1 Introduction

Given a graph, one can determine whether two nodes are equivalent, that is have the same relationships to all others nodes in the graph, by comparing their identifiers. But what if the nodes do not have identifiers? That is, what if the nodes are blank? In this case, one must explore the connections associated with the blank nodes throughout the entire graph to determine whether the nodes are equivalent. This is called the graph isomorphism problem.

This paper proves the correctness of the Universal RDF Dataset Normalization Algorithm 2015 (URDNA2015) [3], an algorithm that has been in use for several years but has not been formally proved until this paper. URDNA2015 explores the connections of blank nodes throughout the graph and uniquely determines a method of assigning canonical identifiers to each blank node. With the blank nodes canonically labeled, one can then compare these labels to determine whether
two nodes are equivalent.

More specifically, URDNA2015 canonically labels an RDF dataset—or collection of RDF graphs. An RDF graph is a collection of triples <s,p,o> that can be depicted via a directed edge from the subject s to the object o (Figure 1). The predicate p indicates the type of edge from s to o.

![Figure 1](image_url)

In [2], an RDF dataset $\mathcal{D}$ is defined as a collection of RDF graphs that comprises:

- Exactly one default graph, being an RDF graph. The default graph does not have a name and may be empty.
- Zero or more named graphs. Each named graph is a pair consisting of an IRI or a blank node (the graph name), and an RDF graph. Graph names are unique within an RDF dataset.

In URDNA2015, an RDF dataset $\mathcal{D}$ is represented as a set of quads of the form <s,p,o,g> where the graph component g is empty if and only if the triple <s,p,o> is in the default graph. Quads in the default graph are denoted <s,p,o,−>, where “−” is used to indicate an empty graph component. This paper will also consider an RDF dataset to be a set of quads. We will ultimately demonstrate that two RDF datasets are the same modulo blank nodes, or isomorphic, if and only if they return the same canonically labeled list of quads via URDNA2015.

URDNA2015 consists of several sub-algorithms. These sub-algorithms are introduced throughout the exposition that follows, translated into the notation and language of this paper. For reference, a notation index is provided in Appendix A. Below we give a very high level summary of URDNA2015; the algorithm is presented in its entirety in Section 3. The specification can be viewed here.

**URDNA2015 (High Level)**

1. **Initialization.** Initialize the state needed for the rest of the algorithm.

2. **Compute first degree hashes.** Compute the first degree hash for each blank node in the dataset using the Hash First Degree Quads (HF) Algorithm.

3. **Canonically label unique nodes.** Assign canonical identifiers via the Issue Identifier algorithm, in lexicographical order, to each blank node whose first degree hash is unique.

4. **Compute N-degree hashes for non-unique nodes.** For each repeated first degree hash (proceeding in lexicographical order), compute the N-degree hash via the Hash N-Degree Quads (HN) algorithm of every unlabeled blank node that corresponds to the given repeated hash.

5. **Canonically label remaining nodes.** In lexicographical order of the N-degree hashes, issue canonical identifiers to each corresponding blank node using the Issue Identifier algorithm. Later, we show that if more than one node produces the same N-degree hash, the order in which these nodes receive a canonical identifier does not matter.
Throughout the algorithm, blank nodes will be issued identifiers via the **Issue Identifiers algorithm**. Some blank nodes may be issued temporary identifiers by this algorithm prior to being assigned a canonical identifier.

### The Issue Identifier Algorithm

This algorithm issues a new blank node identifier for a given existing blank node identifier. It also updates state information that tracks the order in which new blank node identifiers were issued.

This algorithm takes an identifier `issuer`, denoted `I`, and an existing blank node identifier `n` as inputs. The output is a new issued identifier `I(n)`.

1. If there is already an issued identifier for `n` in the issued identifiers list for `I`, return it.
2. Generate `I(n)` by concatenating `identifier prefix` with the string value of `identifier counter`.
3. Append an item to the issued identifiers list for `I` that maps `n` to `I(n)`.
4. Increment `identifier counter`.
5. Return `I(n)`.

In this paper, we denote the issued label `I(n)` by `c_n` if it is a canonical identifier or by `b_n` if it is a temporary identifier.

### 1.1 “Mentions” as the Edges of an RDF Dataset

Because a graph name in a dataset may be blank, the appearance of a blank identifier in the graph component of a quad must be characterized when determining a canonical labeling. Given the quad `<s,p,o,g>`, we cannot use the term “edge” to describe the relationship between `s` and `g` or the relationship between `o` and `g` (as we can when relating `s` and `o`). To include this new relationship type, we use the term **mention** instead of edge. In this way, mentions can be physical edges in a graph or they can simply indicate that two nodes appear in a quad together when one is in the graph position.

Each quad in a dataset `D` describes a set of mentions between the nodes in its components. We say that a quad **mentions a node** `n` if `n` is an entry in one of its components. The set of all quads in `D` that mention a node `n` is called its **mention set**, denoted `Q_n`.

Two nodes `n` and `n'` are **related** if they appear together in the same quad; that is, `Q_n \cap Q_{n'} \neq \emptyset`. Related nodes necessarily have a mention “between them.” Mentions are “directed” in that they are described from each incident node’s perspective. A mention `m` from `n_1` to `n_2` described by the quad `q` indicates the component of `q` in which `n_2` is mentioned. For example, given `q = <n_1,p,n_2,n_3>`,
we could use the mention $m_1$ to indicate “$n_1$ mentions $n_2$ in the object component of $q$” and the mention $m_2$ to indicate “$n_2$ mentions $n_1$ in the subject component of $q$.”

2 Encoding Information Connected to a Blank Node

To determine a canonical labeling, the URDNA2015 considers the information connected to each blank node. Nodes with unique first degree information can be issued a canonical identifier immediately via the Issue Identifier algorithm. When a node has non-unique first degree information, it is necessary to determine all information that is connected to it transitively throughout the entire dataset. Section 2.1 defines a node’s first degree information via its first degree hash.

Hashes are computed from the information of each blank node. In particular, these hashes encode the mentions incident to each blank node. The hash of a string $s$, denoted $h(s)$, is the lowercase, hexadecimal representation of the result of passing $s$ through a cryptographic hash function. URDNA2015 uses the hash algorithm SHA-256.

When performing the steps required by URDNA2015, it is helpful to track the state in a data structure. This is called the normalization state and it consists of three parts.

1. blank node to quads map - A data structure that maps a blank node identifier $n$ to its mention set $Q_n$.

2. hash to blank nodes map - A data structure that maps a hash to a list of blank node identifiers. In Section 2.2.1 we denote this list by $[x]$ where $x$ is a related hash.

3. canonical issuer - An identifier issuer, initialized with the prefix `_:c14n`, for issuing canonical blank node identifiers.

When calling sub-algorithms, the normalization state is often passed.

2.1 First Degree Hashes

To determine whether the first degree information of a node $n$ is unique, a hash is assigned to its mention set, $Q_n$. Specifically, the first degree hash of a blank node $n$, denoted $h_f(n)$, is the hash that results from the Hash First Degree Quads (HF) algorithm when passing $n$. Nodes with unique first degree hashes have unique first degree information.

Computing $h_f(n)$ requires the serialization in N-quads format [1] of all quads in the mention set of $n$, $Q_n$. Prior to serializing a quad $q \in Q_n$, the algorithm makes a replacement for each blank component in $q$. Each component containing $n$ is replaced with $a$ and all other blank components different from $n$ are replaced with $z$ (See step 3.1.1.1 of HF). By first replacing each blank node with $z$ or $a$, HF distinguishes between blank and nonblank components in the quads of $Q_n$. See Example 2.1 for an illustration of this rule.

Each quad in the replacement list is serialized. Then, the serialized list is lexicographically sorted, concatenated, and hashed. This hash is the first degree hash of $n$, $h_f(n)$. The set of all first degree hashes, denoted $H_F$ is called the first degree hash list of $\mathcal{D}$. 
Example 2.1. When executing HF on the blank node $n$, HF step 3.1.1.1 makes a special replacement for each quad in $Q_n$ that mentions $n$. Table 1 illustrates a sample mention set $Q_n$ and its replacement under this rule. Note that $n_1$ and $n_2$ are blank nodes, whereas $s$, $o$, and $g$ are nonblank identifiers.

<table>
<thead>
<tr>
<th>$Q_n$</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$&lt; n, p, n_1, g&gt;$</td>
<td>$&lt; a, p, z, g&gt;$</td>
</tr>
<tr>
<td>$&lt; n, p, n_1, n_2&gt;$</td>
<td>$&lt; a, p, z, z&gt;$</td>
</tr>
<tr>
<td>$&lt; n_2, p, n, n&gt;$</td>
<td>$&lt; z, p, a, a&gt;$</td>
</tr>
<tr>
<td>$&lt; s, p, n, g&gt;$</td>
<td>$&lt; s, p, a, g&gt;$</td>
</tr>
<tr>
<td>$&lt; n_1, p, o, n&gt;$</td>
<td>$&lt; z, p, o, a&gt;$</td>
</tr>
<tr>
<td>$&lt; n_1, p, n, n_1&gt;$</td>
<td>$&lt; z, p, a, z&gt;$</td>
</tr>
</tbody>
</table>

Table 1: $n_1$ and $n_2$ are blank nodes distinct from $n$. $s$, $o$, and $g$ are nonblank identifiers.

The Hash First Degree Quads (HF) Algorithm

This algorithm takes the normalization state and a reference blank node identifier $n$ as inputs.

1. Initialize $\text{nquads}$ to an empty list. It will be used to store quads in N-Quads format.

2. Get the list of quads $Q_n$ associated with the reference blank node identifier $n$ in the blank node to quads map.

3. For each quad $q$ in $Q_n$:
   
   3.1. Serialize $q$ in N-Quads format with the following special rule:
      
      3.1.1. If any component in $q$ is a blank node, then serialize it using a special identifier as follows.
      
      3.1.1.1. If the existing blank node’s identifier is $n$ then use the blank node identifier $a$; otherwise, use the blank node identifier $z$.

4. Sort $\text{nquads}$ in lexicographical order and denote the sorted list by $H_F$.

5. Return the hash $h_f(n)$ that results from passing the sorted, joined list $H_F$ through the hash algorithm $h$.

2.2 $N$-Degree Hashes

When two blank nodes have the same first degree hash, extra steps must be taken to detect global, or $N$-degree, distinctions. All information that is in any way connected to the blank node $n$ through other blank nodes, even transitively, must be considered.

To consider all transitive information, the algorithm traverses and encodes all possible paths of incident mentions emanating from $n$, called gossip paths, that reach every unlabeled blank node connected to $n$. Each unlabeled blank node is assigned a temporary label in the order in which it is reached in the gossip path being explored. The mentions that are traversed to reach connected
2.2 N-Degree Hashes

Blank nodes are encoded in these paths via related hashes. This provides a deterministic way to order all paths coming from \( n \) that reach all blank nodes connected to \( n \) without relying on input blank node labels.

Ultimately, the algorithm selects a shortest gossip path, distributing canonical labels to the unlabeled blank nodes in the order in which they appear in this path. The hash of this encoded shortest path, called the N-degree hash of \( n \), distinguishes \( n \) from other blank nodes in the dataset.

For clarity, we consider a gossip path encoded via the string \( s \) to be shortest\(^1\) provided that

1. The length of \( s \) is less than or equal to the length of any other gossip path string \( s' \).
2. If \( s \) and \( s' \) have the same length (as strings), then \( s \) is lexicographically less than or equal to \( s' \).

For example, \( abc \) is shorter than \( bbc \), whereas \( abcd \) is longer than \( bcd \).

2.2.1 Related Hashes

This section explains how a related hash is assigned to a mention between two blank nodes. Suppose that a blank node \( n \) is related to the blank node \( n_i \neq n \) via the quad \( q \). \( h_r(q, n, n_i, \text{position}) \) is the related hash (with respect to \( n \)) that results from the Hash Related Node (HR) Algorithm when passing the quad \( q \), identifier node \( n \), and the related blank node \( n_i \) where \( \text{position} \) indicates the position of \( n_i \) in \( q \) for the mention that is being encoded. For example, consider the quad \( q = < n, p, n_i, n_i > \). \( h_r(q, n, n_i, o) \) is the related hash that encodes that \( n \) mentions \( n_i \) in the object position of \( q \), whereas \( h_r(q, n, n_i, g) \) is the related hash that encodes that \( n \) mentions \( n_i \) in the graph position of \( q \). Note: it is possible that a single quad \( q \) produces two related hashes if \( n_i \) appears in two components of \( q \).)

If the related node \( n_i \) has already been issued a label, that label is used to compute the related hash. Otherwise, the first degree hash of \( n_i \) is used. In this way, a related hash is a value that encodes a mention. The lexicographically sorted set of all related hashes, or related hash set, for a node \( n \) is denoted \( H_n \). Below, we give an example of related hashes for quads in a mention set \( Q_n \).

Given a related hash \( x \in H_n \), it is possible that \( x \) is repeated (e.g. when two quads mention two blank nodes in the same way). The related hash to blank node list\(^2\), denoted \([x] \), is the set of all blank nodes, including repetitions, that produce the related hash \( x \). For example, if two different quads containing \( n \) and \( n_1 \) produce the related hash \( x \), then \( n_1 \) will be listed twice in \([x] \).

---

\(^1\)The spec in [3] stipulates the condition for skipping to the next permutation due to path length (steps 5.4.4.3 and 5.4.5.5) as \( \text{path} \) is greater than or equal to the length of chosen path and \( \text{path} \) is lexicographically greater than chosen path. Considering path length first before considering lexicographical order is an optimization for selecting a shortest path that we make in this paper’s definition of shortest path. The results of this paper still hold when using the original condition of the spec.

\(^2\)[\([x]\)] is the output of the related hash to blank node list mapping on \( x \) that determines the shortest chosen path string in step 5.4 of the Hash N-Degree Quads algorithm.
2.2 N-Degree Hashes

Hash Related Blank Node (HR) Algorithm

This algorithm creates a hash to identify how a blank node \( n \) is related to another \( n_i \) in a quad \( q \in Q_n \). It takes the normalization state, a related blank node \( n_i \), a quad \( q \) in \( Q_n \), an identifier issuer \( I \), and a string \textit{position} as inputs.

1. Set the identifier \( I(n_i) \) to use for the related node \( n_i \), preferring first the canonical identifier \( c_i \) if issued, second the temporary identifier \( b_i \) if issued, and last, if necessary, the result of the HF algorithm \( h_f(n_i) \).

2. Initialize a string \textit{input} to the value of \textit{position} (s, o, or g).

3. If \textit{position} is not g, append \(<p>\) to \textit{input}.

4. Append \( I(n_i) \) to \textit{input}.

5. Return the related hash \( h_r(q,n,n_i,\text{position}) \) that results from passing \textit{input} through the hash algorithm \( h \).

Example 2.2. Suppose that \( q = <n,p,n_1,n_2> \). We will consider the case where \( n_1 \) has been issued a label, namely \( b_1 \), but \( n_2 \) has not. From \( n \)'s perspective, \( q \) contains two mentions:

1. \( m_1 \): \( n \) mentions \( n_1 \) in the object position with predicate \( p \).

2. \( m_2 \): \( n \) mentions \( n_2 \) in the graph position.

Then, \( h_r(q,n,n_1,o) = h("o < p > b_1") \) is the related hash assigned to \( m_1 \), and \( h_r(q,n,n_2,g) = h("gh_f(n_2)") \) is the related hash assigned to \( m_2 \). Table 2 shows all related hashes that would result for each quad in an example mention set \( Q_n \) when assuming \( n_1 \) has label \( b_1 \) and \( n_2 \) has no label yet.

<table>
<thead>
<tr>
<th>( Q_n )</th>
<th>(&lt;n,p,n_1,n_2&gt;)</th>
<th>(&lt;n_2,p,n,n&gt;)</th>
<th>(&lt;s,p,n,g&gt;)</th>
<th>(&lt;n_1,p,o,n&gt;)</th>
<th>(&lt;n_1,p,n,n_1&gt;)</th>
</tr>
</thead>
<tbody>
<tr>
<td>( H_n )</td>
<td>( h(&quot;o &lt; p &gt; b_1&quot;) ) ( h(gh_f(n_2)) )</td>
<td>( h(&quot;s &lt; p &gt; h_f(n_2)&quot;) )</td>
<td>–</td>
<td>( h(&quot;s &lt; p &gt; b_1&quot;) ) ( h(gh_1) )</td>
<td></td>
</tr>
</tbody>
</table>

Table 2: The related hashes in \( H_n \) for mention list \( Q_n \). \( n_1 \) has issued label \( b_1 \) and \( n_2 \) has not been labeled yet. \( s, o, \) and \( g \) are nonblank.

2.2.2 Gossip Paths

When two blank nodes have the same first degree hash, it is necessary to describe their information from a degree greater than one. To do so, URDNA2015 explores paths of incident mentions. The generalization of edges to mentions necessitates an analogous generalization of paths. A gossip path \( p_{nn'} \) from a blank node \( n \) to a blank node \( n' \) in the dataset \( D \) is an alternating sequence of blank nodes and mentions \((n_0,m_1,n_1,m_2,n_2,...,m_k,n_k)\) such that

1. \( n_0 = n \) and \( n_k = n' \);
2. For each \( i \), \( 0 \leq i \leq k - 1 \), \( n_i \) is related to \( n_{i+1} \) via \( m_{i+1} \). Note that this implies that \( Q_{n_i} \cap Q_{n_{i+1}} \neq \emptyset \) for each \( i \); and

3. \( n_i \neq n_j \) if and only if \( i \neq j \).

**Remark 2.3.** Notice that two blank nodes need only appear in a quad together for there to be a gossip path between them. It is not necessary that the path proceeds in the direction from subject to object. Mentions may be traversed from object to subject, for example.

When multiple nodes have the same first degree hash, the Hash \( N \)-Degree Quads (HN) Algorithm is executed on each one. Given an unlabeled blank node \( n \), HN explores all gossip paths from \( n \) to any other blank node \( n_i \) that is reachable by paths through only blank nodes that have not yet been issued a canonical identifier. The set of all such paths is called the **gossip class** \([P_n]\) of \( n \). Although a gossip class is a set of paths, we will say “a blank node \( n_i \) is in \([P_n]\)” when \( n_i \) appears in a path in \([P_n]\). Note that it is possible for \([P_n]\) to contain paths that terminate at canonically identified nodes.

**Example 2.4.** Figure 2 shows a graph with unlabeled blank nodes \( n, n_1, n_2, n_3, \) and \( n_4 \). There are also two canonically labeled blank nodes \( c_1 \) and \( c_2 \). The gossip class \([P_n]\) of \( n \) is

\[
[P_n] = \{(n, m_1, c_1), (n, m_2, n_1), (n, m_2, n_1, m_4, c_2), (n, m_2, n_1, m_5, n_2), (n, m_3, n_3)\}.
\]

Note that \( n_4 \) is not reachable from \( n \) via only unlabeled nodes. Therefore, the gossip class of \( n \) does not contain any paths from \( n \) to \( n_4 \). We do, however, include paths to \( c_1 \) and \( c_2 \) since they are blank nodes that can be reached from the unlabeled blank nodes \( n \) and \( n_1 \), respectively.

2.2.3 Computing \( N \)-Degree Hashes at a High Level

The **\( N \)-degree hash** of a blank node \( n \), denoted by \( h_N(n) \), is the hash that results from executing the Hash \( N \)-Degree Quads (HN) algorithm when passing the identifier node \( n \) and its temporary issuer \( I_n \). The data to hash \( D_n \) corresponds to the shortest gossip path for distributing labels to the unlabeled blank nodes connected to \( n \). The hash of \( D_n \) is precisely the \( N \)-degree hash of \( n \). That is, \( h_N(n) = h(D_n) \). In Section 2.2.4, we describe the terms of \( D_n \), which encode the nodes and mentions that comprise gossip paths in \([P_n]\).

Below, we first give a high level summary of the HN algorithm. Example 2.5 illustrates this high level summary and the data to hash that results from HN. In Section 2.2.5, we present the full algorithm and explain in further detail how gossip paths are encoded.
The (High Level) Hash $N$-Degree Quads (HN) Algorithm

The following provides a high level outline for how the $N$-degree hash of $n$ is computed along the shortest gossip path. Note that the full algorithm considers all gossip paths, ultimately returning the hash of the shortest encoded path.

1. **Compute related hashes.** Compute the related hash $H_n$ set for $n$, i.e. all first degree mentions between $n$ and another blank node. Note that this includes both labeled and unlabeled blank nodes.

2. **Explore mentions.** Given the related hash $x$ in $H_n$, record $x$ in the data to hash $D_n$. Determine whether each blank node reachable via the mention with related hash $x$ has already received a label.
   
   (a) **Record the labels of labeled nodes.** If a blank node already has a label, record its label in $D_n$ once for every mention with related hash $x$. Skip to the next related hash in $H_n$ and repeat step 2.
   
   (b) **Distribute and record temporary labels to unlabeled nodes.** For each unlabeled blank node, assign it a temporary label according to the order in which it is reached in the gossip path, recording its given label in $D_n$ (including repetitions). Add each unlabeled node to the recursion list $R_n(x)$ in this same order (omitting repetitions).
   
   (c) **Recurse on newly labeled nodes.** For each $n_i$ in $R_n(x)$
      
      i. Record its label in $D_n$
      
      ii. Append $<r(i)>$ to $D_n$ where $r(i)$ is the data to hash that results from returning to step 1, replacing $n$ with $n_i$.

3. **Compute the $N$-degree hash of $n$.** Hash $D_n$ to return the $N$-degree hash of $n$, namely $h_N(n)$. Return the updated issuer $I_n$ that has now distributed temporary labels to all unlabeled blank nodes connected to $n$.

We call the pair $(h_N(n), I_n)$ returned from the algorithm the **result of HN**, where $I_n$ is the final issuer state at the conclusion of HN (see step 3 above). At this time, $I_n$ has issued temporary labels $b_i$ for each unlabeled node that is visited in computing $h_N(n)$. That is, for each $n_i \in [P_n]$. These temporary labeled nodes are denoted $n_0, n_1, \ldots, n_k$, where $I_n(n_i) = b_i$ for each $i$. Note that $n_0 = n$, necessarily.

As described above in step 2(c), HN **recurses** on each unlabeled blank node when it is first reached along the gossip path being explored. This recursion can be visualized as moving along the path from $n$ to the blank node $n_i$ that is receiving a temporary identifier. If, when recursing on $n_i$, another unlabeled blank node $n_j$ is discovered, the algorithm again recurses. Such a recursion traces out the gossip path from $n$ to $n_j$ via $n_i$.

The **recursive hash** $r(i)$ is the hash returned from the completed recursion on the node $n_i$ when computing $h_N(n)$. Just as $h_N(n)$ is the hash of $D_n$, we denote the data to hash in the recursion on $n_i$ as $D_i$. So, $r(i) = h(D_i)$. For each related hash $x \in H_n$, $R_n(x)$ is called the **recursion list** on
which the algorithm recurses.

Example 2.5. Figure 3 shows a graph with one nonblank node $s$ and seven blank nodes. Of the blank nodes, $c_1$ is canonically labeled, and $n$ and each $n_i$ are unlabeled. This example computes the data to hash when executing $HN$ on $n$. Note that while $n$ is related to four blank nodes, only three distinct related hashes are produced since we assume that the mentions $m_3$ and $m_4$ correspond to the same related hash. In particular, $h_f(n_4) = h_f(n_5)$. Table 3 shows each related hash $x$, a permutation of its related hash to blank node list $[x]$, and its data to hash term $T(x)$.

<table>
<thead>
<tr>
<th>$H_n$</th>
<th>$[x]$</th>
<th>$T(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = h(\text{&quot;o &lt; p &gt; c_1&quot;})$</td>
<td>$[x_1] = {c_1}$</td>
<td>$x_1 c_1$</td>
</tr>
<tr>
<td>$x_2 = h(\text{&quot;s &lt; p &gt; h_f(n_1)&quot;})$</td>
<td>$[x_2] = {n_1}$</td>
<td>$x_2 b_1 b_1 &lt; r(1)$</td>
</tr>
<tr>
<td>$x_3 = h(\text{&quot;s &lt; p &gt; h_f(n_4)&quot;})$</td>
<td>$[x_3] = {n_4, n_5}$</td>
<td>$x_3 b_4 b_5 b_4 &lt; r(4) &gt; b_5 &lt; r(5)$</td>
</tr>
</tbody>
</table>

Table 3

If we assume that $H_n$ is lexicographically ordered as $x_1, x_2, x_3$, then the following data to hash results from $HN$. 

$$D_n = x_1 c_1 x_2 b_1 b_1 < r(1) > x_3 b_4 b_5 b_4 < r(4) > b_5 < r(5) >.$$ 

Note that the blank nodes $n_2$ and $n_3$ receive the labels $b_2$ and $b_3$, respectively, when recursing on $n_1$. These labels appear in the recursive data to hash $< r(1)$>. In Section 2.2.5, Example 2.10 shows the steps of executing $HN$ on $n$ that ultimately produce $D_n$.

2.2.4 The Terms of a Node’s Data to Hash

$N$-degree hashes are used to determine the order in which canonical labels are distributed to blank nodes with non-unique first degree hashes. Recall that a blank node’s $N$-degree hash is computed from its data to hash string. Example 2.5 provided an example of the data to hash that might result from the algorithm. In this section, we characterize generally the terms of a blank node $n$’s data to hash $D_n$. This characterization will be essential for proving that URDNA2015 yields a canonical labeling of the dataset.
**Lemma 2.6.** Suppose that $n$ is a blank node with related hash list $H_n$. Let $x \in H_n$ be a related hash and $[x]$ be an arbitrary permutation of the related hash to blank node list for $x$. If a node $w \in [x]$ is already labeled when computing $x$, then $w$ is the only distinct node in $[x]$.

*Proof.* When computing the related hash of a previously labeled node $w$, its unique label (canonical or temporary) $I_n(w)$ is used in the input string. That is,

$$x = \begin{cases} 
  h(“s < p > I_n(w)”)) & \text{if } s = w \\
  h(“o < p > I_n(w)”)) & \text{if } o = w \\
  h(“gI_n(w)”)) & \text{if } g = w
\end{cases}$$

No other blank node (labeled or unlabeled) can return $w$’s label from $I_n$. Consequently, only $w$ can produce the related hash $x$. In this case, $[x] = \{w, w, \ldots, w\}$, where $w$ appears once for each quad whose related hash equals $x$.

\[\square\]

**Theorem 2.7.** Let $n$ be a node with related hash list $H_n$, let $x$ be a related hash in $H_n$, and let $[x]$ be the related hash to blank node list for $x$. Then, $x$ contributes a term $T(x)$ of exactly one of the following forms to $D_n$ when executing HN.

1. If $[x]$ contains a blank node $w$ that was issued label $I_n(w)$ prior to computing $x$, then

$$T(x) = x I_n(w) I_n(w) \cdots I_n(w).$$

Note: the number of copies of $I_n(w)$ is equal to the number of times that $w$ appears in $[x]$.

2. Otherwise, $[x]$ contains only blank nodes that were unlabeled when computing $x$. In this case, suppose $[x] = \{n_{i_1}, \ldots, n_{i_t}\}$ with recursion list $R_n(x) = \{n_{j_1}, n_{j_2}, \ldots, n_{j_t}\}$. Then,

$$T(x) = x b_{i_1} b_{i_2} \cdots b_{i_t} b_{j_1} < r(j_1) > b_{j_2} < r(j_2) > \cdots b_{j_t} < r(j_t) >.$$

Note that if $R_n(x) = \emptyset$, then the remainder terms $b_{j_1} < r(j_1) > b_{j_2} < r(j_2) > \cdots b_{j_t} < r(j_t) >$ are omitted from $T(x)$.

*Proof.* Given $x \in H_n$, regardless of the value of $[x]$, the first entry in the term $T(x)$ contributed to $D_n$ will be $x$, per step 5.1 of HN. The appended data that follows $x$ depends on whether $[x]$ contains a previously labeled node or not.

**Case 1.** $[x]$ contains a node that was labeled prior to computing $x$.

By Lemma 2.6, if $[x]$ contains a node that has already been labeled prior to computing $x$, then $[x] = \{m, m, \ldots, m\}$. Either $m$ is canonically identified or it is temporarily identified.

If $w$ has canonical identifier $I_n(w) = c_w$, then for each copy of $w$ in $[x]$, $c_w$ is appended to $D_n$ in step 5.4.4.1 of HN. In this case,

$$T(x) = x c_w c_w \cdots c_w.$$

Otherwise, $w$ has temporary identifier $I_n(w) = b_w$. For each copy of $w$ in $[x]$, $b_w$ is appended to $D_n$ in step 5.4.4.2 of HN. So,

$$T(x) = x b_w b_w \cdots b_w.$$

Either way, $T(x) = x I_n(w) I_n(w) \cdots I_n(w)$.
2.2 N-Degree Hashes

Case 2. \([x]\) contains only nodes that were unlabeled when computing \(x\).

In this case, suppose \([x] = \{n_{i_1}, \ldots, n_{i_r}\}\) with recursion list \(R_n(x) = \{n_{j_1}, n_{j_2}, \ldots, n_{j_k}\}\). Note that \(R_n(x)\) is an ordered subset of the distinct nodes from \([x]\) on which the algorithm has not yet recursed when executing HN step 5 for \(x\). For each related node \(n_{i_k}\) in the permutation \([x]\) (including repetitions), the algorithm first checks whether \(n_{i_k}\) has been issued a label. If not, \(n_{i_k}\) is appended to the recursion list (HN 5.4.4.2.1) and issued a label (HN 5.4.4.2.2). Then, regardless of whether \(n_{i_k}\) was appended to \(R_n(x)\), its issued label \(b_{i_k}\) is appended to \(D_n\) (note: nodes that are repeated in the permutation will have their labels repeated accordingly in the data string). Once HN 5.4.4 has completed, a recursion term of the form \(b_{j_k} < r(k)\) is also appended for each node in \(R_n(x)\) (in order) in HN 5.4.5. That is,

\[
T(x) = x b_{i_1} b_{i_2} \cdots b_{i_r} b_{j_1} < r(j_1) > b_{j_2} < r(j_2) > \cdots b_{j_t} < r(j_t) > .
\]

Note that if \(R_n(x) = \emptyset\), then

\[
T(x) = x b_{i_1} b_{i_2} \cdots b_{i_r} .
\]

For example, if the permutation of \([x]\) is \(\{n_1, n_2, n_3, n_1, n_3, n_1\}\) and \(R_n(x) = \{n_1, n_2, n_3\}\). Then, the data appended is

\[
T(x) = x b_1 b_2 b_3 b_1 b_3 b_1 b_1 < r(1) > b_2 < r(2) > b_3 < r(3) > .
\]

\(\Box\)

Remark 2.8.

1. Whenever the algorithm recurses on a blank node \(n_i\), its recursive hash \(r(i)\) appears in \(D_n\). The recursion \(r(i)\) is itself the hash of a data string \(D_i\), whose terms follow the same form when applying Theorem 2.7 to \(n = n_i\).

2. The term \(T(x)\) appended to \(D_n\) is ultimately the lexicographically shortest string produced when running over all permutations of \([x]\). This is guaranteed by step 5.4 of HN.

Corollary 2.9. For each blank node \(n_i \in [P_n]\) every related hash in its related hash list \(H_{n_i}\) appears in a string that is hashed when executing HN on \(n\). That is, each of \(n_i\)'s mentions is encoded and ultimately influences the N-degree hash of \(n\).

Proof. If \(n_i = n\) (that is, \(i = 0\)), Theorem 2.7 demonstrates that every \(x \in H_n\) contributes a term \(T(x)\) to \(D_n\). Since \(x\) appears in \(T(x)\), \(x\) appears in \(D_n\), the data string whose hash yields the N-degree hash of \(n\). If \(n_i \neq n\) (that is, \(i > 0\)), Remark 2.8(i) notes how each \(x \in H_{n_i}\) contributes a term \(T(x)\) to the recursive data to hash string \(D_i\) used to compute \(r(i)\). Because \(n_i\) is recursed on when it is first issued the label \(b_i\), the term \(< r(i) >\) necessarily appears in a data to hash string that is ultimately used to compute the N-degree hash of \(n\).

\(\Box\)

2.2.5 Encoding Gossip Paths with the Hash N-Degree Quads Algorithm

Theorem 2.7 characterizes the terms of a node’s data to hash \(D_n\) for each of its related hashes in \(H_n\). In this section, we explain exactly how the data to hash \(D_n\) that results from executing the Hash N-Degree Quads (HN) algorithm on a blank node \(n\) encodes gossip paths. We also provide the full HN algorithm and an example for reference. Please refer to the high level summary of
2.2 N-Degree Hashes

Section 2.2.3 as needed.

HN explores each gossip path beginning at $n_i$ issuing temporary labels to unlabeled nodes in the path, until reaching a node that has already been labeled (either with a temporary or a canonical identifier). To begin, the lexicographically first related hash $x \in H_n$ is appended to $D_n$. Nodes related to $n$ with the related hash $x$ are each mentioned by $n$ via the same type of mention $m_1$. Either $m_1$ mentions a canonical node $w$ or $m_1$ mentions at least one unlabeled blank node.

In the former case, a term $T(x)$ of form (i) in Theorem 2.7 is appended to $D_n$. Each copy of $c_w$ in $T(x)$ corresponds to a gossip path $(n, m_1, c_w)$, one for each mention $m_1$ relating $n$ and $w$. Because gossip paths terminate when a labeled node is reached, the next hash in $H_n$ is selected. In this way, HN begins exploring a new path in search of unlabeled nodes. If the next related hash also leads to a canonically labeled node, the process repeats until a related hash encodes a mention to an unlabeled blank node.

Therefore, we assume that $x \in H_n$ encodes the mention $m_1$ that leads to the discovery of an unlabeled blank node. Each unlabeled node $n_i \in [P_i]$ that is mentioned by $n$ via $m_1$ if issued a temporary label and a gossip path $(n, m_1, n_i)$ is initialized. These paths are indicated in $T(x)$ by $xb_i_1b_i_2\ldots b_i_{\ell}$ where each $b_i_j$ corresponds to the path $(n, m_1, n_i_j)$. After initializing each path, HN recurses on $n_1$, which is indicated by the term “$b_1 < r(1)$” in $T(x)$. Recursing on $n_1$ can be visualized as moving along $m_1$ to $n_1$ and next looking for unlabeled nodes that are related to $n_1$.

The first related hash $x_2 \in H_{n_1}$ encodes the next mention $m_2$ to move along. Again, if $m_2$ mentions a canonically labeled node, $T(x_2)$ is appended to $n_1$’s data to hash $D_1$ and the next hash in $H_{n_1}$ is selected. Similarly, if $m_2$ mentions a node $n_i$ that has already received a temporary label, the term $T(x_2)$ is recorded and the next hash in $H_{n_1}$ is selected. In particular, Note that when recursing on $n_1$, the related hash that encodes the mention from $n_1$ to $n$ (from the perspective of $n_1$) will appear in the recursive data to hash, however $n$ will not be appended to the recursion list as it has already received a temporary label.

As long as there is a hash in $H_{n_1}$ that encodes a mention $m_2$ that leads to the discovery of a new unlabeled node, the algorithm will continue to recurse on the newly discovered node, extending the gossip path by the new mention and the new node. For example, if $m_2$ is the first mention that leads to the discovery of an unlabeled node $n_i$, then the algorithm will move to $n_i$. This movement corresponds to the path $(n, m_1, n_1, m_2, n_i)$.

If, however, each hash in $H_{n_1}$ only leads to previously labeled nodes, the recursion on $n_1$ completes and the algorithm returns to explore paths beginning at $n$ via the next related hash in $H_n$.

HN explores gossip paths beginning at $n$ by moving to a related node via a related hash. HN will continue to step from node to node along a gossip path until the only related nodes have already been labeled. If for example, the path moves from $n_i$ and discovers the labeled node $n_j$, the algorithm returns to $n_i$ to explore other mention options (i.e. the next related hash in $H_{n_i}$). If there are no other mentions to explore, the algorithm returns to the previous node $n_{\ell}$ in the path from which $n_i$ was reached, exploring its mention options. In this case, returning to $n_{\ell}$ signals that the recursion of HN on $n_{\ell}$, $r(i)$, has completed. The algorithm continues to explore other “branches” until all recursions along the gossip path have been completed, i.e. backing out along the path until returning to the initial node $n$. 

13
After returning to \( n \), a new path is selected for exploration via the next related hash in \( H_n \). This continues until the related hash list \( H_n \) has been exhausted. In this way, \( HN \) visits every unlabeled node \( n_i \in [P_n] \) that is reachable from \( n \) and encodes every mention in \([P_n]\).

When a mention \( m \) leads to the discovery of more than one unlabeled node, all possible ways for issuing temporary identifiers to the adjacent nodes are explored. For example, suppose that \( w \) and \( w' \) are each unlabeled. The algorithm first explores the result of labeling \( w \) as \( b_1 \) and \( w' \) as \( b_2 \), recording the data to hash until no new nodes can be reached along that path. Then, it explores the result of labeling \( w' \) as \( b_1 \) and \( w \) as \( b_2 \). A permutation labeling that yields the shortest data to hash path is ultimately selected. So, \( HN \) explores all the gossip paths in \([P_n]\) and all of the ways to issue labels to unlabeled nodes along those paths.

### Hash N-Degree Quads (HN) Algorithm

This algorithm inputs the normalization state, a blank node identifier \( n \) on which to recurse, and a path identifier \( \text{issuer} \) that issues temporary node identifiers.

1. Create a hash to related blank nodes map for storing hashes that identify related blank nodes. The set of all related hashes that are ultimately stored in this map is denoted by \( H_n \).

2. Get a reference to the list of quads \( Q_n \) that mention the blank node identifier \( n \).

3. For each quad \( q \) in \( Q_n \):
   
   3.1. For each component in \( q \), if the component is the subject \( s \), object \( o \), or graph name \( g \), and it is a blank node identified by \( n' \) where \( n' \neq n \):
      
      3.1.1. Set \( \text{hash} \), denoted by \( x \), to the result of the Hashed Related Blank Node Algorithm, passing the related blank node \( n' \), quad \( q \), path identifier \( \text{issuer} \), and the component position of \( n' \) in \( q \) as either \( s \), \( o \) or \( g \). That is, \( x = h_{r}(q, n, n_i, \text{position}) \).
      
      3.1.2. Add a mapping of \( \text{hash} \) to the blank node for \( n' \) to the hash to related blank nodes map, adding an entry as necessary. That is, add \( x \) to the related set \( H_n \) (if it is not already a member of the set) and append \( n' \) to the related hash to blank node list \([x]\).

4. Create an empty string \( \text{data to hash} \) denoted by \( D_n \).

5. For each related hash \( x \) in \( H_n \), sorted lexicographically by related hash:
   
   5.1. Append the related hash \( x \) to \( D_n \).
   
   5.2. Create an unset string \( \text{chosen path} \). This will later be used to store the current shortest gossip path through the gossip class \([P_n]\) of \( n \) via a mention with related hash \( x \) that produces the shortest data to hash string \( D_n \).
   
   5.3. Create an unset \( \text{chosen issuer} \) variable. This will ultimately be the state of the issuer \( I_n \) that is returned from \( HN \).
5.4. For each permutation of the related hash to blank node list \([x]\)

5.4.1. Create a copy of issuer, issuer copy.

5.4.2. Create an unset string path to store the gossip path through \([P_n]\) that is being explored along this permutation.

5.4.3. Create an unset recursion list \(R_n(x)\) to store blank node identifiers that must be recursively processed by this algorithm.

5.4.4. For each related blank node \(n'\) in the permutation of \([x]\):

5.4.4.1. If a canonical identifier \(c_{n'}\) has been issued for \(n'\), append it to path.

5.4.4.2. Otherwise:

5.4.4.2.1. If issuer copy has not issued an identifier for \(n'\), append \(n'\) to \(R_n(x)\).

5.4.4.2.2. Use the Issue Identifier algorithm, passing issuer copy and \(n'\), and append the result to path.

5.4.4.3. If chosen path is not empty and chosen path is shorter than or equal to path, then skip to the next permutation.

5.4.5. For each related blank node \(n'\) in \(R_n(x)\):

5.4.5.1. Set result to the result of recursively executing the HN algorithm, passing \(n'\) and issuer copy for the path identifier issuer. This result is denoted by \((r(n'), I_{n'})\) where \(r(n')\) is the returned hash \(h_N(n')\) from the recursion.

5.4.5.2. Use the Issuer Identifier algorithm, passing the issuer copy and \(n'\), and append the result \(I_{n'}(n')\) to path.

5.4.5.3. Append \(<r(n')>\) to path.

5.4.5.4. Set issuer copy to the identifier issuer \(I_{n'}\) in result.

5.4.5.5. If chosen path is not empty and chosen path is shorter than or equal to path, then skip to the next permutation.

5.4.6. If chosen path is empty or path is shorter than chosen path, set chosen path to path and chosen issuer to issuer copy.

5.5. Append chosen path to \(D_n\).

5.6. Replace issuer with chosen issuer.

6. Return issuer and the hash that results from passing \(D_n\) through the hash algorithm. This result is denoted by \((h_N(n), I_n)\)

Example 2.10. The following example illustrates the process of executing the Hash N-Degree Quads (HN) algorithm on the unlabeled blank node \(n\) of the graph in Figure 4. In this graph, \(s\) is nonblank, \(c_1\) is a canonically labeled blank node, and each \(n_i\) is an unlabeled blank node. Just prior to executing HN on \(n\), URDNA2015 assigns the temporary label \(b_0\) to \(n\).

1. Initialize \(H_n\).

2. \(Q_n = \{< n, p, s, g >, < n, p, c_1, g >, < n_1, p, n, g >, < n_4, p, n, g >, < n_5, p, n, g >\}\). 

3. Compute the lexicographically ordered related hash list \(H_n = \{x_1, x_2, x_3\}\) shown in Table 4. Note that because \(n_4\) and \(n_5\) have the same first degree hash, they both correspond to the related hash \(x_3\).
Table 4: The related hashes of $H_n$ and their hash to blank node lists.

<table>
<thead>
<tr>
<th>$H_n$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_1 = h(\text{&quot;o &lt; p &gt; c_1&quot;})$</td>
<td>$x_1 = {c_1}$</td>
</tr>
<tr>
<td>$x_2 = h(\text{&quot;s &lt; p &gt; h_f(n_1)&quot;})$</td>
<td>$x_2 = {n_1}$</td>
</tr>
<tr>
<td>$x_3 = h(\text{&quot;s &lt; p &gt; h_f(n_4)&quot;})$</td>
<td>$x_3 = {n_4, n_5}$</td>
</tr>
</tbody>
</table>

4. Initialize $D_n$.

5. The first gossip path explored is $(n, m_1, c_1)$ (see Figure 5). For $x_1$ in $H_n$:

5.1. Append $x_1$ to $D_n$. Note: $D_n = x_1$.

5.2. Initialize chosen path.

5.3. Initialize chosen issuer.

5.4. For the unique permutation $\{c_1\}$ of $[x_1]$:

5.4.1. Create issuer copy.

5.4.2. Create path. Note there is only one path to explore $(n, m_1, c_1)$.

5.4.3. Initialize $R_n(x_1)$.

5.4.4. For $c_1$ in $\{c_1\}$:
5.4.4.1. Append $c_1$ to path.

5.4.5. $R_n(x_1) = \emptyset$ since $c_1$ has already been issued a canonical identifier.

5.4.6. chosen path = $c_1$ and chosen issuer = issuer copy.

5.5. Append chosen path to $D_n$. So, $D_n = x_1c_1$.

5.6. Replace issuer with chosen issuer.

Figure 6: Exploring the gossip path from $n$ to $n_1$ via the mention $m_2$.

The next gossip path explored is $(n, m_2, n_1)$. For $x_2$ in $H_n$:

5.1. Append $x_2$ to $D_n$. Note: $D_n = x_1c_1x_2$.

5.2. Initialize chosen path.

5.3. Initialize chosen issuer.

5.4. For the unique permutation $\{n_1\}$ of $[x_2]$:

5.4.1. Create issuer copy.

5.4.2. Create path. Note there is only one path to explore $(n, m_2, n_1)$.

5.4.3. Initialize $R_n(x_2)$.

5.4.4. For $n_1$ in $\{n_1\}$:

5.4.4.1. There is no canonical identifier for $n_1$.

5.4.4.2.

5.4.4.2.1. Append $n_1$ to $R_n(x_2)$. Note: $R_n(x_2) = \{n_1\}$.

5.4.4.2.2. Issue $n_1$ the temporary identifier $b_1$ via the Issue Identifier algorithm passing issuer copy. Append $b_1$ to path. Note: path = $b_1$.

5.4.5. For $n_1$ in $R_n(x_2)$:

5.4.5.1. Recurse on $n_1$, passing issuer copy. Return the resulting $N$-degree hash $r(1)$ and issuer $I_{n_1}$. Note that recursing on $n_1$ leads to the distribution of temporary labels $b_2$ and $b_3$ to $n_2$ and $n_3$. Example 2.11 shows the steps of HN when computing $r(1)$.

5.4.5.2. Append $b_1$ to path. Note: path = $b_1b_1$.

5.4.5.3. Append $< r(1) >$ to path. That is, path = $b_1b_1 < r(1) >$.

5.4.5.4. issuer copy = $I_{n_1}$.
5.4.5.5. chosen path = $\emptyset$.
5.4.6. chosen path = $b_1b_1 < r(1)$ and chosen issuer = $I_{n_1}$.
5.5. Append chosen path to $D_n$. Note: $D_n = x_1c_1 x_2 b_1 b_1 < r(1) >$.
5.6. Replace issuer with $I_{n_1}$.

Figure 7: Exploring the gossip paths along the mentions $m_4$ and $m_5$ that each correspond to the related hash $x_3$.

Because the next related hash corresponds to two mentions, there are two paths to explore, $p_{nn_4} = (n, m_3, n_4)$ and $p_{nn_5} = (n, m_4, n_5)$, shown in Figure 7. There are two orders in which these paths can be explored; that is, there are two permutations of $[x_3]$ that must be encoded and compared.

For $x_3$ in $H_n$:
5.1. Append $x_3$ to $D_n$. Note: $D_n = x_1c_1 x_2 b_1 b_1 < r(1) > x_3$.
5.2. Initialize chosen path.
5.3. Initialize chosen issuer.
5.4. There are two permutations, $\{n_4, n_5\}$ and $\{n_5, n_4\}$, of $[x_3]$.

For the permutation $\{n_4, n_5\}$ that explores $p_{nn_4}$ and then $p_{nn_5}$:
5.4.1. Create issuer copy.
5.4.2. Create path.
5.4.3. Initialize $R_n(x_3)$.
5.4.4. For $n_4$ in $\{n_4, n_5\}$:
5.4.4.1. There is no canonical identifier for $n_4$.
5.4.4.2.
5.4.4.2.1. Append $n_4$ to $R_n(x_3)$. Note: $R_n(x_3) = \{n_4\}$.
5.4.4.2.2. Issue $n_4$ the temporary identifier $b_4$ via the Issue Identifier algorithm passing issuer copy. Append $b_4$ to path. Note: path = $b_4$. Note that the labels $b_2$ and $b_3$ were already issued when recursing on $n_1$ via the related hash $x_2$.

For $n_5$ in $\{n_4, n_5\}$:
5.4.4.1. There is no canonical identifier for \( n_5 \).

5.4.4.2.

5.4.4.2.1. Append \( n_5 \) to \( R_n(x_3) \). Note: \( R_n(x_3) = \{ n_4, n_5 \} \).

5.4.4.2.2. Issue \( n_5 \) the temporary identifier \( b_5 \) via the Issue Identifier algorithm passing issuer copy. Append \( b_5 \) to path. Note: path = \( b_4b_5 \).

5.4.5. \( R_n(x_3) = \{ n_4, n_5 \} \).

For \( n_4 \) in \( R_n(x_3) \):

5.4.5.1. Recurse on \( n_4 \), passing issuer copy. Return the resulting \( N \)-degree hash \( r(4) \) and issuer \( I_{n_4} \).

5.4.5.2. Append \( b_4 \) to path. Note: path = \( b_4b_5b_4 \).

5.4.5.3. Append \( < r(4) > \) to path. That is, path = \( b_4b_5b_4 < r(4) > \).

5.4.5.4. issuer copy = \( I_{n_4} \).

5.4.5.5. chosen path = \( \varnothing \).

For \( n_5 \) in \( R_n(x_3) \):

5.4.5.1. Recurse on \( n_5 \), passing \( I_{n_4} \). Return the resulting \( N \)-degree hash \( r(5) \) and issuer \( I_{n_5} \).

5.4.5.2. Append \( b_5 \) to path. Note: path = \( b_4b_5b_4 < r(4) > b_5 \).

5.4.5.3. Append \( < r(5) > \) to path. That is, path = \( b_4b_5b_4 < r(4) > b_5 < r(5) > \).

5.4.5.4. issuer copy = \( I_{n_5} \).

5.4.5.5. chosen path = \( \varnothing \).

5.4.6. chosen path = \( b_4b_5b_4 < r(4) > b_5 < r(5) > \) and chosen issuer = \( I_{n_5} \).

For the permutation \( \{ n_5, n_4 \} \) that explores \( p_{n_5} \) and then \( p_{n_4} \):

5.4.1. Create issuer copy.

5.4.2. Create path.

5.4.3. Initialize \( R_n(x_3) \).

5.4.4. For \( n_5 \) in \( \{ n_5, n_4 \} \):

5.4.4.1. There is no canonical identifier for \( n_5 \).

5.4.4.2.

5.4.4.2.1. Append \( n_5 \) to \( R_n(x_3) \). Note: \( R_n(x_3) = \{ n_5 \} \).

5.4.4.2.2. Issue \( n_5 \) the temporary identifier \( b_4 \) via the Issue Identifier algorithm passing issuer copy. It is important to note that for this permutation, \( n_5 \) receives the label \( b_4 \) (whereas \( n_4 \) was labeled \( b_4 \) in the other permutation). Append \( b_4 \) to path. Note: path = \( b_4 \).

For \( n_4 \) in \( \{ n_5, n_4 \} \):

5.4.4.1. There is no canonical identifier for \( n_4 \).

5.4.4.2.

5.4.4.2.1. Append \( n_4 \) to \( R_n(x_3) \). Note: \( R_n(x_3) = \{ n_5, n_4 \} \).

5.4.4.2.2. Issue \( n_4 \) the temporary identifier \( b_5 \) via the Issue Identifier algorithm passing issuer copy. Append \( b_5 \) to path. Note: path = \( b_4b_5 \).
5.4.5. \( R_n(x_3) = \{n_5, n_4\} \).

For \( n_5 \) in \( R_n(x_3) \):

5.4.5.1. Recurse on \( n_5 \), passing issuer copy. Return the resulting \( N \)-degree hash \( r(5) \) and issuer \( I_{n_5} \).

5.4.5.2. Append \( b_4 \) to path. Note: path = \( b_4 b_5 b_4 \).

5.4.5.3. Append \( < r(5) > \) to path. That is, path = \( b_4 b_5 b_4 < r(5) > \).

5.4.5.4. issuer copy = \( I_{n_5} \).

5.4.5.5. chosen path = \( b_4 b_5 b_4 < r(4) > b_5 < r(5) > \), so do not skip.

For \( n_4 \) in \( R_n(x_3) \):

5.4.5.1. Recurse on \( n_4 \), passing \( I_{n_5} \). Return the resulting \( N \)-degree hash \( r(4) \) and issuer \( I_{n_4} \).

5.4.5.2. Append \( b_5 \) to path. Note: path = \( b_4 b_5 b_4 < r(4) > b_5 \).

5.4.5.3. Append \( < r(4) > \) to path. That is, path = \( b_4 b_5 b_4 < r(5) > b_5 < r(4) > \).

5.4.5.4. issuer copy = \( I_{n_4} \).

5.4.5.5. Recall that chosen path = \( b_4 b_5 b_4 < r(4) > b_5 < r(5) > \). The symmetry of \( n_4 \) and \( n_5 \) implies that \( r(4) = r(5) \). Therefore, chosen path = path, so do not skip.

5.4.6. Note that path = chosen path, so do not replace chosen path or chosen issuer.

Note: chosen issuer = \( I_{n_5} \) (the final state of \( I_{n_5} \) for the permutation \( \{n_4, n_5\} \)).

5.5. Append chosen path to \( D_n \).

\[
D_n = x_1 c_1 x_2 b_1 b_1 b_1 < r(1) > x_3 b_4 b_5 b_4 < r(4) > b_5 < r(5) > .
\]

5.6. Replace issuer with \( I_{n_5} \).

6. Return \((h_N(n), I_n)\) where \( h_N(n) = h(D_n) \) and \( I_n = I_{n_5} \).

**Example 2.11.** In this example, we show the steps of recursing on \( n_1 \) that were omitted in Example 2.10. That is, we compute \( r(1) \) by executing \( H \) on \( n_1 \). Note that \( c_1 \) is canonically labeled and that \( n \) has been issued the temporary label \( b_0 \).

1. Initialize \( H_{n_1} \).

2. \( Q_{n_1} = \{< n_1, p, n, g >, < n_1, p, n_2, g >, < n_1, p, n_3, g >\} \).

3. Compute the lexicographically ordered related hash list \( H_{n_1} = \{x_4, x_5\} \) shown in Table 5. Note that because \( n_2 \) and \( n_3 \) have the same first degree hash, they both correspond to the related hash \( x_5 \).

<table>
<thead>
<tr>
<th>( H_{n_1} )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_4 = h(\text{“o &lt; p &gt; b}_0\text{”}) )</td>
<td>( [x_4] = {n} )</td>
</tr>
<tr>
<td>( x_5 = h(\text{“o &lt; p &gt; h}_f(n_2)\text{”}) )</td>
<td>( [x_5] = {n_2, n_3} )</td>
</tr>
</tbody>
</table>

Table 5: The related hashes of \( H_{n_1} \) and their hash to blank node lists.
4. Initialize $D_{n_1}$.

5. The first gossip path explored is $(n_1, m_2, n)$, shown in Figure 8. For $x_4$ in $H_{n_1}$:

   5.1. Append $x_4$ to $D_{n_1}$. Note: $D_{n_1} = x_4$.
   5.2. Initialize chosen path.
   5.3. Initialize chosen issuer.
   5.4. For the unique permutation $\{n\}$ of $[x_4]$:
      
      5.4.1. Create issuer copy.
      5.4.2. Create path. Note there is only one path to explore $(n_1, m_2, n)$.
      5.4.3. Initialize $R_{n_1}(x_4)$.
      5.4.4. For $n$ in $\{n\}$:
         
         5.4.4.1. There is no canonical identifier for $n$.
         5.4.4.2. $n$ has already been issued the temporary label $b_0$. Thus, $R_{n_1}(x_4) = \emptyset$.
         5.4.4.2. Append $b_0$ to path. Note: path = $b_0$.
         5.4.4.3. chosen path = $\emptyset$, so do not skip.
         5.4.5. $R_{n_1}(x_4) = \emptyset$.
         5.4.6. chosen path = $b_0$ and chosen issuer = issuer copy = $I_n$.
   5.5. Append chosen path to $D_{n_1}$. Note: $D_{n_1} = x_4b_0$.
   5.6. Replace issuer with $I_n$.

Because the next related hash corresponds to two mentions, there are two paths to explore, $p_{nn_2} = (n_1, m_5, n_2)$ and $p_{nn_3} = (n_1, m_6, n_3)$, shown in Figure 9. There are two orders in which these paths can be explored; that is, there are two permutations of $[x_5]$ that must be encoded and compared.

For $x_5$ in $H_{n_1}$:

   5.1. Append $x_5$ to $D_{n_1}$. Note: $D_{n_1} = x_4b_0x_5$.  

Figure 8: Exploring the gossip path from $n_1$ to $n$ via the mention $m_2$. 

Figure 9: Exploring the gossip path from $n_1$ to $n$ via other means.
5.2. Initialize chosen path.
5.3. Initialize chosen issuer.
5.4. There are two permutations, \{n_2, n_3\} and \{n_3, n_2\}, of \([x_5]\).

For the permutation \{n_2, n_3\} that explores \(p_{n_2}\) and then \(p_{n_3}\):

5.4.1. Create issuer copy.
5.4.2. Create path.
5.4.3. Initialize \(R_{n_1}(x_5)\).
5.4.4. For \(n_2\) in \{n_2, n_3\}:
   5.4.4.1. There is no canonical identifier for \(n_2\).
   5.4.4.2.1. Append \(n_2\) to \(R_{n_1}(x_5)\). Note: \(R_{n_1}(x_5) = \{n_2\}\).
   5.4.4.2. Issue \(n_2\) the temporary identifier \(b_2\) via the Issue Identifier algorithm passing issuer copy. Append \(b_2\) to path. Note: path = \(b_2\).

For \(n_3\) in \{n_2, n_3\}:
5.4.4.1. There is no canonical identifier for \(n_3\).
5.4.4.2.1. Append \(n_3\) to \(R_{n_1}(x_5)\). Note: \(R_{n_1}(x_5) = \{n_2, n_3\}\).
5.4.4.2.2. Issue \(n_3\) the temporary identifier \(b_3\) via the Issue Identifier algorithm passing issuer copy. Append \(b_3\) to path. Note: path = \(b_2b_3\).

5.4.5. \(R_{n_1}(x_5) = \{n_2, n_3\}\).

For \(n_2\) in \(R_{n_1}(x_5)\):
5.4.5.1. Recurse on \(n_2\), passing issuer copy. Return the resulting \(N\)-degree hash \(r(2)\) and issuer \(I_{n_2}\).
5.4.5.2. Append \(b_2\) to path. Note: path = \(b_2b_3b_2\).
5.4.5.3. Append \(<r(2)>\) to path. That is, path = \(b_2b_3b_2 <r(2)>\).
5.4.5.4. issuer copy = \(I_{n_2}\).
2.2 N-Degree Hashes

5.4.5. chosen path = ∅.

For n3 in Rn1(x5):
5.4.5.1. Recurse on n3, passing I_{n_2}. Return the resulting N-degree hash r(3) and issuer I_{n_3}.
5.4.5.2. Append b_3 to path. Note: path = b_2b_3b_2 < r(2) > b_3.
5.4.5.3. Append < r(3) > to path. That is, path = b_2b_3b_2 < r(2) > b_3 < r(3) >.
5.4.5.4. issuer copy = I_{n_3}.
5.4.5.5. chosen path = ∅.
5.4.6. chosen path = b_2b_3b_2 < r(2) > b_3 < r(3) > and chosen issuer = I_{n_3}.

For the permutation \{n_3, n_2\} that explores p_{nn_3} and then p_{nn_2}:

5.4.1. Create issuer copy.
5.4.2. Create path.
5.4.3. Initialize R_{n_1}(x_5).
5.4.4. For n_3 in \{n_3, n_2\}:
5.4.4.1. There is no canonical identifier for n_3.
5.4.4.2.
5.4.4.2.1. Append n_3 to R_{n_1}(x_5). Note: R_{n_1}(x_5) = \{n_3\}.
5.4.4.2.2. Issue n_3 the temporary identifier b_2 via the Issue Identifier algorithm passing issuer copy. It is important to note that for this permutation, n_3 receives the label b_2 (whereas n_2 was labeled b_2 in the other permutation). Append b_2 to path. Note: path = b_2.

For n_2 in \{n_3, n_2\}:
5.4.4.1. There is no canonical identifier for n_2.
5.4.4.2.
5.4.4.2.1. Append n_2 to R_{n_1}(x_5). Note: R_{n_1}(x_5) = \{n_3, n_2\}.
5.4.4.2.2. Issue n_2 the temporary identifier b_3 via the Issue Identifier algorithm passing issuer copy. Append b_3 to path. Note: path = b_2b_3.
5.4.5. R_{n_1}(x_5) = \{n_3, n_2\}.

For n_3 in R_{n_1}(x_5):
5.4.5.1. Recurse on n_3, passing issuer copy. Return the resulting N-degree hash r(3) and issuer I_{n_3}.
5.4.5.2. Append b_2 to path. Note: path = b_2b_3b_2.
5.4.5.3. Append < r(3) > to path. That is, path = b_2b_3b_2 < r(3) >.
5.4.5.4. issuer copy = I_{n_3}.
5.4.5.5. chosen path = b_2b_3b_2 < r(2) > b_3 < r(3) >, so do not skip.

For n_2 in R_{n_1}(x_5):
5.4.5.1. Recurse on n_2, passing I_{n_3}. Return the resulting N-degree hash r(2) and issuer I_{n_2}.
5.4.5.2. Append b_3 to path. Note: path = b_2b_3b_2 < r(3) > b_2.
5.4.5.3. Append < r(2) > to path. That is, path = b_2b_3b_2 < r(3) > b_3 < r(2) >.
5.4.5.4. issuer copy = I_{n_2}.

5.4.5.5. Recall that chosen path = b_2 b_3 b_2 < r(2) > b_3 < r(3) >. The symmetry of n_2 and n_3 implies that r(2) = r(3). Therefore, chosen path = path, so do not skip.

5.4.6. Note that path = chosen path, so do not replace chosen path or chosen issuer. Note: chosen issuer = I_{n_3} (the final state of I_{n_3} for the permutation \{n_2,n_3\}).

5.5. Append chosen path to D_{n_1}.

\[ D_{n_1} = x_4 b_0 x_5 b_2 b_3 b_2 < r(2) > b_3 < r(3) >. \]

5.6. Replace issuer with I_{n_5}.

6. Return (r(1), I_{n_1}) where r(1) = h(D_{n_1}) and I_{n_1} = I_{n_5}.

2.2.6 Distributing Canonical Labels Via N-Degree Hashes

When a first degree hash is non-unique, all nodes with this hash are grouped together. For each group, the N-degree hash of each node in this group is computed and the state of their temporary issuers is returned. The lexicographically sorted list of N-degree hashes for these nodes is called the hash path list. Proceeding in lexicographical order, for each N-degree hash in the hash path list and its corresponding blank node n, every unlabeled blank node in the gossip class [P_n] of n will receive a canonical label. The order in which these nodes receive labels is the same as the order in which their temporary issuer I_n issued their temporary labels in HN.

This process is uniquely determined so long as the hash path list contains only distinct N-degree hashes. But, what if two nodes n and n’ that have the same first degree hash also have the same N-degree hash? We will show that the order in which each node’s temporary issuer is used to issue canonical labels does not matter. The labeled lists that result will be the same. This result is called the Temporary Labeling Theorem and is proven in Section 3.2.

3 The Canonical Labeling of URDNA2015

This chapter is devoted to demonstrating that the labeling that results from URDNA2015 is indeed canonical. That is, two RDF datasets will be labeled the same if and only if the datasets are isomorphic. We present the full algorithm here for reference.

**URDNA2015**

1. Create the normalization state.

2. For every quad q in the dataset D:

   2.1. For each blank node n that occurs in q, add a reference to q in the blank node to quads map, create a new entry if necessary.

3. Create a list of non-normalized blank node identifiers non-normalized identifiers and populate it using the keys from the blank node to quads map.
4. Initialize \( \text{simple} \), a boolean flag to \( \text{true} \).

5. While \( \text{simple} \) is \( \text{true} \), issue canonical identifiers for blank nodes:
   5.1. Set \( \text{simple} \) to \( \text{false} \).
   5.2. Clear hash to blank nodes map.
   5.3. For each blank node identifier \( n \) in non-normalized identifiers:
      5.3.1. Create its first degree hash \( h_f(n) \) via the Hash First Degree Quads algorithm.
      5.3.2. Add \( h_f(n) \) to the hash to blank nodes map, creating a new entry if necessary. Add \( h_f(n) \) to the list of first degree hashes \( H_F \), including repetitions.
   5.4. For each hash in \( H_F \), lexicographically-sorted by hash:
      5.4.1. If the hash appears more than once in \( H_F \), continue to the next hash.
      5.4.2. Use the Issue Identifier algorithm, passing canonical issuer and the single blank node identifier \( n \) such that \( h_f(n) = \text{hash} \).
      5.4.3. Remove \( n \) from non-normalized identifiers.
      5.4.4. Remove \( \text{hash} \) from the hash to blank nodes map.
      5.4.5. set \( \text{simple} \) to \( \text{true} \).

6. For each hash in \( H_F \), lexicographically-sorted by hash:
   6.1. Create hash path list where each item will be a result of running the Hash \( N \)-degree Quads algorithm.
   6.2. For each blank node identifier \( n \) in hash to blank nodes map, lexicographically sorted by hash:
      6.2.1. If a canonical identifier has already been issued for \( n \), continue to the next blank node identifier.
      6.2.2. Create temporary issuer \( I_n \), an identifier issuer initialized with \( {}_b \).
      6.2.3. Use the Identifier Issuer algorithm, passing temporary issuer \( I_n \), to issue a new temporary blank node identifier \( b_n \) to \( n \).
      6.2.4. Run the Hash \( N \)-Degree Quads algorithm, passing temporary issuer \( I_n \), and append the result to the hash path list.
   6.3. For each result in the hash path list, lexicographically-sorted by the \( N \)-degree hashes in result:
      6.3.1. For each blank node identifier \( n \) that was issued a temporary identifier by identifier issuer in result, issue a canonical identifier, in the same order using the Issue Identifier algorithm, passing canonical issuer and \( n \).

7. For each quad \( q \) in \( \mathcal{D} \):
   7.1. Create a copy, quad copy, of \( q \) and replace any existing blank node identifier \( n \) using the canonical identifier \( C(n) \) previously issued by canonical issuer.
   7.2. Add quad copy to the normalized dataset \( \mathcal{C}(\mathcal{D}) \).

8. Return the normalized dataset \( \mathcal{C}(\mathcal{D}) \).
3.1 RDF Dataset Comparison

To establish that URDNA2015 yields a canonical labeling, we must first give the conditions under which two datasets are isomorphic. We use the notations $I$, $L$, and $B$ to refer to the pairwise disjoint sets of IRIs, literals, and blank nodes, respectively in a given RDF dataset. For simplicity, we denote $I \cup L \cup B$ by $\text{ILB}$.

**Definition 3.1.** Let $\mathcal{D}$ and $\mathcal{D}'$ be RDF datasets. We say that $M : \text{ILB} \rightarrow \text{ILB}$ is a **dataset-isomorphism** provided that $M$ is a bijection between the terms of $\mathcal{D}$ and the terms of $\mathcal{D}'$ such that

1. $M$ is the identity on IRIs, literals, and the default graph symbol “−”;
2. $M$ maps blank nodes in $\mathcal{D}$ to blank nodes in $\mathcal{D}'$; and,
3. The image of the dataset $\mathcal{D}$ under $M$ is $M(\mathcal{D}) = \{< M(s), p, M(o), M(g) > \mid < s, p, o, g > \text{ is in } \mathcal{D} \}$

$\mathcal{D}$ and $\mathcal{D}'$ are **dataset-isomorphic** if there exists a dataset-isomorphism $M$ such that $M(\mathcal{D}) = \mathcal{D}'$.

The definition of dataset isomorphism can also be used to determine whether a subset of quads in one dataset is isomorphic to a subset of quads in another dataset. Given a subset $Q$ of $\mathcal{D}$ and a subset $Q'$ of $\mathcal{D}'$, we use the notation $Q \cong Q'$ to denote that $Q$ is isomorphic to $Q'$. Note that $Q \cong Q'$ need not imply that $\mathcal{D} \cong \mathcal{D}'$.

3.2 The Case of Equal $N$-Degree Hashes

As a first distinction, URDNA2015 uses first degree hashes to issue canonical identifiers. When multiple nodes have the same first degree hash, their $N$-degree hashes are computed. Then, canonical labels are issued according to the lexicographically sorted hash path list of $N$-degree hashes. In this section, we will demonstrate that when two nodes have the same $N$-degree hash, the order in which their temporary issuers are used to distribute canonical labels does not matter.

Below, with the assistance of a few lemmas, the Temporary Labeling Theorem asserts that when two nodes have the same $N$-degree hash, the subsets of quads associated with their gossip classes are isomorphic via the identification of nodes that receive the same temporary identifier by $I_n$ and $I_{n'}$, respectively. Because of this result, repeated $N$-degree hashes in the hash path list can be processed in any order.

**Lemma 3.2.** Suppose that the blank nodes $n$ and $n'$ have equal first degree hashes and equal $N$-degree hashes. If $I_n$ issues the label $b_i$ to $n_i \in [P_n]$ and $I_{n'}$ issues the label $b_i$ to $n'_i \in [P_{n'}]$, then $h_f(n_i) = h_f(n'_i)$.

**Proof.** By assumption, $h_f(n) = h_f(n')$. Therefore the claim is true when $i = 0$. Let $i \geq 1$ be arbitrary. When $n_i$ and $n'_i$ are each first issued the label $b_i$, their first degree hash is used to compute the related hash $x$ to which $b_i$ is appended to their data to hash strings $D_n$ and $D_{n'}$. Because $n$ and $n'$ have the same $N$-degree hash, it must be that $D_n = D_{n'}$. Furthermore, because $b_i$ must appear with exactly the same related hashes in both data strings, it must be that $h_f(n_i) = h_f(n'_i)$. Therefore, the first degree hashes of all corresponding nodes in $[P_n]$ and $[P_{n'}]$ must have the same first degree hash. □
Theorem 3.3 (The Temporary Labeling Theorem). Suppose that two blank nodes \( n \) and \( n' \) that have equal first degree hashes also have equal \( N \)-degree hashes. Then, \( \bigcup_{n_i \in [P_n]} Q_{n_i} \) is isomorphic to \( \bigcup_{n'_i \in [P_{n'}]} Q_{n'_i} \) via the map \( M(n_i) = n'_i \) where \( I_n(n_i) = I_{n'}(n'_i) \).

Proof. The proof is by contradiction. Assume that \( h_N(n) = h_N(n') \) but \( M \) is not an isomorphism. For simplicity, let \( Q = \bigcup_{n_i \in [P_n]} Q_{n_i} \) and \( Q' = \bigcup_{n'_i \in [P_{n'}]} Q_{n'_i} \). So, there exists a quad \( q = < s, p, o, g > \) in \( Q \) such that \( M(q) = < M(s), p, M(o), M(g) > \) is not in \( Q' \). Because \( Q \) is the collection of quads that mention each blank node \( n_i \), at least one component of \( q \) contains a blank node \( n_i \) for some \( i \).

Case 1. The only blank node mentioned by \( q \) is \( n_i \).

Let \( n_i \) be the blank node that \( q \) mentions. Without loss of generality, assume that \( n_i \) appears only in the subject of \( q \). So \( q = < n_i, p, o, g > \) where \( o \) and \( g \) are nonblank. Further, \( < n'_i, p, o, g > \) is not in \( Q' \). By Lemma 3.2, \( h_f(n_i) = h_f(n'_i) \). Because \( q \) mentions \( n_i \), its serialization \( < a, p, o, g > \) is included in the first degree hash of \( n_i \). Therefore, the serialization of \( < a, p, o, g > \) must also appear in \( h_f(n_i) \) since \( h_f(n_i) = h_f(n'_i) \). When computing \( h_f(n'_i) \), \( n'_i \) is the unique node that can produce the label \( a \) in the component of a quad. Furthermore, because \( o \) and \( g \) are nonblank, they also uniquely carry their respective identifiers. Therefore, the only quad that can produce this serialization is \( < n'_i, p, o, g > \). We assumed that no such quad existed. \( \Rightarrow \equiv \)

Case 2. \( q \) mentions \( n_i \), a blank node \( w \), and a nonblank node \( \eta \).

Without loss of generality, assume \( q = < n_i, p, w, \eta > \). Then, there is no quad of the form \( < n'_i, p, M(w), \eta > \) in \( Q' \). However, \( n_i \) and \( n'_i \) must have the same related hash list \( H_i \) since each of their related hashes is appended to the data string in \( r(i) \). Any discrepancy in their related hash lists would result in a related hash that appears in one string but not the other. Because \( n_i \) is related to \( w \) via \( q \), the related hash \( x = h(\text{"o} < p > I_n(w)) \) must appear in \( H_i \), where \( I_n(w) = h_f(w) \) if \( w \) did not have a label at the time \( x \) was computed. Furthermore, \( T(x) \) must have the same number of copies of the label \( I_n(w) \) appended to \( x \) in both \( D_n \) and \( D_{n'} \). Because \( q \) will produce one copy of \( I_n(w) \) in \( D_n \), there must be a quad \( q' \in Q' \) that contributes a copy of \( I_n(w) \) to \( T(x) \). Thus, \( q' \) must mention \( n'_i \) and \( M(w) \) together, with \( M(w) \) in the object. And, if \( q' \) is to account for the missing copy of \( I_n(w) \) in \( D_{n'} \), \( M^{-1}(q') \) cannot be in \( Q \).

Any introduction of a quad \( q' \) cannot change the first degree hash of \( n'_i \) since \( h_f(n_i) = h_f(n'_i) \).

Note: if \( q' \) were to contribute a first degree hash different from \( q \), we would need to introduce another quad to account for this change. This process would end in a contradiction because there are only finitely many quads.

So, because \( < n_i, p, w, \eta > \) is replaced with \( < a, p, z, \eta > \), \( q' \) must make this same replacement. Therefore, \( q' \) must have \( n'_i \) in the subject and \( \eta \) in the graph component (since \( \eta \) is a unique nonblank identifier). Recall, however, that \( M(w) \) must also be in the object to produce the related hash \( x \). Thus, \( q' = < n'_i, p, M(w), \eta > \) necessarily. \( \Rightarrow \equiv \) We assumed that no such quad existed.

Case 3. \( q \) mentions \( n_i \) and two blank nodes \( w_1 \) and \( w_2 \).

Without loss of generality, assume \( q = < n_i, p, w_1, w_2 > \) with \( w_1 \neq n_i \) and \( w_2 \neq n_i \). Note that
if \( w_2 = n_i, \) for example, this case would be similar to case 2 where \( q \) would be replaced with \(<a, p, z, a>\) in \( h_f(n_i) \), forcing the graph component of \( q' \) to be \( n'_j \).

By assumption, there is no quad of the form \(<n'_j, p, M(w_1), M(w_2)>\) in \( Q' \). As in case 2, \( n_i \) and \( n'_j \) must have the same related hash list \( H_i \). Because \( n_i \) is related to \( w_1 \) and \( w_2 \) via \( q \), the related hashes \( x_1 = h(o < p > I_n(w_1)) \) and \( h(gI_n(w_2)) \) must appear in \( H_i \). Note: \( I_n(w_1) = h_f(w_1) \) if \( w_1 \) did not have a label at the time \( x \) was computed, and similarly for \( I_n(w_2) \). Furthermore, \( T(x_1) \) must have the same number of copies of the label \( I_n(w_1) \) appended to \( x_1 \) and \( T(x_2) \) must have the same number of copies of \( I_n(w_2) \) appended to \( x_2 \) in both \( D_n \) and \( D_{n'} \). Because \( q \) will produce one copy of \( I_n(w_1) \) in \( T(x_1) \) and one copy of \( I_n(w_2) \) in \( T(x_2) \), there must be a quad \( q' \in Q' \) that contributes these copies in \( D_{n'} \).

We note here that exactly one quad \( q' \) must produce both these related hashes. For if two quads were used, that would necessitate the existence of yet another quad in \( q'' \in Q \) to maintain the length of the first degree hash for \( n_i \). Every time a quad is introduced in one set, we must find another quad in the other. Because there are only finitely many nodes, this would terminate in a contradiction.

Thus, \( q' \) must mention \( n'_j \) together with \( M(w_1) \) and \( M(w_2) \) in the graph. If either of \( M(w_1) \) and \( M(w_2) \) have previously been issued a label or if they have distinct first degree hashes, the related hashes \( x_1 \) and \( x_2 \) force \( M(w_1) \) to be in the object and \( M(w_2) \) to be in the graph. This would require \( q' = <n'_j, p, M(w_1), M(w_2)>, \) a contradiction.

Therefore, it must be that the first degree hashes \( h_f(M(w_1)) = h_f(M(w_2)) \) were used to compute \( x_1 \) and \( x_2 \), allowing for \( q' = <n'_j, p, M(w_1), M(w_2)>. \) Because their first degree hashes were used to compute these related hashes, \( M(w_1) \) and \( M(w_2) \) were unlabeled and can be denoted by some \( n'_j \) and \( n'_\ell \) in \( [P_{n'}] \). Thus, \( w_1 = n_j \) and \( w_2 = n_\ell \).

Summarizing, we now have that \( q = <n_i, p, n_j, n_\ell> \) and \( q' = <n'_j, p, n'_\ell, n'_j>. \) This leads to several contradictions. For example, when recursing on \( n'_\ell \), \( q \) and \( q' \) will contribute different related hashed for \( n_j \) and \( n'_j \) since the former will hash \( "o < p > b_j" \) and the latter will hash \( "gb_j" \). Additionally, the first degree hashes of \( n'_j \) and \( n'_\ell \) would be altered since the location of the \( a \) in the replacement of \( q' \) would be different from \( q \). Regardless, we reach a contradiction. \( \equiv \Leftarrow \equiv \)

Thus, in all three cases, we have shown that it is impossible for \( q \in Q \) and \( M(q) \notin Q' \). Therefore, \( M \) is an isomorphism between \( Q \) and \( Q' \). \( \Box \)

Remark 3.4. The argument of the Temporary Labeling Theorem demonstrates that if \( Q \) contains a quad \(<s, p, o, g>\), but \(<M(s), p, M(o), M(g)> \) does not appear in \( Q' \) it is impossible for \( D_n = D_{n'} \). It is important to note that every mention described by a gossip class will be encoded in the data to hash when computing an N-degree hash.

Corollary 3.5. Suppose that two blank nodes \( n \) and \( n' \) that have equal first degree hashes also have equal N-degree hashes. Then, \( I_n(Q) = I_{n'}(Q') \) where \( Q = \bigcup_{n_i \in [P_n]} Q_{n_i} \) and \( Q' = \bigcup_{n_i \in [P_{n'}]} Q_{n'_i} \).

Proof. By Theorem 3.3, \( Q \cong Q' \) via the subisomorphism \( M(n_i) = n'_i \) where \( I_n(n_i) = I_{n'}(n'_i) \). That is, \( q = <s, p, o, g> \) is in \( Q \) if an only if \( M(q) = <M(s), p, M(o), M(g)> \) is in \( Q' \). So, the
Corollary 3.5 shows that when two nodes labeled the same (namely with their nonblank identifier). And, because the identity map on nonblank nodes, corresponding nonblank components in q and M(q) must be labeled the same (namely with their nonblank identifier). And, because \( I_n(n_i) = I_{n'}(M(n_i)) \) for all blank nodes \( n_i \) in Q, all blank components must be labeled the same. Therefore, \( I_n(q) = I_n(M(q)) \). Because this is true for any \( q \in Q \), \( I_n(Q) = I_{n'}(Q') \).

The Case of Equal N-Degree Hashes

**Theorem 3.6 (The Repeated N-Degree Labeling Theorem).** Suppose that \( n \) and \( n' \) have the same N-degree hash in the hash path list for a first degree hash \( h_f \). Then, the order in which \( I_n \) and \( I_{n'} \) are used to issue canonical labels for the nodes of \([P_n]\) and \([P_{n'}]\) does not matter. That is, the labeled list of quads that results in each case will be identical.

**Proof.** Suppose that \( n \) and \( n' \) are such that \( h_N(n) = h_N(n') \). Let \( L_n = \{n_0, n_1, \ldots, n_k\} \) and \( L_{n'} = \{n_0', n_1', \ldots, n_k'\} \) be the order in which \( I_n \) and \( I_{n'} \) label the nodes of their respective gossip classes. Let \( C_1 = L_n \oplus L_{n'} \) and \( C_2 = L_{n'} \oplus L_n \) be the label orderings that result from concatenating \( L_n \) and \( L_{n'} \) in opposite order. We will show that

\[
C_1(Q \cup Q') = C_2(Q \cup Q')
\]

where \( Q = \bigcup_{n_i \in [P_n]} Q_{n_i} \) and \( Q' = \bigcup_{n_i' \in [P_{n'}]} Q_{n_i'} \).

That is, the labeled list of quads that results from issuing canonical labels according to \( C_1 \) will be the same as the labeled list of quads that results from issuing canonical labels according to \( C_2 \).

Because \( n \) and \( n' \) have the same N-degree hash, the Temporary Labeling Theorem implies that \( M(n_i) = n_i' \) defines an isomorphism between \( Q \) and \( Q' \). Furthermore, Corollary 3.5 implies that \( I_n(Q) = I_{n'}(Q') \).

1. **Case 1.** \( L_n \cap L_{n'} \neq \emptyset \).

The Hash N-Degree Quads algorithm recurses until all nodes in the gossip class of the starting node are issued a temporary label. So, if two gossip classes share a common unlabeled node, those gossip classes necessarily consist of exactly the same list of unlabeled nodes. Therefore, \( L_n \) and \( L_{n'} \) are comprised of the same nodes. That is, \( Q = Q' \).

Therefore, when issuing canonical labels for the ordering \( C_1 \), the issuer first issues labels to the nodes in \( L_n \) via \( I_n \). When the issuer reaches the nodes in \( L_{n'} \), they have already been issued canonical labels and their existing labels are returned by the canonical issuer (as new labels are only distributed to unlabeled nodes). A similar claim is true for the ordering \( C_2 \). So, because \( I_n(Q) = I_{n'}(Q') \) and \( Q = Q' \), the canonically labeled list of quads produced in either case are identical. That is,

\[
C_1(Q \cup Q') = C_1(Q) = I_n(Q) = I_n(Q') = C_2(Q') = C_2(Q \cup Q').
\]

2. **Case 2.** \( L_n \cap L_{n'} = \emptyset \).
In this case, \( Q \cap Q' = \emptyset \). In \( C_1(Q \cup Q') \), the temporary labels \( b_0, b_1, \ldots, b_k \) in \( I_n(Q) \) are replaced with the canonical labels \( c_0, \ldots, c_k \) and the temporary labels \( b_0, b_1, \ldots, b_k \) in \( I'_n(Q') \) are replaced with the canonical labels \( c_{k+1}, c_{k+2}, \ldots, c_{2k+1} \). In \( C_1(Q \cup Q') \), the temporary labels \( b_0, b_1, \ldots, b_k \) in \( I'_n(Q') \) are replaced with the canonical labels \( c_0, \ldots, c_k \) instead, and the temporary labels \( b_0, b_1, \ldots, b_k \) in \( I_n(Q) \) are replaced with the canonical labels \( c_{k+1}, c_{k+2}, \ldots, c_{2k+1} \). But, recall that \( I_n(Q) = I'_n(Q') \). So, in either case, an identical list of quads is ultimately produced.

In either case, \( C_1(Q \cup Q') = C_2(Q \cup Q') \). Therefore, when an \( N \)-degree hash is repeated in the hash path list, the associated nodes can be issued canonical labels in any order. Note that this result extends inductively when more than two results correspond to the same hash.

\[ \square \]

**Lemma 3.7.** Suppose that \( D \) and \( D' \) are isomorphic RDF datasets. Then, \( H_f = H'_f \). That is, \( D \) and \( D' \) have the same first degree hash list.

**Proof.** Assume that \( D \cong D' \). Then, there exists a dataset isomorphism \( M : D \rightarrow D' \).

Let \( n \) be a blank node in \( D \), and let \( q \in Q_n \). Because \( M \) is a dataset isomorphism, \( q \) is in \( D' \) if and only if \( M(q) = < M(n), p, M(n'), M(g) > \) is in \( D' \). Furthermore, \( M \) is the identity on nonblank components, and \( M \) maps blank nodes to blank nodes. Without loss of generality, suppose that \( q = < n, p, n', g > \) where \( n' \neq n \) is a blank node and \( g \) is nonblank. Then, \( M(q) = < M(n), p, M(n'), g > \) where \( M(n) \neq M(n') \) since \( n \neq n' \) and \( M \) is a bijection. Thus, \( q \) is replaced with \( < a, p, z, g > \) in \( h_f(n) \) and \( M(q) \) is replaced with \( < a, p, z, g > \) in \( h_f(M(n)) \). Because this will be true for a quad \( q \) of any form in \( Q_n \), \( h_f(n) = h_f(M(n)) \). And, because \( n \) was an arbitrary blank node in \( D \), we conclude that \( H_f = H'_f \).

\[ \square \]

**Theorem 3.8** (The Canonical Labeling Theorem). Suppose that \( D \) and \( D' \) are RDF datasets. Let \( C \) and \( C' \) denote the final issuer states at the conclusion of URDNA2015 on \( D \) and \( D' \), respectively. Then, \( C(D) = C'(D') \) if and only if \( D \) is isomorphic to \( D' \). That is, URDNA2015 produces a canonical labeling for an RDF dataset.

**Proof.**

\( \rightarrow \) Suppose that \( C(D) = C'(D') \). Then, Define the mapping \( M : D \rightarrow D' \) as follows.

1. \( M \) maps the blank node \( n \) in \( D \) to the blank node \( n' \) in \( D' \) where \( C(n) = C'(n') \).
2. \( M \) is the identity on URIs and literals.

Note that \( M \) is well defined because \( C \) and \( C' \) are bijections on the blanks nodes in \( D \) and \( D' \), respectively. Then, \( q = < s, p, o, g > \) is in \( D \) if and only if \( C(q) \) is in \( C(D) \) if and only if there exists a \( q' \) in \( D' \) such that \( C'(q') = C(q) \). By definition, \( q' = < s', p, o', g' > \) where \( C'(q') = C'(s'), p, C'(o'), C'(g') = C(s), p, C(o), C(g) > \). Thus, \( q' = < M(s), p, M(o), M(g) > = M(q) \). So, we have shown that \( q \in D \) if and only if \( M(q) \in D' \). By definition, \( M \) is a dataset isomorphism and \( D \cong D' \).

\( \leftarrow \) Suppose that \( D \cong D' \). Then, there exists a dataset isomorphism \( M : D \rightarrow D' \). By Lemma 3.7, \( H_f = H'_f \) (that is, their first degree hash lists agree). URDNA2015 immediately distributes canonical identifiers to those blank nodes whose first degree hash is unique. Thus, if \( n \) has a
unique first degree hash, \( C(n) = C'(M(n)) \) since \( h_f(n) = h_f(M(n)) \) (by Lemma 3.7). Therefore, \( C \) and \( C' \) necessarily agree on any blank node whose first degree hash is unique.

Now, consider a first degree hash \( h_f \) that is repeated in \( H_F = H'_F \). Given \( n \in D \) and \( M(n) \in D' \) with first degree hash \( h_f \), we will show that \( D_n = D_{M(n)} \). Then, \( n \) and \( M(n) \) will have the same \( N \)-degree hash.

Suppose toward a contradiction that \( D_n \neq D_{M(n)} \). Then, there exists a related hash \( x \in H_{n_i} \) for some \( n_i \) such that \( x \) contributes \( T(x) \) to \( D_{n_i} \) but \( T(x) \) does not appear in \( D_{M(n_i)} \). Because \( x \) is computed via mentions and \( M \) is mention-preserving, it must be that \( x \in H_{M(n_i)} \). So, the distinction of \( T(x) \) in \( D_{M(n_i)} \) must be due to the appearance of a different label \( \ell \) (canonical or temporary). However, each appearance of the label \( \ell \) indicates the existence of a mention of \( \ell \) with related hash \( x \). But again, \( n \) has all the same mentions as \( M(n) \) since \( M \) is an isomorphism. In particular, their related hash list is identical. \( \Rightarrow \Longleftrightarrow \)

Therefore, \( D_n = D_{M(n)} \). Therefore, the hash path list for \( h_f \) in \( D \) will be the same as the hash path list for \( h_f \) in \( D \). If \( n \) and \( M(n) \) produce the same \( N \)-degree hash, then they will produce the same temporary labeled lists of quads by Corollary 3.5. And, if \( n \) and \( n' \) each have the same \( N \)-degree hash, so will \( M(n) \) and \( M(n') \), and by the Repeated \( N \)-Degree Labeling Theorem, the order in which \( n \) and \( n' \) (respectively, \( M(n) \) and \( M(n') \)) are issued canonical identifiers does not matter. Putting these facts together, the quads that are labeled by the hash path list of \( h_f \) in \( D \) will receive the same labels from \( C \) as the quads \( M(q) \) in the hash path list of \( h_f \) in \( D' \) will receive from \( C' \).

Because this is true for the hash path list for any repeated first degree hash \( h_f \), we may conclude that \( C(D) = C(D') \).

\[ \square \]

**Remark 3.9.** Note that the final issuer state \( C \) is inherently a dataset isomorphism between \( D \) and \( C(D) \). Therefore, since \( D \cong C(D) \) and \( C(D) = C'(D') \), the labeling is indeed canonical.

### 3.3 URDNA2015 Terminates

Finally, it is important to note that URDNA2015 does indeed terminate. If all first degree hashes are unique, then step 6 of the algorithm is skipped. That is, HN is never run. So, HN never recurses and the canonicalized dataset is returned in step 8 after looping over finitely many first degree hashes. So, it suffices to show that when HN is necessary, the algorithm terminates.

When a first degree hash is non-unique, there are only finitely many blank nodes that produce it. HN is run on each of these nodes in step 6 of URDNA2015. Any node is related to finitely many blank nodes and therefore there are finitely many permutations over which HN loops. Furthermore, within each permutation HN only recurses once on each blank node (the first time it is issued a label). Therefore, HN necessarily terminates when running on a blank node \( n \). So, step 6 of URDNA2015 will be complete after looping over all nodes that produced the non-unique hash, and ultimately the canonicalized dataset is returned in step 8. That is, URDNA2015 terminates, returning an RDF dataset with canonical labels issued to all blank nodes. The same graph will
produce the same labeling (regardless of which party runs the algorithm) and different graphs will produce different labelings (as guaranteed by the Canonical Labeling Theorem).

A Notation Index

\( \mathcal{D} \)  an RDF dataset
\( q \)  a quad \(<s,p,o,g>\) where \( s = \) subject, \( p = \) predicate, \( o = \) object, and \( g = \) graph
\( n \)  a blank node in \( \mathcal{D} \)
\( Q_n \)  the set of all quads in \( \mathcal{D} \) that mention \( n \)
\( p_{nn'} \)  a gossip path from \( n \) to \( n' \).
\([P_n]\)  the gossip class of \( n \)
\( h \)  cryptographic hash function
\( HF \)  the Hash First Degree Quads algorithm
\( h_f(n) \)  the first degree hash of \( n \)
\( H_f \)  the set of all first degree hashes in \( \mathcal{D} \)
\( HR \)  the Hash Related Nodes algorithm
\( h_r(q,n,n',\text{position}) \)  the related hash of \( n \) describing how \( q \in Q_n \) mentions \( n' \)
\( H_n \)  the set of all related hashes for \( n \)
\( HN \)  the Hash N-Degree Quads algorithm
\( h_N(n) \)  the \( N \)-degree hash of \( n \)
\( D_n \)  the HN data to hash such that \( h_N(n) = h(D_n) \)
\( I_n \)  the issuer that issues temporary labels when executing \( HN \) on \( n \)
\( b_i \)  a temporary label for a node \( n_i \) issued by \( I_n \)
\( C \)  the canonical label issuer for \( \mathcal{D} \)
\( c_n \)  the canonical label for \( n \) issued by \( C \)

References

