Versioning Version Trees: The provenance of actions that affect multiple versions

David Koop

University of Massachusetts Dartmouth, Dartmouth MA, USA,
dkoop@umassd.edu

Abstract. Change-based provenance captures how an entity is constructed; it can be used not only as a record of the steps taken but also as a guide during the development of derivative or new analyses. This provenance is captured as a version tree which stores a set of related entities and the exact changes made in deriving one from another. Version trees are generally viewed as monotonic–new nodes may be added but none are modified or deleted. However, there are a number of operations (e.g., upgrades) where this constraint leads to inefficient and unintuitive new versions. To address this, we propose a version tree without monotonicity where nodes may be modified and new actions inserted. We also propose to track the provenance of these tree changes to ensure that past version trees are not lost. This provenance is change-based; it links versions of version trees by the actions which transform the trees. Thus, we continue to track every change that impacts the evolution of an entity, but the actions are split between direct edits and changes to the version tree that affect multiple entity definitions. We show how this provenance leads to more intuitive and efficient operations on workflows and how this hybrid provenance may be understood.

Keywords: provenance, version tree, workflows

1 Introduction

As the number of documents, source trees, and images continues to grow, it is important to understand when and how individual items are related to each other. If a digital entity has changed over time, there are different versions of it, and the relationships between these versions help organize the information they contain. Version graphs encode derivation histories of the entities and may also relate different objects that were derived from a similar source. Usually, these graphs are used to archive past versions, often encoded for efficient storage. However, past versions may also be re-examined and integrated with current and future versions. In most cases, *one* new version is generated when an entity is modified or merged with another version. For example, in versioned source code, a commit defines a single new version with the updates to the files.

A version graph most basically defines when one version is derived from another, but this information need not contain *how* the versions are related.

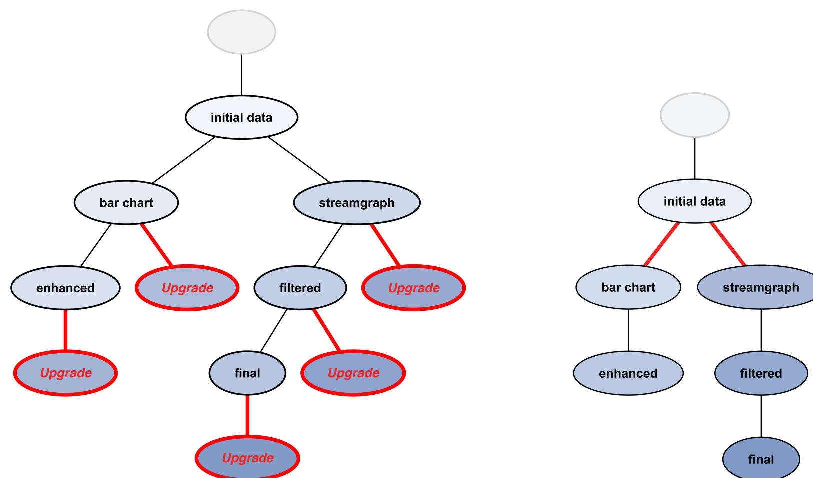

Fig. 1: It is often necessary to update or upgrade collections of related documents or workflows. Even though these changes can be automated, each version must be modified often with the same changes (left, new nodes and edges outlined in red). If we can instead edit the version tree, we can update past changes to reflect the updates (right, modified edges highlighted in red).

Change-based schemes, however, store the changes that transform one version to another. With this richer history, we can not only understand the difference but also directly edit the change to update the derivative version. When that edit impacts a version that itself has many derivative versions, *all* of those versions are also affected. Such edits can correct past errors, update an old approach, or introduce additional functionality to multiple versions. For example, in a source code version tree, we might replace an algorithm added early in development with a more efficient version. Instead of applying these changes to each branch, we modify the tree itself so branches inherit the update.

While this rewriting of history brings the potential for efficient and intuitive edits over a collection of versions, it also presents the problem of how to preserve the old collection. This is particularly problematic when the past versions are tied to other provenance information. For example, a collection of workflows may need to be upgraded, but if the provenance of past runs is associated with the old versions, we do not want to lose the definitions of them. To address this problem, we suggest *versioning version trees* by storing the evolution provenance of the version tree. Figure 1 shows how this can declutter workflow upgrades. Any version can be obtained by first materializing the version tree and then materializing the version in that tree. To simplify navigation, we propose *links* between a version in one version tree and its "derivative" version in another version tree. Because simple additions to a current version tree can be interleaved with transformations of version trees, the provenance of any single version becomes more involved.

We introduce meta version trees (MVTs), define intuitive operations enabled by the new trees, test their efficiency using synthetic version trees, and discuss the implications in understanding the provenance of entities created and modified in this framework.

2 Preliminaries

2.1 Versioning Background

Keeping track of different versions of documents, code, and workflows is commonplace, but the strategies for doing so have evolved over time. With a central authority for changes, it is possible to fully order the versions according to the time they were submitted. However, distributed version control systems allow changes to be evolve independently from a central repository. Branching allows users to develop new features in a branch, and then merge these changes back into a "master" branch. Version graphs track versions of evolving objects and any merges [5]; merging operations require the history to be represented as a directed acyclic graph. Because we will be leveraging change-based versioning where explicit user changes specify the derivation of a version, we focus on version trees which do not contain merges.

A *version tree* is a tree where each node represents a version of a particular entity (S) and an edge from one version to another indicates that the child version was derived from the parent version. Recall that a tree $T = (V, E)$ is a directed acyclic graph where each node $v \in V$ has at most one edge ending at v. In a version tree, each node v has an associated version S_v. If, for a given v, there exists u such that $(u, v) \in E$, then S_u is the *parent version* of S_v and S_v is a *child version* of S_u. Note that in general, there is no specific restriction on how two versions must relate to each other. A version tree may represent a human-curated understanding of derivations or enforce specific derivation requirements that permits automated construction.

2.2 Change-based Version Trees

A version tree indicates relationships between versions, but these relationships can be further defined as transformations from one version to another. These functions define the *changes* and may be inferred or prescribed. For example, version control software like svn [17] computes differences between the current and previous version of a file, inferring the lines added and deleted. Thus, a search-and-replace of a single word would be recorded by most version control software as a sequence of line modifications. VisTrails, on the other hand, stores the exact actions it makes when a user changes a workflow [8]. For example, when a module is added to a workflow, the exact detail is recorded. Such prescriptive changes allow greater understanding of the process involved in creating a workflow.

Formally, a change-based version tree $T = (V, E, S_0)$ is a version tree where S_0 is a default version, and for each edge $(u, v) \in E$, there exists an associated function f such that $f(S_u) = S_v$. In other words, f tells us how S_v can be derived from S_u. Associating the function with the edge instead of the node will

provide more intuitive operations in manipulating change-based version trees. The root of the change-based tree often corresponds to an empty state, e.g., an empty repository or an empty workflow. In a general version tree, we might need to store the associated version for each node, but a change-based version tree requires storing *only* the edge functions and the default state S_0. Let $P(0, v)$ denote the edges $e_1, \ldots, e_n$ along the path from the root to version v. Then, given a change-based version tree $T = (V, E, S_0)$ and a node v,

$$S_v = f_n \circ \cdots \circ f_1(S_0)$$

where each f_i is the function associated with the edge $e_i \in P(v)$. Although we do not need to store any versions S_v except the default version, we will need to be able to *materialize* any version via the above construction.

In many cases, we may also have inverse functions that allow us to transform from a version to its parent version. Specifically, the *inverse action* f^{-1} associated with an edge (u, v) satisfies $f^{-1}(S_v) = S_u$. This allows greater flexibility in materializing versions as we can move between states in both directions. Note that we may have some actions where inverses exist and others where they do not in the same change-based version tree. If there exists an inverse for *every* edge, we say the change-based version tree is *invertible*. Given an invertible change-based version tree $T = (V, E, S_0)$ and two nodes u and v with common parent p,

$$S_v = f_n \circ \cdots \circ f_1 \circ g_1^{-1} \circ \cdots \circ g_m^{-1}(S_u)$$

where each f_i is the function associated with the edge $e_i \in P(p, v)$ and g_j^{-1} is the inverse associated with the edge $e_j \in P(p, u)$. The construction corresponds to applying inverses up to a common parent p and then applying forward actions down to v.

We can also *compress* edges in a change-based version tree by composing their functions. Specifically, suppose v has a single parent u and a single child w. We can compress edges (u, v) and (v, w) with associated actions f and g, respectively, into a single edge (u, w) with the associated function $g \circ f$. The node v can then be eliminated from the change-based version T.

This allows us, given a set of nodes $\{v_i\}$, to construct a *skeleton* of a change-based version tree T, $\texttt{skel}(T, \{v_i\})$. The skeleton consists of all nodes $\{v_i\}$, the root, and the compressed edges between them. Often selected nodes include those that have been annotated or are at a branch point (have more than one child node).

2.3 Identifiers and Labeling

Unique identifiers make it possible to refer to a particular version, and labels provide users with the ability to annotate versions with memorable titles. For histories with centralized control, integers can be used to identify versions, but when versions may be distributed, we need to assign identifiers that are universally unique. Git uses hashes of content and commits to identify versions [9], but universally unique identifiers (UUIDs) can also be generated randomly with minimal probability of overlap.

In addition to an identifier, each node of a version tree may also be labeled. We will assume that a single label may be associated with each node, but clearly, associating a set of attributes is also possible. Note that labels may change over time; for example, a user who creates an updated version of a workflow may wish to move the label to the new version in the same way as one would overwrite a file with updated information. Formally, all version trees may have an associated labeling function $\mathcal{L} : V \to \Sigma^*$.

2.4 Provenance

Provenance captures how a particular result was achieved—the steps involved in the derivation of that result. Version trees naturally integrate with this goal as they capture dependencies between the different versions. Change-based version trees go further, presenting descriptions of the actions that transform one version to another. Change-based provenance further limits this to a monotonic change-based version tree. In change-based provenance, a user may add new actions to the tree but not edit or reorganize existing actions.

The distinction between change-based provenance [8] and change-based version trees is intentional because the latter offers more latitude in reorganizing or editing. Specifically, change-based provenance seeks to capture the exact changes that occurred and maintain the monotonicity of the tree. Each change is recorded and cannot be relocated or mutated. A change-based version tree requires a function to exist for each edge but does not enforce any restraints how this was derived or inferred. However, in many cases, one can obtain provenance about how an entity was constructed directly from the change-based version tree.

3 Manipulating Change-Based Version Trees

Instead of viewing version trees as a historical, immutable record, we propose operations that allow users to manipulate and update the trees. In the same way that a user might keep versions of code files or workflows, we argue that versions of version trees provide powerful new ways to manipulate collections of entities. Our goal is to allow users to modify the version tree itself. Some operations, like labeling and pruning are agnostic to the versioned entities, but others, like those where changes are being modified, require some understanding of the domain. In either case, an operation takes one version tree and produces another.

At the lowest level, we propose three pairs of primitive operations:

1. AddNode, DelNode: Add/delete a node
2. AddEdge, DelEdge: Add/delete an edge
3. AddLabel, DelLabel: Add/delete a label to/from a specific node

These operations provide the ability to construct any tree T' from any other tree T as in the worst case, we can delete everything from T and add everything from T'. In addition, each primitive operation has a clear inverse which means any change to the tree is invertible. While each action produces another tree, some may produce a degenerate tree where a subtree is not connected to the root.

Using these primitive operations, we can generate higher-level operations that transform the version tree. Two operations that act without any understanding of the versioned entities are relabeling and pruning. Relabeling involves moving a label ℓ from a version u to a version v. This can be accomplished by the pair of actions $\textsc{DelLabel}(u)$ and $\textsc{AddLabel}(v, \ell)$. Pruning v is a deletion of all nodes and edges in the subtree rooted at v. Again, we can rewrite this in terms of DelNode and DelEdge operations.

Other operations that act on the changes in a change-based version tree require some information about the underlying entities being manipulated. These include operations that rewrite past changes or reorganize versions. The *remap* operation takes pairs of changes (f, g) and replaces any instance of f on an edge in the version tree with g. For example, in a version tree of sets, we may wish to replace any occurrences of an element n with n'. Once a matching edge is identified, remap requires a $\textsc{DelEdge}(u, v)$ operation followed by a $\textsc{AddEdge}(u, v, g)$. Since version trees reflect the chronological order of user-initiated changes, reorganizing versions by similarity can aid in producing more compact and more intuitive trees [10]. This reorganization involves moving nodes and rewriting edges.

4 Framework

4.1 Versioning Version Trees

While allowing users to modify version trees grants some intuitive and efficient operations, we lose the original state of the version tree upon modification. Since a new version of the version tree has been created, the same versioning procedures can also be used to manage versions of version trees. Furthermore, since we have identified a set of primitive, invertible operations that change version trees, we can create an invertible change-based version tree to store versions of version trees. While one may question whether a tree is necessary here, the overhead in keeping a tree versus a list is minimal, and thus it seems reasonable to keep all versions of the version trees.

Formally, a *meta version tree* (MVT) $\mathcal{T} = (V, E, S_0)$ is an invertible change-based version tree where each edge defines a change to a version tree and S_0 is an empty version tree with only a root node. While the definition is straightforward and parallel to a standard version tree, working with the entities stored by the MVT of version trees comes with more overhead. Specifically, the creation of a new version of an entity triggers a new version in the current version tree, T, which in turn triggers updates to the MVT about the new node and edge added to T. This is represented by *two* new nodes and *two* new edges in the MVT, one pair for the new node and one pair for the new edge in T. With edge compression, we can package all of these changes together, but there is some extra overhead. Operations on the version trees are encoded as changes as described in the previous section.

Finally, note that with an MVT, identifying a specific version of an entity requires two identifiers—one to identify the version of the version tree and the other to identify the version of the entity in that tree. To materialize this entity,

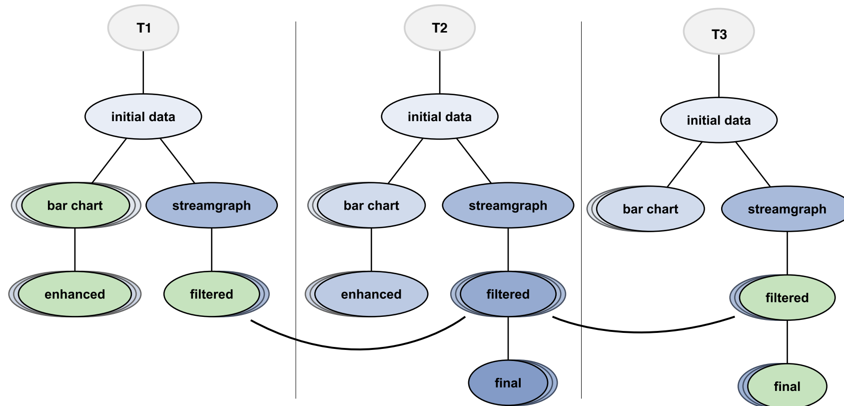

Fig. 2: Three versions of a version tree. Layered nodes indicate the presence of links to older versions (layered left) or newer ones (layered right). Green nodes are linked nodes that change from T1 to T2 or T2 to T3. Note that `filtered` has links to both older and newer versions.

we must first materialize the version tree and then use that version tree to materialize the version of the entity.

4.2 Linking Versions Across Trees

Systems like git and VisTrails have demonstrated that users can understand and interact with trees, but using similar interfaces for a meta version tree would likely be confusing. In a parallel interface, when a user wishes to select a specific object, that user would both choose a version tree and then select a specific version in that tree. As it is cumbersome to keep track of both of these levels—especially as both are trees, we propose an interface where users are encouraged to navigate between trees by identifying a particular version for which they wish to see a previous version in an older version tree.

To track relationships between versions that appear in different versions of the version tree, we propose a *link* that directionally links a version v in one version tree T_1 with a version v' in another version tree T_2. Suppose we have a meta version tree $\mathcal{T}$, a specific tree $T \in \mathcal{T}$, and a specific version $v \in T$. Then if T' is a child version of T and $v' \in T'$, we can define a *link* between v and v' to denote that v' was *indirectly derived* from v via the set of actions that transformed T to T'. In other words, v' is not the result of changes made directly in T', but it is the result of the changes made to the version tree T in deriving T'.

Links allow a user to navigate between versions of version trees by following them from a node in one version tree to a node in another. In effect, this is a different dimension of a derivation; instead of parent-child relationships, links show tree relationships. Figure 2 shows how a user interface may indicate the presence of linked versions with layered nodes. Upon clicking the lower layers to indicate a desire to see a version in a past or future version tree, we can show all transitively linked versions. Selecting one of those versions can then materialize not only that entity but also its associated version tree.

5 Workflow Applications

To demonstrate the potential of versioned version trees, we present intuitive operations they enable in the context of scientific workflows. In this section, we define scientific workflows as composed of computational modules and connections that link an output of one module with an input of another; each module may also have configurable parameters. Thus, changes to the workflow include the addition or deletion of modules, connections, or parameters.

5.1 Bulk Edits and Upgrades

Suppose a user made a number of workflows using a particular module, but decided later that a different module would have worked better. Instead of changing every version that contained the module, a user may instead wish to edit the action where that module was originally introduced, replacing it with the alternate module. Without meta version trees, replacing each version would at least require actions that remove the original module and add the new module, and could also require elements that depend on that module to be deleted and re-added after the change actions.

As software and libraries are updated, it may be necessary for workflows to also be updated to match them [11]. For example, if a library changes the interface for a particular call, we may also need to update the corresponding module. Furthermore, even if the module's interface does not change, it is important that the execution provenance capture exactly the version used. It is common, then, for an older workflow to need an upgrade to reflect the current interfaces. Without MVTs, upgrading an entire version tree can lead to a number of new branches that can drastically alter the appearance of the tree as shown in Figure 1 (left). When collaborators are working with different package versions (perhaps because they have different operating systems), this can be especially distracting. With MVTs, the upgrades can be encoded as updates to the *changes*, effectively replacing any action that added an old version of a module with a new action that adds the new version of the module as shown in Figure 1 (right).

5.2 Parameter Exploration

Parameter exploration is often viewed as a transient state whereby a number of versions are explored but only a select few are preserved, added to the version tree, and examined further. Otherwise, the many versions would clutter the version tree. We can store the parameter ranges explored as annotation on the version being explored, but the version tree is only updated when a selected workflow of interest is persisted. Not only is the provenance of unselected workflows lost, but storing information about the exploration in an annotation does not match how a user might manually carry out the same operations.

We propose representing parameter exploration in an MVT by creating an intermediate version tree to uniformly encode all parameter combinations tested and then pruning that version tree to eliminate all non-selected versions. In other words, from a node v of version tree T, we create a version tree T' with nodes

$v_0, \ldots, v_n$ as children of v but also each with links to v in T. If a user decides to use v_i for future work, it is persisted as a new version in the resulting version tree T^* but unselected versions are pruned.

5.3 Reorganization

Reorganizing a version tree by moving nodes and rewriting edges may allow a clearer understanding of relationships between workflows and/or a more efficient encoding. The minimization of version trees allows operations that cancel each other out to be removed, leading to a smaller version tree. Refactoring is an operation where nodes are relocated in order to represent the versions with fewer actions [10]. In the original implementation, the reorganized tree was not linked with the starting tree, and this made it difficult to determine which nodes had been moved or edges minimized. Using the actions in MVTs, we can not only link corresponding versions but we can highlight those that changed.

6 Provenance

As how an entity is created or derived is a question about the provenance of that entity, it is important to understand how meta version trees impact an understanding of that entity's evolution. We may either make a very literal interpretation of the origin of an entity or look to project this literal provenance into a form that may be more understandable.

The *literal provenance* of an entity derived from a version tree of version trees is exactly the steps in materializing that entity. Specifically, this is a sequence of actions describing the construction and transformation of the version tree the entity lives in, following the sequence of actions from the path through the version tree that actually construct the entity. While this provenance is correct, and following the steps will create the entity, its use is limited. Literal provenance is a chronological log of all activity in the version tree followed by the materialization of a specific version.

If we wish to dispense with a provenance view that involves multi-layered construction, we must project the operations down to the entity-level. Workflow evolution provenance is the sequence of operations involved in constructing a workflow [8]. While those operations live in a version tree, the provenance for any specific workflow involves only the changes related to that workflow. This is in contrast to literal provenance which keeps track of operations that may be unrelated to the entity in question. If we ignore the other versions of the version tree, we can generate *updated provenance* that is exactly the changes from the path through that tree. However, such a derivation is not accurate when the version tree has been transformed. Suppose a remap operation that mapped change A to Z occurred between T and T'. If the version v in T was created via a sequence BCAE, v' is created via BCZED. However, the change from A to Z was made *after* E and D. Thus, we want the sequences to look like BCAEA^{-1}Z.

We define *projected provenance* as the entity-level provenance that seeks to translate the effects of any tree operations into the entity-level while maintaining the correct order. For tree operations like remap, this equates to a inverse-forward sequence as shown in the previous example. In general, we can examine

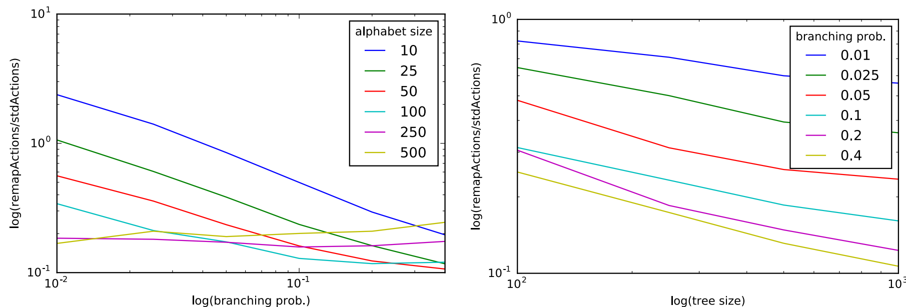

Fig. 3: Results for running remap operations over synthetically generated version trees capturing set manipulation. Generally, remap operations do better on trees with larger alphabets and higher branching probabilities.

the version before the version tree operation and after and infer the necessary entity-level operations. Note that such provenance introduces actions that did not actually occur. However, it may still be faithful in communicating the evolution of that entity.

7 Evaluation

In addition to providing intuitive operations over collections of versioned entities, meta version trees enable more efficient storage because they do not duplicate the same work in many branches. To evaluate this, we used synthetically-generated version trees and applied remap operations, comparing the resulting trees with those where the remap was applied to individual versions independently.

We used sets of integers from a bounded range as the domain with two change actions: add value and delete value. The version trees were randomly generated; each edge was an add or delete value of a randomly selected integer from the range (if the integer was in the set, delete value was inserted, otherwise add value was inserted). Based on a branching probability, the new edge was either appended to the current branch or the start of a new branch from an existing node. Then, a remap which changed a few of the values used in the actions was applied to the tree T. The same remap was also accomplished individually on nodes appearing in the skeleton of T. Tags were generated at a probability of 0.02; the skeleton includes tagged nodes.

We ran tests that combined different branching probabilities (0.01, 0.025, 0.05, 0.1, 0.2, 0.4), alphabet sizes (10,25,50,100,250,500), tree sizes (100,250,500,1000), and number of remapped values (1,2,4,8). For each test, the number of new MVT nodes created by the remap operation was compared with the number of MVT nodes created individually. Each of the 576 tests was run 200 times, and the average ratio between the new MVT nodes in the two approaches was computed.

In most cases, remap operations use fewer actions than conventional version updates (see Figure 3). Interestingly, there are scenarios where the remap fares worse. Specifically, when the alphabet for the set is small, the same integers

are being added and deleted over and over so many need to be changed in the remap operation but the replacement at the end of a long branch needs only happen once. For larger alphabets (when items aren't being constantly added and removed), the remap operations needs fewer actions since the operations update multiple branches at once.

8 Related Work

Version graphs have been used in a number of contexts, including source code management (e.g., git [9] and darcs [6]), web content versioning [15], and web services [12]. Conradi and Westfechtel's survey on versioning for software configuration management provides both background and an overview of different approaches for versioning including the distinction between state-based and change-based [5]. Version control system provenance from git can also be represented in the PROV standard [7].

In the context of data management, versioning has focused on data lineage [3, 4] and changes over time [14]. Recently, the DataHub project has been working to support collaborative data analysis with a view to versioning evolving datasets [2]. Because of the cost of storing both versions and changes, the project seeks to examine the tradeoff between storing versions and materializing them using change information. Ba et al. describe methods for incorporating uncertainty into version control [1].

The problem of determining impacts and conflicts of operations on versions that are themselves graphs, like workflows, is complicated by the subgraph isomorphism problem. Previous work focused on reorganizing versions by using the given changes [10]. Metrics based on maximal common subgraphs may also be used to compare workflow graphs [13]. Taentzner et al. have investigated versioning graphs and resolving conflicts in software modeling [18]. darcs uses patch theory to reorder and merge different changes [6].

9 Conclusion

We present meta version trees to allow more intuitive and efficient operations on collections of versions. Instead of editing multiple versions individually, users may edit the change and create a new version of the version tree. Future work includes examining applications beyond workflows and considering the process of applying analogies to multiple versions. Specifically, we envision allowing a user to edit a single entity and then propagate those changes to multiple versions. While this could be done using workflow analogies [16], it should be possible to encode the analogy as an edit to the version tree instead.

Another important consideration is potential conflicts introduced by an edited operation. For example, when an action adding a specific value is removed from the version tree, descendant actions that delete that value are invalid. One could check for such conflicts before allowing the operation to proceed, or it might be possible to separate those versions that are affected and put them in a subtree unaffected by the tree modification.

Acknowledgements. The author thanks Juliana Freire for her suggestions and the anonymous reviewers for their helpful comments. This work was supported in part by NSF CNS-1405927.

References

1. Ba, M.L., Abdessalem, T., Senellart, P.: Uncertain version control in open collaborative editing of tree-structured documents. In: Proc. 2013 ACM Symp. Document Engineering. pp. 27–36. ACM (2013)
2. Bhattacherjee, S., Chavan, A., Huang, S., Deshpande, A., Parameswaran, A.: Principles of dataset versioning: Exploring the recreation/storage tradeoff. Proc. VLDB Endow. 8(12), 1346–1357 (2015)
3. Bose, R., Frew, J.: Lineage retrieval for scientific data processing: A survey. ACM Comput. Surv. 37(1), 1–28 (2005)
4. Buneman, P., Khanna, S., Tan, W.C.: Why and where: A characterization of data provenance. In: Proc. 8th Int'l Conf. on Database Theory. pp. 316–330. Springer-Verlag (2001)
5. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Comput. Surv. 30(2), 232–282 (1998)
6. Darcs. http://darcs.net/
7. De Nies, T., Magliacane, S., Verborgh, R., Coppens, S., Groth, P., Mannens, E., Van de Walle, R.: Git2PROV: Exposing version control system content as W3C PROV. In: Poster & Demo Proc. 12th Int'l Semantic Web Conf. (2013)
8. Freire, J., Silva, C., Callahan, S., Santos, E., Scheidegger, C., Vo, H.: Managing rapidly-evolving scientific workflows. In: IPAW 2006. pp. 10–18. LNCS 4145, Springer Verlag (2006)
9. git. http://git-scm.com/
10. Koop, D., Freire, J.: Reorganizing workflow evolution provenance. In: 6th USENIX Workshop on the Theory and Practice of Provenance (TaPP 2014) (2014)
11. Koop, D., Scheidegger, C.E., Freire, J., Silva, C.T.: The provenance of workflow upgrades. In: Provenance and Annotation of Data and Processes, pp. 2–16. Springer (2010)
12. Leitner, P., Michlmayr, A., Rosenberg, F., Dustdar, S.: End-to-end versioning support for web services. In: IEEE Int'l Conf. on Services Computing. pp. 59–66 (2008)
13. Lins, L.D., Ferreira, N., Freire, J., Silva, C.T.: Maximum common subelement metrics and its applications to graphs. CoRR abs/1501.06774 (2015)
14. Özsoyoğlu, G., Snodgrass, R.T.: Temporal and real-time databases: a survey. IEEE Transactions on Knowledge and Data Engineering 7(4), 513–532 (1995)
15. Sabel, M.: Structuring wiki revision history. In: Proc. 2007 Int'l Symp. on Wikis. pp. 125–130. ACM, New York, NY, USA (2007)
16. Scheidegger, C.E., Vo, H.T., Koop, D., Freire, J., Silva, C.T.: Querying and creating visualizations by analogy. IEEE Trans. Vis. Comp. Graph. 13(6), 1560–1567 (2007)
17. Subversion (svn). https://subversion.apache.org
18. Taentzer, G., Ermel, C., Langer, P., Wimmer, M.: A fundamental approach to model versioning based on graph modifications: from theory to implementation. Software & Systems Modeling 13(1), 239–272 (2014)