Computing a Stable Distance on Merge Trees

Brian Bollen, Pasindu Tennakoon, and Joshua A. Levine

Abstract—Distances on merge trees facilitate visual comparison of collections of scalar fields. Two desirable properties for these distances to exhibit are 1) the ability to discern between scalar fields which other, less complex topological summaries cannot and 2) to still be robust to perturbations in the dataset. The combination of these two properties, known respectively as stability and discriminativity, has led to theoretical distances which are either thought to be or shown to be computationally complex and thus their implementations have been scarce. In order to design similarity measures on merge trees which are computationally feasible for more complex merge trees, many researchers have elected to loosen the restrictions on at least one of these two properties. The question still remains, however, if there are practical situations where trading these desirable properties is necessary. Here we construct a distance between merge trees which is designed to retain both discriminativity and stability. While our approach can be expensive for large merge trees, we illustrate its use in a setting where the number of nodes is small. This setting can be made more practical since we also provide a proof that persistence simplification increases the outputted distance by at most half of the simplified value. We demonstrate our distance measure on applications in shape comparison and on detection of periodicity in the von Kármán vortex street.

Index Terms—Merge trees, scalar fields, distance measure, stability, edit distance, persistence

1 INTRODUCTION

Topological descriptors in topological data analysis (TDA) have been used extensively to identify and summarize features of interest in scalar fields in a wide variety of domains such as nuclear energy [16], turbulent mixing [28], shape analysis [27], porous materials [25], combustion [8], and chemistry [22]. Most topological descriptors fall into one of three categories: the set-based descriptors, such as the persistence diagram [11, 19]; the graph-based descriptors such as the Reeb graph [32, 33, 41], contour tree [9], and merge tree; and the complex-based descriptor such as the Morse-Smale complex [17, 24].

In a visualization setting, we are often posed with the question of how similar two datasets are to one another. Since these topological descriptors have been used for individual analysis of features in the dataset, we can use measures of similarity between the topological descriptors to produce a similarity measure between the underlying datasets [43]. For instance, on persistence diagrams, distances such as the bottleneck distance and Wasserstein distance have been used effectively to gauge this similarity. Graph-based distances have also seen a wide variety of distances such as the interleaving [12, 29], functional distortion [4], universal [5], and multiple edit distances [3, 14, 15]. These graph-based and set-based descriptor distances have all been proven to be stable—a property which indicates that the distance is robust to perturbations of the underlying dataset. Furthermore, the graph-based distances have all been shown to be more discriminative.
than the bottleneck distance, i.e. they can discern between differences in the datasets which the bottleneck distance may not. The combination of retaining stability and discriminativity to the bottleneck distance makes graph-based distances desirable.

However, these theoretical distances all have related problems which imply that these distances are computationally complex. For example, the functional distortion distance is a version of the Gromov-Hausdorff distance [21] which is known to be NP-hard to approximate within a factor of 3 [1]. Similarly, the Reeb graph edit distance is heavily related to the graph edit distance (GED) which is known to be NP-hard [20, 45] and determining if two Reeb graphs have an interleaving distance of $\epsilon$ is known to be NP [12].

In order to construct similarity measures which are computationally feasible, researchers have constructed new distances on the graph-based descriptors (most notably the merge tree) which have ultimately loosened the restrictions on stability while attempting to instead only maintain discriminativity to the bottleneck distance [6, 34, 36]. These distances sacrifice theoretical properties for computational feasibility. In this work, we ask if it is possible to construct a distance on merge trees that is still practical to use despite bounded computational complexity. We couple this with an approximation bound on this distance based on persistence simplification.

1.1 Contributions

We choose to focus our efforts on constructing a distance which can experimentally be shown to retain stability and discriminativity. To the best of our knowledge, this is the first distance with an implementation which is shown to be both discriminative and stable. Instead of designing a distance for all graph-based descriptors, we follow the lead of several other experimental distances [6, 35, 36] and focus on the simplest – the merge tree. Our algorithm matches the features of one merge tree to another and computes a cost of this matching which is heavily inspired by the universal distance [2].

More specifically, this work will contribute the following:

- Define an extended semipseudometric on merge trees by first encoding the features of the merge tree using branch decomposition trees;
- Construct an algorithm for this distance which utilizes the A*-search algorithm to find a matching between vertices of two branch decomposition trees;
- Prove that persistence simplification of the dataset increases our distance by at most half the simplified value – allowing us to move larger datasets into more practical settings;
- Experimentally show that this distance is stable and more discriminative than the bottleneck distance while still retaining a similar “largest feature difference” approach to similarity measuring;
- Show the usefulness of stable, discriminative distances on several datasets.

2 RELATED WORK

2.1 Graph-based Topological Descriptors

Graph-based topological descriptors include merge trees (sometimes specifically referred to as split trees or join trees), Reeb graphs, contour trees, and mapper graphs. Each descriptor is designed to show the changes of the topological structure in the underlying dataset. Reeb graphs are, arguably, the most complex descriptor in this family. Reeb graphs contract each component of each level set into a single point. The contour tree is simply the Reeb graph defined on a simply connected domain – making the contour tree a well-defined tree rather than a directed multigraph. Merge trees are then the simplest (both in structure and in computational cost) of these in that it encodes the sublevel (or superlevel) set topology rather than the levelset topology.

The visualization community has a long history of providing effective computations of these descriptors as well as using them for data analysis. Heine et al. recently surveyed many of their uses [26]. In this section, we highlight some of the more recent works as they relate to applications of level set topology, rather than providing an exhaustive survey. Oesterling et al. construct topological landscapes of high-dimensional point clouds using join trees [30]. Bremer et al. capture the behavior of turbulent mixing by developing hierarchical techniques for merge trees [8]. Thomas et al. explore symmetry detecting using contour trees [39]. Widanagamaachchi et al. study atmospheric phenomena by constructing a tracking on merge trees [42]. Yan et al. compute a structural average of merge trees for understanding statistical properties of collections [44].

2.2 Distances on Merge Trees

Stability of merge trees was proven when the interleaving distance between merge trees was introduced [29]. Afterwards, functional distortion distance was introduced for Reeb graphs [4], the interleaving distance was extended to Reeb graphs [12], and several edit distances were introduced for Reeb graphs [5, 14, 15]. While the Reeb graphs are inherently different summaries of the scalar field, merge trees are still a 1-dimensional graph and thus many of the definitions introduced in these works can be applied directly to merge trees. Researchers have actually shown the equivalence of interleaving, functional distortion, and the universal distance on merge trees [2]. Each of these aforementioned distances have been proven to be both stable and discriminative to the bottleneck distance [4, 5, 7, 12, 14, 15, 29].

Unfortunately, implementations of these distances have been scarce due to their computational complexity. In order to have distances which are practical, other researchers have focused their efforts on defining distances specifically on merge trees due to their simplicity. As stated before, these distances loosen the restriction on either stability or discriminativity in order to have distances which are computationally feasible. To avoid confusion, we will call the collection of distances consisting of the interleaving, functional distortion, and universal distance as the theoretical merge tree distances. We call the collection of distinct distances we discuss below the experimental merge tree distances.

Sridharamurthy et al. introduced an edit distance between merge trees which is experimentally shown to be more discriminative than both the bottleneck and 1-Wasserstein distance [36]. This distance loosens the restriction on stability which makes it computationally feasible. Cases of instability are still addressed by introducing an adjustable parameter which combines saddles which are within the parameters value – simplifying the topology of the merge tree. The distance was also proven to be a well-defined metric on the space of merge trees. This idea was later expanded upon with the introduction of the local merge tree edit distance – a well-defined metric specifically designed to study the local similarities at multiple resolutions rather than providing a global measure [37].

Beketayev et al. provides a computation of a similarity measure for merge trees by first computing all of its branch decomposition trees – data structures which encode features of the merge tree as nodes in a new tree – and then finding pairwise matchings between these trees. The matching imposes a restriction that if $x$ matches to $y$ and $x'$ is a child of $x$, then $x'$ must be matched to a child of $y$ (or be deleted). This is similar to the ‘ancestor preserving’ restriction imposed by standard tree edit distance (TED) [38]. Our distance that we propose in Sect. 4 similarly uses branch decomposition trees in order to encode the feature of a merge tree. We divert from this work by removing the ‘ancestor preserving’ restriction which allows us to maintain stability of our distance.

Saikia et al. [34, 35] produces a similar distance to the one defined by Beketayev et al. They introduce a dynamic programming algorithm to create an extended branch decomposition tree – a data structure which encodes similar data to the conglomerate of all possible branch decomposition trees without having to store all these possibilities in memory. They show the application of this distance on self-similarity of scalar fields and detecting periodicity in time-varying datasets.
3 Technical Background

3.1 Scalar Fields and Merge Trees

To ensure that our resulting structures are well-behaved, we elect to focus our attention towards piecewise linear scalar fields: scalar fields in which the domain $\mathbb{X}$ is triangulable and the function $f$ is piecewise linear. Furthermore, we will focus our work on scalar fields in which the domain is a simply connected, two-dimensional manifold and the function $f$ is a simple Morse function [18]. These conditions ensure that the merge tree is a well-defined, one-dimensional graph [12].

**Definition 1.** A scalar field (equivalently an $\mathbb{R}$-space) is a pair $(\mathbb{X}, f)$ where $\mathbb{X}$ is a topological space and $f : \mathbb{X} \to \mathbb{R}$ is a continuous real-valued function.

**Definition 2.** A sublevel set of $\mathbb{X}$ at $a \in \mathbb{R}$, denoted as $\mathbb{X}_a$, is the pre-image of the set $(-\infty, a]$ under $f$. Similarly, a superlevel set of $\mathbb{X}$ at $a$ is $f^{-1}[a, \infty)$ and is denoted as $\mathbb{X}^a$.

**Definition 3.** We define an equivalence relation $\sim_f$ on $\mathbb{X}$ by stating that $x \sim_f y$ if $x, y \in \mathbb{X}^a$ and $x$ and $y$ both lie in the same connected component of the superlevel set. We define $\mathbb{X}_f$ to be the quotient space $\mathbb{X} / \sim_f$ and define $\hat{f} : \mathbb{X}_f \to \mathbb{R}$ to be the restriction of $f$ to the domain $\mathbb{X}_f$. The pair $\mathbb{F} := (\mathbb{X}_f, \hat{f})$ is called the split tree of $(\mathbb{X}, f)$. The join tree $\mathbb{J}$ is defined analogously using sublevel sets rather than superlevel sets. The split and join tree make up the class of merge trees. We denote a general merge tree of the scalar field $(\mathbb{X}, f)$ as $\mathcal{M}_f$.

In what follows, we will be working solely with the split tree. Some definitions, theorems, and parts of our algorithm work for both the join and split tree, while others are specific to the split tree due to aspects such as increasing paths from saddle to extremum rather than decreasing paths. However, if we were to negate the original function defined on the split tree, we could provide a distance for the join tree as well. To this end, we will elect to use the term merge tree and use the notation $\mathcal{M}_f$ for a merge tree defined on a scalar field $(\mathbb{X}, f)$.

Since merge trees can be considered as labeled graphs, we will often denote the vertices and edges of $\mathcal{M}_f$ as $V(\mathcal{M}_f)$ and $E(\mathcal{M}_f)$, respectively. From this, we can define the notion of merge tree isomorphism.

**Definition 4.** Two merge trees $\mathcal{M}_f$ and $\mathcal{M}_g$ are isomorphic if there exists a bijection $\alpha : V(\mathcal{M}_f) \to V(\mathcal{M}_g)$ such that 1) the edge $e(u, u') \in E(\mathcal{M}_f)$ if and only if $e(\alpha(u), \alpha(u')) \in E(\mathcal{M}_g)$ and 2) for every $u \in \mathcal{M}_f$, we have $f(u) = g(\alpha(u))$.

### 3.2 Persistence Diagrams and Bottleneck Distance

Instead of defining the persistence diagram on the scalar field, we elect to define the persistence diagram by using the merge tree as a scalar field itself since it reduces the number of classes of points in the persistence diagram and overall makes the comparison between distances on persistence diagrams and distances on merge trees simpler; see Bollen et al. [7] for a discussion on defining the persistence diagram of graph-based descriptors. For the sake of brevity, we show how to construct a persistence diagram from a merge tree and its properties rather than theoretical definitions.

The persistence diagram $\text{Dgm}(\mathcal{M}_f)$ of a merge tree $\mathcal{M}_f$ is a multiset of points $(a, b)$ which each represent a pair of vertices of the merge tree. These pairs intuitively represent different features of the merge tree. To determine which vertices are paired together, we introduce the elder rule.

**Definition 5.** The elder rule is a pairing scheme between saddles and extrema of a merge tree that says $x$ is paired with $y$ if there exists a monotone increasing path from $y$ to $x$ and if for all saddles $y'$ on the same path with $f(y') > f(y)$, they are paired with an extrema $x'$ such that $f(x') < f(x)$. The persistence diagram is a multiset $\text{Dgm}(\mathcal{M}_f)$ where $(f(x), f(y)) \in \text{Dgm}(\mathcal{M}_f)$ if $x, y$ are a saddle extremum pair based on the elder rule with the addition of the pair $(f(g_1), f(g_2))$ where $g_1$ is the global minimum and $g_2$ is the global maxima.

### 3.3 Branch Decomposition Trees

The branch decomposition tree (BDT) is a data structure which, in topological data analysis, attempts to pair the saddles of contour trees or merge trees to the extrema of that tree. Each node in the BDT would then represent a feature of the original scalar field. Pascucci used the branch decomposition trees to inform a layout for complex contour trees with many self-intersections [31]. Since then, BDTs have seen additional use as representations of merge trees for their comparison [6, 35].

Each merge tree or contour tree has precisely $2^n - 1$ different possible BDTs, where $n$ is the number of nodes [6]. Often, a unique BDT is constructed by weighting the choice of pairing based on a particular measurement – such as persistence of the branch or the number of voxels of the branch in the scalar field [35].

**Definition 7.** A branch is a monotone (in function value) path traversing a sequence of nodes in the merge tree $\mathcal{M}_f$. The first and last nodes of this sequence are called the endpoints of the branch.

**Definition 8.** A branch decomposition of a merge tree is a set of branches such that every edge $e \in E(\mathcal{M}_f)$ appears in exactly one branch.

**Definition 9.** A branch decomposition of a merge tree is a hierarchical decomposition if (1) there is exactly one branch which connects two extrema to one another (called the root branch) and (2) every other branch connects an extremum to a node that is interior to another branch.

**Definition 10.** Let $H_f$ be a hierarchical decomposition of a merge tree $\mathcal{M}_f$. The branch decomposition tree (BDT), $b_f$, with respect to $H_f$ is a rooted tree $b = (V, E)$ where $v = (v_1, v_2) \in V$ represent the branches of $H_f$. The edge $e(v, u) \in E$ if and only if $u$ has an endpoint interior to the branch $v$.

Persistence diagrams have been shown to be stable under the well-studied bottleneck distance [10, 11]. The bottleneck distance assigns a cost to a matching between the points of two persistence diagrams. We allow for each point to also be matched to an empty node which can be thought of as deleting or inserting that feature.

**Definition 6 (Bottleneck Distance).** Let $D_1, D_2$ be two persistence diagrams and let $\lambda$ denote an empty node. We define $D_1 := D_1 \cup \{\lambda\}$. A matching $M$ between $D_1$ and $D_2$ is a binary relation $M \subseteq D_1 \times D_2$ such that each element from $D_1$ and $D_2$ appear in exactly one pair $(x, y) \in M$.

The cost of a pair $(x, y) \in M$ is defined as

$$c(x, y) = \begin{cases} \max\{|x_1 - y_1|, |x_2 - y_2|\} & x \in D_1, y \in D_2 \\
\frac{|y_1 - y_2|}{|x_1 - x_2|} & x \in D_1, y = \lambda \\
\frac{|y_1 - y_2|}{|x_1 - x_2|} & x = \lambda, y \in D_2 \end{cases}$$

The cost of a matching $M$, denoted as $c(M)$, is then the largest cost of all pairs in the matching.
For every hierarchical decomposition of a merge tree \( M_f \), we obtain a unique BDT \( b_f \). Each node \( u \in b_f \) corresponds to two vertices of \( M_f \). If \( u \in b_f \) is not the root node, then there is a corresponding saddle \( u_s \in M_f \) and a corresponding maxima \( u_m \in M_f \). The root node \( e \in b_f \) corresponds to the global minimum \( r_s \in M_f \) and a maxima \( r_m \in M_f \). We denote the set of all possible branch decomposition trees of a merge tree \( M_f \) as \( B_f \). Fig. 3 shows the eight different BDTs for a merge tree with eight nodes.

### 3.4 Stability and Discriminativity

**Definition 11.** A distance \( d \) defined merge trees is said to be stable if and only if

\[
d(M_{f_1}, M_{f_2}) \leq ||f_1 - f_2||_\infty,
\]

where \( f_1, f_2 \) are the corresponding functions for the scalar fields of \( M_{f_1} \) and \( M_{f_2} \), and \( M_{f_1} \) and \( M_{f_2} \) are defined on the same domain \( \mathcal{X} \).

Stability of a distance guarantees that point-wise perturbations introduced into the dataset will not drastically change the merge tree. Saikia et al. [34] defined two different types of instabilities which are exhibited in merge trees: horizontal instabilities and vertical instabilities.

**Definition 12.** Let \( (\mathcal{X}, f) \) be a scalar field with respective merge tree \( M_f \) such that there exists a pair \( s_1, s_2 \in V(M_f) \), with \( \deg (s_1) = \deg (s_2) = 3 \) and such that \( |f(s_1) - f(s_2)| < 2\epsilon \). If \( e(s_1, s_2) \in E(M_f) \), then \( (\mathcal{X}, f) \) is horizontally \( \epsilon \)-unstable.

**Definition 13.** Let \( (\mathcal{X}, f) \) be a scalar field with respective merge tree \( M_f \) such that there exists a pair of vertices \( m_1, m_2 \in V(M_f) \), with \( \deg (m_1) = \deg (m_2) = 1 \) and such that \( |f(m_1) - f(m_2)| < 2\epsilon \). Let \( (s_1, m_1) \in Dgm(M_f) \) and \( (s_2, m_2) \in Dgm(M_f) \) be the persistence pairs corresponding to \( m_1 \) and \( m_2 \), for some \( s_1, s_2 \in V(M_f) \). If there exists monotone paths \( \rho_1: s_1 \rightarrow m_2 \) and \( \rho_2: s_2 \rightarrow m_1 \), then \( (\mathcal{X}, f) \) is vertically \( \epsilon \)-unstable.

For our algorithm, we will use branch decomposition trees to organize the features of the scalar field. Due to this setup, we may have a situation where a small change in the function values of extrema switches their total ordering – possibly altering the topology of BDTs. This is called a vertical instability. Fig. 4 depicts the affects of perturbations in a vertical and horizontal manner on a function \( f \).

**Definition 14.** A distance \( d \) defined on merge trees is said to be more discriminative than a baseline distance \( d_0 \) if there exist some constant \( c > 0 \) such that

\[
d_0(M_{f_1}, M_{f_2}) \leq c \cdot d(M_{f_1}, M_{f_2}),
\]

for all merge trees \( M_{f_1}, M_{f_2} \), and if there does not exist a constant \( c' \) such that \( d_0 = c' \cdot d \).

If a similarity measure is strictly bounded below by a baseline distance (up to a constant \( c \)), then there are cases in which the baseline distance is not able to discern between two merge trees while the distance \( d \) does detect some dissimilarity. Furthermore, this implies that if the distance \( d \) detects no difference between two merge trees, then the baseline will not detect any difference as well.

A core position on discriminativity being desirable is that we expect these merge tree distances to inherently be more computationally complex than persistence diagram distances since the graph-based descriptors are strictly more complex than set-based descriptors. Thus, these merge tree distances will trade off their computational efficiency for encoding more information in the similarity measure. As with the theoretical merge tree distances, we will use the bottleneck distance as our baseline since the theoretical merge tree distance and bottleneck distance all use a "max-feature-difference" approach to similarity measuring.

A similar notion to discriminativity is isomorphism invariance.

**Definition 15.** A distance \( d \) on merge trees is isomorphism invariant if

\[
d(M_{f_1}, M_{g_1}) = 0 \text{ if and only if } M_{f_1} \text{ and } M_{g_1} \text{ are merge tree isomorphic}.
\]

It has been shown that the bottleneck distance is not isomorphism invariant on the space of merge trees while all of the theoretical graph-based distances are [7]. The merge tree edit distance is also isomorphism invariant [36].

### 3.5 Zigzag Diagrams

The universal distance (originally referred to as the Reeb graph edit distance [5]) is a stable, discriminative distance defined on Reeb graphs and merge trees which has been shown to be the largest stable distance defined on Reeb graphs – a property known as universality. On merge trees, it was shown to be equivalent to the interleaving and functional distortion distance [2] – thus making all of these distances universal.

The universal distance is defined by constructing a zigzag diagram of topological spaces which connects a source merge tree \( M_f \) to its target \( M_g \). These zigzag diagrams can be intuitively thought of as a sequence of operations carrying one merge tree to another. We use a simplified version of the zigzag diagram for use with our distance. The term carry is used to state that we are transforming a source merge tree \( M_f \) into a merge tree \( M_g \) which is isomorphic to \( M_g \).

**Definition 16.** Let \( M_f, M_g \) be two merge trees. A zigzag diagram \( Z \) is a sequence of merge trees \( M = \{M_f \rightarrow M_1, M_2, \ldots, M_{n-1}, M_n = M_g \} \) coupled with a sequence of 1-dimensional graphs \( X = \{X_1, \ldots, X_{n-1} \} \) such that for each \( X_i \), there are two valid maps \( q_{i,i+1} : V(X_i) \rightarrow V(M_{i+1}) \) and \( g_{i,i+1} : V(X_i) \rightarrow V(M_{i+1}) \) which respect edge assignments. That is, \( e(x_i, x_j) \in X_i \) then \( e(g_i(x_i), g_{i+1}(x_j)) \in M_{i+1} \). The
sequence $X$ will be called the connecting spaces of $Z$ while $M$ is called the merge trees of $Z$.

Each merge tree $M \in M$ will have an associated function $f_i$. In general, these merge trees need not be Morse. Specifically, we will call merge trees with vertices of degree 4 which is not permitted under the definition of Morse functions on 2-manifolds. The connecting spaces are responsible for changing adjacencies in the merge trees. Fig. 5 depicts two different zigzag diagrams. The first carries a merge tree $M_f$ to $M_g$, while the bottom diagram is in reverse order.

**Definition 17.** The limit $L$ of a zigzag diagram $Z$ with $n$ merge trees is the $(n-1)$-dimensional space where $x = (x_1, \ldots, x_{n-1}) \in L$ if $q_{i+1}(x_i) = q_{i+1}(x_{i+1})$ for all $x_i$. We define $y_i = x_i = q_{i+1}(x_i)$.

**Definition 18.** The spread $S$ of an element $x \in L$ is the difference between the maximum function value and minimum function value it attains in the zigzag diagram. That is,

$$S(x) = \max_{i=1, \ldots, n} f_i(y_i) - \min_{i=1, \ldots, n} f_i(y_i).$$

The cost of the zigzag diagram $Z$ is then the largest spread of its limit $L$.

$$c(Z) = \max_{x \in L} S(x).$$

When the choice of zigzag diagram is not clear, we use the notation $L(Z)$ to denote the limit of the zigzag diagram $Z$. See Appendix A.2 for another example of a zigzag diagram with the corresponding spread.

### 4 Merge Tree Matching Distance

The bottleneck distance constructs a similarity measure between scalar fields by 1) effectively encoding the features of the scalar field as a multiset of points, 2) constructing a way to match the encoded features of one scalar field to the features of another, 3) computing a cost on this matching by computing the largest difference between two matched features, and 4) taking the distance to be the lowest cost over all possible matchings. The theoretical merge tree distances can be thought of in a similar fashion. For example, the universal distance requires a choice of which features to transform into others, provides a way to carry out this transformation by using zigzag diagrams, and then computes a cost of this zigzag diagram [5].

Our distance is motivated by three main objectives:

- create a distance which is similar to the bottleneck distance and the theoretical merge tree distances in that it 1) properly encodes the features of the merge tree, 2) matches features of one merge tree to another, 3) assigns a cost to this matching by computing the largest feature difference, and 4) minimizes this cost over all possible matchings;
- construct it in such a way that it is isomorphism invariant on the set of merge trees as well as being more discriminative than the bottleneck distance; and
- make sure that the distance handles cases of instability correctly.

**Distance Definition**

When constructing a distance between merge trees, we need to make sure that the distance captures the difference based on the relationship between features that are in the original scalar field rather than solely on the paired critical points. We encode this hierarchical relationship between features using the BDT, which also captures topological features (pairs of critical points) as individual vertices in the BDT.

Unlike persistence diagrams, there are many different BDTs for each merge tree. Using only one can lead to vertical instabilities in the distance. Thus, in order to adequately find the distance between two merge trees, enumeration of all the BDTs is needed, similar to Beketayev et al. [6]. Let $M_f, M_g$ be two merge trees with respective sets of BDTs $F_f, F_g$. A matching between a fixed $b_f \in F_f$ and $b_g \in F_g$ gives a matching between the features of $M_f$ and $M_g$.

**Definition 19.** Let $M_f, M_g$ be two merge trees with respective sets of BDTs $F_f, F_g$. Let $b_1 \in F_f, b_2 \in F_g$ be two BDTs and let $\lambda$ be an empty node not in $V(b_1)$ nor $V(b_2)$. We let $x_1, x_2$ denote the extrema and saddle nodes of a vertex $x \in V(b_i)$. We define $b_i := b_i \cup \{\lambda\}$. A matching $M$ between $b_1$ and $b_2$ is a binary relation $M \subseteq b_1 \times b_2$ such that the following conditions hold:

1. $(r_1, r_2) \in M$, where $r_1 \in V(b_1), r_2 \in V(b_2)$ are the respective roots of $b_1, b_2$.
2. Each element in $V(b_1)$ and $V(b_2)$ appear in exactly one pair in $M$.
3. If $(x, \lambda) \in M$, then $(f(x), f(x)) \in Dgm(M_f)$.
4. If $(\lambda, y) \in M$, then $(g(y), g(y)) \in Dgm(M_g)$.

A partial matching $M$ is a matching between BDTs with condition 1) loosened to have each element of $V(b_1)$ and $V(b_2)$ appear in at most one pair of $M$.

Elements of a matching $M$ fall into three different categories: insertion pairs which have the form $(\lambda, v)$, deletion which have the form $(u, \lambda)$, and relabel which have both the form $(u, v)$. If $(u, v) \in M$ and $(u_p, v_p) \notin M$, where $u_p, v_p$ are the parents of nodes $u, v$, then $(u, v)$ is further categorized as a movement relabel pair.

An inserted zigzag diagram is the zigzag diagram which arises from carrying one merge tree to another. This induced zigzag diagram follows the protocol that we apply insertions, non-movement relabels, movement relabels, and then deletions. We apply insertions first and deletions last since we cannot create disconnected merge trees in our zigzag diagram.

Each matching $M$ between two BDTs $b_i \in F_f, b_j \in F_g$ induces two zigzag diagrams: the forward zigzag diagram $Z_i$ from $M_f$ to $M_g$, and the backward zigzag diagram $Z_j$ from $M_g$ to $M_f$. The difference in these two induced zigzag diagrams is when the relabel pairs are applied. Taking the minimum cost over the forward and backward zigzag diagram ensures that we handle the instability in a way that keeps the distance below the $L^\infty$ distance. Fig. 5 depicts an example of the possible difference. Since these merge trees are horizontal and unstable, we need to make sure that our distance is less than or equal to $\epsilon$ for us to have a distance below the $L^\infty$ distance. In this case, $\epsilon = |\tilde{g}(y_1) - \tilde{g}(y_2)|$. Note that the backward zigzag diagram has a spread less than $\epsilon$.

We define the cost of a matching as follows:

**Definition 20.** The cost of a matching $M$ is the minimum value between the costs of the induced zigzag diagrams. That is,

$$c(M) = \min_{Z \in \{Z_i, Z_j\}} c(Z) = \min_{\lambda \in \{L(Z_i), L(Z_j)\}} \max_{x \in L(Z)} S(x).$$

**Definition 21.** The merge tree matching distance is defined as the minimum cost of all matchings between all pairs of BDTs. That is,

$$d_M(M_{f}, M_{g}) = \min_{b_1 \in F_f, b_2 \in F_g, M \subseteq b_1 \times b_2} c(M),$$

where $M_{i,j}$ denotes the set of all possible matchings between $b_i \in F_f$ and $b_j \in F_g$.

**Proposition 22.** The merge tree matching distance is isomorphism invariant. That is, $d_M(M_f, M_g) = 0$ if and only if $M_f$ and $M_g$ are merge tree isomorphic.

**Proof.** Suppose $M_f$ and $M_g$ are isomorphic. Then, each have identical sets of BDTs. Let $b_f, b_g$ be two such identical BDTs of $M_f$ and $M_g$, respectively. We define $M$ to be the matching induced by the isomorphism between $b_f$ and $b_g$. Since they are isomorphic trees, there are no movement relabels in the induced zigzag diagram. Then, the induced zigzag diagrams will both consist of only the single merge tree since $M_f$ and $M_g$ are already isomorphic to one another. Now, suppose that
Thus, vertices by deducing an edit sequence between the graphs. From this values on each of their corresponding saddles and extrema. Thus, each node finds the path \( p_v \) for all maps from the connecting spaces to the respective merge trees. With horizontal instabilities, it is desirable that moving \( x_i \) up to its final position or \( y_j \) down to its final position. \( Z_f \) achieves this cost, while \( Z_b \) achieves a larger cost due to relabeling the saddle downwards to begin with, and then performing the movement.

the merge tree matching distance between \( M_f \) and \( M_g \) is 0. Then, there must exist a pair of BDTs \( b_f \) and \( b_g \) such that the matching induces no deletions, insertions, movement relabels, or relabels which incur a cost. Thus, \( b_f \) and \( b_g \) must be tree isomorphic as well as having the same values on each of their corresponding saddles and extrema. Thus, \( M_f \) and \( M_g \) must also be isomorphic.

5 ALGORITHM

Our algorithm is dependent on finding the cheapest matching between the vertices of all possible BDTs of two input merge trees. We are motivated by the well-studied graph edit distance (GED) in order to solve this. As stated in Section 4, the BDTs allow us to convert merge trees into data structures where the features of interest are now single vertices rather than pairs of vertices. GED has a similar problem statement: given two graphs \( G_1 \) and \( G_2 \), find a matching between their vertices by deducing an edit sequence between the graphs. From this edit distance a cost is computed.

Our algorithm is split into three main components:

1. Construction of all BDTs from the input merge trees \( M_f, M_g \).
2. Finding full matchings between two BDTs \( b_f, b_g \) using the A* algorithm.
3. Computing the cost of a full matching \( M \) between \( b_f \) and \( b_g \) by first constructing the elements of the limit \( L(Z) \).

5.1 Constructing All Branch Decomposition Trees

Suppose we have a merge tree \( M_f \). As an example, we describe how to construct a persistence-based BDT. We first take the local min \( v_0 \) and find the path \( p_{v_0,v_n} \) to the global max \( v_n \). The pair \( (v_0, v_n) \) is placed as the root of the BDT. For every vertex \( v \) along this path, we recursively call the construction of the BDT algorithm with \( v \) being the new root. Each node \( v \) in \( p_{v_0,v_n} \), contributes a single child node to the root node. The algorithm ends when all nodes are paired.

For general BDTs, we must consider every maxima \( v \) that is in the up-path of a root \( u \) as a possible pairing rather than just the global maxima of that branch. In order to avoid redundant computations of the same branches, we add another recursive layer to the BDT algorithm.

Suppose that \((v_i,v_j)\) is a pair corresponding to a branch \( B \) which has just been placed into a BDT \( b \). For computation of a single BDT, we would recursively call this operation for all \( v \in p_{v_0,v_n} \). However, we know that for every node \((v_i,v_j)\), there are multiple different configurations of its children. To avoid redundant computations, we generate a specific number of BDTs for each possible pairing of a saddle \( v \), for each \( v \in p_{v_0,v_n} \). First, assume \( v \) is the only vertex in \( p_{v_0,v_n} \) which is not one of the endpoints. Let \( \{u_1, \ldots, u_n\} \) be the set of extrema in the up-path of \( v \). For each pair \((v_i,u_j)\), we construct a new copy of \( b \) and add \((v_i,u_j)\) as the child of \((v_i,v_j)\).

Now, suppose instead that \( \{v_1, \ldots, v_k\} \) is the set of vertices in \( p_{v_0,v_n} \) which are not the endpoints. We proceed to carry out the same step as if there was only one non-endpoint vertex \( v \), except instead of making \( m \) copies of \( b \), we need to make a copy for the possible combinations of choices of extrema pairing for all \( v_i \in \{v_1, \ldots, v_k\} \). More specifically, if \( m_i \) denotes the number of extrema that \( v_i \) can be paired with, we make \( m_i m_2 \ldots m_{k-1} m_k \) copies of \( b \). Each \( b \) gets a unique set of nodes added as the children of \((v_i,v_j)\).

5.2 Finding the Best Matching Using A*

The A* algorithm maintains a priority queue \( Q \) which holds a list of partial matchings \( M' \) with a current cost \( c(M') \). Since our distance requires the full matching before a true cost can be determined, we use the bottleneck distance between the currently matched nodes in order to under approximate cost. More specifically, let \( P \) be a partial matching between \( b_f \) and \( b_g \). Then \( c(M') = \max_{(u,v) \in M} c(u,v) \), where \( c(u,v) \) is the cost function in Equation 6. As stated earlier, providing an under approximation to the true cost will guarantee that we still reach an optimal solution. However, since our under approximation will not necessarily converge to the true cost when we reach a full matching (unlike standard GED), we have to have an additional step which computes the true cost of the matching and to determine whether to continue finding better matchings dependent on this cost. Appendix A.3 shows an overview of our A* algorithm with this additional module.

The efficiency of A* is heavily dictated by introducing pruning techniques to reduce the number of possible matchings and using a good heuristic function which approximates the future cost along a particular path in the search tree. In our case, we can effectively prune the search space by not matching two nodes to another if it would be cheaper to simply insert or delete both the nodes. As for a heuristic function, we introduce the function \( h(M') \) which finds the lowest possible future cost based on the fact that if \( b_f \) has \( n \) unmatched nodes and \( b_g \) has \( m \) unmatched nodes, we must insert or delete the difference in the number of nodes. See Appendix A.1 for more information on our pruning and heuristic function. As opposed to GED, our “approximate cost” would then be \( \max\{c(M'), h(M')\} \) rather than \( c(M') + h(M') \).

5.3 Computing the Cost by Constructing the Limit \( L(Z) \)

Let \( b_f \) and \( b_g \) be the branch decomposition trees that we are comparing and let \( M \) be the current full matching between them. As stated before, the forward zigzag diagram \( Z_f \) is constructed by applying insertions, non-movement relabels, movement relabels, and then deletions to \( b_f \) which ultimately creates \( b_g \). Just as the graph edit distance can be thought of as constructing edit sequences to carry one graph to another, our distance can be thought of as altering the branch decomposition \( b_f \) with this set of operations in order to construct \( b_g \).

There is a one-to-one mapping between the vertices of any two connecting spaces. Thus, we keep the labeling of each connecting space the same. When we say that \( x_i = x_j \) for two vertices in different connecting spaces, this implies that \( i = j \). We denote the set of vertices in the connecting spaces as \( V(X) \).

Let \( x^i, x^{i+1} \) be vertices of \( X_i, X_{i+1} \), respectively. Note that \( x_j = \{x_j, \ldots, x_j^{i-1}\} \) is an element of \( L(Z) \) for all \( x_j \in V(X) \). Additional elements are added if \( q_{i+1}(X) = q_{i+1}(x_i^{i-1}(x_i^{i+1})) \), for some \( x_i \in X_i \) and \( x_i^{i+1} \in X_{i+1} \) where \( x_i \neq x_i^{i+1} \). We call these swaps. For example, Fig. 5 has a swap in both the forward and backward zigzag diagram. For the forward zigzag diagram, a swap occurs in \( \delta f \) where \( q_{1,2}(x_1) = q_{2,3}(x_2) \). This implies \( \{x_1, x_1, x_2\} \) is also an element of the limit. For every swap, there are two options to continue constructing the limit: continue with the same vertex or move to the swapped vertex.

In what follows, each pair \((u,v) \in M \) would create at least one connecting space and one merge tree. If \((u,v) \) is a movement relabel, we may create more. Each time we would create a merge tree and

---

Fig. 5. Depiction of the forward zigzag diagram \( Z_f \) (top) and the backward zigzag diagram \( Z_b \) (bottom) of a matching between merge trees. The connecting spaces are viewed as 1-dimensional graphs in between each of the merge trees of the sequence. Here, we always have the mapping \( x_i \rightarrow y_i \) for all maps from the connecting spaces to the respective merge trees. With horizontal instabilities, it is desirable that moving the branch \( \varepsilon(y_1, y_2) \) has a cost equal to the largest difference between moving \( y_2 \) up to its final position or \( y_1 \) down to its final position. \( Z_f \) achieves this cost, while \( Z_b \) achieves a larger cost due to relabeling the saddle downwards to begin with, and then performing the movement.
connecting space, instead we create a copy the previous BDT and alter it according to the pair (inserting a node, deleting a node, relabeling, or moving a node). By the end of the algorithm, we are left with a set of BDTs \( \{B_f = b_1, b_2, \ldots, b_{n-1}, b_n = b_g\} \) and a list \( S \) of length \( m \) which will contain our swaps. These two data structures are enough to determine the elements of \( L(Z) \).

To begin, we construct a copy of \( b_g \) and apply all \( n \) insertions. We sort the list of insertions by increasing depth order to make sure that if \( u \) is to be inserted on \( u' \), then \( u' \) already exists. Non-moving relabels are then conducted in no particular order.

Let \( b_1 \) be the current BDT. In a movement relabel, we have a source branch \( u \in b_1 \), its parent \( v \in b_2 \), a target branch \( v \in b_2 \), and the parent of the target \( u \in b_2 \). Let \( u' \) be such that \( (u', v) \in M \) and \( u' \) be such that \( (u', v, u) \in M \). Our goal is then to move \( u \) to have parent \( u' \). We find the path \( p : u \vdash u'_p \). The node closest to the root of \( b_1 \) in depth is known as the intersection. The intersection is the branch where \( u \) does not need to attach to its saddle. For every other branch \( w \) in the path \( p \), we know we must add an additional connecting space and add the swap tuple \( (u'_w, w_w) \), where \( u_w \) and \( w_w \) are saddles of the respective merge trees, to the list. Fig. 6 depicts an example of recording the intersection and swaps for moving a branch.

Movement relabels need to be conducted in an order that makes sure that the BDT stays connected. Our A* algorithm has the restriction that if \( u \) is an ancestor of \( u' \), then \( u \) cannot become a descendant of \( u' \) since it is suboptimal (i.e., another choice of BDT should be used if this is the case). However, once we begin altering the original BDT, there are situations where this \( u \) needs to be moved onto its parent. To alleviate this, we maintain another priority queue which holds all the movements needed to be performed. If \( u \) is the parent of some node \( v \), then we push this pair back into this priority queue with a lower priority (if \( u \) is the case). However, once we begin altering the original BDT, there are situations where this \( u \) needs to be moved onto its parent. To alleviate this, we maintain another priority queue which holds all the movements needed to be performed. If \( u \) is the parent of some node \( v \), then we push this pair back into this priority queue with a lower priority (if \( u \) is the case).

5.4 Putting It All Together

Algorithm 1 shows the pseudocode of how we first choose pairs of branch decomposition trees and then subsequently feed these pairs into our A* algorithm. Note that we introduce a “cutoff” variable which indicates when to stop the A* computation of a pair of BDTs. If the cheapest current matching is ever larger than this cutoff, we stop the computation and move onto the next pair.

Algorithm 1

```
mergeTreeMatchingDistance

Input: Two merge trees \( M_f, M_g \)
Output: Merge tree matching distance between \( M_f, M_g \)
1: \( R_f = \text{ConstructBDTs}(M_f), R_g = \text{ConstructBDTs}(M_g), \text{cutoff} = 0 \)
2: for \( b_f \in R_f \) do
3: for \( b_g \in R_g \) do
4: \( \text{currCost, completeMatch} = \text{aStat}(b_f, b_g, \text{cutoff}) \)
5: if completeMatch == true then
6: \( \text{cutoff} = \text{currCost} \)
7: \( \text{dist} = \text{currCost} \)
8: return dist
```

Theorem 23. Let \( M_f, M_g \) be two merge trees whose distance \( d_M \) is \( A \) and let \( 0 < \varepsilon < A \) be fixed. Then

\[
A \leq d_M(P_f(M_f), P_g(M_g)) \leq A + \frac{1}{2} \varepsilon.
\]

Proof. Let \( b_f, b_g \) be the optimal branch decomposition choices along with the optimal matching \( M \). Suppose \( u = (u_1, u_2) \in V(b_f) \) and \( f(u_1) - f(u_2) \leq \varepsilon \). Then, persistence simplifying \( M_f \) by \( \varepsilon \) removes \( u \) from \( b_f \) and removes the pair \( (u_1, u_2) \in M \), where \( \star \) may be the empty node \( \lambda \) or some node \( v \in V(b_e) \). If \( (u_1, u_2, \lambda) \in M \), then removal of the pair via simplification will not increase the distance since \( |f(u_1) - f(u_2)| < \varepsilon \). Therefore cannot be the largest cost pair in \( M \). Now, suppose \( (u_1, u_2) \in M \). If \( |g(v) - g(v_2)| > \varepsilon \), then the cost may change depending on where \( v \) is now assigned. In the worst case scenario, we delete \( v \). In this case, the largest increase from deletion of \( v \) and assigning \( u \) to \( v \) comes when they share a midpoint. Thus, the difference in deletion of \( v \) to the relabel is at most \( \frac{1}{2} \varepsilon \).

Simplification by the same value of \( \varepsilon \) is not necessary to achieve a bound on the distance. There may arise situations in which simplification by the same value of \( \varepsilon \) yields merge trees with too little information for data analysis. The less features that exist in the merge tree means the les less information we may glean from the matching provided by the distance.

Corollary 24. Let \( M_f, M_g \) be two merge trees whose distance \( d_M \) is \( A \) and let \( 0 < \varepsilon_1 < A \) and \( 0 < \varepsilon_2 < A \) be fixed. Then

\[
A \leq d_M(P_f(M_f), P_g(M_g)) \leq A + \frac{1}{2} \max \{\varepsilon_1, \varepsilon_2\}
\]

Note that since the merge tree matching distance is always bounded below by the bottleneck distance, we can compute the bottleneck distance between the two merge trees and use the resulting value to gauge the value of \( \varepsilon \).

7 Experiments

We implemented the algorithm described above using Python. We use the Topology Toolkit [40] to visualize and extract merge trees from the input datasets. The bottleneck distance was computed in python through Persim [13]. Each experiment was run using on a single AMD EPYC 7642 machine at 2.4GHz using 32 of the cores. We split the distance computations in batches to work on subsets of the data in parallel. Within a batch, we also took advantage of the on node parallelism to compute computing multiple distances at the same time. In each of these experiments, we have decided to apply persistence simplification in order to move our merge trees down to 14 nodes each.

In order to determine a size for the graphs which provided us with a good balance of computational feasibility and low persistence thresholding, we evaluated the computation times of 8,10,12,14,16 and 18 node graphs and the corresponding persistence thresholds which achieved these sizes. We randomly pulled 175 pairs of scalar fields from our shape comparison experiment (Sect. 7.1) and computed the distance between pairs to track the average computation time. Fig. 8 shows these values as line charts. We can see that using 16 and 18 node graphs begins to increase the computation time dramatically while the
7.1 Shape Comparison

We took the TOSCA non-rigid world dataset, which contains a collection of shapes of animals and humans in different poses, and then we computed the average geodesic distance on each of them using the method suggested by Hilaga et al. [27]. We randomly sampled a subset of 100 vertices from each mesh, calculated the geodesic distance from every vertex to the subset and then took the average. Next we persistence simplified the data, by using a custom threshold for each mesh so that we get 14 nodes in the split tree. The largest persistence simplification value was 7.34 which implies that the distances we have computed are at most 3.67 above the true distance.

7.1.1 Results

We computed pairwise distance between 132 shapes, separating the 17424 distance computations into 12 batches. Each batch, with on-node parallelization from the 32 cores, took an of average 44.84 minutes.

Fig. 7 shows the pairwise distances computations of our merge tree matching distance compared to the merge tree edit distance from Sridharamurthy et al. [36]. We note that the distance matrix for merge tree edit distance produced here differs from the distance matrix produced in their original paper. This can be due to several reasons: 1) we used more vertices of the scalar field in order to compute the average geodesic distance, 2) the merge tree edit distance does not simplify the resulting scalar field, and 3) a difference in color scale.

We can expect a difference between our distance and the merge tree edit distance due to our distance being more stable as well as merge tree distance summing the values of the feature differences rather than taking the largest feature difference.

We would like to note that our distance and the merge tree edit distance produce similar global patterns. For example, comparisons to the seahorse produce relatively large distances and comparisons between humanoid shapes produce relatively low distances. In our distance matrix, we can see that there is a low distance for comparison between two of the same classes of shapes, regardless of the pose it takes. Another interesting point is the relationship between the centaur to other shapes. The shape has a similar distance to each of the other shapes, besides the seahorse and shark, which may be expected due to half of the centaur’s shape being similar to each of the other shapes.

In Fig. 9, we depict the bottleneck distance compared to our merge
tree matching distance for a single horse and single seahorse to all other poses. We note that the only time that our distance achieves the bottleneck distance exactly is when we compare the horse to other horse poses, centaur poses, or seahorse. This is likely due to the large features being able to be matched to one another with little need for adjacency changes. The case is similar for the seahorse pose.

7.2 von Kármán Vortex Street

We obtained a von Kármán Vortex Street dataset from [23] and calculated the vorticity scalar field for a set of uniformly sampled timesteps. Then we persisted simplified each timestep so that the resulting merge would have exactly 14 nodes. Since we were only planning on obtaining 14 nodes, we decided to clip the vortex data so that more information from the clipped timestep could be obtained. Then we computed the pairwise distance using our distance between time-steps. The distance matrix is shown in Fig. 10. The largest persistence simplification value was 6.60, which implies that the distances we have computed are 3.30 above the true distance.

7.2.1 Results

As shown in Fig. 10 (right) we can see that there is an initial time period where there are no vortices in the data. This is reflected in the upper left hand cluster of the distance matrix in Fig. 10 (left) which shows low distance values between early consecutive timesteps. Once the vortices have started forming, we can see that their positions periodically alternate as the vortices move forward. The collection of alternating high and low values in the bottom right section of the distance matrix demonstrates this periodicity.

7.3 Stability Testing

To test the stability of our distance, we constructed a baseline scalar field \((\phi_0, f_0)\) with three maxima, two saddles, and the global minimum. The two connected saddles are within \(\epsilon\) of each other – making the scalar field horizontally \(\epsilon\)-unstable. We generated 36 new scalar fields by applying a random plane multiplied by a Gaussian, to simulate random noise. The bottleneck, merge tree matching, and \(L^\infty\) distance was computed for each scalar field when compared to the baseline and plotted in Fig. 1. We found that, as desired, our distance lay between the bottleneck and \(L^\infty\) distance.

The perturbation was chosen with several specifics in mind in order to correctly mimic the case of horizontal stability. Let \(\epsilon = |f(x_0) - f(x_1)|\). The extrema \(x_3, x_4, x_5\) are assigned function values such that each are more than \(2\epsilon\) greater than their connected saddle. Otherwise, the perturbation which we apply would create a new scalar field \((\phi, g)\) such that deletion of \(\epsilon(x_2, x_4)\), inserting \(\epsilon(x_2, x_4)\), and adjusting the function value of \(x_1\) to be the function value of \(y_1\) would be optimal. Furthermore, since we wanted to focus on horizontal instabilities in this experiment, we made sure that the difference between \(x_3\) and \(x_3\) is larger than \(\epsilon\). Otherwise, the perturbation would essentially just be “reflecting” the merge tree, i.e. mapping \(x_3\) to \(y_3\) and \(x_5\) to \(y_5\).

It is worth noting that if there was no topology change in the data and our perturbation was only causing differences in function value, then our merge tree matching distance would be equal to the bottleneck distance. Thus, this distance is sufficiently capturing a perturbation which changes the topology of the merge tree.

8 DISCUSSION

Here we have constructed a distance on merge trees which has experimentally been shown to be both stable and more discriminative than the bottleneck distance. Not only was our distance less than the \(L^\infty\) distance during our experimentation, but it was specifically designed to identify and quantify scenarios where perturbations in the dataset may cause topological changes in the merge tree or pairing changes in the persistence diagram and branch decomposition trees.

Properties of a Metric While we do not explicitly prove the triangle inequality and symmetry property, we would like to note that we verified that these properties hold on each of the datasets that we provided here.

Comparison to Beketayev Distance Beketayev et al. introduced a distance on merge trees which also computes and compares all branch decomposition trees to one another [6]. They are able to reduce the computation time of comparing all BDTs by not repeating comparisons of subtrees of specific BDTs. One fundamental difference that makes this possible is that once a node \(x\) is paired to a node \(y\), the children of \(x\) must be mapped to the children of \(y\), or be inserted/deleted. In our case, we cannot necessarily re-utilize comparisons of BDT subtrees since our nodes are always able to be mapped outside of any given subtree. This particular restriction of ancestor-descendant relationships is exactly what may cause horizontal instabilities while reducing the computation time. It is for a similar reason that a direct application of tree edit distance is unstable on merge trees.

Translation to Contour Trees and Reeb Graphs When translating to contour trees and Reeb graphs, the A* algorithm would still be able to adequately match features of one graph-based descriptor to another. Furthermore, we can introduce additional pruning since these theoretical graph-based distances always have the restriction that we cannot match features of different types to one another (e.g. an up-leaf cannot be matched to a down-leaf in a contour tree or Reeb graph). The hurdle that we run into is that of properly encoding the features of these descriptors. There is some nuance on choosing the pairing in contour trees. For example, the path from the global min to the global max may be a non-monotone path. To the best of our knowledge, there has been no generalization of the BDT for Reeb graphs.

Scalability We would like to make note that our algorithm still suffers from the issue of scalability. While a strength of our approach is that we can use persistence simplification to reduce the number of vertices while retaining accuracy, even small increases in the size of the merge tree may cause our computation time to increase in an exponential fashion (see Fig. 8). Nevertheless, while analysis of small trees may be practical in some settings, we see developing a more efficient approach as an important, open challenge.

ACKNOWLEDGMENTS

We thank Raghavendra Sridharamurthy and Vijay Natarajan for providing the comparison results of their algorithm [36] on experiment used in Sect. 7.1. We also thank our anonymous reviewers for provided their detailed feedback and suggestions. This work is supported in part by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Award Number(s) DE-SC-0019039.
