# King's Research Portal

# Incremental Inference of Provenance Types[*]

David Kohan Marzagão[1][0000−0001−8475−7913], Trung Dong
Huynh[1][0000−0003−4937−2473], and Luc Moreau[1][0000−0002−3494−120X]

King's College London, London WC2B 4BG, UK
{david.kohan,dong.huynh,luc.moreau}@kcl.ac.uk

**Abstract.** Long-running applications nowadays are increasingly instrumented to continuously log provenance. In that context, we observe an emerging need for processing fragments of provenance continuously produced by applications. Thus, there is an increasing requirement for a mode of incremental processing of provenance, while the application is still running, to replace batch processing of a complete provenance dataset available only after the application has completed. A process of particular interest is summarising provenance graphs, which has been proposed as an effective way of extracting key features of provenance and storing them in an efficient manner. To that goal, summarisation makes use of provenance types, which, in loose terms, are an encoding of the neighbourhood of nodes.
This paper shows that the process of creating provenance summaries of continuously provided data can benefit from a mode of incremental processing of provenance types. We also introduce the concept of a library of types to reduce the need for storing copies of the same string representations for types multiple times. Further, we show that the computational complexity associated with the task of inferring types is, in most common cases, the best possible: only new nodes have to be processed. We also identify and analyse the exception scenarios. Finally, although our library of types, in theory, can be exponentially large, we present empirical results that show it is very small in practice.

**Keywords:** Provenance Summaries · Provenance Types · Incremental Processing of Provenance.

## 1 Introduction

Let us imagine an application continuously monitoring a system that records all sorts of hospital data. Tracking patient flows, chains of procedures, and staff rotation are examples of such data. The application monitoring this system aims to help identify issues, such as bottlenecks, helping hospital administration to channel resources where needed the most. For that, however, this application needs to process and present the data in a meaningful way, given its potentially

---

very large size and complexity. That is a domain in which provenance can offer analysing tools and techniques, thanks to its widespread use in recording what influenced the generation of data or information. In particular, summarisation of provenance graphs can be of use to extract information from large quantities of data.

More specifically, the World Wide Web Consortium (W3C) has defined provenance as "the record about entities, activities, and people involved in producing a piece of data or thing, which can be used to form assessments about its quality, reliability or trustworthiness" [7]. It has been widely used in different domains, including climate science [10], computational reproducibility [1], and emergency responses [15].

Like in our hospital data example, increasingly, applications are generating provenance information continuously [8], and there is an emerging need for processing incoming data in a similar fashion, i.e., without having to wait for the application to terminate for data to be processed altogether. The need for reprocessing data is costly, as well as the need to store large quantities of provenance information [12].

A process of particular interest is summarising provenance graphs [13]. It has been proposed as a way to extract features of the original graphs and store them efficiently. Summaries can be easily compared to one another, and be used to identify common patterns or find outliers. In order to create provenance summaries, we make use of provenance types, which can be described as an abstraction of the shapes of the neighbourhood of nodes. The idea is that nodes that have similar neighbourhoods will be given the same type and thus will be treated similarly when a summary is created.

Provenance summaries, therefore, could be an important tool to analyse data from a domain such as the hospital scenario described above. For example, we might want to investigate 'how do patient flows compare week after week?', or 'what are the differences and similarities of hospital procedures day after day?'. For these particular examples, we need to compare summaries over dynamic sliding windows over time. The standard summarisation techniques (e.g. [13]), however, are currently not designed to process data incrementally. In this paper, we will propose an efficient way to infer provenance types, which consist of the main ingredients for the creation of provenance summaries. We are interested in addressing the following questions:

**Q1** Considering our goal of inferring provenance types over continuously provided provenance, is it necessary to store all provenance information from the beginning of our application?

**Q2** Is there need to reprocess provenance types of any previously seen provenance expressions with the addition of more provenance data?

**Q3** How can we optimise the need for storage space in case multiple copies of the same provenance types appear as the application runs?

With regards to Questions **Q1** and **Q2**, we show that in the most common cases, there is no need to store all provenance information, nor to reprocess

previously seen nodes. We then identify the exception conditions and provide an algorithm to address these scenarios. Regarding Question **Q3**, we propose the creation of a library of types that enumerate all provenance types encountered from the start and that can be updated, if needed, with the incoming of new data. Finally, we will provide empirical results that support the claim that the use of such a library can indeed optimise storage space.

The structure of this paper is as follows. Section 2 provides the technical background and sets the underlying definitions used in the remaining of the paper. Section 3 introduces the variant of provenance types considered in this paper, as well as presenting an algorithm to infer them. Subsequently, in Section 4 we present the theoretical results with respect to inferring provenance types incrementally at the same time as maintaining libraries of types that we later show, in Section 5, are very limited in size compared the set of all provenance documents from which the library was created. The related work is discussed in Section 6, followed by conclusions and future work (Section 7).

## 2 Background and Definitions

We will consider $G = (V, E, T, L)$ a **provenance graph** in which $G(V)$, or simply $V$, corresponds to the set of nodes of $G$, $E(G)$, or $E$, its set of its edges, and $T$ and $L$ correspond, respectively, to the sets of labels of nodes and edges in $G$. An edge $e \in E$ is a triplet $e = (v, u, l)$, where $v \in V$ is its **starting point**, $u \in V$ is its **ending point**, and $l \in L$, also denoted $lab(e)$, is the edge's **label**. Each node $v \in V$ can have more than one label, and thus $lab(v) \in \mathbb{P}(S)$, where $\mathbb{P}(S)$ denotes the power-set of $S$, i.e., the set of subsets of $S$.[1] Note that provenance graphs are **finite**, **directed**, and **multi-graphs** (as there might exist more than one edge between the same pair of nodes). We do not make the assumption that provenance graphs are acyclic.

In provenance, we typically have $T = \{ag, act, ent, \dots\}$, where 'ag' denotes an agent, 'act' denotes an activity, and 'ent' denotes an entity. There are also application specific labels (e.g. 'hospital:Nurse' to denote a specific label for agents in our hospital example) that also belong to set $T$. On the other hand, typically $L = \{abo, used, waw, wro, \dots\}$, where the edge $(v, u, abo)$, for example, indicates that agent $v$ acted on behalf of agent $u$. We assume sets $T$ and $L$ are totally ordered, which implies that for any two elements $l_1, l_2 \in L$, either $l_1 < l_2$, $l_2 < l_1$, or $l_1 = l_2$ (analogously for $T$). In this paper, we will choose the alphabetical ordering for node and edge labels. Finally, we denote $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{S}, \mathcal{L})$ as a (finite) family of graphs, where $\mathcal{V}, \mathcal{E}, \mathcal{T}$, and $\mathcal{L}$, are the union of the sets of, respectively, nodes, edges, node labels, and edge labels of graphs in $\mathcal{G}$. For a node $v \in \mathcal{V}$, we will refer to the **forward-neighbourhood** of $v$ as $v^+$, where $v^+ = \{u \mid (v, u, l) \in \mathcal{E}\}$. Analogously, the **backward-neighbourhood** of $v$ is denote by $v^- = \{u \mid (u, v, l) \in \mathcal{E}\}$. We say a node $u$ is **distant from** $v$ by $x$ if there

---

[1] Note that when there is the use of application types, a provenance expression may have more than one label, for e.g. $lab(v) = \{ag, Prov:Operator\}$. When $lab(v)$ is a singleton set, we will abuse notation and omit the set-brackets.
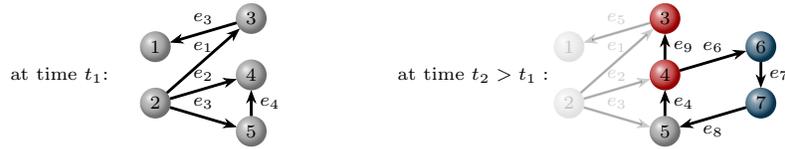
Fig. 1: An abstract graph (left) processed before window slides (right). Formally, $\mu_{\mathcal{V}} = \{1, 2\}$, $\mu_{\mathcal{E}} = \{e_1, e_2, e_3, e_5\}$, $\delta_{\mathcal{V}} = \{6, 7\}$, $\delta_{\mathcal{E}} = \{e_6, e_7, e_8, e_9\}$, $\lambda_{\mathcal{V}} = \{3, 4\}$.

is a sequence of $x$ concatenated edges starting at $v$ and ending at $u$. Finally, we extend the notion of forward-neighbourhood (resp. backward-neighbourhood) for **sets of nodes** $W \subset \mathcal{V}$, i.e., $W^+ = \{u \mid u \in v^+ \text{ for some } v \in W\}$ (resp. $W^- = \{u \mid u \in v^- \text{ for some } v \in W\}$).

When there is the addition of unprocessed (new) data to a database of already processed provenance information, we define $\delta_{\mathcal{V}}$ as the **set of new nodes**, i.e., that have not been processed yet, and $\delta_{\mathcal{E}}$ the **set of new edges**. We now introduce the notation for removal of provenance information. This will be particularly necessary to study provenance types over dynamic sliding windows over time. When there is removal of already processed provenance, we define $\mu_{\mathcal{V}}$ as the **set of removed nodes**, $\mu_{\mathcal{E}}$ as the **set of removed edges**.[2] Finally, we define $\lambda_{\mathcal{V}}$ as the set of **previously processed (and non removed) nodes** $v$ **that either received a new edge starting at** $v$ **or had an edge starting at** $v$ **removed**, i.e., $\lambda_{\mathcal{V}} = \{v \mid v \notin \mu_{\mathcal{V}} \cup \delta_{\mathcal{V}} \text{ and } \exists e \in \mu_{\mathcal{E}} \cup \delta_{\mathcal{E}} \text{ s.t. } e = (v, u, l) \text{ for some } u \text{ and } l\}$. Figure 1 shows an abstract example with $\lambda_{\mathcal{V}}$ highlighted in red and $\delta_{\mathcal{V}}$ in blue.

## 3   Provenance Types

In this section, we will first present a similar definition of provenance types to the one introduced in [13]. Subsequently, we will provide an algorithm that infers provenance types for nodes in a family of graphs. As we will demonstrate, this alternative definition allows us to improve the computational complexity of inferring nodes' types from an exponential to a polynomial function on the number of edges in our family of graphs. Lastly, we define the notion of a library of types, that records all different provenance types seen up to a given point in time, as well as allowing a more efficient way to store types of all nodes in a family of graphs.

As a motivation for the main definition presented in this section, consider the provenance graph in Figure 2 extracted from [3]. It depicts a scenario in which a blogger is analysing the provenance of an online newspaper article, including a chart produced from a government agency dataset. The blogger, the newspaper, the chart generator company, and the government agency are the different sources from which the provenance information was obtained.

Consider nodes *composer*1 and *illustrate*1. We can say that they share some similarity as both represent activities in this provenance graph. Further, we can

---

[2] Note that removing a node automatically removes all edges connected to it
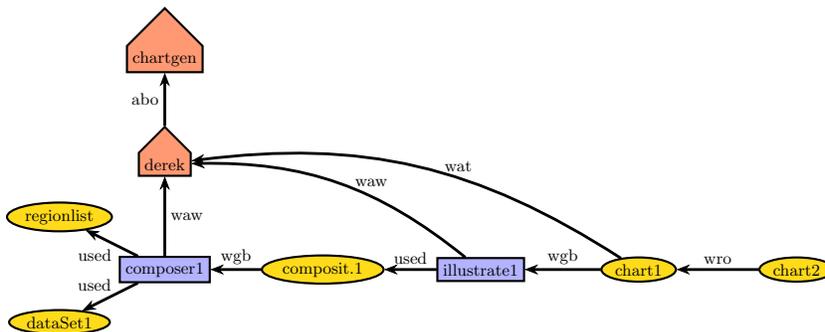
Fig. 2: Example of Provenance Graph from [3]

say that they share even more similarities as they related to entities (via the *used* relation) to some agent (via the *waw* edge label). Note that here we are ignoring the number of times a given pattern appears, since *composer*1 *used* two entity nodes, whereas *illustrate*1, only one. Going one step further, however, these nodes do not present the same 'history': *illustrate*1 used an entity that was generated by some other activity, whereas *composer*1 did not. Later, we will formally define these patterns that we denote the **provenance types** of a node.

We will be looking into a variant of the provenance types defined in [13]. In loose terms, a provenance type of depth $k$ describes the neighbourhood of a node $v$ up to distance $k$ from $v$. Another way of viewing such structures is to think of a subtree rooted at $v$, in which all branches have exactly depth $k$ and no branch is repeated. A more precise (recursive) definition of $k$-types is given as follows.

**Definition 1 (Provenance $k$-type of a node).** *Let $G$ be a provenance graph and $v \in V$ a node. Firstly we define, according to definition in Section 2,*

$$0\text{-}type(v) = lab(v) \tag{1}$$

*Further, we define $k$-type$(v)$ recursively. Consider $v^+$, the forward-neighbourhood of $v$. We will make use of pairs that combine the label of an edge $e$ starting at $v$ with the $(k-1)$-type of the destination node of $e$. We define*

$$k\text{-}type(v) = \{(l, (k\text{-}1)\text{-}type(u)) \mid e = (v, u, l) \in E \text{ and } (k\text{-}1)\text{-}type(u) \neq \emptyset\} \tag{2}$$

*Example 1.* We will now formally present what we discussed at the begining of this section. Consider nodes *composer*1 and *illustrate*1 in Figure 2 and note that they have the same $k$-type: for $k = 0$ and $k = 1$:

$$0\text{-}type(composer1) = 0\text{-}type(illustrate1) = \{\text{act}\}$$
$$1\text{-}type(composer1) = 1\text{-}type(illustrate1) = \{(used, \{\text{ent}\}), (waw, \{\text{ag}\})\}$$

However, they differ with regards to their 2-*type*:

$$2\text{-}type(composer1) = \{(waw, \{(abo, \{\text{ag}\})\})\}$$
$$2\text{-}type(illustrate1) = \{(used, \{(wgb, \{\text{act}\})\}), (waw, \{(abo, \{\text{ag}\})\})\}$$

Algorithm 1.1: Non Incremental Inferring of types$(\mathcal{V}, \mathcal{E}, k)$

```
1  initialise for all v ∈ V and for all i ≤ k
2       i-type(v) ← ∅
3  for v ∈ V
4       0-type(v) ← lab(v)
5  for 1 ≤ i ≤ k
6       for each edge e = (v, u, l) ∈ E such that (i − 1)-type(u) ≠ ∅
7            add (l, (i − 1)-type(u)) to set i-type(v)
8  return i-type(v) for all v ∈ V and 0 ≤ i ≤ k
```

In this case, this difference is a result of the fact that the nodes *regionlist* and *dataSet*1, part of of the out-neighbourhood of *composer*1, have an empty 1-type.

Note that if $v$ is a leaf node in a provenance graph, then $k\text{-}type(v) = \emptyset$ for all $k \geq 1$. Also, note that nodes might coincide with regards to $i\text{-}type$s but differ with regards to $j\text{-}type$s, for some $j < i$. Clearly, they can also differ for some $j > i$. The parameter $k$ may be referred to as the depth of a provenance type.

The choice of looking into edge labels comes from the fact that, in the context of provenance, a sequence of edges culminating at a given node provides a good description of the transformation that occurred to this particular node, or the information that it contained. And thus the $k$-type of a node $v$ provides the set of transformations acting on different provenance elements of the graph leading to the existence of $v$.

Note that in the definition of a $k$-type, repetitions of pairs $(l, (k\text{-}1)\text{-}type(u))$ are discarded. For example, both *composer*1 and *illustrate*1, in Figure 2, have the same 1-*type*, regardless of the fact that *composer*1 is related to two activities (in the same way). The intuition behind that is that the nature of the transformations that generated $v$ are more important than the number of occurrences of a particular transformation.

Algorithm 1.1 infers types according to Definition 1. It takes as input a set of nodes $\mathcal{V}$, a set of edges $\mathcal{E}$, and a parameter $k$. It infers all $i\text{-}type(v)$, for all $v \in \mathcal{V}$ and $0 \leq i \leq k$. This is a non-incremental algorithm, as it is batch-processing all nodes in $\mathcal{V}$. We first initialise our sets $i\text{-}type(v) = \emptyset$ for all nodes. Lines 3-4 infer the 0-*type*s according to Definition 1. The loop in starting in line 5, for each $i$, visits all the edges $(v, u, l)$ that terminate at a node $u$ which has been assigned a non-empty $(i\text{-}1)\text{-}type(u)$ and add the pair $(l, (i\text{-}1)\text{-}type(u))$ to $i\text{-}type(v)$. Note that this is a sequential loop, and thus cannot be run in parallel. This restriction comes from the recursive definition of types. It is, however, possible to run this loop (or the entire algorithm) in parallel for different graphs $G$ of $\mathcal{G}$.

### 3.1  Library of Types

In the previous section, we have shown how to infer $k\text{-}type$s of nodes in a graph or family of graphs. Note, however, that for large sizes of $\mathcal{G}$, we expect a significant

| $\mathcal{T}_0$ | $\mathcal{T}_1$ | $\mathcal{T}_2$ | $\mathcal{T}_3$ |
|---|---|---|---|
| $1_0 \to (\text{ag})$ | $1_1 \to ((used, 3_0), (waw, 1_0))$ | $1_2 \to ((wgb, 1_1))$ | $1_3 \to ((wgb, 3_2))$ |
| $2_0 \to (\text{act})$ | $2_1 \to ((wgb, 2_0))$ | $2_2 \to ((used, 2_1), (waw, 5_1))$ | $2_3 \to ((used, 1_2))$ |
| $3_0 \to (\text{ent})$ | $3_1 \to ((wat, 1_0), (wgb, 2_0))$ | $3_2 \to ((waw, 5_1))$ | $3_3 \to ((wgb, 2_2))$ |
| | $4_1 \to ((wro, 3_0))$ | $4_2 \to ((wro, 3_1))$ | $4_3 \to ((wro, 5_2))$ |
| | $5_1 \to ((abo, 1_0))$ | $5_2 \to ((wat, 5_1), (wgb, 1_1))$ | |

Table 1: Libraries of Types Generated From Graph in Figure 2

recurrence rate with respect to nodes' types. With that in mind, we propose the creation of library that records all types seen up to a given point, as well as a function that maps each node to the library index of its type.

*Example 2.* Consider once more graph $G$ in Figure 2. Table 1 presents four library of types of nodes in $G$. Note that elements from list $\mathcal{T}_k$ will make reference to elements in map $\mathcal{T}_{(k-1)}$.[3] Also, although libraries may not be sorted in any particular way, each entry is. The reason is that we need to be able to uniquely identify each type. In this particular example, the ordering of edge-labels is the alphabetical one. As before, if more than one pair has the same edge label, the second coordinate is compared.

Note for example that none of the 3-*type*s have two branches. That is a result of node *derek* having an empty 2-*type*.

Like in Example 2, libraries make use of the recursive definition of types to further simplify their representation. A key characteristic is that entries in our libraries in Table 1 are not exactly provenance $k$-types, but a compact representation of them. For example, entry $2_2 \to ((used, 2_1), (waw, 5_1))$ corresponds to the $k$-type defined by $\{(used, \{(wgb, \{\text{act}\})\}), (waw, \{(abo, \{\text{ag}\})\})\}$. The definitions of library of types and compact types are somehow intertwined. We now formally define a library of types that would give us the table above.

**Definition 2 (Library of Types and Compact Types).** *Given a family of graphs $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathcal{S}, \mathcal{L})$, and given $k \geq 0$, we create $k + 1$ libraries $\mathcal{T}_i$, $0 \leq i \leq k$, which are one-to-one mappings from integers to compact type expressions encountered in nodes of $\mathcal{G}$. We provide the definition recursively.*

*Regarding the compact representation of 0-types, note that there is nothing to further simplify, so the set of compact 0-types is defined by $\mathcal{C}_0 = \mathbb{P}(\mathcal{S})$, i.e., set of possible values for $lab(v)$.[4] For $i = 0$, $\mathcal{T}_0 : N_0 \to \mathcal{C}_0$ is a mapping from integers ($N_0 \subset \mathbb{N}$) to compact 0-types.*

*For $1 \leq i \leq k$, $\mathcal{T}_i : N_i \to \mathcal{C}_i$, where $\mathcal{C}_i$ is the set of compact type-expressions of depth $i$, i.e. $t_i \in \mathcal{C}_i$ if $t_i$ is an ordered sequence of the form*

$$t_i = \left( \left( l_1, \mathcal{T}_{i-1}^{-1}(t_{i-1}^1) \right), \ldots, \left( l_x, \mathcal{T}_{i-1}^{-1}(t_{i-1}^x) \right) \right) \tag{3}$$

---

[3] For readability, we index elements of map $\mathcal{T}_k$ with $k$

[4] Recall that nodes may have more than one label.

where $l_1 \leq \cdots \leq l_x \in \mathcal{L}$, and $t_{i-1}^1, \ldots, t_{i-1}^x \in \mathcal{C}_{i-1}$. Also, $\mathcal{T}^{-1}$ denotes the inverse mapping that takes a compact type to its index. If $l_j = l_{j'}$ for two different pairs in $t_i$, their order is defined by their respective library index.

*Remark 1.* Note that in order to transform a compact *k-type* into its full form, a serialiser would need to make use of libraries $\mathcal{T}_0, \ldots, \mathcal{T}_{k-1}$. The total number of calls to libraries is bounded by $O(\Delta^k)$, where $\Delta$ is the maximum forward-degree of nodes in $\mathcal{V}$.

Note that, although a library is created from actual data (as opposed to including all possible theoretical *k-types*), its definition does not include any mapping from nodes to their types. It is useful, therefore, to create and maintain such a mapping from nodes to the indexes of their types in $\mathcal{T}_i$.

**Definition 3.** *Let $\mathcal{G}$ be a family of graphs and $\mathcal{T}_k$ a library of types that cover types in $\mathcal{G}$. For each $v \in \mathcal{V}$, $t_k(v) \in \mathcal{C}_k$ the compact representation of its k-type. We define $\theta_k : \mathcal{V} \to \mathbb{N}$ the function that takes a node $v$ and outputs the library index associated with its k-type, i.e., for all $v \in \mathcal{V}$,*

$$\mathcal{T}_k(\theta_k(v)) = t_k(v) \tag{4}$$

*When considering node increments $\delta_{\mathcal{V}}$, we denote $\theta_{\mathcal{V},k}$ as the function associated with the set of previously processed nodes only, whereas $\theta_{\delta_{\mathcal{V}},k}$ is the function related to new nodes only.*

*Example 3.* Consider once more the provenance graph in Figure 2 and the library in Example 2. We have $t_2(illustrate1) = ((used, 2_1), (waw, 5_1))$, we have $\theta_2(illustrate1) = 2_2$, and, finally,

$$\mathcal{T}_2(\theta_2(illustrate1)) = \mathcal{T}_2(2_2) = ((used, 2_1), (waw, 5_1)) = t_2(illustrate1) \tag{5}$$

The definition of a library partially addresses Question **Q3**. In the the following section we will show how to construct such a library, and in Section 4 how to maintain it in the context of an incremental mode of processing provenance types.

### 3.2   Creating a Provenance Types Library

In this section we will present an efficient algorithm for creating libraries of types as defined Section 3.1, as well as maintaining functions $\theta_k$ that map nodes to the library index associated with their (compact) *k-type*. We use an auxiliary Algorithm 1.2 receiving a set of nodes $\mathcal{V}$ and edges $\mathcal{E}$ as input, as well as the depth $i$. This algorithm first infers the compact representations of $v$, $t_i(v)$, for $v \in \mathcal{V}$, and then check if this type is already part of the current library $\mathcal{T}_i$. If yes, mapping $\theta_i$ is updated for $v$. If not, a new entry is added to the library and $\theta_i$ is also updated accordingly.

   With that, we are able to present Algorithm 1.3, that creates all libraries of types up to $k$ from $\mathcal{V}$ and $\mathcal{E}$. It calls our auxiliary Algorithm 1.2 $k + 1$ times, creating overall libraries $\mathcal{T}_0, \ldots, \mathcal{T}_k$ and functions that assign a node's (compact) type-index $\theta_0, \ldots, \theta_k$.

Algorithm 1.2:  TYPE LIBRARY $(\mathcal{V}, \mathcal{E}, i, \theta_{i-1}, \mathcal{T}_i, \theta_i)$

```
1   if  i == 0
2        for  all  v ∈ V
3             t₀(v) ← lab(v)
4   else
5        for  each  edge  e = (v, u, l) ∈ E  such  that  θᵢ₋₁(u) ≠ 0
6             add  (l, θᵢ₋₁(u))  to  ordered  sequence  tᵢ(v)
7   for  all  v ∈ V
8        if  tᵢ(v) == Tᵢ(x)  for  some  x
9             add  (v → x)  to  θᵢ
10       else
11            add  (y → tᵢ(v))  to  Tᵢ  for  new  index  y
12            add  (v → y)  to  θᵢ
13  return  Tᵢ, θᵢ
```

Algorithm 1.3: CREATING LIBRARIES AND MAPPINGS$(\mathcal{V}, \mathcal{E}, k)$

```
1   initialise  for  all  v ∈ V  and  for  all  i ≤ k
2        tᵢ(v) ← ∅
3        Tᵢ ← ∅
4        θᵢ ← ∅
5   for  0 ≤ i ≤ k
6        TYPE LIBRARY  (V, E, i, θᵢ₋₁, Tᵢ, θᵢ)
7   return  T₀, ..., Tₖ, θ₀, ..., θₖ
```

**Running Time of Algorithm 1.3** We are now showing that we need $O(k\,|\mathcal{E}|)$ operations (amortised time) to create all $k+1$ libraries and functions $\theta_0, \ldots, \theta_k$, i.e., to infer all (compact) $i$-types for all $v \in \mathcal{V}$, and all $0 \leq i \leq k$. This proof is similar to the one for Weisfeiler-Lehman graph kernels [16]. Note that we can infer the $i$-types on each graph in parallel.

The first iteration ($i = 0$) of auxiliary Algorithm 1.2 can be run in $O(|\mathcal{V}|)$ amortised time. Lines 2-3 take $O(|\mathcal{V}|)$. In loop starting at line 7, for each node, we can check whether its type has been recorded in the library (line 8) in amortised constant time using a suitable hash map. Lines 9-12 run in constant time, so overall complexity is amortised $(\mathcal{V})$ as we enter the loop $|\mathcal{V}|$ times. The other $k$ iterations ($1 \leq i \leq k$) of auxiliary Algorithm 1.2 takes (amortised) $O(|\mathcal{E}|)$ time. It visits each edge at most once (line 5). To order each set $t_i(v)$ for all $v \in \mathcal{V}$, we execute bucket sorting twice (i.e., a version of radix sort) in all lists at the same time, recording from which vertex each pair $(l, \theta_{i-1}(u))$ came from. We perform the first bucket sort in which buckets represent values $\theta_{i-1}(u)$ for some $u \in \mathcal{V}$ (takes $O(|\mathcal{E}|)$). Note that the size of each library, and therefore the image

of $\theta_{i-1}$ is bounded by the number of nodes $|\mathcal{V}|$. The second iteration of this sort orders the partially ordered set into buckets representing the edge-labels $l \in \mathcal{L}$ (again takes $O(|\mathcal{E}|)$). Recall that we record the vertex from which each pair comes from. Then, as in its first iteration, looking up whether a compact type is already part of the library (line 8) takes amortised constant time. Then, iterations for $1 \leq i \leq k$, Algorithm 1.2 runs on average in $O(|\mathcal{E}|)$.

Finally, assuming $\mathcal{V} = O(|\mathcal{E}|)$, Algorithm 1.3 can then be run in amortised $O(k\,|\mathcal{E}|)$, because auxiliary Algorithm 1.2 is called $k$ times for $i > 0$.

## 4   Incremental Inference of Provenance Types

In this section, we look into the main motivation of this paper: how to process provenance types incrementally. We have proposed three research avenues: The need for storing all provenance data from the beginning (**Q1**), the need for reprocessing previously seen nodes (**Q2**), and how to efficiently deal with the possibly multiple occurrence of provenance types among all nodes (**Q3**).

At this point, we need to clarify what we define as increments of provenance data. We can have either a stream of provenance graphs, or a stream of nodes and edges of a provenance graph. We will show that the main difference, however, is whether previously seen nodes have any deleted edges or added edges that start at $v$ (recall definition of $\lambda_{\mathcal{V}}$ from Section 2). We thus propose the study of the following cases:

**Monotonically Increasing Stream (case $\lambda_{\mathcal{V}} = \emptyset$)):** As there is new provenance being received, there are no new edges starting at a previously processed node. There is also no deletion of previously seen edges.

**Non-monotonically Increasing Stream (case $\lambda_{\mathcal{V}} \neq \emptyset$):** As new provenance is received, there is at least one new edge starting at a previously processed node, or at least one edge removed that started at a previously processed node.

In the next sections, we will show that the answer to Question **Q1** is negative for monotonically increasing streams. For, non-monotonically increasing ones, however, we might need to revisit all previously processed provenance. Similarly, we will show that Question **Q2** is negative for monotonically increasing streams, as there no need to reprocess previously seen nodes. However, when $\lambda_{\mathcal{V}} \neq \emptyset$, there might be the need to reprocess nodes.

### 4.1   Monotonically Increasing Streams

In this section, we are studying the cases in which we have monotonically increasing streams of provenance data. This includes the introduction of entirely new provenance graphs, but also the addition of new nodes to existing graphs as long as $\lambda_{\mathcal{V}} = \emptyset$. This definition is broad and includes situations in which a single node (or edge) is added, or situations in which entire new graphs together with nodes in previously seen graphs are added. We first show that, when $\lambda_{\mathcal{V}} = \emptyset$, there is no need to reprocess previously seen nodes.

Algorithm 1.4: INCREMENTAL INFERENCE UNDER MONOT. CASE$(\mathcal{V}, \mathcal{E}, \delta_{\mathcal{V}}, \delta_{\mathcal{E}}, k)$

```
1  initialise  for  all  v ∈ δ_V  and  for  all  i ≤ k
2       t_i(v) ← ∅
3  for  0 ≤ i ≤ k
4       define  T_i = T_i(V)  and  θ_i = θ_(V,i)
5       TYPE LIBRARY (δ_V, δ_E, i, θ_{i-1}, T_i, θ_i)
6  return  T_0, …, T_k, θ_0, …, θ_k
```

**Lemma 1 (Addressing Q2).**  *No previously processed node will have their $k$-types altered in a monotonically increasing increment. Thus, they do not need to be reprocessed.*

*Proof.* We prove by induction on $k$. For $k = 0$, the result follows since all nodes' provenance types remain the same. Assume the result holds for $k = i$, we shall prove it also holds for $i + 1$. From the recursiveness of the definition of $k$-types (Definition 1), for any $v$, $(i + 1)$-$type(v)$ rely only on (1) the label on edges starting at $v$, and (2) the $i$-$type(u)$, for $u \in v^{+}$. Since there is no new edge starting on $v$, all $u \in v^{+}$ have all previously processed. That fact, together with the induction hypothesis ($i$-$type(u)$ unchanged) we conclude that $(i+1)$-$type(u)$ will not been altered. The result follows by induction.  □

Lemma 1 suggests that not much information needs to be stored to deal with incremental processing of monotonically increasing streams. The following lemma formalises this idea.

**Lemma 2 (Addressing Q1).**  *In order to infer $i$-types, $0 \leq i \leq k$, of newly added nodes (equivalent to constructing $\theta_{\delta_{\mathcal{V}},i}$) there is no need to store previously seen edges, but only the set of maps $\theta_{\mathcal{V},i}$, $0 \leq i \leq k$.*

*Proof.* From Lemma 1, no previously processed node needs reprocessing, therefore no edge starting at them will be visited. Therefore there is no need to store previously seen edges.  □

**Algorithm for Incremental Inference of Types and Libraries** Given the results of the lemmas above, the algorithm for incrementally referring types of monotonically increasing streams is simple if we have maintained libraries of types (and mapping functions) from the already processed data.

Consider Algorithm 1.4. It takes as input the set of already processed nodes $\mathcal{V}$ and edges $\mathcal{E}$, as well as the new ones ($\delta_{\mathcal{V}}$ and $\delta_{\mathcal{E}}$). It also takes depth $k$ as input. We consider $\mathcal{T}_i(\mathcal{V})$, library over nodes in $\mathcal{V}$, and $\theta_{(\mathcal{V},i)}$, mapping from nodes of $\mathcal{V}$ to library indexes, as global variables, for all $0 \leq i \leq k$. The algorithm calls auxiliary Algorithm 1.2 only with the new sets of nodes and edges as inputs. Note that the algorithm deals with the introduction of new node labels by updating $\mathcal{T}_0$ and $\theta_0$ accordingly.

**Complexity of Algorithm 1.4** As expected from the lemmas above, the complexity of inferring types of incoming data (and updating libraries and mappings, when needed) is given by $O(k\,|\delta_{\mathcal{E}}|)$ amortised time.

*Remark 2.* Note that the only entries of $\theta_i$ that will be accessed in Algorithm 1.4 are the ones for nodes in set $\delta_{\mathcal{V}}^+$. Therefore, further storage optimisations can be achieved when there is previous knowledge with regards to $\delta_{\mathcal{V}}$.

### 4.2   Non-monotonically Increasing Streams

In this section, we analyse the case of non-monotonically increasing increments, i.e., $\lambda_{\mathcal{V}} \neq \emptyset$. A particular example of such a scenario is the consideration of dynamic sliding windows of time, in which 'old' provenance is deleted as new provenance is added. We will then show that the answer for Question **Q2**, in this case, is positive: we might need to revisit (in the worst case, all) provenance expressions and possibly update their types. This implies that, because previously seen nodes may need reprocessing, it is required that all provenance data, including edges, is kept accessible (Question **Q1**).

Under non-monotonically increasing streams, Algorithm 1.5 infers the provenance types of new nodes as well as reprocess previously seen provenance expressions that may have had their types altered. In line 4, it infers all 0-*type*s of newly added nodes. Line 5 flags such nodes since all nodes in their backward-neighbourhood will need their 1-*type* to be (re)processed. The loop in line 6 start by marking all nodes in the backward-neighbourhood of previously marked nodes, making sure to add all provenance expressions from set $\lambda_{\mathcal{V}}$. That last part is needed because such nodes need reprocessing regardless of the types of their forward-neighbours. Notation in line 11 refers to whether the updated value $\theta_i'(v)$ has changed or not compared to its value before the auxiliary function in line 9. It is has not changed, then it will not contribute for a change in $(i+1)$-types of nodes in its backwards neighbourhood. Although nothing prevents the same node $v$ to be added back to $M$ in line 7 of the next iteration of the loop.

**Correctness of Algorithm 1.5** We show that the algorithm will correctly reprocess the $k$-types of all nodes that were directly or indirectly affected by the addition or removal of edges. The bottom line of the algorithm is to take in account that not only nodes that were immediately affected will need to have their $k$-types updated, but that the effect may cascade down along the graph to a distance up to $k$.

We first consider $v \in \lambda_{\mathcal{V}}$ . Even though we may have no change in, for example, 1-*type*(v), that does not imply that 2-*type*(v) will be also unchanged (the added or removed edge might connect or have connected $v$ to different branches), and thus we need to reprocess this node for all $0 \leq i \leq k$. Line 7 guarantees that by adding $\lambda_{\mathcal{V}}$ to all $M_i$. Now consider all other nodes, including the ones previously processed. We are going to show, by induction, that Algorithm 1.5 correctly identifies the need for reprocessing. For $i=0$, all new nodes (and only those) need 0-*type*(v) inferred (line 4). Note that, for $i>1$, a node $v$'s $k$-*type*(v)

Algorithm 1.5: INCREMENTAL INFERENCE UNDER NON-MONOT. CASE$(\mathcal{V}, \mathcal{E}, \delta_{\mathcal{V}}, \delta_{\mathcal{E}}, k)$

```
1   initialise  for  all  v ∈ δ_V  and  for  all  0 ≤ i ≤ k
2         t_i(v) ← ∅
3   define  T_i = T_i(V)  and  θ_i = θ_(V,i) ,  for  0 ≤ i ≤ k
4   TYPE LIBRARY  (δ_V, δ_E, 0, ∅, T_0, θ_0)
5   define  set  M_0 = δ_V                          [these are marked nodes]
6   for  i = 1  until  k
7         define  M_i = M_{i-1}^- ∪ λ_V
8         define  L_i  set  of  edges  starting  at  nodes  in  M_i
9         TYPE LIBRARY  (M_i, L_i, i, θ_{i-1}, T_i, θ_i)
10        for  v ∈ M_i            [compare types for v before and after line 9]
11              if  (θ'_i(v) == θ_i(v))
12                    remove  v  from  M_i
13  return  T_0, …, T_k, θ_0, …, θ_k
```

needs to be reprocessed if and only if at least one of its forward-neighbours $u \in v^+$ had their $(k-1)$-$type(u)$ updated, i.e., reprocessed and changed. For $i > 1$, assume that all nodes that required were reprocessed. We show that this is also true for $i+1$. Indeed, no node that had their $i$-$type$ modified was removed from $M_i$ (lines 11-12), and thus, $v$ will be in set $M_{i+1}$ if and only if at least one of its neighbours was not removed from $M_i$ (or, of course, if $v \in \lambda_{\mathcal{V}}$).

**Complexity of Algorithm 1.5** In the worst case, this algorithm may need to reprocess all nodes that have been previously seen. Line 4 takes $O(|\delta_{\mathcal{V}}|)$ operations on average. Let $L_i$ be the set of edges starting at nodes in $M_i$, i.e, $|L_i| = \sum_{v \in M_i} \deg^+(v)$. Then, each iteration of the loop starting in line 6 takes $O(|L_i|)$, similarly to algorithm 1.3. Finally, denoting $|L| = \sum_{i=1}^k |L_i|$, we conclude that Algorithm 1.5's running time complexity is $O(|\delta_{\mathcal{V}}| + |L|)$ amortised time. Note that visiting all nodes in $M_i$ (line 10) does not increase complexity as $|L_i| > |M_i|$.

In Section 4, we investigated the different modes of incremental processing of provenance types, showing that, in the context of monotonically increasing streams, provenance types of new nodes can be inferred fast and without the need to reprocess previously seen nodes or access all past provenance data. In non-monotonically increasing streams, however, reprocessing of old nodes might be necessary, as well as access to edges of previously seen graphs.

## 5   Empirical Evaluation

In this section, we show that the size of a library is indeed much smaller than $|\mathcal{V}|$. We present empirical results of the processing of more than $36,000$ graphs, that

| Datasets | Number of graphs | Nodes with non-empty *i-type* for $i =$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 |
| CM-buidings | 5175 | 94k | 89k | 71k | 58k | 51k | 38k |
| CM-routes | 4997 | 105k | 100k | 90k | 70k | 44k | 40k |
| CM-Routeset: | 4710 | 101k | 97k | 76k | 49k | 45k | 37k |
| MIMIC: | 21892 | 1208k | 865k | 843k | 821k | 788k | 753k |
| PG: | 80 | 18k | 14k | 14k | 14k | 14k | 14k |
| Total | 36854 | 1526k | 1165k | 1094k | 1012k | 943k | 883k |

Table 2: Number of graphs in each dataset, as well as the number of nodes with a non-empty *i-type* for each $i = 0, \ldots, 5$

show that the number of distinct *i*-types, for each $i \leq 5$, is approximately $5,000$. The challenge of choosing $i$ to best analyse and compare summary graphs left for future work. We analysed datasets from 3 different domains. CollabMap [14] (CM) is a database of provenance graphs for evacuation planning generated in a crowd-sourcing platform. We separate CollabMap graphs into buildings, routes, and route sets. MIMIC [9] is a database of information of patients in critical hospital care. Finally, our last dataset of graphs describes actions of players on Pokemón Go (PG) simulations.

We first provide an overview of the size of our datasets in Table 2. We have a total of $36,854$ different provenance graphs, with an average of $41.4$ nodes each. Note that, although all nodes have a non-empty 0-*type* associated with them, the same is not valid for deeper types. For example, leaf nodes have an empty 1-*type* (and thus an empty *i-type* for $i \geq 1$). The quantity of nodes with a non-empty *i-type* for each $i = 0, \ldots, 5$ is given rounded up to the nearest thousand. Observe, for example, that of the nodes in graphs of the MIMIC dataset, approximately $343,000$ ($\approx 28\%$) are leaf nodes, as they have an empty 1-*type*.

Table 3 shows the sizes of libraries of types $\mathcal{T}_i$, for $0 \leq i \leq 5$, for each dataset separately, as well as them combined, using one common library of types. In these experiments, we ignore primitive labels of provenance expressions, and thus $\mathcal{T}_0 = \{\text{ag}, \text{act}, \text{ent}\}$ for the combined dataset, although there are no agents

| Datasets | $|\mathcal{T}_0|$ | $|\mathcal{T}_1|$ | $|\mathcal{T}_2|$ | $|\mathcal{T}_3|$ | $|\mathcal{T}_4|$ | $|\mathcal{T}_5|$ |
|---|---|---|---|---|---|---|
| CM-buidings | 2 | 3 | 3 | 5 | 14 | 102 |
| CM-routes | 2 | 4 | 10 | 35 | 155 | 577 |
| CM-Routeset: | 2 | 4 | 8 | 23 | 58 | 262 |
| MIMIC: | 3 | 5 | 11 | 39 | 405 | 4328 |
| PG: | 2 | 3 | 5 | 10 | 23 | 54 |
| sizes of common libraries for all domains | 3 | 9 | 23 | 84 | 601 | 5197 |
| sum of individual library sizes | 11 | 19 | 37 | 112 | 655 | 5323 |

Table 3: Number of entries in libraries of types for different datasets, as well as for a common one.

in neither CollabMap nor PokemonGo datasets. The 6th row gives us the size of libraries generated from all datasets, which implies that if the same type appears in more than one dataset, it will be counted only once in our joint library. In contrast, the last row gives the sum of sizes of the 5 individual libraries. We can see that, especially for low-depth types, there is a significant overlap of types across datasets.

This experiment shows that maintaining a library of types is an efficient way of avoiding the need to store multiple copies of the same types. As the depth increases, so does the number of different patterns to be stored and, although this is expected due to increased complexity of neighbourhoods of greater radius, the final library sizes continue to be much more compact than the size of original data. Note also that there is a significant overlap of types between different domains, which indicates that keeping a library of types across different domains can contribute to further storage optimisation. The full library of types for each of the datasets can be found at https://openprovenance.org/typelibrary/.

## 6   Related Work

To the best of our knowledge, this is the first work that proposes the study of incremental processing of provenance types, which contributes to the study of incremental provenance sumaries. There is, however, literature on dynamic analysis of provenance, such as [8], which proposes the analysis of dynamic sliding windows to identify behaviour anomalies. Also, MaMaDroid [11] builds a Markov chain over continuously provided data for malware detection. Provenace data streams without the focus on data incremental processes for contructing summaries were also studied in [4] and [18].

Beyond the domain of provenance, there is also work on incremental inferences of summaries. In particular, a DataGuide [5] provides summaries of databases, in both incremental and non-incremental modes, although their model may take require exponential time and space complexity on the number of nodes and edges of input graphs. Also, [2] studies pattern matching in incremental scenarios. The main difference with our work is that they focus on finding a specific pattern within a large (and changing) graph, rather than inferring each node's type. On a similar domain, [6] proposes the incremental processing of a summary graph in which nodes are associated to a hash value, although they do not consider edge labels in their work. Song and Ge [17], on the other hand, do consider edge labels and construct graph sketches over sliding windows. The main difference compared to provenance types is that they do not consider patterns within graphs, but only information encoded in edges such as their label and endpoints. In the context of machine learning and Weisfeiler-Lehman graph kernels, [20] provides graph classification with continuously provided data. Unlike in our work, they discard nodes with less discriminatory power to facilitate the classification process.

The concept of provenance types was introduced in [13] as the main step to construct summary graphs, which are, in turn, a way to extract the essence of

provenance graphs. Based on their definition of provenance types, however, the algorithm presented in [13] to infer types has an exponential time complexity in function of the parameter $k$. This drawback comes from the fact that all walks of lengh $k$ from a given node may need to be inspected. In this paper, we offer similar but recursive definition of provenance types which allows us infer them in polynomial time. This recursive definition is similar to the one used in the context of Weisfeiler-Lehman Graph Kernels [16], with the differences that our sets $\theta_k(v)$ consist of pairs instead of single values, and that we discard repetitions in such sets. Another difference, also explored in [19], is that we work with graphs with labelled edges, and therefore they are taken in account when processing nodes' types.

## 7    Conclusions and Future Work

In this paper, we studied incremental processing of provenance in the context of summarisation of provenance graphs. Our contribution focused on the inference of provenance types, as we leave incremental computation of summary graphs for future work.

First, we suggested an alternative definition of provenance types (Definition 1), which, in loose terms, consist of an abstraction of the forward-neighbourhood of nodes in a provenance graph. This definition allows provenance types to be inferred in polynomial time taking into account the size of the input data (Algorithm 1.3). To avoid storing the same provenance types multiple times, we suggest the creation of a library of types for each parameter $k$ (Definition 2). Such libraries record all seen (compact) provenance types. We also define, for each parameter $k$, a function that maps each node to the library entry associated with the node's compact type (Definition 3).

In order to study the different modes of data increments, we considered two broad scenarios: when previously processed edges are removed from - or when new edges start at - previously seen nodes (non-monotonically increasing streams), and when that is not the case (monotonically increasing streams). For the former case, we provide Algorithm 1.5 that reprocess nodes when needed. The latter scenario, on the other hand, was shown to allow the processing of incoming data without the need to reprocessing previous provenance information (Question **Q2**). In that case, there is also no need to keep stored previously seen provenance relations (Question **Q1**).

Subsequently, we presented an analysis of more than 36.000 provenance files and showed that the size of libraries of types is small compared to the size of our datasets (Question **Q3**). We also give the number of nodes in each dataset that has a non-empty type for each level from 0 to 5.

As future work, it would be useful to develop an empirical analysis of the time it takes to run Algorithm 1.5 in practice, i.e., to understand what proportion of the graph needs to be reprocessed. Another important further step is to extend the incremental inference to constructing summary graphs over continuously provided provenance.

# References

1. Chirigati, F., Shasha, D., Freire, J.: Reprozip: Using provenance to support computational reproducibility. In: Presented as part of the 5th USENIX Workshop on the Theory and Practice of Provenance (2013)
2. Fan, W., Wang, X., Wu, Y.: Incremental graph pattern matching. ACM Trans. Database Syst. **38**(3) (Sep 2013). https://doi.org/10.1145/2489791, https://doi.org/10.1145/2489791
3. Gil, Y., Miles, S., Belhajjame, K., Deus, H., Garijo, D., Klyne, G., Missier, P., Soiland-Reyes, S., Zednik, S.: Prov model primer. W3C Working Group Note (2013)
4. Glavic, B., Sheykh Esmaili, K., Fischer, P.M., Tatbul, N.: Ariadne: Managing fine-grained provenance on data streams. In: Proceedings of the 7th ACM International Conference on Distributed Event-Based Systems. p. 39–50. DEBS '13, Association for Computing Machinery, New York, NY, USA (2013). https://doi.org/10.1145/2488222.2488256, https://doi.org/10.1145/2488222.2488256
5. Goldman, R., Widom, J.: Dataguides: Enabling query formulation and optimization in semistructured databases. In: 23rd International Conference on Very Large Data Bases (VLDB 1997) (1997), http://ilpubs.stanford.edu:8090/232/
6. Gou, X., Zou, L., Zhao, C., Yang, T.: Fast and accurate graph stream summarization. In: 2019 IEEE 35th International Conference on Data Engineering (ICDE). pp. 1118–1129. IEEE (2019)
7. Groth, P., Moreau (eds.), L.: PROV-Overview. An Overview of the PROV Family of Documents. W3C Working Group Note NOTE-prov-overview-20130430, World Wide Web Consortium (Apr 2013), http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/
8. Han, X., Pasquier, T., Ranjan, T., Goldstein, M., Seltzer, M.: Frappuccino: fault-detection through runtime analysis of provenance. In: 9th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 17) (2017)
9. Johnson, A.E., Pollard, T.J., Shen, L., Li-wei, H.L., Feng, M., Ghassemi, M., Moody, B., Szolovits, P., Celi, L.A., Mark, R.G.: Mimic-iii, a freely accessible critical care database. Scientific data **3**, 160035 (2016)
10. Ma, X., Fox, P., Tilmes, C., Jacobs, K., Waple, A.: Capturing provenance of global change information. Nature Climate Change **4**, 409–413 (01 2014). https://doi.org/10.1038/nclimate2141
11. Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G.J., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models. CoRR **abs/1612.04433** (2016), http://arxiv.org/abs/1612.04433
12. Moreau, L.: The foundations for provenance on the web. Foundations and Trends in Web Science **2**(2–3), 99–241 (Nov 2010). https://doi.org/http://dx.doi.org/10.1561/1800000010
13. Moreau, L.: Aggregation by provenance types: A technique for summarising provenance graphs. In: Graphs as Models 2015 (An ETAPS'15 workshop). pp. 129–144. Electronic Proceedings in Theoretical Computer Science, London, UK (Apr 2015). https://doi.org/10.4204/EPTCS.181.9
14. Ramchurn, S., Huynh, T.D., Venanzi, M., Shi, B.: Collabmap: Crowdsourcing maps for emergency planning. In: Proceedings of the 3rd Annual ACM Web Science Conference, WebSci 2013. pp. 326–335 (05 2013). https://doi.org/10.1145/2464464.2464508

15. Ramchurn, S.D., Huynh, T.D., Wu, F., Ikuno, Y., Flann, J., Moreau, L., Fischer, J.E., Jiang, W., Rodden, T., Simpson, E., et al.: A disaster response system based on human-agent collectives. Journal of Artificial Intelligence Research **57**, 661–708 (2016)
16. Shervashidze, N., Schweitzer, P., Leeuwen, E.J.v., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-lehman graph kernels. Journal of Machine Learning Research **12**(Sep), 2539–2561 (2011)
17. Song, C., Ge, T.: Labeled graph sketches. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE). pp. 1312–1315. IEEE (2018)
18. Vijayakumar, N.N., Plale, B.: Towards low overhead provenance tracking in near real-time stream filtering. In: Moreau, L., Foster, I. (eds.) Provenance and Annotation of Data. pp. 46–54. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)
19. de Vries, G.K.: A fast approximation of the weisfeiler-lehman graph kernel for rdf data. In: JEC on ML and Knowledge Discovery in Databases. pp. 606–621. Springer (2013)
20. Yao, Y., Holder, L.: Scalable svm-based classification in dynamic graphs. In: 2014 IEEE International Conference on Data Mining. pp. 650–659 (Dec 2014). https://doi.org/10.1109/ICDM.2014.69