Technical Report on the Universal RDF Dataset Normalization Algorithm

Prepared by Mirabolic Consulting

Submitted to Digital Bazaar on 10/16/2020
Last edited on 10/19/2020

Introduction

Summary of Findings

Canonicalization and Isomorphism

Correctness up to Hashing Function

Correctness of Arnold/Longley Proof
- Hash First Degree Quads (HF) Algorithm
- Hash Related Blank Node (HR) Algorithm
- Hash N-Degree Quads (HN) Algorithm
- URDNA2015

Practical Considerations
- Engineered Hash Collisions
- Validation on Specific Graph Topologies
  - Directed Cycle Graphs
  - Undirected Cycle Graphs
  - Complete Graphs
  - Petersen Graph
  - Feedforward Graph
- Runtime Performance

Correctness by Design

References

Introduction

The goal of this technical report is to review the Universal RDF Dataset Normalization Algorithm (URDNA2015) for correctness and to provide satisfactory evidence that possible issues with URDNA2015 have been considered and dismissed. We do not lay out the algorithm in its considerable technical detail here, but refer the reader to the proposed technical specification

[Longley], a set of proofs by Rachel Arnold and Dave Longely [Arnold], and a reference implementation in Python [DigitalBazaar].

Summary of Findings

A detailed report follows, but our summary findings are as follows:

- We found no substantive errors in the proof in [Arnold] that the URDNA2015 algorithm canonicalizes an RDF dataset.
- As with any algorithm utilizing a hash function into a finite set, there is the possibility that hash collisions could cause URDNA2015 to fail in practice; this proves more of a nuisance than a substantive issue and we discuss an approach to handle it.
- The URDNA2015 algorithm, as implemented, is extremely slow in certain simple cases. The performance is sufficiently drastic that it may warrant modifications in some future updating of the specification.

Canonicalization and Isomorphism

The immediate goal of URDNA2015 is to provide a ***canonical labeling*** of an RDF dataset. By a canonical labeling, we mean the following:

- All blank nodes in the original RDF dataset will receive a label.
- If the same RDF dataset is processed by the algorithm more than one time, but perhaps fed into the algorithm with a different ordering of the graphs and/or labelling of the nodes, the labels for blank nodes that are produced by the algorithm will be the same, up to symmetries between blank nodes in the original, unlabeled dataset.

In simpler terms, URDNA2015 labels all of the unlabeled nodes of a dataset in a consistent, repeatable fashion.

In the above, when we describe symmetries between blank nodes, we are referring to graph automorphisms in which two or more blank nodes are indistinguishable from one another based on the data in the graphs, on the names of the graphs, and on their connectivity within the graphs.

We stated that a canonical labeling was the *immediate goal* of the algorithm, but in reality this is but a means to an end; the ultimate goal is to be able to take two RDF datasets as input and to decide whether or not they are identical. The canonicalization of the dataset provides a direct method to do this by providing labels for blank nodes so that all parts of the two datasets can be compared, say by representing the datasets as sets of N-quads and comparing the sets of quads. Thus, the canonicalization provides a dataset isomorphism, as defined by Arnold and Longley, between two datasets; if the canonicalized datasets are different, no such isomorphism exists.

Correctness up to Hashing Function

The description above would be completely correct if URDNA2015 did not contain multiple steps in which a hashing algorithm, in this case SHA-256, were used. However, this hashing algorithm, though well-accepted for practical cryptographic and other uses, is, like all hashing algorithms, non-injective. That is, there is a non-zero probability of hash collisions within the algorithm. This fact poses a difficulty for proving correctness, because hash collisions can produce identical canonicalizations for non-isomorphic graphs (so the proofs as stated are not technically correct). Given the super-exponential running time of the algorithm (and thus super-exponential evaluation of the hash function), it is even conceivable that hash collisions might occur regularly for ordinary-sized graphs if the algorithm were run to termination.

With this in mind, the reader will bear in mind that this report in no way warrants that URDNA2015 produces a canonicalized graph in an absolute sense; rather, we believe that it produces a canonicalization that is correct *up to hash collisions within the algorithm*. The probability of such a collision increases with the number of blank nodes within an RDF dataset; nevertheless, we feel that this vulnerability of the algorithm in no way limits its practical usefulness.

There are several approaches to handling this nuisance. One method is as follows. Rather than specifically defining the hash function *h()* as implementing SHA-256, we can instead define it as an arbitrary injective function on the space of strings, with the property that the range of the function is disjoint from the set of literals, blank nodes, and IRIs. In that case, the proofs stand as they are written. As a final corollary, Digital Bazaar can then note that, absent a hash collision, SHA-256 will behave as an injective function when restricted to the particular terms evaluated in the function.

For our own analysis of the algorithm, we treated the proof as though such an injectivity approach had been adopted.

Correctness of Arnold/Longley Proof

The design of URDNA2015 consists of a main algorithm that utilizes, in a recursive way, a number of sub-algorithms. Each of these pieces is an opportunity for a problem to occur in the proof; mainly, the question is whether the sub-algorithm is itself injective (up to hash). We consider each piece in turn in this section.

Hash First Degree Quads (HF) Algorithm

The HF algorithm provides a hash associated with a blank node in an RDF dataset using only the first degree information associated with that node, i.e. the first degree hash of a blank node n. In practice, this information will almost always suffice to uniquely identify a blank node within a dataset. We note, however, that URDNA2015 makes no guarantee that the HF algorithm is

an injective mapping on blank nodes; in fact, the rest of the algorithm is entirely dedicated to handling the case in which two blank nodes receive an identical first degree hash from HF.

Hash Related Blank Node (HR) Algorithm

The HR algorithm encodes a mention of a blank node, from the standpoint of another given blank node; alternatively, it could be said that HR encodes a quad in which two blank nodes are referenced. This algorithm is careful to use both distinct identifiers for the second blank node and to distinguish the position of the second node as related to the first node in the quad. Thus, different mentions should receive unique results from this algorithm. The one exception to this is when two blank nodes have the same first degree hash as well as the same relationship to the input node n. For example, assume that we have three blank nodes n, n_1, and n_2. If $h_f(n_1) = h_f(n_2)$ and two quads in a dataset are given by $<n,p,n_1,g>$ and $<n,p,n_2,g>$, those two quads would have identical output from the HR algorithm, from the perspective of node n. As with the HF algorithm, this illustrates the need to encode more global information about a blank node n and its mentions than either the HF or HR algorithm provides.

Hash N-Degree Quads (HN) Algorithm

The HN algorithm yields the N-degree hash of a blank node n, $h_N(n)$. This encodes all of the information reachable to n within the RDF dataset via mentions of other blank nodes, up to and including (but not proceeding past) blank nodes that have already been canonically labeled. We believe the HN algorithm, unlike the HF and HR algorithms discussed above, does, in fact, yield a distinct hash for any blank node within an RDF dataset *which is distinguishable via either data in the dataset or the topology of the RDF dataset*. (Specifically, if two nodes are equivalent under a graph automorphism, they will have the same value under the HN algorithm.) This is due to

- The recursive nature of the algorithm that explores mentions through blank nodes until these are exhausted
- The care taken in creating the data to hash D_n, which in turn uses the HR algorithm
- The fact that the shortest (gossip) path is chosen through the blank nodes, ensuring a canonical ordering of the data to hash

We note that, at first glance, it seems surprising that the HN algorithm does not include data on *non-blank nodes* reachable from a blank node n via mentions in the RDF dataset. After all, this is data that could help to uniquely identify n! However, because URDNA2015 first performs HF on all blank nodes and then *recursively canonicalizes* blank nodes distinguishable via first degree information, this information has effectively been exhausted by the time the HN algorithm comes into play.

URDNA2015

The full RDF dataset normalization algorithm rests on top of HF, HR, and HN, which it calls as subroutines. Without reiterating the algorithm in full, we can summarize the algorithm as follows:

- First, give out canonical identifiers, in a prescribed order, to all blank nodes that have unique first degree hashes. This step will presumably label most blank nodes in most circumstances.
- Second, proceeding methodically through groups of blank nodes having identical first degree hashes, use the N-degree hash of each node in a group to give out canonical identifiers, again in a prescribed order.

Arnold and Longley then address the remaining case of nodes having identical first and N-degree hashes. In this case they seek to prove that the order in which such nodes are given canonical identifiers does not matter.

After careful consideration we believe their proof to be correct; moreover, we put considerable time into creating potential counterexamples and could not construct one. Our understanding of this concept can be summarized by saying that groups of nodes with identical first and N-degree hashes will share a symmetry with respect to the rest of the RDF dataset; it is this symmetry that renders the order of the canonical labeling of these nodes ultimately of no consequence.

Practical Considerations

The previous sections were concerned with the correctness of the proof, up to hash collisions. In the next sections we address potential concerns with using the algorithm in practice.

Engineered Hash Collisions

The URDNA2015 algorithm invokes hash functions at multiple stages of the process. This structure exposes the algorithm to several potential vulnerabilities. On balance, these possibilities do not seem to pose undue risk to the proposal, but we discuss them here for completeness. When we discuss vulnerabilities and attacks against these vulnerabilities, we are not just concerned with adversarial attacks in the cryptographic sense; we are also concerned with accidental hash collisions.

First we consider two brute force attacks for generating hash collisions. Note that after canonicalizing the blank nodes and producing a canonical representation of the quads, a final application of the SHA-256 hash algorithm produces a single hash for the RDF dataset that we would hope would be unique. To that end, consider a single-line quad of the form:

```
_:a <urn:a:b> “X” _:a
```

We now proceed with some numerology to understand the exposure to hash collisions. If X is a 43 byte (344 bit) UTF-8 literal, then the quad is 64 bytes (including the newline).[1] To be a valid literal, X must be a UTF-8 string. A string of B bits can represent approximately $2^{0.9B}$ UTF-8 strings, so X assumes approximately 2^{310} possibilities. Since we are using a 256-bit hash function and we have more than 2^{256} possibilities, the pigeonhole principle guarantees hash collisions of this form exist, i.e. we can find $X_1 \neq X_2$ with the same SHA-256 hash. However, actually finding such a pair of bit strings appears prohibitive given current technology. Specifically, no SHA-256 hash collisions are currently known, and the brute force approach to generating a hash collision would require about 2^{128} bits of work.

Second, we may be concerned that someone might discover several SHA-256 hash collisions, and that these collisions could be escalated into an attack. For example, this might occur if the pair of values in a hash collision both corresponded to the prefix of a valid N-quad string. For example, what is the probability that a set of random bits will parse as an IRI scheme? Consider quads of the form:

<AX://foo.com> _:a <ftp://a> _:a _:a

where "A" is one of the 52 alphabetic ASCII/UTF-8 characters, and X is an arbitrary 62-byte (496 bit) UTF-8 string. There are approximately $2^{5.7} \times 2^{0.9(496)} = 2^{452}$ possibilities. Therefore, if a hash collision were generated at random, the odds of it parsing correctly are approximately $2^{452-512}=2^{-60}$. Given a random pair of strings producing a SHA-256 hash collision, the odds of both parsing in this form are approximately 2^{-120}, which is comparable to the naive work for producing a SHA-256 hash collision itself (namely, 2^{-128}). Analogously, random strings are also unlikely to parse as a prefix of a blank node.

Third, we may concern ourselves with the possibility of two carefully constructed strings causing a collision within the algorithm itself. To provide a concrete example, consider the Hash Related Blank Node algorithm. Might it be possible to choose two distinct sets of input values that would produce the same output? (This type of collision would be in the spirit of a code injection attack.) Fortunately, this particular attack appears impossible. The URDNA2015 algorithm uses strings corresponding to 4 different types of objects: IRIs, literals, blank nodes and hashes. Each set is distinct and distinguishable, forming a prefix-free code. Because each string can be uniquely decomposed into its constituent elements, ambiguous or adversarial string injection is avoided.

Validation on Specific Graph Topologies

As an additional check on correctness, we examined several graph topologies[2]. We considered quads in which

- All the predicates were identical
- All the subject and object nodes were unlabelled
- All quads used the default graph

---

[1] Note that SHA-256 operates on 512 bit / 64 byte buffers.

[2] We apologize that the word "graph" is somewhat overloaded; here, we mean "graph" in the sense of "graph theory".

This perspective allowed us to view the subject and object nodes as a set of graph vertices and each quad as a directed edge.

We can optionally restrict to undirected edges as follows:

- Each <s,p,o,g> has a paired <o,p,s,g>. (This step effectively makes the edges undirected.)

This reduction allows us to express any classic (directed or undirected) graph isomorphism problem as a graph isomorphism problem on quads.

In each case, we generated a single hash, then permuted and rehashed the graph 100 times to confirm that the hashes matched. In all cases they did: no counterexamples were found.

We used the reference Python implementation available at [DigitalBazaar].

Directed Cycle Graphs

We considered a directed cycle graph on N nodes, for N=3,...,10. Note that all nodes are isomorphic.

Undirected Cycle Graphs

We considered an undirected graph on N nodes with a single cycle, for N=3,...,10. Note that all nodes are isomorphic.

Complete Graphs

We considered the complete (undirected) graph on N nodes, for N=3, 4, 5, 6. Note that all nodes are isomorphic.

Petersen Graph

We considered the Petersen graph on 10 nodes. (The Petersen graph is a notorious counterexample for many graph theoretical problems.)

Feedforward Graph

We constructed a feedforward graph with 5 layers, two nodes per layer, and the nodes in layer (i) connected to the nodes in layer (i+1) in a directed fashion. If we consider the three internal layers, all the nodes share the same first degree hash, and the nodes within each layer are isomorphic, but the nodes on different layers are non-isomorphic.

Runtime Performance

On real-world RDF datasets, we expect the URDNA2015 algorithm to compute a set of canonical node labels extremely quickly. The URDNA2015 algorithm begins by constructing labels with the HF sub-algorithm; frequently, all nodes will have a unique label, in which case

HF completes the canonical labelling. The much slower HN sub-algorithm is only invoked when multiple HF labels match exactly, so frequently this step can be avoided completely.

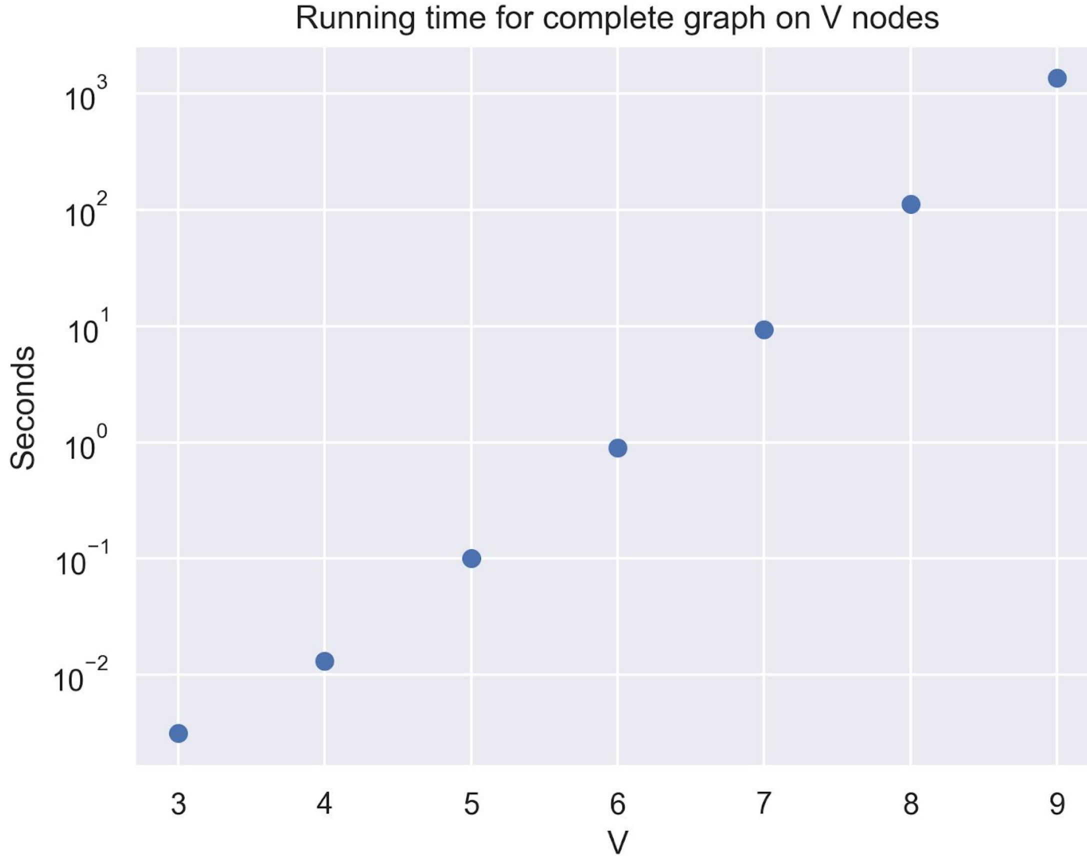
For practical purposes, therefore, the URDNA2015 algorithm appears sufficiently performant for applications on real-world data. In this section, we examine the worst case performance because of its theoretical and adversarial implications.

We begin by examining the theoretical complexity. As mentioned above, in a typical case, the HF sub-algorithm may completely label an RDF dataset, so we begin by examining this sub-algorithm. Suppose our RDF dataset contains a list of E quads and V blank nodes. If we apply the HF sub-algorithm to each node, we will examine each quad at most 3 times. Because each non-terminating iteration through all nodes isolates at least one blank node, we iterate over all nodes at most V times. Therefore the complexity is O(EV), which is polynomial in our parameters.

The other dominant step is the HN sub-algorithm. From a complexity perspective, the theoretical performance of URDNA2015 is dominated by step 5.4.4 of the Hash N-Degree Quads algorithm, in which the algorithm considers all permutations [x]. This list is potentially as large as the set of blank nodes. Therefore, the complexity of this single step is $O(2^{V\log(V)})$, which is super-exponential.

However, we may be less concerned with the theoretical performance of the algorithm than its behavior in practice. To that end, we examined the running time of URDNA2015 on the “complete graphs” as described in the previous subsection. These graphs are meaningless from a practical perspective, but are still well-formed RDF datasets.

Note that the number of quads is (V choose 2), or approximately $V^2/2$. Running time is displayed in the figure below; please note that the running time axis (the y-axis) is displayed on a logarithmic scale.



In particular, a 9-node complete graph, using the reference Python implementation, running on an Intel core i9, canonicalized the graph in about 22 minutes. This seems remarkably slow for such a tiny graph.

This performance suggests that ideally one would consider using an algorithm that exploits graph automorphisms intelligently, such as Traces, as a starting point. Such algorithms typically have sub-second running times on graphs with thousands of edges, per [McKay]. Typically, it can be challenging even finding graphs that slow down modern algorithms [Neuen].

We should be concerned about performance for at least three reasons. First, if the graph canonicalization process induces a large computational load, it may deter broader acceptance. (Again, we expect this not to be an issue in practice.) Second, from a systems perspective, it is not clear what the intended behavior should be if URDNA2015 does not terminate in practice. Finally and most concerningly, this state of affairs enables an adversary to take an otherwise innocuous RDF dataset, inject a small, unlabelled complete graph disconnected from the rest of the RDF dataset, and render it effectively impossible to canonicalize the labels. The specification does not seem to describe a maximum timeout period, so an insufficiently paranoid implementation of the specification would expose a user to the risk of this attack. Note that [DigitalBazaar] does not have a default timeout option.

Correctness by Design

The reality of algorithmic design against the backdrop of approval by a standards body is complex at best, and the design acquires historical and organic baggage in its development. That said, we would like to share a few thoughts toward streamlining future endeavors.

Ideally, when designing a complex algorithm that requires a proof of correctness, one begins by identifying the elements necessary to prove correctness and then ensuring that the algorithm enforces these rules. The goal is to facilitate external review and confidence in correctness.

The requirements of a specification, such as the RDF datasets definition are very specific. However, when designing an algorithm, we are at liberty to design a more general algorithm so long as it can handle the particular specification as a special case.

In our case, the subject, object and graph name fields in the tuple are handled separately. This may be in part because of the history of the RDF dataset and the migration from triples to quads; it may be in part because there are some minor differences (e.g., objects can be literals but subjects cannot). For example, the “Hash Related Blank Node (HR) Algorithm” appends the predicate p when the location is the subject or object, but does not append it when it is a graph name. This makes the analysis substantially more complicated because the correctness depends on many more conditional interactions.

Instead, suppose we consider a slightly more general problem. Instead of canonicalizing only RDF datasets, let us generalize to the set of quads in which the subject, object and graph name fields can *all* contain IRIs, literals or blank nodes. In the generalized problem, we allow the algorithm to handle all the fields uniformly, so it is easier to ascertain correctness (and, incidentally, easier to implement in code). When we wish to apply our algorithm to the particular specification, we simply restrict to proper RDF datasets.

We can gain additional benefits from further generalization. Because RDF Datasets are defined in terms of quads, there is a conflation between the label of a node (e.g., an IRI) and the node itself. For example, in the case of a subject, object or graph name, a label (IRI or literal) is guaranteed to be unique to a node. On the other hand, in the case of a predicate, the IRI is not unique to the corresponding hyperedge but is only a label. This situation makes the relationship between the quad syntax and the underlying graphical structure fairly opaque. Instead, it would be much clearer to assign an index to each node (possibly just the integers 1,...,N) along with a label. Furthermore, we can generalize our problem and allow multiple nodes to have the same label.

This generalized problem is much more in line with the standard approaches taken by the software packages and corresponding algorithms of Nauty, Traces, and similar tools [McKay].

In those approaches, all nodes start with the same label, and the labels are progressively refined until each non-automorphic node has a unique label. Therefore, this generalized problem allows us to leverage the work of the graph isomorphism community much more directly and to adhere to the more common terminology used in the field.

If we wish to canonicalize a specific RDF dataset, we would label all blank nodes with a single label and label all other nodes with their corresponding IRI or literal (which would happen to be unique). This approach suppresses the distinction between blank and non-blank nodes, treating all nodes uniformly. We would hope such a more general approach might sidestep some of the analytical complexity around canonical, blank, temporary and other node types that crops up when focussing on the details of the specification.

We add a final observation about correctness by design. Many algorithms (such as URDNA2015) can be factored into a set of modular sub-algorithms. By clearly defining the algorithm's internal state, it is possible to view each algorithm as a well defined function, mapping the initial state to the final state. The URDNA2015 algorithm, as with many graph isomorphism algorithms, is designed to act as an equivariant map on the serialized graph. In other words, if we permute the original blank labels, they cause a particular corresponding action on the internal state. If the algorithm is structured so that it is straightforward to establish equivariance for each sub-algorithm, then verifying correctness of the entire proof becomes much more straightforward.

References

Arnold, Rachel and Longley, Dave. *RDF Dataset Canonicalization*. October 9, 2020.

Digital Bazaar. *PyLD*. GitHub Repository, retrieved October 2, 2020 at git hash 316fbc2c9e25b3cf718b4ee189012a64b91f17e7.

Longley, Dave, and Manu Sporny. *RDF dataset normalization: A Standard RDF Dataset Normalization Algorithm, Draft Community Group Report 27 February 2019*. Technical Report. World Wide Web Consortium. Retrieved from http://json-ld.github.io/normalization/spec, October 16, 2016.

McKay, Brendan D., and Adolfo Piperno. "Practical graph isomorphism, II." *Journal of Symbolic Computation* 60 (2014): 94-112. arXiv: https://arxiv.org/pdf/1301.1493.pdf

Neuen, Daniel, and Pascal Schweitzer. "Benchmark graphs for practical graph isomorphism." *arXiv preprint arXiv:1705.03686* (2017). arXiv: https://arxiv.org/pdf/1705.03686.pdf