A Deformation-based Edit Distance for Merge Trees

Florian Wetzels\textsuperscript{*}  
TU Kaiserslautern  

Christoph Garth\textsuperscript{†}  
TU Kaiserslautern

\textbf{Abstract}

In scientific visualization, scalar fields are often compared through edit distances between their merge trees. Typical tasks include ensemble analysis, feature tracking and symmetry or periodicity detection. Tree edit distances represent how one tree can be transformed into another through a sequence of simple edit operations: relabeling, insertion and deletion of nodes. In this paper, we present a new set of edit operations working directly on the merge tree as an geometrical or topological object: the represented operations are deformation retractions and inverse transformations on merge trees, which stands in contrast to other methods working on branch decomposition trees. We present a quartic time algorithm for the new edit distance, which is branch decomposition-independent and a metric on the set of all merge trees.

\textbf{Keywords:} Scalar data, Topological data analysis, Merge trees, Edit distance

1 Introduction

Measuring similarity or dissimilarity of scalar fields, as well as finding similar features or mappings between them, is an important tool in scientific visualization for the analysis of ensemble data or time series [27], specifically for tasks such as feature tracking, clustering, and the detection of periodicity or self-similarity. Both problems can be and have been addressed through the use of edit distances on merge trees [11, 14, 19, 26], an abstract representation of the topology of sub-level sets or super-level sets of scalar functions. Tree edit distances, which come in a large variety of different forms [4], are well-suited for these tasks since, typically (i.e. for most variants), they are efficiently to compute, induce mappings between the edges of the trees (which correspond to topological features), fulfill the metric properties, are very intuitive to understand and have great flexibility through the use of different base metrics on the labels of the trees. Furthermore, working on topological abstractions such as merge trees has a huge impact on performance, as these structures usually stay rather small in comparison to the actual data domain.

The operations in classic tree edit distances are node-insertion, node-deletion and node-relabel. For merge trees, this set of edit operations is not coherent with the intuitive way to transform them, since merge trees are actually continuous objects, whereas node-labeled trees are not. Formally, this means that applying the classic edit operations to a merge tree may not result in another merge tree, or an invalid one. Figure 2 illustrates this in more detail. Previous approaches [14, 16, 19] overcame this issue by working on branch decomposition trees (BDTs) of merge trees rather than working on the merge tree itself. However, this comes with the downside of using fixed branch decompositions, which are very susceptible to small-scale perturbations in the data. Although this problem has also been overcome recently [26] through the new concept of branch mappings, the solution came at the cost of losing the metric property.

Furthermore, both approaches (using fixed BDTs or branch mappings) lose the direct connection to modifying operations on merge trees. While some connection is still there (e.g. all operations on BDTs can be interpreted as non-primitive operations on merge trees), which will be discussed in Section 5, the corresponding operations differ significantly from the classic model on arbitrary trees, and are not as intuitive. In particular, the branch based edit distances focus more on the induced mappings than the actual edit operations.

This paper introduces an edit distance for merge trees based on a new set of edit operations which are specifically tailored to deformations of merge trees, with a cost function based on the typical drawings or embeddings of merge trees, making it highly intuitive.

\textbf{Contribution}

In particular, we present an edit distance between merge trees with its corresponding mappings, which we call \textit{path mappings}, that

\begin{itemize}
  \item corresponds to an optimal sequence of deformations,
  \item is independent of a fixed branch decomposition,
  \item fulfills the metric properties,
  \item shows better practical performance than branch mappings.
\end{itemize}

\textsuperscript{*}e-mail: f.wetzels13@cs.uni-kl.de
\textsuperscript{†}e-mail: garth@cs.uni-kl.de

Figure 1: Illustration of the edit operations on abstract merge trees. For all three types, the cost function is based on persistence change, i.e. change in y-range of the modified edge (see coordinate grid).
Furthermore, we provide an open-source implementation of our distance and showcase its utility by replicating previous results for clustering and periodicity detection.

In the remainder of this section, we cover related work. Section 2 recapts important definitions and concepts. In Section 3.2, we introduce the new edit distance and path mappings, and show their core properties needed for implementation and application. Section 4 discusses the actual algorithms and the results of our experiments. In Section 5, we discuss the choice of edit operations and the relation to previous methods. We conclude the paper with an outlook to future work in Section 6.

Related Work

Topological descriptors and abstractions play a key role for the analysis of scalar fields in scientific visualization. Many of them have been used for the task of scalar field comparison: merge trees or contour trees in an ensemble [10, 11]. Wetzels et al. defined the concept of branch mappings [26], an edit distance that works on BDTs and adapted for unordered rooted trees by Zhang [30].

A survey on topology based visualization methods in general is given by Heine et al. in [8]. Tree edit distances in general have been introduced by Tai for ordered rooted trees [21] and adapted for unordered rooted trees by Zhang [30]. In this paper, we use the one-degree edit distance, which has been introduced by Selkow [17] for ordered rooted trees and which is a special variant of the constrained edit distance [29]. A survey on the various versions of tree edit distances and related problems can be found in [4]. We now quickly review those methods from the field of scientific visualization that are closest to the here presented distance measure.

The constrained edit distance on merge trees (actually working on a fixed BDT) has been applied by Sridharamurthy et al. [19] to various visualization tasks including periodicity detection or clustering. They adapted their method for the use in self-similarity detection. Related Work can be found in [3, 12, 22, 28].

Figure 2: Illustration of the difference between the classic discrete edit operations and the new continuous edit operations, exemplary for a deletion. If we apply a classic node-deletion or edge-contraction to the middle tree, we get the tree on the left. The result is not a merge tree. The desired result is shown in the right tree. Note that the problem cannot be fixed through deleting the remaining node with another classic edit operation, because a deletion always removes an edge as well.

just a special case, an edit distance was proposed in [1]. Actually, the here presented edit distance can be seen as an adaptation of the reeb graph edit distance to merge trees (note however that tree edit distances are more than just a special case of graph edit distances, as the allowed operations differ significantly and the graph edit distance has a much higher complexity). Other examples for distances between reeb graphs can be found in [2, 6, 18]. In [13], a distance measure for extremum graphs was introduced.

2 Preliminaries

In this section, we recap the core definitions from computational topology and graph theory that are needed to define the edit distance for merge trees and to study its properties. For basic notions on topological spaces and simplicial complexes, we refer to [7].

Merge Trees

Given a d-manifold $\mathcal{X}$ with a continuous map $f : \mathcal{X} \to \mathbb{R}$, its Join Tree and Split Tree represent the connectivity of its sub-level sets and super-level sets.

The join tree of $\mathcal{X}$, $f$ is the quotient space $\mathcal{X}/\sim$ under the equivalence relation $\sim$, where $x \sim y$ if $f(x) = f(y)$ and $x$ and $y$ belong to the same connected component of the sublevel set $f^{-1}((\sim, f(x))]$. A split tree of $\mathcal{X}$, $f$ is defined in the same way by just replacing the sub-level set $f^{-1}((\sim, f(x))]$ with the super-level set $f^{-1}((f(x), \sim))$.

We use the terms critical point, maximum, minimum, saddle and path (in a topological space, not in discrete graphs) following the definitions in [5].

A merge tree is, in essence, a 1-dimensional simplicial complex. I.e. for each merge tree $\mathcal{X}/\sim$, there is a simplicial complex $K$ of dimension 1 with a scalar function $f$ such that its underlying space $|K|$ is not only homeomorphic to $\mathcal{X}/\sim$, but there is a homeomorphism $h : \mathcal{X}/\sim \to |K|$ that preserves the scalar function, $f(x) = f(h(x))$ for all $x \in \mathcal{X}/\sim$, and with $Vert(K)$ being exactly the critical points of $\mathcal{X}, f$. We call this simplicial complex $\mathcal{T}(\mathcal{X}, f)$, and we will refer to the underlying spaces $|\mathcal{T}(\mathcal{X}, f)|$ of these structures as merge trees. This means that by merge trees we do not only denote quotient spaces of scalar fields but actually all spaces that are homeomorphic to them through a scalar function-preserving homeomorphism.

As a next step, we define an abstract model for merge trees and introduce basic notation for graphs and trees used in this paper. After that, we will discuss the relation of continuous merge trees (as quotient spaces) and abstract merge trees. Abstract Merge Trees

Throughout this paper, we will consider rooted trees as directed graphs with parent edges. I.e. a rooted tree $T$ is a directed graph with vertex set $V(T)$, edge set $E(T) \subseteq V(T) \times V(T)$ and a unique root, denoted root($T$). We call a node $c \in V(T)$ a child of node $p \in V(T)$, if $(c, p) \in E(T)$, and, conversely, $p$ the parent of $c$. For a node $p$, we denote its number of children by $\text{deg}_T(p) := |\{c \mid (c, p) \in E(T)\}|$. Furthermore, we denote the empty tree, consisting only of a single node and no edges, by $\bot$.

Rooted trees that can be interpreted as merge trees for some domain of dimension at least 2 will be called abstract merge trees. They are the center objects of this paper.

Definition 1. An unordered, rooted tree $T$ of (in general) arbitrary degree with edge labels $\ell : V(T) \to \mathbb{R}_{\geq 0}$ is an Abstract Merge Tree if the following properties hold:

- The root node has degree one, $\text{deg}_T(\text{root}(T)) = 1$
- All inner nodes have a degree of at least two, $\text{deg}_T(v) \neq 1$ for all $v \in V(T)$ with $v \neq \text{root}(T)$

Note that we do not use node labels representing the scalar value of the original critical points but rather edge labels directly representing the persistence of edges (we use the term persistence for
the length of edges and paths, since this similarity measure is just an adaptation of the persistence of branches and to distinguish it from the length of paths, i.e. number of edges, in abstract trees). We chose this for two reasons: first, persistence of edges or paths are the properties of interest anyway and second, this gives a lot of flexibility in terms of representing similar properties like edge volume or region size via the same abstract definition. Since we are usually not interested in shifts of the whole merge tree to higher scalar values, this should not influence the practicality of the definition. Throughout this paper, we will often just write \( T \) instead of \( T, \ell \), but if we do, it should be clear from the context that the abstract merge tree has a label function attached.

Since the root of an abstract merge tree always has degree one and inner nodes do not, subtrees rooted in an inner node are not abstract merge trees themselves. Therefore, we identify subtrees by root edges, rather than root nodes: Formally, for a node \( p \in V(T) \) with child \( c \in V(T) \), the subtree rooted in \((c,p)\) is defined by the vertex set

\[
V(T[(c,p)]) = \{c,p\} \cup \{v \mid c \text{ is an ancestor of } v\}
\]

and the induced edge set. Given an abstract merge tree \( T \) with subtrees \( T' \) rooted in the edge \((c,p)\), we define \( T - T' \) to be the tree \( T'' \), which we obtain by removing all edges and all vertices of \( T' \) from \( T \) except the root \( p \). If \( \deg_p(p) = 2 \), then we also remove it from \( T'' \), as otherwise \( p \) would be an inner node of degree one in \( T'' \). With this definition, it holds that \( T' \) and \( T'' \) are abstract merge trees as well.

As for general graphs, a path of length \( k \) in an abstract merge tree \( T \) is a sequence of vertices \( p = v_1 \ldots v_k \in V(T)^k \) with \( v_i, v_{i+1} \in E(T) \) for all \( 2 \leq i \leq k \) and \( v_i \neq v_j \) for all \( 1 \leq i, j \leq k \). Note the strict root-to-leaf direction: we only consider monotone paths. For a path \( p = v_1 \ldots v_k \), we denote its first vertex by \( \text{start}(p) := v_1 \), its last vertex by \( \text{end}(p) := v_k \) and the set of all paths of a tree \( T \) by \( \mathcal{P}(T) \).

In an abstract merge tree \( T \), each node \( v \in V(T) \) has a unique path connecting it to the root of the tree \( \text{root}(T) \). Each node \( u \neq v \) on this path is called an ancestor of \( v \) and we denote the unique path connecting \( u \) and \( v \) by \( \text{path}(v,u) \).

We lift the label function \( \ell \) of an abstract merge tree \( T \) from edges to paths in the following way: \( \ell(v_1 \ldots v_k) = \sum_{2 \leq i \leq k} \ell((v_i,v_{i-1})) \).

### Continuous Merge Trees to Abstract Merge Trees

Now, we show how the two introduced concepts of merge trees relate. As stated above, abstract merge trees represent those trees that can be interpreted as merge tree for some domain. Formally, we denote the continuous merge tree of a scalar field \( f \) by \( \mathcal{T}(\mathbb{R}, f) \) and the corresponding abstract merge tree by \( T(\mathbb{R}, f) \), which we define in the following. Again, we closely stick to the definitions for contour trees in [5].

The vertex set \( V(\mathcal{T}(\mathbb{R}, f)) \) is the set of critical points in \( \mathcal{T}(\mathbb{R}, f) \). For the edges, we have \( (u,v) \in E(\mathcal{T}(\mathbb{R}, f)) \) if and only if \( f(v) < f(u) \) and there is an \( f \)-monotone path connecting \( u \) and \( v \) such that \( u \) and \( v \) are the only critical points on this path. We define the label \( \ell_f((u,v)) \) to be \( f(u) - f(v) \). Note the strong correspondence between an actual merge tree \( \mathcal{T} \) and the abstract merge tree \( T \): the vertices in \( V(T) \) are exactly the vertices of the simplicial complex \( \mathcal{T} \) and the edges \( E(T) \) are exactly the 1-simplices in \( \mathcal{T} \). If only a merge tree \( \mathcal{T} \) is given without the original domain, we can therefore denote by \( T(\mathcal{T}) \) its corresponding abstract merge tree. Furthermore, for each abstract merge tree \( T \), we can define a simplicial complex \( \mathcal{T}(T) \) such that \( |\mathcal{T}(T)| \) is a merge tree and \( T(\mathcal{T}(T)) = T \) by using an arbitrary embedding/drawing of the abstract merge tree.

### 3 Deformation-based Edit Distance

In this section, we introduce the new edit distance for merge trees. First, we define two models of edit operations: an intuitive description for a set of deformations on continuous merge trees, which is mainly used to guide the motivation of the new distance, as well as a formal definition of edit operations on abstract merge trees. Next, we study the relationship of the two models before presenting the underlying algorithmic concept of path mappings and their recursive structure. We postpone an in-depth discussion of the choice of edit operations to Section 5.

#### 3.1 Edit Operations

We begin by defining two sets of edit operations on merge trees, one of which works on the continuous objects, i.e. merge trees as the quotient spaces of the original domains, and the other one works on discrete abstract merge trees and is used for computation.

**Continuous Edit Operations**

For continuous trees, the goal is to define an intuitive set of edit operations, which should be able to transform any two merge trees into each other and operate only locally on edges and nodes. Similar to deletions and insertions on arbitrary trees, we need edit operations changing the size of the tree. Since the labels in merge trees are the lengths of edges, relabel operations fall into the same category. Therefore, we identified only two types of operations:

- Shrinking an arc, possibly deleting it.
- Extending an arc or inserting a new one.

These operations have the following intuition: they strongly resemble deformation retractions on merge trees: since a deformation retraction on a merge tree always yields another merge tree (or a single point, which is formally also a tree), all they can do is shrink edges, or possibly remove them. The corresponding inverse transformations are extending or inserting arcs. Hence, a sequence of shrinking operations transforms \( T_1 \) into \( T_2 \) if and only if \( T_2 \) is a deformation retract of \( T_1 \), and conversely, a sequence of extending operations transforms \( T_1 \) into \( T_2 \) if and only if \( T_1 \) is a deformation retract of \( T_2 \). Next, we define the equivalent model on abstract merge trees and then prove the correspondence to deformation retractions on this model.

**Abstract Edit Operations**

The edit distance for abstract merge trees is now defined in a more formal way: we consider the following edit operations that transform an abstract merge tree \( T, \ell \) into another abstract merge tree \( T', \ell' \):

- **Edge relabel**: changing the length of an edge \((c,p)\) to a new value \( \nu \in \mathbb{R}_{\geq 0} \), i.e. \( T' = T, \ell'((c,p)) = \nu \) and \( \ell'(e) = \ell(e) \) for all \( e \neq (c,p) \).
- **Edge contraction**: remove an edge from the tree and merge the two nodes. Then, remove the parent node if it had only two children originally. Formally, for a node \( p \) with children \( c_0 \ldots c_k \) and parent \( p' \), we define \( T' \) after contracting \((c_1,p)\) as follows: if \( k > 1 \), we have
  \[
  V(T') = V(T) \setminus \{(c_1,p)\}, E(T') = E(T) \setminus \{(c_1,p)\},
  \]
  and otherwise, if \( k = 1 \), we have
  \[
  V(T') = V(T) \setminus \{(c_1,p)\},
  E(T') = E(T) \cup \{(c_1,p')\} \setminus \{(c_1,p),(c_1,p),(p,p')\}
  .
  \]
  Furthermore, \( \ell' = \ell \) if \( k > 1 \), otherwise
  \[
  \ell'((c_1,p')) = \ell((p,p')) + \ell((c_1,p)),
  \]
  and \( \ell'(e) = \ell(e) \) for all \( e \neq (c_1,p') \).
- **Inverse edge contraction**: inverse operation to edge contraction.
We also call edge contractions deletions and inverse edge contractions insertions, to have a more intuitive naming that also fits better to the classic edit operations on node-labeled trees. The three types of edit operations are illustrated on an example tree in Figure 1. If a sequence of edit operations $S$ transforms an abstract merge tree $T_1$ into $T_2$, we denote this by $T_1 \xrightarrow{S} T_2$.

We define the costs of the edit operations between abstract merge trees as the change in persistence. As for classic edit distances, we denote the edit operations by pairs of two labels for relabel operations or pairs of a label and a blank symbol for deletions or insertions. We use $\mathbb{R}_{\geq 0}$ as the label set for abstract merge trees and 0 as the blank symbol. Then, we define the cost function simply as the euclidean distance on $\mathbb{R}_{\geq 0} \cup \{0\}$: $c(l_1, l_2) = |l_1 - l_2|$ for all $l_1, l_2 \in \mathbb{R}_{\geq 0}$. This means, for a deletion or insertion we charge the persistence of the edge, whereas for a relabel we charge the persistence difference between the old and new edge. We should note that abstract merge trees allow for other labels than persistence, but the cost function can be easily adapted to suit this usecase. For this paper, we restrict to persistence labels.

Model Relation

The edit operations from the two models can be transformed into each other in an intuitive way. Since nodes in an abstract merge tree correspond to critical points of the continuous trees, shortening and extending operations are mapped to delete and insert operations if and only if at least one critical point of the continuous tree disappears and they are mapped to relabel operations otherwise. For the other direction, all edit operations on abstract merge trees are mapped to continuous operations in the obvious way.

Furthermore, the abstract operations can also be related to deformation retracts on continuous merge trees. The connection is the following:

**Theorem 1.** Let $\mathcal{T}_1, \ell_1$ and $\mathcal{T}_2, \ell_2$ be merge trees with abstractions $T_1 = T(\mathcal{T}_1), \ell_{\mathcal{T}_1}$, and $T_2 = T(\mathcal{T}_2), \ell_{\mathcal{T}_2}$. If $|\mathcal{T}_2|$ is homeomorphic to a deformation retract of $|\mathcal{T}_1|$, then there is a sequence of edit operations $S$ only containing deletions and relabels that decrease the edge labels such that $T_1 \xrightarrow{\ell_{\mathcal{T}_1}} T_2 \xrightarrow{\ell_{\mathcal{T}_2}} S$.

Furthermore, given two abstract merge trees $T_1, \ell_1$ and $T_2, \ell_2$ with $T_1 \xrightarrow{\ell_{\mathcal{T}_1}} T_2 \xrightarrow{\ell_{\mathcal{T}_2}} S$ and $S$ only containing deletions and relabels that decrease the edge labels, then there are merge trees $\mathcal{T}_1, f_1$ and $\mathcal{T}_2, f_2$ with $T_1 = T(\mathcal{T}_1), \ell_1 = \ell_{\mathcal{T}_1}$, and $T_2 = T(\mathcal{T}_2), \ell_2 = \ell_{\mathcal{T}_2}$, such that $|\mathcal{T}_2|$ is homeomorphic to a deformation retract of $|\mathcal{T}_1|$.

**Proof.** See supplementary material, App. A.

The same also holds for insert operations and label-increasing relabels, with the adaption that $\mathcal{T}_1$ is a deformation retract of $\mathcal{T}_2$. This leads to another interesting property of the new edit distance: classic edit distances or edit mappings represent the largest common subtree, which means that our interpretation of the merge tree edit distances yields something like the largest common deformation retract, where the size of a merge tree is its total persistence.

**Edit Distance**

The edit distance between two abstract merge trees $T_1, T_2$ is defined to be the minimal cost of an edit sequence transforming $T_1$ into $T_2$:

$$\delta_{\mathcal{T}}(T_1, T_2) = \min \{ c(S) \mid T_1 \xrightarrow{S} T_2 \}.$$ 

We call $S$ a one-degree edit sequence, if all insertions and deletions happen on edges connecting a leaf node. The edit distance based on these sequences is called one-degree edit distance:

$$\delta_1(T_1, T_2) = \min \{ c(S) \mid T_1 \xrightarrow{S} T_2, S \text{ is one-degree} \}.$$ 

Since edit sequences can be concatenated and the costs just add up, they are a metric for abstract merge trees. Since one-degree edit sequences do not restrict the possibility of concatenation, the one-degree edit distance is a metric, too.

**Theorem 2.** $\delta_{\mathcal{T}}$ and $\delta_1$ are metrics on the set of all abstract merge trees.

In [30] it has been shown that the problem of computing the general edit distance on unordered, node-labeled trees is NP-hard. Therefore, constrained versions like tree alignments [9] and the constrained edit distance [29] have been introduced. The one-degree edit distance [17] is a special case of the constrained edit distance and has tractable algorithms even for unordered trees. Due to the high complexity of unconstrained tree edit distances, merge tree edit distances are usually defined using one of the three kinds of constrained versions, see [11, 14, 16, 19, 26]. For the same reason, we will use $\delta_1$ instead of $\delta_{\mathcal{T}}$ throughout the rest of this paper. Intuitively, one-degree edit distances capture strongly connected subtrees instead of ancestor-preserving subtrees (which allow for gaps).

### 3.2 Path Mappings

Edit sequences using the edit operations defined in the last section induce mappings between abstract merge trees in a similar way as edit sequences for the classic tree edit distance do. However, in contrast to mappings between the edges or nodes of the trees, we get...
mappings between the paths of two abstract merge trees. To see why, consider the definition of an edge contraction. If the remaining node has only one child, we remove it from the tree and connect its only child to its parent. The new edge in the resulting tree is created by merging two edges in the original tree. This relation is represented by the mapping. Hence, the new edge in the resulting tree is mapped to a path of length 2 in the original tree, consisting of the two edges that were merged. This correspondence also works transitively for multiple operations and also for inverse edge contractions the other way around. Figure 3 illustrates this correspondence and how to derive a path mapping from a sequence of edit operations. Next, we will study path mappings in a more formal manner, starting with a definition.

**Definition 2.** Given two abstract merge trees $T_1$, $T_2$, a path mapping between $T_1$ and $T_2$ is a mapping $M \subseteq \mathcal{P}(T_1) \times \mathcal{P}(T_2)$ such that

1. $p_1 = p_2$ if and only if $q_1 = q_2$ for all $(p_1, q_1), (p_2, q_2) \in M$,
2. $|p_1 \cap p_2| \leq 1$ and $|q_1 \cap q_2| \leq 1$ for all $(p_1, q_1), (p_2, q_2) \in M$,
3. for all $(p, q) \in M$,

   - either there are paths $p' \in \mathcal{P}(T_1)$ and $q' \in \mathcal{P}(T_2)$ such that $(p', q') \in M$ and start$(p) = \text{end}(p')$ and start$(q) = \text{end}(q')$,
   - or start$(p) = \text{root}(T_1)$ and start$(q) = \text{root}(T_2)$.

For a path mapping $M$ between two abstract merge trees $T_1, T_2, T_3, T_4$, we also define its corresponding edit operations $\text{edits}(M)$. They consist of the corresponding relabel, insert and delete operations:

- $\text{rel}(M) = \{(e_1 e_2) \mid (p_1, p_2) \in M\}$,
- $\text{ins}(M) = \{(0, e_2) \mid e_2 \in E(T_2), \exists p_1 \in \mathcal{P}(T_1) : (p_1, e_2) \in M\}$,
- $\text{del}(M) = \{(e_1 0) \mid e_1 \in E(T_1), \exists p_2 \in \mathcal{P}(T_2) : (e_1, p_2) \in M\}$.

Then we have $\text{edits}(M) = \text{rel}(M) \cup \text{ins}(M) \cup \text{del}(M)$. Furthermore, we define the costs of a mapping through the corresponding edit operations:

$$c(M) = \sum_{(l_1, l_2) \in \text{edits}(M)} |c(l_1, l_2)|$$

We say a path $p_1 \in \mathcal{P}(T_1)$ is contained in a path mapping $M \subseteq \mathcal{P}(T_1) \times \mathcal{P}(T_2)$, if there is a path $p_2 \in \mathcal{P}(T_2)$ with $(p_1, p_2) \in M$. Also, $p_2$ is contained in $M$ if the symmetrical condition holds. We say the subtree $T_1[[u, v]]$ is not contained in $M$ if for any $p' \in \mathcal{P}(T_1[[u, v]])$, $p'$ is not contained in $M$.

A core property of optimal path mappings is that the contained paths are not unnecessarily partitioned, i.e. if we interpret them as edges, these edges form an abstract merge tree without degree one nodes (except the root). Intuitively, the reason for this can be seen from the way paths are mapped in Figure 3: all subtrees branching from mapped paths ($ABCD, ABC, ACD$) are either deleted or inserted, i.e. they are not present in the mapping. In other words, mapped branches do not start within other mapped paths. This property is given in Lemma 1. Furthermore, all mapped paths either end in a leaf node (e.g. $ABCD$) or end in the starting vertex of at least two other mapped paths (e.g. $ABC$ splits into $CD$ and $ED$, both also present in the mapping). This is formalized in Lemma 2. The properties are also illustrated in Figure 4 and essential for deriving the recursive structure in Section 3.3. We now discuss them more formally.

First, we note that the following property directly follows from the definition of path mappings (conditions 2 and 3).

**Lemma 1.** Let $M \subseteq \mathcal{P}(T_1) \times \mathcal{P}(T_2)$ be an optimal path mapping between two abstract merge trees $T_1, T_2$. For any path $p = v_1 \ldots v_k \in \mathcal{P}(T_1)$ that is contained in $M$ and any child $v_i \neq v_{i+1}$ of $v_i$ ($1 < i < k$), the subtree $T_1[[v_i, v_{i+1}]]$ is not contained in $M$.

Symmetrically, the same holds for any path $p \in \mathcal{P}(T_2)$.

Furthermore, any path contained in an optimal path mapping branches into at least two other contained branches. A proof for this claim is provided in the supplementary material.

**Lemma 2.** Let $M \subseteq \mathcal{P}(T_1) \times \mathcal{P}(T_2)$ be an optimal path mapping between two abstract merge trees $T_1, T_2$. For any path $p \in \mathcal{P}(T_1)$ that is contained in $M$, there are at least two paths $p', p''$ that are contained in $M$ with start$(p') = \text{start}(p'') = \text{end}(p)$.

Symmetrically, the same holds for any path $p \in \mathcal{P}(T_2)$.

**Proof.** See supplementary material, App. B.

The properties from the last two lemmas are illustrated Figure 4. As a next step, we now focus on the equivalence of path mappings and the here defined one-degree edit distance for abstract merge trees. We show the equivalence by first proving that an optimal edit sequence has a corresponding mapping of lower or equal cost and secondly that each mapping has a corresponding edit sequence. This then allows us to compute optimal path mappings instead of optimal edit sequence.

**Lemma 3.** Let $S$ be a cost-optimal sequence of edit operations that transforms an abstract merge tree $T_1$ into another one $T_2$. Then there exists a path mapping $M \subseteq \mathcal{P}(T_1) \times \mathcal{P}(T_2)$ such that $c(M) \leq c(S)$.

**Proof.** See supplementary material, App. C.

**Lemma 4.** For two abstract merge trees $T_1, T_2$, let $M \subseteq \mathcal{P}(T_1) \times \mathcal{P}(T_2)$ be a path mapping. Then there exists a sequence $S$ of edit operations that transforms $T_1$ into $T_2$, with $c(S) = c(M)$.

**Proof.** See supplementary material, App. D.
We now know that the one-degree edit distance is equivalent to the abstract merge trees $T_1$ and $T_2$.

**Proof.** Follows directly from Lemmas 3 and 4. 

### 3.3 Recursive Structure

We now know that the one-degree edit distance is equivalent to optimal path mappings. Next, we will investigate the recursive structure of path mappings, which can then be exploited to obtain efficient polynomial time algorithms. We omit a formal discussion of the base cases since the optimal path mapping between a non-empty abstract merge tree and an empty one is of course the empty mapping, and for two trees with just one edge, we always map the two unique edges. Furthermore, we only consider binary merge trees for simplicity. The recursion can of course be easily adapted for trees of arbitrary degree.

**Lemma 5.** Given two abstract merge trees $T_1, T_2$ with root$(T_1) = v_1$ and root$(T_2) = u_1$, let $v_2, u_2$ be the unique children of the two roots and let those have children $v_3, v_4$ and $u_3, u_4$. Let $T_1' = T_1([v_2, v_3], [v_2, v_4])$, $T_2' = T_2([u_2, u_3])$, and $T_2'' = T_2([u_2, u_4])$. Then, for the one-degree edit distance between $T_1$ and $T_2$, it holds that $\delta_1(T_1, T_2)$ is either

- $\delta_1(T_1', \perp) + \delta_1(T_1 - T_1', T_2)$ or
- $\delta_1(\perp, T_2') + \delta_1(T_1, T_2 - T_2')$ or
- $\delta_1(T_1', \perp) + \delta_1(T_1 - T_1', T_2)$ or
- $\delta_1(\perp, T_2') + \delta_1(T_1, T_2 - T_2')$ or
- $\delta_1(T_2', T_2') + \delta_1(T_1', T_2)$ or
- $\delta_1(T_1', T_2') + \delta_1(T_1', T_2)$ or
- $\delta_1(T_2', T_2') + \delta_1(T_1', T_2)$ or
- $\delta_1(T_2', T_2') + \delta_1(T_1', T_2)$.

**Proof.** To show this recursion, we consider the optimal path mapping $M$ between $T_1$ and $T_2$. We know that $M$ is not empty, since both $T_1$ and $T_2$ are non-empty trees. Furthermore, we know that there is a path of paths $p_1 \in \mathcal{P}(T_1)$, $p_2 \in \mathcal{P}(T_2)$ in the mapping. $(p_1, p_2) \in M$, such that both begin at the roots of the two trees, i.e. start$(p_1) = v_1$ and start$(p_2) = u_1$. Then, we can make a distinction between two cases: either $(v_1, v_2, u_1, u_2) \in M$ (a) or $(v_1, v_2, u_1, u_2) \not\in M$ (b). Or in other words, either $p_1 = v_1 v_2, p_2 = u_1 u_2$ holds or not.

(a) First, we consider the case that $(v_1, v_2, u_1, u_2) \in M$. By Lemma 2, we know that there are at least two paths $p_1', p_2' \in \mathcal{P}(T_1)$ contained in $M$ with start$(p_1') = start(p_2') = v_2$. Since $p_1'$ and $p_2'$ go through $v_3$ and $v_4$ and start in $v_2$, we can restrict $M$ to the vertices and edges of $T_1' = T_1([v_2, v_3])$ and $T_2'' = T_2([v_2, v_4])$ and obtain again two path mappings $M', M''$. To see that $M', M''$, are indeed path mappings, we only have to check condition 3 for $p_1'$ and $p_2'$ since only for those two paths the parent paths are removed. However, as they both start in the roots of the corresponding trees, condition 3 is still fulfilled. Hence, we get that

- $\delta_1(T_1, T_2) = \delta_1(T_1', T_2') + \delta_1(T_2'', T_2'') + c(\ell_1(v_2, v_1), \ell_2(u_2, u_1))$

(b) In the second case that $(v_1, v_2, u_1, u_2) \not\in M$, we know that $v_1 v_2$ is not contained in $M$ or $u_1 u_2$ is not contained in $M$, which equivalently means that there is a path $v_1 v_2 v_3 \ldots v_k \in T_1'$ contained in $T_1$ with $k \geq 1$ or there is a path $u_1 u_2 u_3 \ldots u_k \notin T_2''$ contained in $T_2$ in $M$ with $k \geq 1$. If $v_1 v_2 v_3 \ldots v_k \in M$ and $v_1 v_2 v_3 \ldots v_k \notin T_1([v_2, v_4])$, then $T_1'' = T_1([v_2, v_4])$ is not contained in $M$ according to Lemma 1. Let $M''$ be an optimal path mapping between $T_1''$ and the empty tree and $M'$ be an optimal path mapping between $T_1' - T_1''$ and $T_2$. We have $\delta(M''') \subseteq \delta(M)$ since $T_1''$ is not contained in $M$, and edits$(M''') = \delta(M)$ \ edits$(M''')$ since $\ell_1(v_1 v_2 v_3 \ldots v_k) = \ell_1(v_1 v_2 v_3 \ldots v_k)$.

Both together give us

- $\delta_1(T_1, T_2) = \delta_1(T_1'', \perp) + \delta_1(T_1' - T_1'', T_2)$.

The other three cases, $v_1 v_2, u_1 u_2, v_1 v_2 u_1 u_2$ all work symmetrically and in total, we get the six cases from the lemma to show. 

This recursion gives rise to a dynamic programming algorithm which we discuss in the next section.

### 4 IMPLEMENTATION AND EXPERIMENTS

We now present an $O(n^2)$ algorithm for computing $\delta_1$ on binary abstract merge trees. It strongly resembles the dynamic programming for branch mappings in [26]. Although branch mappings and path mappings differ significantly from a theoretic point of view (see Section 5), they are algorithmically closely related. For details on how to adapt the algorithm for non-binary trees, we refer to the techniques used in [29] and [26].

Again, the algorithm is based on identifying subtrees through pairs of nodes. Subtrees rooted in an edge as well as subtrees that are created through subtraction are identified by their root and its unique child. E.g. for a binary abstract merge tree $T$ with root$(T) = v_1$, $(v_2, v_3) \in E(T)$ and $(v_3, v_2), (v_2, v_1) \notin E(T)$, we identify $T([v_2, v_3])$ by $v_2 - T - T_1([v_2, v_3])$ by $v_1$. The recursion in Lemma 5 can be adapted to this notation, which is illustrated in Figure 5 for
Let $d$ be the distance between $n_1$ and $n_2$. Let $c$ be the parent of $n_1$ and $n_2$. If $n_1$ and $n_2$ are leaves, then $d = |n_1 - n_2|$. Otherwise, if either $n_1$ or $n_2$ is a leaf, then $d = c(n_1, n_2)$. If both $n_1$ and $n_2$ are inner nodes, then $d = \min(d_1, d_2, d_3)$, where

1. $d_1 = \min(d_1(n_1, n_2) + \delta_1(c_1, n_1, c_2, n_2))$,
2. $d_2 = \min(d_2(n_1, n_2) + \delta_2(c_1, n_1, c_2, n_2))$,
3. $d_3 = \min(d_3(n_1, n_2) + \delta_3(c_1, n_1, c_2, n_2))$.

for each pair of nodes. By returning the minimum of the six results for inner nodes and adding base cases, we obtain Algorithm 1, which computes the here defined one-degree edit distance for abstract merge trees.

Now consider the running time of Algorithm 1. For two binary abstract merge trees $T_1, T_2$, there are at most $|T_1|^2 \cdot |T_2|^2$ pairs of paths or 4-tuples of vertices. Since the number of subproblems four of the six recursive cases. By returning the minimum of the six results for inner nodes and adding base cases, we obtain Algorithm 1, which computes the here defined one-degree edit distance for abstract merge trees.

The second dataset is a time-varying scalar field consisting of 1001 time steps representing the velocity magnitude of the flow around a cylinder that forms a periodic Kármán vortex street. It was simulated by Weinkauf [24] using the GerrisFlowSolver [15].

Table 1: Running times of the branch mapping distance and path mapping distance on datasets from [26]: The synthetic outlier ensemble (O), the cluster example (C), the heated cylinder (HC) and the vortex street (VS). The sizes of the merge trees are shown in brackets. All times were obtained on a standard workstation with an Intel Core i7-7700 and 64GB of RAM.

| Dataset | HC (10) | C (18) | O (20) | VS (68) | HC (233) |
|---------|---------|--------|--------|---------|----------|
| $d_p$   | 17.5 $\times 10^{-5}$ | 7.0 $\times 10^{-5}$ | 10.4 $\times 10^{-5}$ | 3.5 $\times 10^{-5}$ | 6.8s      |
| $d_f$   | 9.6 $\times 10^{-5}$  | 4.0 $\times 10^{-5}$ | 6.3 $\times 10^{-5}$ | 2.0 $\times 10^{-5}$ | 4.0s      |

Experiments

In the following, we demonstrate the utility of our technique as a basis for typical tasks in visualization. We apply the new distance to two datasets that were also used in [26]. The basis for these experiments is a C++ implementation of Algorithm 1. The merge trees were computed using TTK [23]. Our implementation is publicly available on Github [25]. Computation times of single distances for the here used simplified trees were of the same order of magnitude as the closely related branch mapping distance and follow the theoretical bounds. However, the path mapping distance performed slightly better with a speed-up factor of 1.7 on average. Table 1 shows the comparison in more detail. This speedup is due to the simplified branching of the recursion, specifically in the case of deletions of whole subtrees, since the path mapping distance does not have to try all branch decompositions in this case and can just add up all edge persistences.

The first dataset on which we apply our new distance is the outlier ensemble from [26]. It consists of 20 scalar fields with merge trees of 20 nodes. It demonstrates the branch decomposition-independence of a distance measure if no clusters are found except a single outlier. A more detailed discussion on this behavior can be found in [26]. Figure 6 shows that the path mapping distance performs in the expected way and yields very similar results to the branch mapping distance. Furthermore, it can be seen that the results of the two branch decomposition-independent distances differ significantly from the results using fixed branch decompositions, since they do not show false clusters and are therefore able to identify the outlier clearly.

The second dataset is a time-varying scalar field consisting of 1001 time steps representing the velocity magnitude of the flow around a cylinder that forms a periodic Kármán vortex street. It was simulated by Weinkauf [24] using the GerrisFlowSolver [15].
We begin by listing alternative edit operations on merge trees. We are interested in: we are usually not interested in the distance for merge trees [14] or the branch mapping distance [26].

The remaining operations are shrinking and stretching. Due to the strong correspondence to classic edit operations on trees, we chose stretching and shrinking of edges to be the natural model. They are local operations, lead to a metric distance, and can transform any merge tree into any other merge tree, i.e. they can also express all other operations. Furthermore, they naturally correspond to deformation retractions and their inverse deformations.

Figure 7 shows distance matrices of the timeline using different edit distances. The periodic pattern is clearly visible, with the same periods identified by the path mapping distance, the branch mapping distance from [26] and the constrained edit distance on BDTs from [19].

5 Discussion

In this section, we discuss how the here introduced edit distance and path mappings compare to other edit distances, especially branch mappings, and also elaborate further on the choice of edit operations.

Edit Operations

We begin by listing alternative edit operations on merge trees. We identified the following operations to be considerable: stretching and shrinking of arcs or branches, shifting of branches or subtrees, and shifting of nodes. Figure 8 shows examples for these three classes. For identifying the operations, we used the following core assumption: the edit operations and their costs should be rooted the scalar function. Typical tools like persistence, Wasserstein metrics or similar concepts, which are used in other distance measures, do exactly this. Note that the here considered edit operations are all variants of relabel operations. Deletions and insertions should behave roughly the same in all models.

Now we first consider node shifts. Although these are based on the scalar function, they do actually not represent the changes that we are interested in: we are usually not interested in the absolute scalar values, but rather the relative ones, i.e. for the topological similarity of two merge trees, we only want to consider the distance to the root or the length/persistence of features. Hence, node shifts are not the operations we want, at least if we use scalar difference as the cost measure. Not using these operations also does not restrict the expressiveness of the edit distance, i.e. all node shifts can also be expressed as a sequence of branch/subtree shifts or a sequence of stretch and shrink operations.

Next, we consider branch shifts and subtree shifts. They can be expressed through branch based edit mappings and are, in fact, closely related to branch based edit distances like the Wasserstein distance for merge trees [14] or the branch mapping distances [26]. It is actually possible to define a base metric such that the branch mapping distance exactly represents an edit distance based on these operations. However, this edit distance differs from typical ones in various ways.

First, we have to restrict the sequences of edit operation to those that only touch a node once. Since a node in a merge tree can belong to multiple branches, it can be modified multiple times through edit operations using different branches. This leads to the problem that these sequences would no longer correspond to the mappings, hence, we need to restrict them. Second, in contrast to classic edit distances, the branch based operations are not local ones. Typical edit distances modify only one node or edge locally, whereas branch based operations modify a complete branch and, depending on the definition, also the descending branches. One could argue that those two problems are only aesthetic ones, but they are actually the core of a third problem, which leads to disadvantages in practice: an edit distance based on branch shifts is not a metric (if the branch decomposition-independent variant is chosen) or depending on a fixed BDT (see [26] for a detailed discussion on this problem). Intuitively, the reason for this is that through the use of different branch decompositions, it can actually be cheaper to go over an intermediate tree than using the optimal branch mapping between the original and resulting tree, which contradicts the triangle inequality. A formal proof can be found in [26], where they actually use the shifting base metric. To conclude this argument, for branch shifts we have to chose between the metric property and losing the correspondence to mappings (which also means efficient computability).

The comparison between the various edit operations is only aesthetic if they do not suffer from the problems shown in Figure 2, i.e. they are well-defined on merge trees.

In contrast to branch mappings, path mappings lead to a metric distance function.

In contrast to typical edit distances on BDTs, path mappings are branch decomposition-independent.

Comparison

We now compare the here introduced edit distance and path mappings to previous methods. As mentioned in the discussion on edit operations, branch mappings capture, in essence, optimal mappings achieved through shifts of branches in a merge tree. Hence, other methods that are based on fixed BDTs (e.g. those from [14, 16, 19]) do the same, but only allow shifts of certain branches from a fixed decomposition. In contrast to that, path mappings capture edge based operations like stretching or shrinking. The two operation sets differ significantly from a theoretic point of view which shows in the fact that one leads to a metric while the other does not. Furthermore, the intuitions behind the operations are also completely different, as one of them is a local operation while the other one operates on global structures in a merge tree. This is an interesting observation considering the fact that algorithmically, i.e. in their recursive structure, path mappings and branch mappings are almost identical. To sum up, path mappings show the following behavior:

- In contrast to classic edit operations, path mappings do not suffer from the problems shown in Figure 2, i.e. they are well-defined on merge trees.
- In contrast to branch mappings, path mappings lead to a metric distance function.
- In contrast to typical edit distances on BDTs, path mappings are branch decomposition-independent.
- They are less efficient to compute than classic edit distances (either on merge trees or BDTs).
Based on these properties, path mappings can be seen as an alternative or an improvement for branch mappings that has the same advantages over classic edit distances, but also the same increased complexity. Although the improvements over branch mappings seem to be of only theoretical nature (our experiments did not show improved results on the studied datasets), they actually are of practical importance since they allow us to use the distance in more advanced analysis methods. For example, many clustering methods rely on the metric property to obtain meaningful clusters. Furthermore, due to the metric property, path mappings could be used to compute geodesics and barycenters of merge trees using similar techniques to those from [14], which we believe to not be possible or at least much harder with branch mappings. However, we leave this integration for future work. We should note that due to the nature of constrained edit distances, the path mapping distance is susceptible to the same saddle-swap instabilities as other merge tree edit distances in [14, 16, 19, 26], but it is possible to apply the typical preprocessing to reduce this problem.

6 Conclusion

In this paper, we defined a new edit distance for merge trees based on geometric operations on the continuous object, that resembles an intuitive adaptation of classic tree edit distances to merge trees much closer than branch based methods. We summarized its advantages and limitations in Section 5 and presented a short demonstration of its utility in practice in Section 4. We also provide an open-source implementation publicly available on GitHub.

In future work, we want to study stability properties of the new distance (specifically comparing the unconstrained and one-degree versions in this regard), parallel algorithms for more practical running times on complex datasets, and a possible adaptation to contour trees. Furthermore, the path mapping distance could be integrated in more advanced edit distance-based visualization techniques such as barycenter merge trees [14] or alignments [11].

Acknowledgments

The authors wish to thank the anonymous reviewers for their helpful comments and suggestions, and furthermore Markus Anders, Heike Leitte, Jonas Lukaszczyk and Marvin Petersen for their valuable input. This research was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 442077441.

References

[1] U. Bauer, B. Di Fabio, and C. Landi. An edit distance for Reeb graphs. In Proceedings of the Eurographics 2016 Workshop on 3D Object Retrieval, 3DOR ’16, pp. 27–34. Eurographics Association, Goslar Germany, Germany, 2016. doi: 10.2312/3dor.20161084

[2] U. Bauer, X. Ge, and Y. Wang. Measuring distance between Reeb graphs. Proceedings of the Annual Symposium on Computational Geometry, 07 2013. doi: 10.1145/2582112.2582169

[3] K. Beketayev, D. Yeliussizov, D. Morozov, G. H. Weber, and B. Hamann. Measuring the distance between merge trees. In P. Bremer, I. Hotz, V. Pascucci, and R. Peikert, eds., Topological Methods in Data Analysis and Visualization III, Theory, Algorithms, and Applications, pp. 151–165. Springer, 2014. doi: 10.1007/978-3-319-04099-8_10

[4] P. Billi. A survey on tree edit distance and related problems. Theoretical Computer Science, 337(1-3):217–239, 2005. doi: 10.1016/j.tcs.2004.12.030

[5] H. Carr. Topological manipulation of isosurfaces. PhD thesis, University of British Columbia, 2004.

[6] M. Carrière and S. Oudot. Local equivalence and intrinsic metrics between reeb graphs. In B. Aronov and M. J. Katz, eds., 33rd International Symposium on Computational Geometry, SoCG, 2017, July 4-7, 2017, Brisbane, Australia, vol. 77 of LIPIcs, pp. 25:1–25:15. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2017. doi: 10.4230/LIPIcs.SoCG.2017.25

[7] H. Edelsbrunner and J. Harer. Computational Topology - An Introduction, American Mathematical Society, 2010.

[8] C. Heine, H. Leitte, M. Hlawitschka, F. Lurich, L. D. Floriani, G. Scheuermann, H. Hagen, and C. Garth. A survey of topology-based methods in visualization. Comput. Graph. Forum, 35(3):643–667, 2016. doi: 10.1111/cgf.12933

[9] T. Jiang, L. Wang, and K. Zhang. Alignment of trees - an alternative to tree edit. In M. Crochemore and D. Gusfield, eds., Combinatorial Pattern Matching, 5th Annual Symposium, CPM 94, Asilomar, California, USA, June 5-8, 1994. Proceedings, vol. 907 of Lecture Notes in Computer Science, pp. 75–86. Springer, 1994. doi: 10.1007/3-540-58094-8_7

[10] A. P. Lohfink, F. Gartzky, F. Wetzels, L. Vollmer, and C. Garth. Time-varying fuzzy contour trees. In 2021 IEEE Visualization Conference, IEEE VIS 2021 - Short Papers, New Orleans, LA, USA, October 24-29, 2021, pp. 86–90. IEEE, 2021. doi: 10.1109/VIS49827.2021.9623286

[11] A. P. Lohfink, F. Wetzels, J. Łukaszczyk, G. H. Weber, and C. Garth. Fuzzy contour trees: Alignment and joint layout of multiple contour trees. Comput. Graph. Forum, 39(3):343–355, 2020. doi: 10.1111/cgf.13985

[12] D. Morozov, K. Beketayev, and G. Weber. Interleaving distance between merge trees. Discrete and Computational Geometry, 49(22-45):52, 2013.

[13] V. Narayanan, D. M. Thomas, and V. Natarajan. Distance between extremum graphs. In S. Liu, G. Scheuermann, and S. Takahashi, eds., 2015 IEEE Pacific Visualization Symposium, PacificVis 2015, Hangzhou, China, April 14-17, 2015, pp. 263–270. IEEE Computer Society, 2015. doi: 10.1109/PACIFICVIS.2015.7156386

[14] M. Pont, J. Vidal, J. Delon, and J. Tierny. Wasserstein distances, geodesics and barycenters of merge trees. IEEE Trans. Vis. Comput. Graph.
[15] S. Popinet. Free computational fluid dynamics. *ClusterWorld*, 2(6), 2004.

[16] H. Saikia, H. Seidel, and T. Weinkauf. Extended branch decomposition graphs: Structural comparison of scalar data. *Comput. Graph. Forum*, 33(3):41–50, 2014. doi: 10.1111/cgf.12360

[17] S. M. Selkow. The tree-to-tree editing problem. *Inf. Process. Lett.*, 6(6):184–186, 1977. doi: 10.1016/0020-0190(77)90064-3

[18] V. Silva, E. Munch, and A. Patel. Categorified Reeb graphs. *Discrete and Computational Geometry*, 55, 01 2015. doi: 10.1007/s00454-016-9763-9

[19] R. Sridharamurthy, T. B. Masood, A. Kamakshidasan, and V. Natarajan. Edit distance between merge trees. *IEEE Trans. Vis. Comput. Graph.*, 26(3):1518–1531, 2020. doi: 10.1109/TVCG.2018.2873612

[20] R. Sridharamurthy and V. Natarajan. Comparative analysis of merge trees using local tree edit distance. *IEEE Transactions on Visualization and Computer Graphics*, pp. 1–1, 2021. doi: 10.1109/TVCG.2021.3122176

[21] K. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979. doi: 10.1145/322139.322143

[22] D. M. Thomas and V. Natarajan. Symmetry in scalar field topology. *IEEE Trans. Vis. Comput. Graph.*, 17(12):2035–2044, 2011. doi: 10.1109/TVCG.2011.236

[23] J. Tierny, G. Favelier, J. A. Levine, C. Gueunet, and M. Michaux. The topology toolkit. *IEEE Trans. Vis. Comput. Graph.*, 24(1):832–842, 2018. doi: 10.1109/TVCG.2017.2743938

[24] T. Weinkauf and H. Theisel. Streak lines as tangent curves of a derived vector field. *IEEE Trans. Vis. Comput. Graph.*, 16(6):1225–1234, 2010. doi: 10.1109/TVCG.2010.198

[25] F. Wetzels, H. Leitte, and C. Garth. Branch decomposition-independent edit distances (supplementary source code). https://github.com/scivislab/bdi-ed, 2021.

[26] F. Wetzels, H. Leitte, and C. Garth. Branch decomposition-independent edit distances for merge trees. *Computer Graphics Forum*, 41(3):367–378, 2022. doi: 10.1111/cgf.14547

[27] L. Yan, T. B. Masood, R. Sridharamurthy, F. Rasheed, V. Natarajan, I. Hotz, and B. Wang. Scalar field comparison with topological descriptors: Properties and applications for scientific visualization. *Comput. Graph. Forum*, 40(3):599–633, 2021. doi: 10.1111/cgf.14331

[28] L. Yan, Y. Wang, E. Munch, E. Gasparovic, and B. Wang. A structural average of labeled merge trees for uncertainty visualization. *IEEE Trans. Vis. Comput. Graph.*, 26(1):832–842, 2020. doi: 10.1109/TVCG.2019.2934242

[29] K. Zhang. A constrained edit distance between unorderd labeled trees. *Algorithmica*, 15(3):205–222, 1996. doi: 10.1007/BF01975866

[30] K. Zhang, R. Statman, and D. E. Shasha. On the editing distance between unordered labeled trees. *Inf. Process. Lett.*, 42(3):133–139, 1992. doi: 10.1016/0020-0190(92)90136-J