:d	_:g



We now have a core endomorphism (the resulting graph is isomorphic with that from Example 5.10). However, we have no guarantee that our heuristic will find a core endomorphism first time; hence we must re-apply the process again over the resulting graph until we find that there are no proper endomorphisms, confirming that the graph is lean and the solution is a core endomorphism.

In the above example, the DFS heuristic neatly finds its way to a core endomorphism; it is worth noting that in other cases, backtracking may be necessary, and indeed when the input graph is already lean, all solutions will still need to be checked through to confirm leanness. □

As per this example, the DFS strategy pauses the search when a proper endomorphism is found to rewrite the graph before continuing, which requires the additional higher-level recursion in EVALUATE$_{\text{DFS}}$; as per the previous example, this strategy works well when the rewriting greatly simplifies the graph, whereas the other option would be to continue exhaustively through all solutions without rewriting (similar, in principle, to BFS). Our current DFS heuristic is greedy – it tries to minimise the number of blank nodes at each step – but it will not always lead to a global minimum of blank nodes; for example, it is not difficult to construct counterexamples where at the start, the heuristic gets unlucky with its first intermediate solution, which leads it to a non-core proper endomorphism. An interesting question then is if it would be possible to design a similar heuristic that *guarantees* to find a core endomorphism as the first solution, which would then obviate the need for the additional recursion in EVALUATE$_{\text{DFS}}$; we leave this as an open question. In any case, the maximum number of rewritings is bounded by the number of blank nodes in Q, where we would expect the DFS strategy to terminate after zero or one rewritings in most cases.

Algorithm characteristics. We now briefly discuss some properties of the algorithm.

LEMMA 5.12. *For any RDF graph G, Algorithm 6 terminates under either the BFS or DFS strategies.*

PROOF. With respect to BFS, we are using a standard left-deep join process to enumerate the answers to a conjunctive query: we evaluate the query one pattern at a time until all patterns have been processed, at which point the algorithm terminates.

With respect to DFS, first looking at SEARCH$_{\text{DFS}}$, we progress one solution at a time for each pattern. As aforementioned, we know that there exists at least one such solution: the identity map for blank nodes in Q. Hence we will reach the point where Q' is empty and will return at least that

Algorithm 6 Computing a core of the RDF graph

```
1: function LEAN_σ(G)    ▷ returns a core of the RDF graph G; σ denotes strategy: BFS or DFS
2:     if bnodes(G) is empty then return G
3:     (G′, F, cands) ← FINDCANDIDATES(G)    ▷ call Algorithm 5
4:     if F = bnodes(G′) then return G′    ▷ if all blank nodes are fixed, G is lean
5:     if there exists (s, p, o) ∈ G′ such that s, o ∈ B then
6:         μ ← {(b, b) | b ∈ F}
7:         μ ← FINDCOREENDOMORPHISM_σ(G′, cands, μ)    ▷ call Algorithm 6
8:         return μ(G′)    ▷ return the lean RDF graph μ(G′) (removing duplicates)
9:     return G′
10: function FINDCOREENDOMORPHISM_σ(G,cands,μ)    ▷ G an RDF graph, cands from FINDCANDIDATES(G)
11:     Q ← {(s, p, o) ∈ G | s, o ∈ B}    ▷ all triples with connected blank nodes
12:     Q ← orderBySelectivity(Q, G, cands)    ▷ order Q into a list of triples with most selective first
13:     μ ← EVALUATE_σ(G, cands, Q, μ)    ▷ returns a core solution; σ denotes a strategy (BFS or DFS)
14:     return μ
15: function EVALUATE_BFS(G,cands,Q,μ)    ▷ generates all solutions; chooses core endomorphism
16:     return SEARCH_BFS(G,cands,Q,{μ})
17: function SEARCH_BFS(G,cands,Q,M)    ▷ M is a set of blank node mappings (i.e., intermediate solutions)
18:     q ← Q_min    ▷ first element of query
19:     M′ ← JOIN(q, G, cands, M)    ▷ extend M by results for q w.r.t G (and cands)
20:     Q′ ← Q \ q
21:     if Q′ is not empty then
22:         SEARCH_BFS(G,cands,Q′,M′)    ▷ recurse in a breadth-first manner for M′
23:     else
24:         return some μ ∈ M′ with fewest blank nodes from dom(μ) in codom(μ)    ▷ μ is a core endo.
25: function EVALUATE_DFS(G,cands,Q,μ)    ▷ will maintain only one solution at a time chosen from heuristics
26:     μ ← SEARCH_DFS(G, cands, Q, μ)
27:     μ′ ← μ
28:     while dom(μ′) ≠ codom(μ′) do    ▷ while we have some non-leanness
29:         μ″ ← {(b, b) | cands[b] = {b}}    ▷ compute fixed blank nodes again
30:         μ′ ← FINDCOREENDOMOPHISM_DFS(μ(G), cands, μ″)    ▷ rewrite graph (removing duplicates) and recurse
31:         μ ← μ′ ∪ {(b, x) ∈ μ | b ∉ dom(μ′)}    ▷ merge the two mappings (effectively μ′(μ(·)))
32:     return μ
33: function SEARCH_DFS(G,cands,Q,μ)    ▷ μ an intermediate solution chosen from heuristics
34:     q ← Q_min    ▷ first element of query
35:     M ← JOIN(q, G, cands, {μ})    ▷ extend μ by results for q w.r.t G (and cands)
36:     M ← orderByCodom(M)    ▷ order M by fewest unique blank nodes in codomain, then fewest self mappings
37:     Q′ ← Q \ q
38:     if Q′ is not empty then
39:         while M is not empty do    ▷ can never be empty to start with; must always have identity solution
40:             μ′ ← SEARCH_DFS(G, cands, Q′, M_min)    ▷ recurse to try and extend M_min, i.e., the first solution of M
41:             if dom(μ′) ≠ codom(μ′) then    ▷ if μ′ is a proper endomorphism (and not the empty mapping)
42:                 return μ′    ▷ return it to rewrite graph with
43:             M ← M \ M_min    ▷ otherwise remove the first solution and move to next solution
44:     if M is empty then
45:         return ∅    ▷ return the empty mapping
46:     return M_min    ▷ will be proper homomorphism if and only if one exists
47: function JOIN(q,G,cands,M)    ▷ evaluate the given pattern joining with solutions in M
48:     M_q ← {μ | μ(q) ∈ G}    ▷ get initial solutions for q in G
49:     M′_q ← {μ ∈ M_q | for all b ∈ bnodes({q}), μ(b) ∈ cands[b]}    ▷ check cands to ensure compatibility
50:     M′ ← M′_q ⋈ M    ▷ '⋈' denotes standard (inner) join operator
51:     return M′
```

solution. With respect to EVALUATE$_{\text{DFS}}$, we will only recurse if the homomorphism found is proper, meaning that it will simplify the graph by removing the triples involving at least one blank node. Hence the number of recursive calls is bounded by the number of blank nodes in G. □

Indeed, both algorithms have exponential runtime behaviour. However, our hypothesis is that for most real-world graphs, these algorithms will have acceptable runtimes. We will discuss performance issues further in the evaluation section.

We now discuss the correctness of the algorithm.

THEOREM 5.13. *The result of applying Algorithm 6 to G is a core of G.*

PROOF. Algorithms 4 and 5 resolve all unconnected blank nodes by either fixing or removing them, producing G', where we already proved that there exists an endomorphism $\mu \in \text{End}(G)$ such that $\mu(G) = G'$. The identity mapping for fixed blank nodes is used as the initial mapping that seeds a core endomorphism, per Lemma 5.5. Thereafter, both the BFS and DFS strategies extend this initial solution, including the fixed blank nodes in the count for selecting a core endomorphism for G', not just considering the blank nodes in Q.

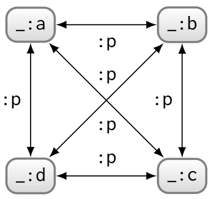
For the BFS strategy, the algorithm computes all solutions in a straightforward brute-force (nested-loop) manner and chooses the core endomorphism as the solution with the fewest blank nodes; hence the mapping μ returned by EVALUATE$_{BFS}$ is a core endomorphism for G' and $\mu(G')$ is thus a core of G following Lemma 3.28.

For DFS, the mapping μ returned by the recursive search function SEARCH$_{DFS}$ will be a proper endomorphism of G if one exists; otherwise all solutions will be checked and an automorphism returned. If a proper endomorphism is found, the graph will be rewritten and the process recursively applied. This will terminate per Lemma 5.12 and will produce a core endomorphism of G' per Lemma 3.28. Again, we thus have that $\mu(G')$ is a core of G per the latter lemma. □

5.4 Pruning the depth-first search using automorphisms

Inspired by Section 4.3, we can also consider applying a pruning optimisation to the DFS strategy based on automorphisms. Every complete solution found in the DFS strategy that maps blank nodes to blank nodes in a one-to-one manner is an automorphism—in other words, every solution that is not a proper endomorphism (triggering a rewriting) is an automorphism. Instead of giving the full algorithm, we instead illustrate this process with an example.

Example 5.14. On the left, we look at an example of a 4-clique of blank nodes, with all edges given in both directions with label :p.



	_:a	_:b	_:c	_:d
0	_:a	_:b	_:c	_:d
1	_:b	_:a	_:d	_:c
2	_:b	_:a	_:c	_:d
3	_:b	_:c	_:d	_:a
4	_:b	_:c	_:a	_:d
5	_:b	_:d	_:a	_:c
6	_:b	_:d	_:c	—
7	...	...	...	...

On the right, we provide a list of solutions found by the DFS strategy thus far, where solution 0 is the trivial automorphism (and thus can be added straight away). In this case, the 4-clique is lean so DFS will not find a proper endomorphism, but instead will typically have to check through all solutions to ensure that they are all automorphisms and that the graph is lean. Thus we can see up to solution 5 that all solutions found thus far are automorphisms.

Let us now assume we are after computing solution 5 in the DFS, and we return to the part of the search where `_:a` $\mapsto$ `_:b` and `_:b` $\mapsto$ `_:d` (i.e., our path thus far is [`_:b`, `_:d`]) looking for another option to map `_:c` to other than `_:a`, where we can consider `_:c` itself (`_:b` or `_:d` would create a self-loop not in the graph). However, in the set of automorphisms, note that mapping between 1 and 3, `_:b` and `_:d` map to themselves and `_:a` maps to `_:c` and vice-versa. Mapping between these two automorphisms, we thus know that we can fix `_:b` and `_:d` and swap `_:a` and `_:c` and end up with precisely the same graph. Thus, since at this point in the DFS search we have [`_:b`, `_:d`] fixed, having visited `_:a` and found only automorphisms, we can prune the `_:c` branch.

Likewise as the process continues, we would expect to find more automorphisms and to prune more frequently. This becomes more valuable for larger graphs. □

We only apply this optimisation in the case of DFS: in the case of BFS, we will not generate solutions until the end of the breadth-first process, meaning that we will not know of any automorphisms until it is, for practical purposes, too late to prune.

5.5 Equi-canonicalising algorithm

Finally, in Algorithm 7, we wrap up by giving the process of computing an equi-canonical form of any RDF graph G. If G does not contain blank nodes, we can return it immediately; otherwise we first lean it per Algorithm 6 and then apply the iso-canonicalisation process of Algorithm 3.

Algorithm 7 Computing an equi-canonical version of an RDF graph

```
1: function equiCanonicalise($G$)                         ▷ $G$ an RDF graph
2:     if bnodes($G$) is empty then return $G$
3:     $G' \leftarrow$ lean(G)                                ▷ calls Algorithm 6
4:     return isoCanonicalise($G'$)                         ▷ calls Algorithm 3
```

The correctness of this algorithm follows directly from Theorem 3.21 and the correctness results for Algorithms 3 and 6.

6 EVALUATION

Having detailed our algorithms for computing an iso-canonical form and an equi-canonical form of an RDF graph, we now evaluate their performance in practice. Both of these algorithms have exponential worst-cases, but our hypothesis is that the types of graphs that produce worst-cases would not be commonly found in typical settings.

Along these lines, we have implemented our methods as a Java package called BLabel made available under Apache Licence 2.0.[15] This package uses the Guava library for hashing methods.[16] We first experiment with a large corpus of RDF graphs crawled from the Web, testing the various algorithms introduced. Thereafter, we present the results of experiments on more challenging synthetic cases – some of which evoke exponential runtimes – to stress-test the algorithms.

To test the correctness of our software, we developed a test framework that takes an input graph and produces four isomorphic shuffles of the input graph by randomly reordering triples and renaming blank nodes, verifying for all shuffles that:

[15]The code repository is available from http://blabel.github.io/, as well as instructions and resources to help reproduce results.
[16]https://github.com/google/guava

(1) our labelling algorithm produces the same iso-canonical graph over all shuffles with and without automorphism-based pruning enabled;
(2) our BFS and DFS leaning algorithms, with and without ordering and pruning optimisations enabled, produce the same equi-canonical graph for all shuffles;
(3) applying the same leaning algorithm over the result of the previous step does not change the graph; i.e., the output of the previous step is deemed lean.

We ran this test framework over the real-world and synthetic graphs introduced in the following, where we found that either (i) no result was returned due to a timeout or out-of-memory exception, or (ii) a result was returned and the above tests passed. In other words, we test over a large collection of graphs of various morphologies and sizes, verifying that when our methods return a result, that that result is consistent according to the checks described above.[17]

6.1 Experiments on real-world graphs

To evaluate our algorithms on a large, diverse collection of real-world graphs, we take the Billion Triple Challenge 2014 (BTC−14) dataset.[18] We first removed all crawling meta-information that is not part of the native Web data, but is rather added by the crawler to the corpus; hence our methods are applied over the RDF graphs as found natively on the Web. The dataset contains a total of 43.6 million RDF graphs crawled from 47,560 pay-level-domains (a domain that must be paid for, such as `dbpedia.org`, `bbc.co.uk`, but not, e.g., `es.dbpedia.org` or `news.bbc.co.uk`, which would be mapped to the former two domains, respectively). The dataset is encoded in the N-Quads syntax [8], where the first three elements represent a standard RDF triple and the fourth element encodes the URL of the document from which the triple was extracted. The corpus consists of approximately 4 billion quads mentioning a total of 132.5 million unique blank nodes across all graphs.

We sort the data by the fourth quad element in order to group the triples of a given document together. Of the 43.6 million graphs, we found that 9.9 million graphs (22.7%) contain blank nodes. The largest graph contains 7.3 million triples and 254,288 blank nodes.[19]. The graph with the most blank nodes contains 1.9 million triples and 494,992 blank nodes.[20]

In the following experiments, we will load each individual RDF graph independently and apply the designated algorithm to it before moving to the next graph. We run each algorithm over all graphs in a given experiment. We use a timeout of 10 minutes per graph at which point the process is interrupted and the next graph processed. When measuring times, since many graphs may take less than 1 ms to process, we take the overall time of processing all graphs rather than taking the sum of the times for the individual runs; along these lines, we ran a control that just parsed the data and loaded the graphs and the blank nodes they contain, logging the size of the graph and the number of blank nodes without further processing, which took 15.2 hours.[21]

The experiments were run in a single-threaded manner on an Intel E5-2407 Quad-Core 2.2GHz machine with 30 GB of heap space and a 3.5-inch SATA hard drive.

6.1.1 Iso-canonical experiments. First we tested Algorithm 3 for the BTC−2014 dataset, computing the iso-canonical form for each graph in turn, where we tested three hashing functions in the

[17]This test framework revealed a bug in the automorphism-based pruning when labelling blank nodes that was present for the previous conference version of this paper [27]. Hence the evaluation results in this version of the paper should be considered as superseding those previously published where differences are present.
[18]This was the most recent version of this dataset available at the time of writing.
[19]http://www.berkeleybop.org/ontologies/ncbitaxon.owl
[20]http://www.berkeleybop.org/ontologies/owl/CHEBI
[21]In the previous conference version of this paper, we reported a control over the same data of 4 hours; however, this did not perform any logging nor did it count blank nodes. We thus argue that this new control is fairer as it allows to subtract the (notable) overhead of generating and logging statistics for the evaluation.

Table 2. High level statistics for iso-canonicalising the BTC–14 graphs

Hash	**Bits**	**Total** (*h*)	**Adjusted** (*h*)	**Average** (*ms*)	**Slowest** (*ms*)	**Timeouts**
MD5	128	23.9	8.8	3.2	48,692	0
Murmur3_128	128	23.6	8.4	3.1	42,925	0
SHA1	160	25.6	10.4	3.8	50,630	0
Murmur3_128*	128	202.3	187.1	68.1	600,000	971

128–160 bit range: MD5, Murmur3_128 and SHA1. These configurations include the automorphism pruning strategy described in Section 4.3; we also ran experiments with pruning disabled for the fastest hashing function to test its efficacy (we denote this configuration Murmur3_128*).

In Table 2, we provide the high-level results. With pruning enabled, the process ran successfully for all graphs without encountering a timeout. We see that the total time taken was between 23.6–25.6 hours for such configurations, but when we adjust by removing the control time for loading the data and logging results (15.2 hours), we see that the total additional time spent computing the iso-canonical graphs was between 8.4–10.4 hours, which equates to between 3.2–3.8 ms per graph (considering only the 9.9 million input graphs that have blank nodes). Comparing the hash functions, we can see that SHA1 is the slowest (perhaps due to having more bits) while Murmur3_128 is slightly faster than MD5. The slowest graph in each case was the largest graph – with 7.3 million triples and 254,288 blank nodes – taking 42.9–50.6 seconds to process.

Table 2 also provides the results for labelling with pruning disabled (Murmur3_128*). In this case, 971 graphs failed to be processed within the timeout; the overall results include such cases, which are considered as taking 10 minutes. In adjusted time, the experiment without pruning enabled took more than 21× longer than the experiment for the same hash function with pruning enabled.

Aside from the average time taken per graph, which may be dominated by millions of trivially simple graphs with a handful of blank nodes, it is interesting to see how many graphs took longer than a few milliseconds to compute. Along these lines, Figure 2 presents a histogram for the runtimes, illustrating how many graphs fell within the presented runtime intervals; note that both the bins on the x-axis and the values on the y-axis are scaled logarithmically and $\top$ denotes the timeout. The results only include graphs with blank nodes. For all configurations, about 98.8% of graphs are processed within 10 ms, and 99.9% are processed within a second. On the other hand, with pruning enabled, 6,026–6,057 graphs took between 1–10 seconds and 516–621 graphs took between 10–100 seconds (recall that the slowest graph took 42.9–50.6 seconds). With pruning disabled, 246 graphs took between 100–600 seconds and 971 timed out.

Finally, we counted the number of graphs that were found to be isomorphic based on hashes over their iso-canonical forms, where we found that of the 9.9 million RDF graphs with blank nodes, 9.4 million (95%) were unique modulo isomorphism.

From these results we can conclude that:

- the vast majority of the 9.9 million real-world graphs studied are relatively trivial to compute an iso-canonical form for, finishing in a few milliseconds;
- with pruning enabled, no graph took longer than 41 seconds and no failures were encountered; however, there are several thousand graphs that take longer than a second, and a few hundred that take longer than 10 seconds;

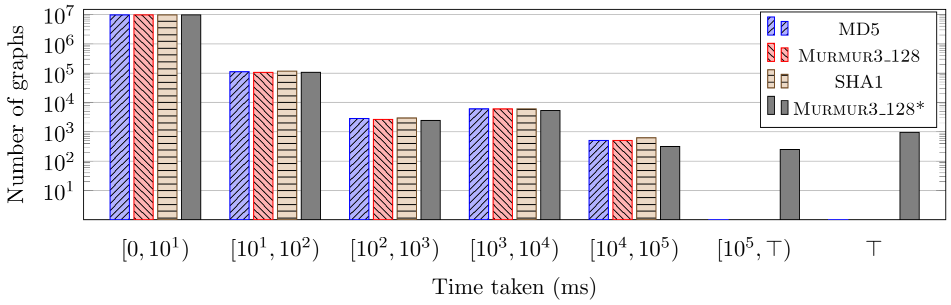


Fig. 2. Histogram of runtimes for iso-canonicalising the BTC−14 graphs

- with pruning disabled, the performance drastically suffers for about one thousand of the real-world input graphs (but pruning has little effect otherwise on the vast numbers of simpler graphs where blank nodes can be trivially distinguished);
- the selection of hashing function can make a moderate difference for performance, where, from those implemented by the Guava library, we found Murmur3_128 to be the fastest hash function in the 128–160 bit range,
- about 5% of the documents in the graph were duplicates when considering isomorphism; these could be, for example, syntactically identical copies of the same RDF document in multiple Web locations.

6.1.2 Equi-canonical experiments. Next we perform experiments for equi-canonicalising the same BTC−14 graphs, where we tested four leaning variants: BFS, DFS with automorphism-based pruning (see Section 5.4), DFS without pruning (denoted DFS*), and DFS with pruning but with random ordering. With respect to DFS with random ordering, instead of selecting intermediate solutions with the fewest blank nodes, we randomly select intermediate solutions; this serves as a baseline to measure the benefit of the ordering heuristic we use in the standard DFS strategy.

We also wish to see if labelling gets easier when the graph is leaned beforehand. Hence we also present results for subsequently computing the iso-canonical form over the output of the leaning algorithm, where we select the standard DFS strategy (with pruning) and the Murmur3_128 hashing function (the fastest hashing function from the previous experiments); given that the iso-canonical computation is independent of the leaning strategy, we can estimate the additional time taken for labelling by subtracting the runtimes for the analogous experiments with and without labelling.

We encountered a lot of problems running the experiments with the BFS strategy. The process continuously timed-out and also ran out of heap space. The experiment had processed 1.8 million graphs with blank nodes, where 19 had timed out; at this point the experiment threw an out-of-memory exception and did not recover. Hence we did not manage to complete the experiment for all 9.9 million graphs, but we can conclude that the time and space required for the rather brute-force BFS strategy can make it prohibitively expensive when leaning some real-world graphs.

Table 3 presents the results for the DFS strategies. We find an adjusted runtime of 14.7 hours for leaning in the best case, averaging 5.3 ms per graph; relative to performing (only) an iso-canonical labelling, this is about 1.8× more expensive compared with the fastest labelling algorithm. In fact, in the case of leaning, we now encounter graphs whose computation surpasses the 10 minute timeout, albeit in relatively few number; we manually inspected the 3 timeout cases for the best-case configuration, where two such graphs were from `berkeleybop.org` with 494,992 and 297,241 blank

Table 3. High level statistics for equi-canonicalising the BTC–14 graphs

Experiment	Total (h)	Adjusted (h)	Average (ms)	Timeouts
DFS	29.8	14.7	5.3	3
DFS*	29.9	14.7	5.3	3
DFS-Random	32.0	16.8	6.1	5
DFS+Label	33.4	18.2	6.6	3
Label	—	3.5	1.3	0

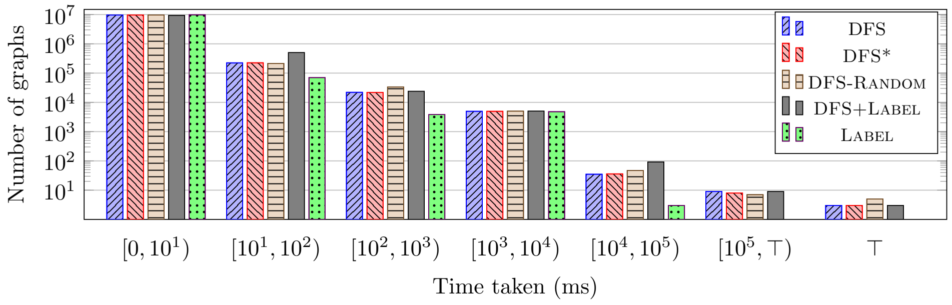


Fig. 3. Histogram of runtimes for equi-canonicalising the BTC–14 graphs

nodes respectively, and another graph contained a 16-clique of blank nodes[22] (we analyse such cases in more detail in the next section). As before, Table 3 counts the graphs that time-out as taking 10 minutes. We see that the pruning optimisation makes little difference to overall runtimes, while randomising the search heuristic adds about a 14% overhead in adjusted times.

Applying labelling after leaning adds a 24% overhead in adjusted times; subtracting the time taken for DFS leaning, the Label row shows the overhead for labelling, which is about 41.6% of the adjusted time taken for labelling without any prior leaning process (see Murmur3_128 in Table 2).

In Figure 3, we present a histogram with the runtimes for the four experiments (and the estimated labelling overhead), where $\top$ denotes the selected timeout of 10 minutes. Not considering labelling, in all three DFS experiments, 97.5% of graphs are leaned within 10 ms; on the other hand, 94.6% are leaned and labelled within 10ms. Across all configurations, 99.9% of graphs are leaned (and labelled) within a second, around 5,000 graphs in each case take 1–10 seconds, while 35–92 graphs take 10–100 seconds and 7–9 take 100–600 seconds.

Referring back to Figure 2, we can see that while 516 graphs took longer than 10 seconds to label without prior leaning for Murmur3_128, after leaning, only 3 graphs take longer than 10 seconds to label with the same hash function (note: this does not consider the additional 3 graphs that timed-out during leaning and for which we thus do not have a labelling time).

[22]http://cliopatria.swi-prolog.org/packs/rdf-mt.ttl: the graph is produced by reasoning over an input graph where all 16 blank nodes are equated to the same IRI, where two partitions of 10 and 6 blank nodes are indistinguishable from ground information; subsequent `owl:sameAs` reasoning creates an undirected clique of pair-wise such relations between all such blank nodes. In our previous paper [27], which included experiments over the same data, we confidently proclaimed that cases such as 16-cliques of undistinguishable blank nodes are unlikely to occur naturally in practice; we did not notice this graph since we only presented results for iso-canonical labelling, which has no problems with such inputs.

Finally, of the 9.9 million RDF graphs with blank nodes, we found 600,839 to be non-lean (6.1%). These non-lean graphs had a total of 314.8 million input triples with 25.2 million unique blank nodes, which were reduced to 273.7 million output triples (86.9%) with 15.2 million unique blank nodes (60.4%). The graph that was most reduced by leaning[23] – in terms of the absolute number of both blank nodes and triples removed – had 1,396,851 input triples with 331,365 unique blank nodes, which was reduced to 588,800 output triples (42.2%) with 119,747 unique blank nodes (36.1%). The total number of unique documents modulo equivalence was 9.4 million (95%), which was only 442 graphs fewer than the analogous number modulo isomorphism.

From these results we can conclude that:

- the vast majority of the 9.9 million real-world graphs studied are relatively trivial to compute an equi-canonical form for, finishing in a few milliseconds;
- however, several thousand graphs take longer than a second, and tens of graphs take longer than 10 seconds, with three non-trivial graphs timing out after 10 minutes;
- leaning is about 1.8× as costly as labelling, where computing the equi-canonical form is about 2.2× times as costly as the iso-canonical form, on average;
- unlike labelling, pruning by automorphism has little impact on leaning;
- however, the ordering heuristic that guides the DFS search towards a core homomorphism has a positive impact on performance;
- labelling after leaning (i.e., excluding the cost of leaning) is considerably faster, on average, than labelling raw graphs; we can conclude that prior leaning can simplify some complex graphs, making them subsequently easier to label;
- about 6.1% of graphs were found to be non-lean; however, only 442 graphs that were not redundant according to isomorphism were found to be redundant with respect to equivalence, which suggests that real-world RDF graphs tend to either be isomorphic copies of each other (e.g., syntactically identical copies in multiple locations) or not equivalent at all—this can probably be considered a negative result for the paper since it suggests that in practice, an iso-canonical form is sufficient in most cases where, e.g., a crawler detecting duplicates can catch the vast majority using the iso-canonical form; all the same, this is an interesting result and the leaning algorithm has other applications aside from duplicate detection in such settings.

6.2 Experiments on synthetic graphs

The vast majority of the millions of RDF graphs experimented with in the previous section are quite trivial to process: we see this as a general trend in real-world data where blank nodes will often be unconnected or will be associated with discriminating ground information, making the search space for leaning and labelling quite small. However, we also saw a few examples of more difficult cases appearing in the BTC−14 dataset, including an RDF graph with a 16-clique of blank nodes. In general, we are thus interested to stress-test our algorithm with more difficult cases.

For this, we apply our algorithms over a selection of synthetic graphs often considered in the graph isomorphism literature; we source our graphs from the Bliss benchmark [30], which was originally defined for evaluating graph isomorphism.[24] More specifically, we take five well-known classes of undirected graphs at various sizes and represent them as RDF graphs (using a single predicate with edges in both directions). We also take a set of Miyazaki graphs, which were constructed to be a particularly tough case for graph isomorphism [45].

We thus consider the following six classes of graph:

[23]http://es.openfoodfacts.org/data/es.openfoodfacts.org.products.rdf
[24]http://www.tcs.hut.fi/Software/bliss/benchmarks/index.shtml

Grid 2D A $k \times k$ grid of blank nodes where each of the k^2 blank nodes is associated with a unique coordinate (a, b) (for $1 \leq a \leq k$, $1 \leq b \leq k$) and a pair of blank nodes are connected if and only if they have a distance of one: the resulting graph has k^2 blank nodes, with $2k(k-1)$ undirected edges and $4k(k-1)$ triples.

Grid-3D A $k \times k \times k$ grid of blank nodes where each of the k^3 blank nodes has a unique coordinate (a, b, c) (for $1 \leq a \leq k$, $1 \leq b \leq k$, $1 \leq c \leq k$) and a pair of blank nodes are connected if and only if they have a distance of one: the resulting graph has k^3 blank nodes, with $3k^2(k-1)$ undirected edges and $6k^2(k-1)$ triples.

Clique A graph of k blank nodes that are then pairwise connected by edges (excluding self-loops), also known as a *complete graph*: the resulting graph has k blank nodes and $\frac{k(k-1)}{2}$ undirected edges and $k(k-1)$ triples.

Rook A $k \times k$ grid of blank nodes, also known as a *lattice graph*,[25] where each of the k^2 blank nodes is associated with a unique coordinate (a, b) (for $1 \leq a \leq k$, $1 \leq b \leq k$) where a pair of blank nodes is connected if and only if they are on the same row or column. The resulting graph has k^2 blank nodes, with $k^2(k-1)$ undirected edges and $2k^2(k-1)$ triples.

Triangle The line graph[26] of the k-clique, also known as a *triangular graph*: the resulting graph has $\frac{k(k-1)}{2}$ blank nodes, $\frac{k(k-1)(k-2)}{2}$ edges and $k(k-1)(k-2)$ triples.

Miyazaki This class of graphs was constructed by Miyazaki [45] to invoke checking an exponential number of labellings in Nauty-style algorithms for deciding graph isomorphism, even when automorphisms are pruned; the constructed (undirected) graphs are 3-regular[27] and 4-colourable[28]. Each graph has $20k$ blank nodes, $30k$ edges, and $60k$ triples.

A timeout of ten minutes was set. The experiments were run on a laptop with 1GB of heap-space and an Intel i7-5600 2.6GHz processor. Experiments are run in a single-threaded manner where the data are loaded into memory before the runtime clock is started. As before, we first present results for labelling the RDF graphs to compute the iso-canonical form, and then present results for leaning the RDF graphs (and labelling them) to compute the equi-canonical form.

6.2.1 Iso-canonical experiments. For computing iso-canonical labels, we use Murmur3_128: the fastest hashing method per Table 2. We perform testing both with (Murmur3_128) and without (Murmur3_128*) automorphism-based pruning.

We plot the results in Figure 4 for the six classes of graphs previously introduced. In each plot, the x-axis denotes graphs with varying levels of k for the class in question and the y-axis shows the total time in milliseconds using log-scale. The maximum y value indicates the 10 minute timeout, where points on the top line of the plot indicate an error for that value of k. Note that as k grows linearly, the number of triples and blank-nodes in the resulting graph often grows much faster, depending on the class of graph; for example, in a Grid-3D graph, for a given value of k, the number of blank nodes is k^3 and the number of triples is $6k^2(k-1)$. From these plots we can see that the automorphism-based pruning is an important optimisation: when compared with pruning disabled, the configuration with pruning enabled is faster in all cases, and allowing larger instances of more challenging classes of graphs – classes with inherently many automorphisms – to be processed.

[25]In our previous work we referred to them as lattice graphs [27] following the source Bliss paper [30], but this can be confused with a more general class of graphs of the same name. The name "rook graph" comes from the intuition that the connectivity of the graph represents the mobility of the rook piece in chess over the grid of nodes.

[26]The line graph of an undirected graph is constructed by converting the edges of the original graph into nodes in the line graph and connecting the nodes in the line graph if and only if the corresponding edges in the original graph share a node.

[27]. . . all nodes have a degree of 3.

[28]. . . one can assign four colours to each node such that no adjacent nodes have the same colour.

Table 4. Largest graphs labelled by MURMUR3_128

Class	k	**Triples**	**BNodes**	**Time** (s)	**Error**
GRID-2D	100	39,600	10,000	43.6	All succeeded
GRID-3D	19	38,988	6,859	27.2	All succeeded
CLIQUE	32	992	32	18.5	Memory ($k = 33$)
ROOK	16	7,680	256	132.3	Time ($k = 17$)
TRIANGLE	17	4,080	136	12.3	Time ($k = 18$)
MIYAZAKI	8	480	160	136.2	Time ($k = 10$)

In Table 4, for each class of graph, we summarise the largest instance that was successfully processed within the limitations of the experiment (1GB of memory, 10 minutes of time) by the MURMUR3_128 configuration with pruning enabled. In particular, the table indicates the class, the largest value of k processed successfully, the number of triples for that value of k, the number of blank nodes for that value of k, the time take to process that graph, and the error incurred (if any) when processing the next largest instance. We can see that with this configuration (found to be optimal in previous experiments), for some classes of graph, the iso-canonicalisation process can handle quite large instances, processing, for example, instances of GRID-2D and GRID-3D with tens of thousands of triples within a minute, and instances of CLIQUE of size up to $k = 32$ within the 1GB memory limit. However, when looking at the hardest cases – MIYAZAKI graphs – we see that the process times-out for moderately sized graphs. One may note that there are large leaps in the time required for some classes, where for example in the TRIANGLE class at $k = 17$, the process succeeded within 12.3 seconds but failed to process $k = 18$ within 600 seconds; we highlight again that the size of the graphs grows much faster than the value of k and that the process has an exponential worst-case complexity, where these factors may combine to explain such jumps. We also see that the class CLIQUE runs of memory when $k = 33$; we believe this is due to the pruning optimisation that stores the orbits for tracking automorphisms.

From these results we can conclude that:

- computing the iso-canonical labelling is feasible for moderately-sized synthetic instances, where using automorphisms to prune the search tree of labellings helps reduce the search space in cases of graphs that are (strongly) regular;
- the size of synthetic cases successfully labelled easily covers the difficult cases found in real-world graphs (in particular, the 16-clique we discovered);
- however, there are certain constructions of graph – where we have looked at MIYAZAKI graphs – that incur the exponential worst case and thus fail for relatively modestly sized graphs in the order of 600 triples and 200 blank nodes.

6.2.2 Equi-canonical experiments. We now present the results for leaning instances of the same classes of graph. We experiment with BFS, DFS, DFS* (pruning disabled), DFS-RAND and LABEL (labelling with MURMUR3_128 *after* DFS leaning).[29]

In fact, the six synthetic classes of graphs we use present some interesting cases since GRID-2D and GRID-3D can be leaned down to a graph of two blank nodes and two triples, ROOK graphs of rank k can be leaned down to a k-clique, certain instances of TRIANGLE graphs can also be leaned[30],

[29]Since LABEL requires a prior leaning step, we only present such results where leaning succeeded. We do not present DFS+LABEL since it would be mostly indistinguishable from DFS in the runtime results.

[30]We are admittedly not sure what pattern the cores of the TRIANGLE instances follow, but empirically it appears that graphs where k is odd are lean whereas graphs where k is even are not.

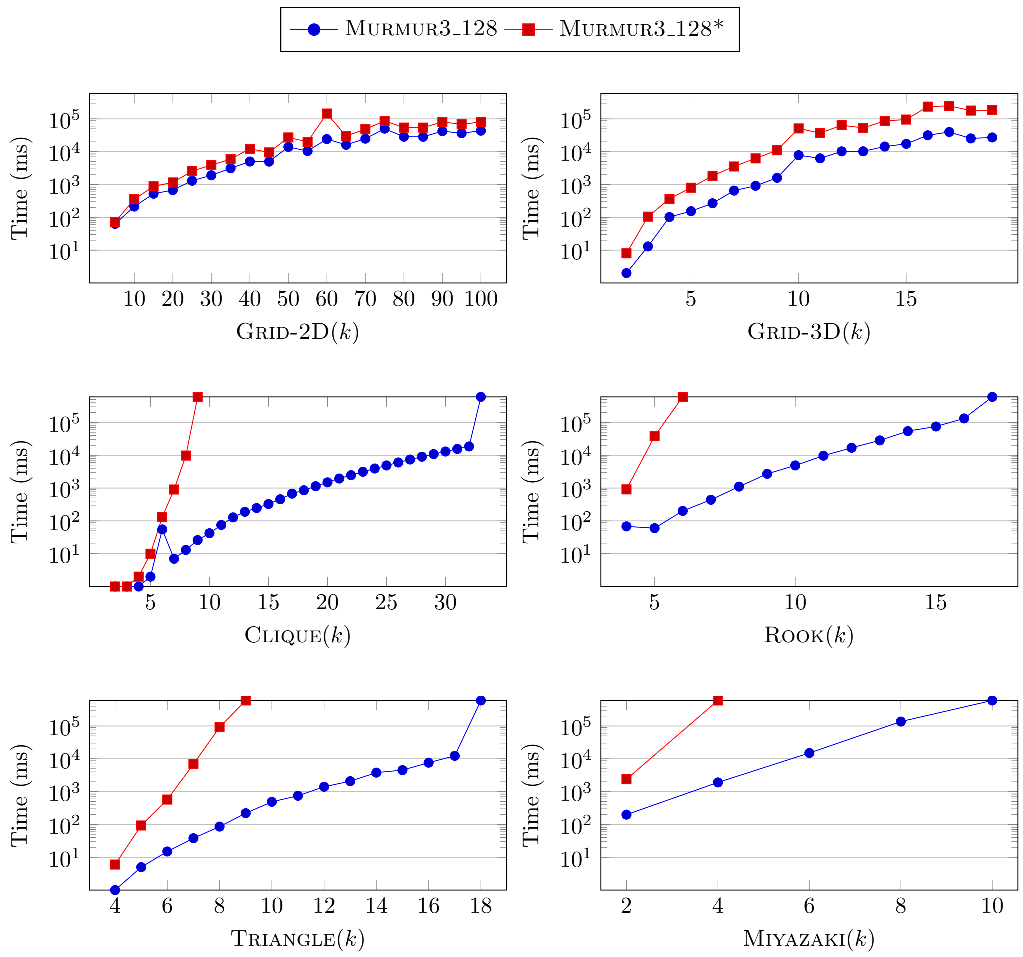


Fig. 4. Iso-canonical results for synthetic graphs

and our MIYAZAKI instances can be leaned down to a cycle of three blank nodes and six triples[31]; on the other hand, CLIQUE graphs are always lean. Thus, within these six synthetic classes, we have a mix of lean graph instances and non-lean graph instances.

The results are summarised in Figure 5. All configurations succeeded on all GRID-2D instances, while only DFS succeeded for all GRID-3D and MIYAZAKI graphs. Other configurations and classes encountered an exception for some instance: in every case the BFS strategy threw an out-of-memory exception since it tries to materialise and store all solutions in memory, whereas the DFS variants mostly tended to time-out. Overall, while for certain classes of graph we see that the DFS pruning and search heuristics have little impact on performance (such as in the case of GRID-2D, or indeed in the case of CLIQUE where the instances are already lean), in other classes of graph, these heuristics become essential: the DFS configuration succeeds for the most instances in all classes of graph,

[31]This is actually a speculation based on the empirical results of DFS rather than something we can prove at the moment; in fact, this implies that the instances of the graphs we take from the BLISS benchmark are 3-colourable, not just 4-colourable.

Table 5. Largest graphs leaned by DFS

Class	k	**Triples**	**BNodes**	**Time** (s)	**Error**
Grid-2D	100	39,600	10,000	3.8	All succeeded
Grid-3D	13	12,168	2,197	9.1	Memory ($k = 14$)
Clique	10	90	10	413.7	Time ($k = 11$)
Rook	4	96	16	1.5	Time ($k = 5$)
Triangle	6	120	15	474.6	Time ($k = 7$)
Miyazaki	50	3,000	1,000	0.6	All succeeded

where in the Grid-3D and Miyazaki classes, the benefits of incorporating these heuristics into the DFS approach are most evident.

From the Label series, we can see that when leaning succeeded, labelling the resulting graph was relatively trivial. In comparison with Figure 4 for the labelling results, we see that leaning is much harder for most classes of graph, particularly Clique, Rook and Triangle. However, in the case of Miyazaki, computing the equi-canonical form by first removing redundancy from the graph is much more efficient (all instances up to $k = 50$ were leaned and labelled in under a second) than computing the iso-canonical form over the raw graph (where the process times-out already at $k = 10$) since the cores of such graphs are simple and thus trivial to label.

Finally, in Table 5, we present details of the largest instances of each class of graph that could be successfully leaned by the DFS configuration. We see that most of the failures again tended to relate to a timeout, though some out-of-memory exceptions were also encountered. Again, relatively large instances of Grid-2D and Grid-3D could be processed, though leaning did fail for relatively small instances of Rook and Triangle graphs. Referring back to the real-world experiments, we found a graph representing a 16-clique of blank nodes, where we can also see from this table that such graphs are out of reach of the current DFS algorithm, which could only process Clique until $k = 10$. On the other hand, in comparison with the labelling results in Table 4, we see that much larger instances of Miyazaki graphs can be leaned.

From these results we can conclude that:

- computing the equi-canonical form of an RDF graph is more difficult in general for these instances than computing the iso-canonical form;
- however, the DFS configuration can efficiently lean Miyazaki graphs, whose cores are then trivial to label—hence in some cases, it may be more efficient to compute the equi-canonical form than the iso-canonical form;
- the size of synthetic cases that were successfully leaned does not cover all difficult cases found in real-world graphs, where we are quite far from being able to process the 16-clique found in real-world data using the DFS configuration;
- in cases where the input RDF graph is already lean, there is no significant difference between the leaning strategies since all such strategies need to check through all endomorphisms to ensure that none are proper.
- the DFS ordering heuristic greatly extends the size of instance that can be processed for certain classes of graph, particularly when the target core is quite small;
- likewise isomorphism-based pruning helps when leaning certain classes of graphs (though perhaps not to the same extent as seen in the labelling process).

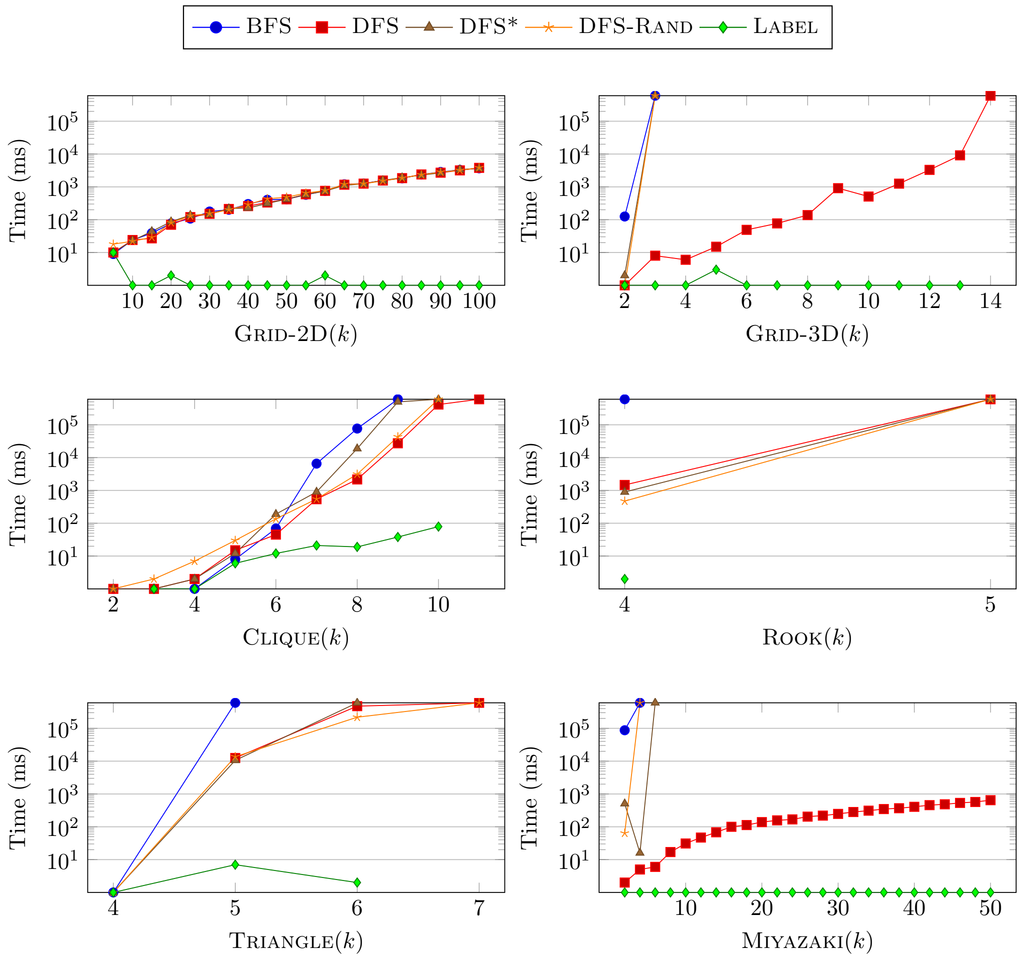


Fig. 5. Equi-canonical results for synthetic graphs

7 RELATED WORK

In the following, we discuss works relating to blank nodes in RDF graphs, canonicalising RDF graphs, signing RDF graphs, etc., as relevant to the current contribution.

Carroll [9] proposed methods to canonicalise RDF graphs with blank nodes in such a manner that they could be digitally signed; thus the goals of his work are quite similar to the present contribution. The method Carroll 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. While perhaps a practical compromise, the end result is that the approach circumvents the problem of isomorphism rather than solving it: isomorphic RDF graphs may end up with different signatures depending on what triples are added.

Sayers and Karp [51] likewise proposed methods to compute the digest of an RDF graph, similar in principle to the idea of computing a signature for an RDF graph. The algorithm is claimed to run in O(γ), for γ the size of the graph. This already suggests that the algorithm is not sound and complete with respect to the isomorphism of RDF graphs. Rather the authors propose to either assume all blank node labels as fixed, or to use a similar method to Carroll [9] where additional triples are assigned to blank nodes to artificially distinguish them. Hence they side-step problems with blank nodes but have similar problems as for the case of Carroll [9].

Tummarello et al. [56] subsequently proposed methods to sign fragments of an RDF graph. The fragments in question are called Minimally Self-contained Graphs (MSGs), which are identical to the graphs contained in our notion of a blank node split (see Definition 4.6) except that each ground triple is considered to form its own split. The authors do not directly deal with the problem of canonicalisation, but rather propose to re-use the method of Carroll [9]; thus the proposed framework does not robustly handle all isomorphic cases of RDF graphs.

Giereth [14] proposed a similar framework that allows for encrypting (and decrypting) selected parts of an RDF graph using standard cryptographic methods; the parts in question can be an RDF term, an RDF triple, or an RDF sub-graph. The author notes that blank nodes can cause complications when encrypting sub-graphs of the original RDF graph but – in a similar vein to the proposals of Carroll [9] and Sayers and Karp [51] – the author proposes to artificially add new triples to distinguish individual blank nodes in the encryption/decryption process.

More recently, Arias-Fisteus et al. [2] proposed methods to canonicalise Notation3 (N3) graphs, which extend RDF graphs with syntax for universally quantified variables and more complex formulae. The core of their approach is very similar in principle to the hashing methods described in Algorithm 1. However, they propose a polynomial-time solution, which they claim to be complete and correct; if this were the case, we could conclude GI = P, solving a long-open problem. The authors acknowledge that the results of a similar hashing process to Algorithm 1 can fail to distinguish blank nodes, as per Example 4.11, and propose a solution to this problem based on the hash partition of blank nodes. However, in this solution, they assume that it does not matter which blank node is selected from the partition; in other words, they assume that in Figure 1, all leafs will always produce the same labelled graph and thus they take the first leaf. As previously discussed, however, counterexamples for such polynomial-time algorithms have been found [7]. Such counterexamples are used in the construction of the Miyazaki class of graphs used earlier, where the approach of Arias-Fisteus et al. will fail to identify isomorphic instances of such classes of graph.[32] It is important to emphasise, however, that known counterexamples are exotic, and thus such an approach – while not formally correct – will work fine for almost all practical cases, though it could be vulnerable to deliberate attacks (depending on the application).

In 2012, Longley and Sporny first proposed the Universal RDF Graph Normalization Algorithm 2012 (URGNA2012); this was later updated to the Universal RDF Dataset Normalization Algorithm 2015 (URDNA2015) [37].[33] Like the present work, the UR*NA series of algorithms centre on computing canonical labels for blank nodes, and follow a similar procedure of recursive hashing of the terms surrounding blank nodes. However, rather than executing a recursive procedure similar to Nauty (as we do here), URDNA2015 instead computes a hash for each individual blank node based on paths emanating from it through other connected blank nodes, basing the blank node hash on the path that is the lowest lexicographically of the potentially non-deterministic options. The URDNA2015 update further supports RDF datasets, rather than RDF graphs as considered here.

[32]In the case of Miyazaki with $k = 6$, for example, we found 228 distinct leaf graphs.

[33]This paragraph was added after publication as an erratum to provide due credit to the earlier work by Longley and Sporny, which though not cited in the published version, establishes an earlier precedent for the canonical labelling of blank nodes.

On the other hand, although the URDNA2015 algorithm appears to offer a sound and complete procedure for computing canonical blank node labels, a formal proof as such has yet to be provided. Furthermore, in the current paper we address semantic redundancy in RDF graphs, offering the possibility to remove non-lean blank nodes from the RDF graph before applying canonical labelling.

Kasten et al. [32] later proposed a framework for signing RDF graphs that is capable of dealing with isomorphic cases through use of a prior canonicalisation step. However, they do not propose a novel canonicalisation procedure, but rather study the practicality of using the existing methods proposed by Carroll [9], Sayers and Karp [51] and Arias-Fisteus et al. [2]. While noting that these methods propose polynomial-time canonicalisation methods, they do not observe that such methods may fail on isomorphic instances of certain classes of graph.

Höfig and Schieferdecker [26] proposed another framework for computing a hash-based digest of an RDF graph, with focus on the issue of blank nodes. Their method is based on the notion of MSGs introduced by Tummarello et al. [56], where starting with a blank node appearing as a subject, the algorithm creates a string by recursively traversing through the predicates to the objects, thus capturing the structure of the entire MSG; this string can then be ordered and hashed. The authors claim that their algorithm runs in time $O(n^n)$, and thus is consistent with isomorphism being a GI-complete problem. However, from our understanding of the proposed approach, this analysis is not correct and the algorithm has a polynomial-time worst-case, essentially performing a depth-first traversal of the MSG for each unique subject that it contains. Likewise we believe that the authors do not consider ties in the ordering of strings caused by non-trivial automorphisms or difficult counterexamples such as Miyazaki graphs. We highlight, however, that these appear to have been initial results and were published at a workshop venue.

Lantzaki et al. [34] 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 their goal of computing deltas and our goals differ somewhat, the signature method that they propose for blank nodes is similar to a non-recursive version of Algorithm 1; they also present results for detecting isomorphism (where the delta is zero). However, their approach is to implement polynomial-time algorithms that offer an approximation of a delta, rather than a complete algorithm as in our case.

Jena [40] 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) iso-canonical labelling. Hence, for example, the methods provided by Jena would not be suitable for detecting isomorphically duplicated RDF graphs in a large collection such as the BTC−14 dataset; with our methods, one can compute a unique iso- or equi-canonical hash for each graph to track duplicates, whereas with Jena, a quadratic number of pairwise isomorphism checks are needed.

Kuhn and Dumontier [33] propose a method to compute a cryptographic hash of an online document – such as an RDF graph – which they then propose to encode in what they call a "*trusty IRI*", such that the content used to mint that IRI can be verified using that IRI. Thus, for example, an IRI can implicitly serve to verify an RDF graph that is intended to be the description of the resource it identifies. In their discussion, they mention the complications of computing hashes over RDF graphs containing blank nodes, where their solution is to simply mint fresh Skolem IRIs for each blank node; hence, their method does not guarantee to produce the same hash for isomorphic RDF graphs and likewise implicitly assumes that the Skolems produced for hashing are known to the person who wishes to verify the graph. As such, trusty IRIs could be a useful application of the work presented herein, allowing to compute trusty IRIs in a consistent manner for isomorphic RDF graphs, additionally allowing the verification of RDF graphs with blank nodes.

We previously conducted an extensive survey of blank nodes [28, 38], covering their theory and practice, their semantics and complexity, how they are used in the standards and in published data, etc. In an analysis of the BTC−12 corpus (a similar dataset as the BTC−14 corpus used herein, but two years antecedent), we found that although the maximum treewidth of blank nodes was 6, most graphs do not contain cycles of blank nodes.[34] We also found that in a merge of the BTC−12 corpus, 6% of the blank nodes were redundant under simple entailment [28]; for this, we used a signature method similar to the naive algorithm proposed earlier, but for a fixed depth. To apply leaning, we also used some methods that inspired the current approaches, but with a slightly different focus: in that paper, we focussed on leaning RDF data in bulk using on disk methods aimed at the vast majority of simple cases, whereas herein we focus on general algorithms for leaning individual graphs in-memory aimed to support more difficult cases.

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 [38]. Booth [6] refers to this proposed profile as "Well Behaved RDF". Although disallowing blank node cycles would simplify matters greatly, in this paper we show that in terms of labelling and leaning, many real-world graphs, even with quite complex blank node structures (including lots of cycles and non-trivial treewidth), can be processed quite cheaply. Our results support the conjecture that exponential cases are unlikely to be found in "real-world" RDF graphs, and thus that a special profile of RDF may not be necessarily warranted if the argument is purely for efficiency reasons.

Compared to these previous works, to the best of our knowledge, our proposal is the first that is formally proven to correctly produce an iso-canonical form for all RDF graphs without having to artificially distinguish blank nodes with fresh ground edges, and is the first work to look at computing a general equi-canonical form for RDF graphs.

8 CONCLUSIONS

In this paper, we proposed methods to compute both an iso-canonical form and an equi-canonical form of an RDF graph. We propose that such forms could serve a variety of applications for RDF graphs, such as for digitally signing RDF graphs [9, 51, 56], or computing trusty IRIs [33], or Skolemising blank nodes in a consistent manner [11], or for finding duplicate RDF graphs in large collections or otherwise comparing RDF graphs [32, 34], and so forth.

The iso-canonical form of an RDF graph will produce the same result for a pair of input RDF graphs if and only if they are isomorphic (barring some extremely unlikely hash collision). Our approach for computing this iso-canonical graph is to apply a canonical labelling scheme to the blank nodes in the RDF graph where we first compute initial hashes for the blank nodes based on ground information in the RDF graph; if these initial hashes fail to distinguish the blank nodes, we then apply a search process – inspired by the Nauty algorithm – to find the lowest possible isomorphic RDF graph based on a total ordering of RDF graphs that does not use blank node labels and that applies certain restrictive rules to narrow the search space; we then discuss how detected automorphisms can be used to further prune the search process. Thereafter, we discussed how global labels can be computed for blank nodes – e.g., for the purposes of Skolemisation – by computing a hash of the output iso-canonical graph and encoding it into the blank nodes labels such that each such label encodes the information of the iso-canonical graph containing it. Despite the worst-case exponential behaviour of our proposed algorithms, in experiments we showed that real-world graphs can be processed, on average, in about 3 milliseconds, and that moderately

[34]In the BTC−14 corpus, the 16−clique would have a treewidth of 15.

sized synthetic graphs can be processed in reasonable time; however, we showed that there are specialised classes of graphs that do incur exponential time.

Likewise the equi-canonical form will produce the same result for pairs of simple-equivalent RDF graphs. Our approach is to first lean the RDF graph and then compute the iso-canonical form of the resulting core. To lean the RDF graph, we propose a method based on the idea of trying to find a *core endomorphism*: a mapping from blank nodes in the graph back to the graph itself that is minimal in the number of unique blank nodes it maps to. To begin with, we process the unconnected blank nodes in the graph and produce a list of candidate mappings based on ground information. If there remain some connected blank nodes with multiple candidates, we then propose a breadth-first strategy and a depth-first strategy for finding a complete core endomorphism, where the former strategy computes all homomorphisms and selects the one with the fewest blank nodes in its codomain, while the latter tries to directly construct the core endomorphism using some search heuristics. Again, these methods have worst-case exponential behaviour. In experiments over real-world graphs, we found that the BFS strategy incurred many timeouts, but that for the DFS strategy, while attempting to process 9.9 million RDF graphs with blank nodes, only 5 failed to be processed within a ten minute timeout, where on average each graph took about 8 milliseconds. We likewise looked at synthetic cases where we found that the DFS strategy – which performs iterative simplified rewritings on the graph – is particularly beneficial when the output core of the graph is small, and can efficiently process some cases that were exponential for our iso-canonical methods. In general, however the number of instances of synthetic graphs that could be leaned were fewer than those that could be labelled in the analogous iso-canonicalisation experiments.

There are various ways in which the current work could be improved. When computing the iso-canonical form of an RDF graph, we currently apply a search process inspired by the Nauty algorithm, where other alternatives – such as Bliss – have been explored in the graph isomorphism literature and could likewise be applied and studied in the context of RDF. Also there have been some results on computing difficult cases such as Miyazaki graphs in a more efficient manner [54]. It would also be interesting to study classes of graphs for which it is known that all leafs in the search tree will produce the same graph; assuming such a classification could be performed efficiently, we could forego the exponential search tree of Figure 1 over such instances, instead taking the first leaf as the canonically-labelled graph (in a similar manner to the algorithm of Kasten et al. [32]).

On the other hand, when computing the equi-canonical form of an RDF graph, we apply the leaning and labelling steps separately, where it may be possible to re-use some of the work done in the former step for the latter; as a trivial example, we could pass the automorphisms found in the former step to help prune the search in the latter. In our DFS strategy, it may be interesting to investigate (the possibility of) search heuristics that guarantee to identify the core endomorphism in the first iteration, obviating the need to verify, for example, that the intermediate output is lean.

Aside from such algorithmic improvements, it would also be interesting to investigate parallelisation of the above processes, taking advantage of – for example – modern multi-core processors or perhaps even GPU processing.

More broadly speaking, there are various directions in which this work could be extended that go beyond computing iso-canonical and equi-canonical RDF graphs.

When comparing RDF graphs, one could further consider equivalence under entailment regimes that interpret datatypes or special (ontological) vocabulary, such as proposed by the RDF(S) [21] and OWL 2 [17] standards. Much like in the present work, the goal would be to deterministically compute a redundancy-free "core" of an RDF graph that permits the same models under more complex interpretations [46]. Likewise, rather than study equivalence relations between graphs, various asymmetric relationships – such as deciding entailment [21] or computing deltas [34] between graphs – are also of practical relevance, where some of the methods described herein – such as for efficiently computing the core of a graph – could be brought to bear on such problems.

Such methods would still not take into the account that there are various (non-isomorphic/non-homomorphic) possibilities in terms of representing equivalent content as a graph. For example, we recently studied various ways in which the Wikidata knowledge-base [57] could be encoded as RDF using various forms of *reification*: various representational forms of encoding higher-arity relations as graphs [25]. This general idea of varying representations being able to encode the same "content" has given rise to the related notions of *information preservation* [13], *design independence* [55] and *representational independence* [10], whereby particular algorithms aim to be as agnostic as possible to the particular representation chosen in the data-modelling process. A more speculative direction, then, would be to define and implement methods for comparing or canonicalising RDF graphs in a representationally independent manner, for example, to consider (i) *invertibility* [13]: can two RDF graphs be mapped to each other under a given mapping language, and (ii) *query preservation* [13]: can the same answers be generated over both graphs for a given query language?

Finally, the methods presented in this paper could be used for canonicalising a variety of structures other than RDF graphs. For example, in the area of data exchange, we previously discussed how many such methods are based on the notion of computing core solutions [50], where the equi-canonical methods proposed herein could be adapted to such settings. Likewise, the current work directly suggests strategies for canonicalising SPARQL queries, both in terms of removing redundancy and computing canonical labels for variables, which would help solve the equivalence problem for queries [36] but without requiring pairwise-comparison. Graph models used in other settings, such as the property graph model [1], could likewise be canonicalised with adaptations of the methods proposed herein. More generally, any (semi-)structured representation that can be encoded to (and decoded from) an RDF graph – through a deterministic translation that preserves the isomorphism (or equivalence) relation – can be canonicalised using our framework.

To conclude, although the problems we tackle – computing iso-canonical and equi-canonical forms for RDF graphs – are intractable, in this paper we provide complete algorithms for both cases that, although exponential in the worst-case, we have demonstrated to be practical for a large collection of real-world RDF graphs and indeed a wide variety of more challenging synthetic instances. Thus, if we can try to generalise the experiences gained this paper, although blank nodes do complicate various comparisons and operations on RDF graphs, we can observe that the sorts of worst-cases predicted in theory are not likely to occur naturally in practical settings, and that efficient – albeit worst-case exponential – algorithms, such as those presented in this paper, are achievable for many such problems. Generalising further, we can argue that complexity classes and worst case analyses – though useful – do not by themselves refute the practicality of sound and complete methods for solving a particular problem: worst-case analyses never tell the full story.

ACKNOWLEDGMENTS

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 input as well as the anonymous referees whose feedback helped improve the paper. He would also like to thank Antoine Zimmermann, Ivan Herman, and Manu Sporny for pointing out various errata that have been corrected in the present version.

REFERENCES

[1] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan L. Reutter, and Domagoj Vrgoc. 2016. Foundations of Modern Graph Query Languages. *CoRR* abs/1610.06264 (2016), 1–50. http://arxiv.org/abs/1610.06264

[2] Jesús Arias-Fisteus, Norberto Fernández García, Luis Sánchez Fernández, and Carlos Delgado Kloos. 2010. Hashing and canonicalizing Notation 3 graphs. *J. Comput. Syst. Sci.* 76, 7 (2010), 663–685.

[3] László Babai. 2015. Graph Isomorphism in Quasipolynomial Time. *CoRR* abs/1512.03547 (2015), 1–89. http://arxiv.org/abs/1512.03547

[4] László Babai, Paul Erdös, and Stanley M. Selkow. 1980. Random Graph Isomorphism. *SIAM J. Comput.* 9, 3 (1980), 628–635.

[5] David Beckett, Tim Berners-Lee, Eric Prud'hommeaux, and Gavin Carothers. 2014. RDF 1.1 Turtle – Terse RDF Triple Language. W3C Recommendation. (25 Feb. 2014). http://www.w3.org/TR/turtle/.

[6] David Booth. 2012. Well Behaved RDF: A Straw-Man Proposal for Taming Blank Nodes. (2012). http://dbooth.org/2013/well-behaved-rdf/Booth-well-behaved-rdf.pdf.

[7] Jin-yi Cai, Martin Fürer, and Neil Immerman. 1992. An optimal lower bound on the number of variables for graph identifications. *Combinatorica* 12, 4 (1992), 389–410.

[8] Gavin Carothers. 2014. RDF 1.1 N-Quads. W3C Recommendation. (2014). http://www.w3.org/TR/n-quads/.

[9] Jeremy J. Carroll. 2003. Signing RDF Graphs. In *International Semantic Web Conference*. 369–384.

[10] Yodsawalai Chodpathumwan, Amirhossein Aleyasen, Arash Termehchy, and Yizhou Sun. 2016. Towards Representation Independent Similarity Search Over Graph Databases. In *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management, CIKM 2016, Indianapolis, IN, USA, October 24-28, 2016*. ACM, 2233–2238.

[11] Richard Cyganiak, David Wood, and Markus Lanthaler. 2014. RDF 1.1 Concepts and Abstract Syntax. W3C Recommendation. (25 Feb. 2014). http://www.w3.org/TR/rdf11-concepts/.

[12] R. Fagin, P. G. Kolaitis, and L. Popa. 2005. Data exchange: getting to the core. *TODS* 30, 1 (2005), 174–210.

[13] Wenfei Fan and Philip Bohannon. 2008. Information preserving XML schema embedding. *ACM Trans. Database Syst.* 33, 1 (2008), 4:1–4:44.

[14] Mark Giereth. 2005. On Partial Encryption of RDF-Graphs. In *The Semantic Web - ISWC 2005, 4th International Semantic Web Conference, ISWC 2005, Galway, Ireland, November 6-10, 2005, Proceedings*. Springer, 308–322.

[15] Georg Gottlob. 2005. Computing cores for data exchange: new algorithms and practical solutions. In *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*. ACM, 148–159.

[16] Georg Gottlob and Alan Nash. 2008. Efficient core computation in data exchange. *J. ACM* 55, 2 (2008).

[17] Bernardo Cuenca Grau, Boris Motik, Zhe Wu, Achille Fokoue, and Carsten Lutz. 2009. OWL 2 Web Ontology Language: Profiles. W3C Recommendation. (27 Oct. 2009). http://www.w3.org/TR/owl2-profiles/.

[18] Claudio Gutierrez, Carlos A. Hurtado, Alberto O. Mendelzon, and Jorge Pérez. 2011. Foundations of Semantic Web databases. *J. Comput. Syst. Sci.* 77, 3 (2011), 520–541.

[19] Steve Harris, Andy Seaborne, and Eric Prud'hommeaux. 2013. SPARQL 1.1 Query Language. W3C Recommendation. (21 March 2013). http://www.w3.org/TR/sparql11-query/.

[20] Patrick Hayes. 2004. RDF Semantics. W3C Recommendation. (10 Feb. 2004). http://www.w3.org/TR/2004/REC-rdf-mt-20040210/.

[21] Patrick Hayes and Peter F. Patel-Schneider. 2014. RDF 1.1 Semantics. W3C Recommendation. (25 Feb. 2014). http://www.w3.org/TR/2014/REC-rdf11-mt-20140225/.

[22] Tom Heath and Christian Bizer. 2011. *Linked Data: Evolving the Web into a Global Data Space*. Vol. 1. Morgan & Claypool. 1–136 pages. Issue 1.

[23] Pavol Hell and Jaroslav Nešetřil. 1992. The core of a graph. *Discrete Mathematics* 109, 1-3 (1992), 127–126.

[24] Ivan Herman, Ben Adida, Manu Sporny, and Mark Birbeck. 2013. RDFa 1.1 Primer – Second Edition – Rich Structured Data Markup for Web Documents. W3C Working Group Note. (22 Aug. 2013). http://www.w3.org/TR/rdfa-primer/.

[25] Daniel Hernández, Aidan Hogan, and Markus Krötzsch. 2015. Reifying RDF: What Works Well With Wikidata?. In *Proceedings of the 11th International Workshop on Scalable Semantic Web Knowledge Base Systems co-located with 14th International Semantic Web Conference (ISWC 2015), Bethlehem, PA, USA, October 11, 2015. (CEUR Workshop Proceedings)*, Vol. 1457. 32–47. http://ceur-ws.org/Vol-1457/SSWS2015_paper3.pdf

[26] Edzard Höfig and Ina Schieferdecker. 2014. Hashing of RDF Graphs and a Solution to the Blank Node Problem. In *Proceedings of the 10th International Workshop on Uncertainty Reasoning for the Semantic Web (URSW 2014) co-located with the 13th International Semantic Web Conference (ISWC 2014), Riva del Garda, Italy, October 19, 2014. (CEUR Workshop Proceedings)*, Vol. 1259. 55–66. http://ceur-ws.org/Vol-1259/method2014_submission_1.pdf

[27] Aidan Hogan. 2015. Skolemising Blank Nodes while Preserving Isomorphism. In *International Conference on World Wide Web (WWW)*. 430–440.

[28] Aidan Hogan, Marcelo Arenas, Alejandro Mallea, and Axel Polleres. 2014. Everything you always wanted to know about blank nodes. *J. Web Sem.* 27 (2014), 42–69.

[29] Aidan Hogan, Jürgen Umbrich, Andreas Harth, Richard Cyganiak, Axel Polleres, and Stefan Decker. 2012. An empirical survey of Linked Data conformance. *J. Web Sem.* 14 (2012), 14–44.

[30] Tommi A. Junttila and Petteri Kaski. 2007. Engineering an Efficient Canonical Labeling Tool for Large and Sparse

Graphs. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*.
[31] Tobias Käfer and Andreas Harth. 2014. Billion Triples Challenge data set. http://km.aifb.kit.edu/projects/btc-2014/. (2014).
[32] Andreas Kasten, Ansgar Scherp, and Peter Schauß. 2014. A Framework for Iterative Signing of Graph Data on the Web. In *The Semantic Web: Trends and Challenges - 11th International Conference, ESWC 2014, Anissaras, Crete, Greece, May 25-29, 2014. Proceedings*. Springer, 146–160.
[33] Tobias Kuhn and Michel Dumontier. 2014. Trusty URIs: Verifiable, Immutable, and Permanent Digital Artifacts for Linked Data. In *ESWC*. 395–410.
[34] Christina Lantzaki, Panagiotis Papadakos, Anastasia Analyti, and Yannis Tzitzikas. 2017. Radius-aware approximate blank node matching using signatures. *Knowl. Inf. Syst.* 50, 2 (2017), 505–542.
[35] Ora Lassila and Ralph R. Swick. 1999. Resource Description Framework (RDF) Model and Syntax Specification. W3C Recommendation. (22 Feb. 1999). http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/.
[36] Andrés Letelier, Jorge Pérez, Reinhard Pichler, and Sebastian Skritek. 2013. Static analysis and optimization of semantic web queries. *ACM Trans. Database Syst.* 38, 4 (2013), 25:1–25:45.
[37] Dave Longley and Manu Sporny. 2019. RDF Dataset Normalization: A Standard RDF Dataset Normalization Algorithm. W3C Draft Community Group Report. (18 Jan. 2019). https://json-ld.github.io/normalization/spec/.
[38] Alejandro Mallea, Marcelo Arenas, Aidan Hogan, and Axel Polleres. 2011. On Blank Nodes. In *International Semantic Web Conference*. 421–437.
[39] Bruno Marnette, Giansalvatore Mecca, and Paolo Papotti. 2010. Scalable Data Exchange with Functional Dependencies. *PVLDB* 3, 1 (2010), 105–116. http://www.comp.nus.edu.sg/~vldb2010/proceedings/files/papers/R09.pdf
[40] Brian McBride. 2002. Jena: A Semantic Web Toolkit. *IEEE Internet Computing* 6, 6 (2002), 55–59.
[41] Brendan McKay. 1980. Practical Graph Isomorphism. In *Congressum Numerantium*, Vol. 30. 45–87.
[42] Brendan D. McKay and Adolfo Piperno. 2014. Practical graph isomorphism, II. *J. Symb. Comput.* 60 (2014), 94–112.
[43] Giansalvatore Mecca, Paolo Papotti, and Salvatore Raunich. 2012. Core schema mappings: Scalable core computations in data exchange. *Inf. Syst.* 37, 7 (2012), 677–711.
[44] Robert Meusel, Petar Petrovski, and Christian Bizer. 2014. The WebDataCommons Microdata, RDFa and Microformat Dataset Series. In *International Semantic Web Conference (ISWC)*. 277–292.
[45] Takunari Miyazaki. 1997. The Complexity of McKay's Canonical Labeling Algorithm. In *Groups and Computation, II*. 239–256.
[46] Reinhard Pichler, Axel Polleres, Sebastian Skritek, and Stefan Woltran. 2013. Complexity of redundancy detection on RDF graphs in the presence of rules, constraints, and queries. *Semantic Web* 4, 4 (2013), 351–393.
[47] Reinhard Pichler, Axel Polleres, Fang Wei, and Stefan Woltran. 2008. dRDF: Entailment for Domain-Restricted RDF. In *ESWC*. 200–214.
[48] Reinhard Pichler and Vadim Savenkov. 2010. Towards practical feasibility of core computation in data exchange. *Theor. Comput. Sci.* 411, 7-9 (2010), 935–957.
[49] Adolfo Piperno. 2008. Search Space Contraction in Canonical Labeling of Graphs (Preliminary Version). *CoRR* abs/0804.4881 (2008). http://arxiv.org/abs/0804.4881
[50] Vadim Savenkov. 2013. Algorithms for Core Computation in Data Exchange. In *Data Exchange, Integration, and Streams*. Dagstuhl Follow-Ups, Vol. 5. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 38–68.
[51] Craig Sayers and Alan H. Karp. 2004. Computing the digest of an RDF graph. HP Technical Report. (23 March 2004). http://www.hpl.hp.com/techreports/2003/HPL-2003-235R1.pdf.
[52] Max Schmachtenberg, Christian Bizer, and Heiko Paulheim. 2014. Adoption of the Linked Data Best Practices in Different Topical Domains. In *International Semantic Web Conference (ISWC)*. 245–260.
[53] Stephen B. Seidman. 1983. Network structure and minimum degree. *Social networks* 5 (1983), 269–287.
[54] Greg Daniel Tener. 2009. *Attacks on Difficult Instances of Graph Isomorphism: Sequential and Parallel Algorithms*. Ph.D. Dissertation. Orlando, FL, USA. Advisor(s) Narsingh Deo.
[55] Arash Termehchy, Marianne Winslett, Yodsawalai Chodpathumwan, and Austin Gibbons. 2012. Design Independent Query Interfaces. *IEEE Trans. Knowl. Data Eng.* 24, 10 (2012), 1819–1832.
[56] Giovanni Tummarello, Christian Morbidoni, Paolo Puliti, and Francesco Piazza. 2005. Signing individual fragments of an RDF graph. In *Proceedings of the 14th International Conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005 – Special interest tracks and posters*. ACM, 1020–1021.
[57] Denny Vrandecic and Markus Krötzsch. 2014. Wikidata: a free collaborative knowledgebase. *Commun. ACM* 57, 10 (2014), 78–85.

A PROOF OF THEOREM 3.3

The following proof appeared in the conference version of this work [27].

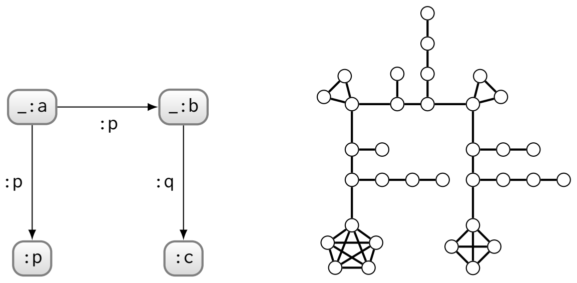
Proof. The proof relies on a polynomial-time many-one reduction both to and from the GI-complete problem of deciding graph isomorphism for undirected graphs.

The reduction from standard graph isomorphism to RDF isomorphism can be done quite straightforwardly. More specifically, let $\mathrm{G} = (V, E)$ denote an undirected graph and let $\nu : V \to \mathbf{B}$ denote an injective (one-to-one) mapping from the vertexes of G to blank nodes. For the undirected graph G, we can create a surrogate RDF graph G such that $(v, v') \in V$ if and only if $(\nu(v), p, \nu(v')) \in G$, where p is an arbitrary fixed IRI. Now given two undirected graphs G and H and their RDF versions G and H respectively, it is not difficult to show that $\mathrm{G} \cong \mathrm{H}$ (under graph isomorphism) if and only if $G \cong H$ (under RDF isomorphism). Hence RDF isomorphism is GI-hard.

The reduction from RDF isomorphism to standard graph isomorphism is trickier. 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 provide an example of one such encoding mechanism that satisfies the proof.

First, let SO denote the set of ground terms 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 will encode the RDF graph on the left as the undirected graph on the right (we keep the intuitive "shape" of both graphs intact for reference). The idea is to use 3-cliques to represent blank nodes in G, n-cliques (for $n > 3$) to represent nodes in SO, and path graphs to indicate the direction and predicate of each edge. During the encoding, we assume that each clique and path graph we create has a fixed *external node* that is used to connect it to other parts of G; for path graphs, the external node is a terminal node.



We start with an empty undirected graph G.

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 $\mathrm{sorank}(x) \coloneqq \mathrm{card}\{y \mid y \in SO \text{ and } y \leq x\}$. 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 $x \in SO$.

We now encode the RDF edges, capturing predicates and direction. Define a total ordering over P and let prank(p) denote the rank of p in P such that $\mathrm{prank}(p) \coloneqq \mathrm{card}\{q \mid q \in P \text{ and } q \leq p\}$ (e.g., with lexical ordering, prank(:p) = 1, prank(:q) = 2). For each $(s, p, o) \in G$, add two path graphs of length prank(p) + 1 and $|P| + 1$ to G and connect their external nodes. Connect the external node of the clique of s to the short path and the external node for the clique of o to the long path.

Once all triples are processed, the encoding is completed in time polynomial in the size of G.

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 finally argue why $G \cong H$ (under RDF isomorphism) if and only if $\mathrm{G} \cong \mathrm{H}$ (under graph isomorphism).

We first argue that $G \cong H$ 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 isomorphism. In particular, G can be decoded back to a G' such that $G \cong G'$. The decoding relies on two main observations:

(1) No ($\geq$ 3)-cliques can be 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 to fresh blank nodes or back 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.

The existence of this decoding shows that two non-isomorphic RDF graphs cannot be encoded to isomorphic standard graphs under fixed vocabulary: the translation preserves isomorphism.[35]

This polynomial-time encoding thus completes the proof of Theorem 3.3. □

Received June 2016; accepted March 2017

[35]Decoding is not part of the reduction. In particular, listing all maximal cliques may not be possible in polynomial time. Instead our goal is to merely demonstrate the *existence* of an unambiguous decoding to establish that isomorphism is preserved between both forms.