Measuring the Distance between Merge Trees

Kenes Beketayev, Damir Yeliussizov, Dmitriy Morozov, Gunther H. Weber, and Bernd Hamann

Abstract Merge trees represent the topology of scalar functions. To assess the topological similarity of functions, one can compare their merge trees. To do so, one needs a notion of a distance between merge trees, which we define. We provide examples of using our merge tree distance and compare this new measure to other ways used to characterize topological similarity (bottleneck distance for persistence diagrams) and numerical difference (L_∞-norm of the difference between functions).

Kenes Beketayev
Lawrence Berkeley National Laboratory, One Cyclotron Rd, Berkeley, CA 94720, USA
Nazarbayev University, 53 Kabanbay Batyr Ave, Astana, Kazakhstan, 010000
e-mail: KBeketayev@lbl.gov

Damir Yeliussizov
Kazakh-British Technical University, 59 Tole Bi St, Almaty, Kazakhstan, 050000
e-mail: yeldamir@gmail.com

Dmitriy Morozov
Lawrence Berkeley National Laboratory, One Cyclotron Rd, Berkeley, CA 94720, USA
e-mail: DMorozov@lbl.gov

Gunther H. Weber
Lawrence Berkeley National Laboratory, One Cyclotron Rd, Berkeley, CA 94720, USA
Institute for Data Analysis and Visualization (IDAV), Department of Computer Science, University of California, Davis, CA 95616-8562, USA
e-mail: GHWeber@lbl.gov

Bernd Hamann
Institute for Data Analysis and Visualization (IDAV), Department of Computer Science, University of California, Davis, CA 95616-8562, USA
e-mail: hamann@cs.ucdavis.edu

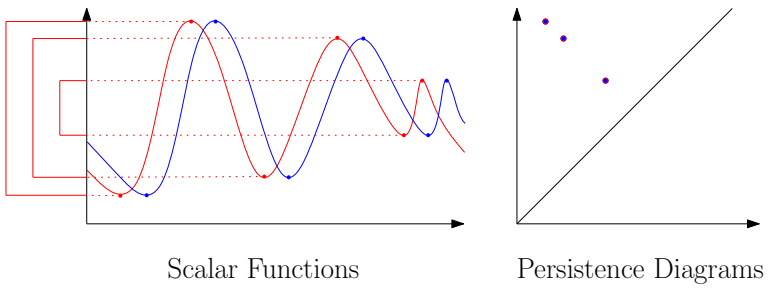

Fig. 1 Consider two scalar functions, where one is a slightly shifted version of the other. Comparing them directly, e.g., via the L_∞ norm, results in a large difference. Their persistence diagrams are the same, thus capturing the topological similarity of these functions.

1 Introduction

Many aspects of physical phenomena are described and modeled by scalar functions. Computational and experimental capabilities allow us to approximate scalar functions at increasing levels of detail and resolution. This fact makes it necessary to analyze and also compare such function automatically, when possible, and to include more abstract analysis methods. Topological methods, based on the characterization of a scalar function via its critical point behavior, are gaining in importance, and we were therefore motivated to investigate the feasibility of comparing scalar functions using their topological similarity. Computational chemistry, physics and climate sciences are just a few applications where our ideas presented here should be valuable.

We address the generic problem of comparing the topology of scalar functions. Fig. 1 demonstrates this problem. The figure shows slightly shifted versions of the same function, colored red and blue. Commonly used analytical distances (e.g., norms of the difference) between these functions would result in a non-zero value, failing to highlight the fact that they have the same sub-level set topology.

One well-established distance that expresses the topological similarity in the above example is the bottleneck distance between persistence diagrams, introduced by Cohen-Steiner et al. [8]. Computing the bottleneck distance for the example in Fig. 1 results in zero. Originally motivated by the shape matching problem, where the goal is to find how similar shapes are based on similarity of their topology, the bottleneck distance also has an important property — robustness to noise; see Fig. 2.

However, the bottleneck distance does not incorporate sub-level set nesting information, often necessary for analysis. Fig. 3 shows two functions that differ by the nesting of the maximum m. The bottleneck distance between the corresponding persistence diagrams is again zero. Nevertheless, the corresponding merge trees cannot be matched exactly, hinting at a positive difference.

To resolve this problem, we introduce a new definition of the distance between merge trees. This distance resembles the bottleneck distance between the persistence diagrams of sub-level sets of the function, but it also respects the nesting relationship

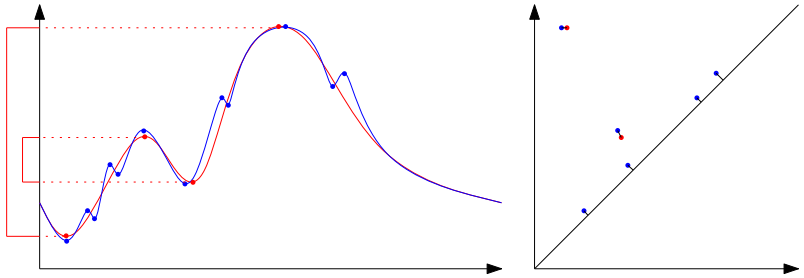

Fig. 2 Consider two close scalar functions on the left, where one contains additional noise. If we construct their persistence diagrams and find the bottleneck distance (which corresponds to the longest black line segment between paired points on the right), the result is small, correctly reflecting the closeness of the functions. In fact, the difference is the same as the level of the noise, which in this example is small.

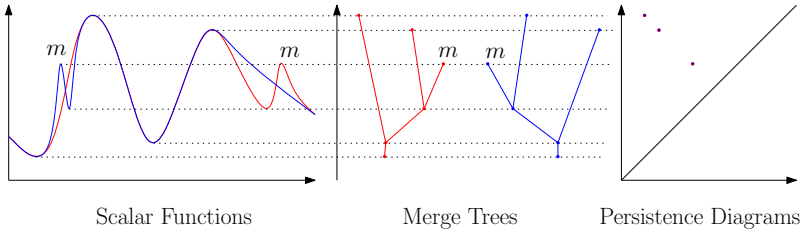

Fig. 3 Consider two scalar functions on the left. The bottleneck distance between persistence diagrams on the right equals zero, as points of two diagrams overlap. However, comparing the corresponding merge trees reveals a difference, since we cannot match them exactly. This difference highlights existence of additional nesting information in merge trees. Quantifying it is the main goal of this work.

between sub-level sets. Furthermore, the proposed distance implicitly distinguishes the noise in the data, similar to the bottleneck distance, resulting in robust measurements resilient to perturbations of the input. This property is crucial when working with scientific data, where noise is a serious problem for any analysis.

The main contributions of this chapter are: a definition and an algorithm for computing the distance between merge trees; computation of the number of branch decompositions of the merge tree; an experimental comparison between the proposed distance, the bottleneck distance, and the L_∞ norm on analytical and real-world data sets.

Section 2 presents related work and background in scalar field topology, persistent homology, graph theory, and shape matching. Section 3 provides the definition and the algorithm for computing the distance between merge trees. Section 4 demonstrates several use cases and presents the results of comparing the distance between merge trees to the bottleneck distance between persistence diagrams, as well as the L_∞ norm. Finally, Section 5 summarizes the work and suggests ideas for future work.

2 Related Work

2.1 Scalar Field Topology

Scalar field topology characterizes data by topological changes of its level sets. Given a smooth, real-valued function without degenerate critical points, level set topology changes only at isolated critical points [16]. Several structures relate critical points to each other.

The *contour tree* [5, 7] and the *Reeb graph* [21, 20] track the level sets of the function by recording their births (at minima), merges or splits (at saddles), and deaths (at maxima). The contour tree is a special case of the Reeb graph, as the latter permits loops in the graph to handle holes in the domain. Both structures are used in a variety of high-dimensional scalar field visualization techniques [23, 18].

Alternatively, the Morse–Smale complex [10, 9] segments the function into the regions of uniform gradient flow and encodes geometric information. It is also used for analysis of high-dimensional scalar functions [12].

We focus on a structure called *merge tree* (sometimes called a barrier tree [11, 13]), as it tracks the evolution of sub-/super-level sets, while still being related to the level-set topology through critical points [16].

2.2 Persistent Homology

The concept of homology in algebraic topology offers an approach to studying the topology of the sub-level sets. We refer to Munkres [17] for the detailed introduction to homology. Informally, it describes the cycles in a topological space: the number of components, loops, voids, and so on. We are only interested in 0–dimensional cycles, i.e., the connected components.

Persistent homology tracks changes to the connected components in sub-level sets of a scalar function. We say that a component is born in the sub-level set $f^{-1}(-\infty, b]$ when its homology class does not exist in any sub-level set $f^{-1}(-\infty, b-\varepsilon]$. This class dies in the sub-level set $f^{-1}(\infty, d]$ if its homology class merges with another class that exists in a sub-level set $f^{-1}(-\infty, b']$ with $b' < b$. When a component is born at b and dies at d, we record a pair (b,d) in the (0–dimensional) persistence diagram of the function f, denoted $D(f)$. For technical reasons, we add to $D(f)$ infinitely many copies of every point (a,a) on the diagonal.

Persistence diagrams reflect the importance of topological features of the function: the larger the difference $d-b$ of any point, the more we would have to change the function to eliminate the underlying feature. Thus, persistence diagrams let us distinguish between real features in the data and noise.

In Cohen-Steiner et al. [8], the authors prove the stability of persistence diagrams with respect to the bottleneck distance, $d_B(D(f), D(g))$. This distance is defined as the infimum over all bijections, $\gamma : D(f) \to D(g)$, of the largest distance between the

corresponding points,

$$d_B(D(f),D(g)) = \inf_{\gamma} \sup_{u \in D(f)} ||u - \gamma(u)||_\infty.$$

Their result guarantees that the bottleneck distance is bounded by the infinity norm between functions:

$$d_B(D(f),D(g)) \leq \|f - g\|_\infty.$$

We use the bottleneck distance between persistence diagrams as a comparison baseline for the distance between merge trees.

2.3 Distance between Graphs

Graph theory offers several approaches for comparing graphs and defining a notion of a distance between them.

A common approach for measuring a distance between graphs is based on an edit distance. It is computed as a number of edit operations (add, delete, and swap in the case of a labeled graph) required to match two graphs [6], or, in a special case, trees [4]. The edit distance focuses on finding an isomorphism between graphs/subgraphs, while for merge trees we can have two isomorphic trees with a positive distance (see the example in Fig. 3).

Alternatively, in a specific case of rooted trees, one can consider the generalized tree alignment distance [15], which, in addition to the edit distance, considers the minimization of the sum of distances between labeled end-points of any edge in trees. However, it is not clear how to adapt this distance definition for our purposes.

2.4 Using Topology of Real Functions for Shape Matching

The field of shape matching offers several methods related to our work. Generally, these methods focus on developing topological descriptors by treating a shape as a manifold, defining some real function on that manifold, and computing topological properties of the function. The selection of the particular function usually depends on which specific topological and shape properties of interest [2].

While the majority of the mentioned descriptors are not directly related to our work, two topological descriptors use similar approaches in defining a similarity measure. One is called a multiresultion Reeb graph, proposed by Hilaga et al. [14], which encodes nesting information into nodes of a Reeb graph for different hierarchy resolutions. Here, the hierarchy is defined by the simplification of Reeb graph. Another descriptor is based on an extended Reeb graph (ERG), proposed by Biasotti et al. [3]. It starts by computing the ERG of the underlying shape, which is basically a Reeb graph with encoded quotient spaces in its vertices. It couples various geo-

metric attributes with the ERG, resulting in an informative topological descriptor. In both cases, similarity of shapes is measured by applying a specialized graph matching (based on embedded/coupled information) to descriptors. However, we focus only on the sub-level set topology information, and design a matching algorithm, tailored specifically for this case.

Thomas and Natarajan [22] focus on symmetry discovery in a scalar function based on its contour tree. The authors develop a similarity measure between subtrees of the contour tree, which in some regards is similar to our proposed measure. However, they consider a single pre-processed branch decomposition, and focus on discovering symmetry in a sole function.

3 Defining a Distance Between Merge Trees

In this section, we provide a formal definition of the distance between merge trees and provide an algorithm (with optimizations) for computing it. In short, to compute the distance between two merge trees, we consider all branch decompositions of both trees and try to find a pair that minimizes the matching cost between them. Additionally, we provide the details of computing the number of branch decompositions of a merge tree, used in complexity analysis of our algorithm.

3.1 Definition

Let K be a simplicial complex; let $f : K \to \mathbb{R}$ be a continuous piecewise-linear function, defined on the vertices and interpolated in the interior of the simplices. Furthermore, assume all vertices have unique function values; in practice, we can simulate this by breaking ties lexicographically.

Let T_f be a merge tree of the function f; every vertex of K is mapped to a vertex in the merge tree. Every vertex of the merge tree has a degree of either one, two, or more, corresponding to a minimum, a regular point, or a merge saddle. Our definition works for higher-dimensional saddles (degenerate critical points) as well, and they need explicit consideration only in the complexity analysis of the algorithm (Sect. 3.4). A merge tree with purged regular vertices is called *reduced*.

A branch decomposition B [19] of a reduced merge tree T is a pairing of all minima and saddles such that for each pair there exists at least one descending path from the saddle to the minimum. We consider a rooted tree representation R of the branch decomposition B, such that the rooted tree representation R is obtained by translating each branch $b = (m,s) \in B$ into a vertex $v \in R$, where m and s are minimum and saddle that form the branch b. The edges of the rooted tree representation describe parent–child relationships between branches, see Fig. 4.

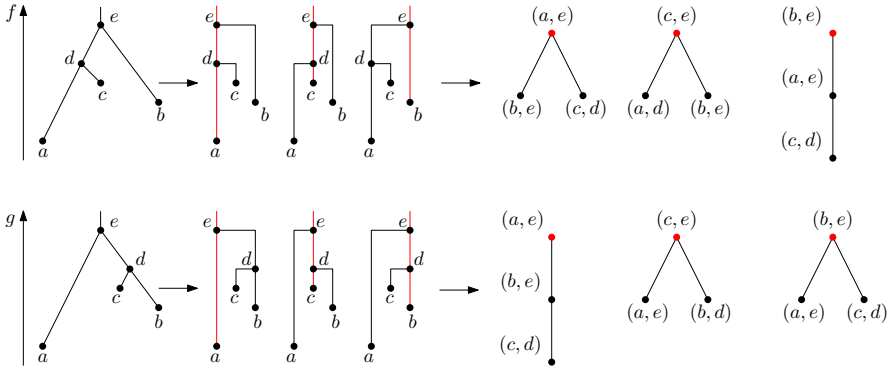

Fig. 4 Merge trees T_f (top) and T_g (bottom), all their possible branch decompositions, and corresponding rooted tree representations. Root branches are colored red, demonstrating the mapping of branches to vertices.

Given two merge trees, T_f and T_g, consider all their possible branch decompositions, $B_{T_f} = \{R_1^f, ..., R_k^f\}$ and $B_{T_g} = \{R_1^g, ..., R_k^g\}$, respectively; see Fig. 4.We need two auxiliary definitions to describe the matching of rooted branch decompositions.

Definition 1 (Matching cost). The cost of matching two vertices $u = (m_u, s_u) \in R_i^f$ and $v = (m_v, s_v) \in R_j^g$ is the maximum of the absolute function value difference of their corresponding elements,

$$mc(u,v) = max(|m_u - m_v|, |s_u - s_v|).$$

Definition 2 (Removal cost). The cost of removing a vertex $u = (m_u, s_u) \in R^{f,g}$ is

$$rc(u) = |m_u - s_u|/2.$$

We say that a partition (M^f, E^f) of the vertices of a rooted branch decomposition R^f is *valid*, if the subgraph induced by the vertices M^f is a tree. Here, the vertices M^f are mapped vertices, while the vertices E^f are reduced vertices. We say that an isomorphism of two rooted trees preserves order when it maps children of a vertex in one tree to the children of its image in the other tree.

Definition 3 (ε-Similarity). Two rooted branch decompositions R^f, R^g are ε-similar, if we can find two valid decompositions (M^f, E^f) and (M^g, E^g) of their vertices, together with an order-preserving isomorphism γ between the trees induced by the vertices M^f and M^g, such that the distance between each matched pair of vertices and the maximum cost for reduced vertices does not exceed ε:

$$\max_{u \in M^f} mc(u, \gamma(u)) \leq \varepsilon \tag{1}$$

$$\max_{u \in E^f \cup E^g} rc(u) \leq \varepsilon \tag{2}$$

The smallest epsilon, for which the above two inequalities hold, denoted $\varepsilon_{min}(R_i^f, R_j^g)$.

Definition 4 (Distance between merge trees). The distance between two merge trees T_f, T_g is:

$$d_M(T_f, T_g) = \min_{R_i^f \in B_{T_f}, R_j^g \in B_{T_g}} (\varepsilon_{min}(R_i^f, R_j^g)).$$

3.2 Distance Computation

To compute the distance d_M, we design an algorithm that is based on Definition 4. In particular, our algorithm constructs all possible pairs of branch decompositions, computes ε_{min} for each pair, and selects the minimum among them.

We use a recursive construction of the branch decompositions of a merge tree. The main operation is to pair a given saddle, one by one, with each minimum in its subtree. We start by pairing the highest saddle s_r with all minima in a tree. Each pair acts as a root branch (s_r, m_i) in the recursive operation. For each child saddle s_j on the root branch, we recursively repeat the pairing until all the saddle–minimum pairs are fixed, producing a unique branch decomposition b_i.

To compute $\varepsilon_{min}(R_i^f, R_j^g)$, we design a function ISEPSSIMILAR$(\varepsilon, R_i^f, R_j^g)$ that, for a predefined ε, determines whether two branch decompositions match. We start by setting ε to a high value — for example, the maximum of the amplitudes of the two functions — and perform a binary search to determine ε_{min}.

The function ISEPSSIMILAR is the core of the algorithm. It works by matching the vertices and the edges at each level of the tree. We recall that each vertex $u = (m_u, s_u) \in r_i, v = (m_v, s_v) \in r_j$ is a minimum-saddle pair. There are only two vertices at the root levels of R_i^f and R_j^g, so we determine whether their endpoints can be matched, i.e., $\max(|m_u - m_v|, |s_u - s_v|) \leq \varepsilon$. If not, ISEPSSIMILAR returns false. Otherwise, we consider all the child vertices (see Fig. 5). Since there are several potential matches, we compute a bipartite graph between the child vertices such that the edge between a pair of children $u \in R_i^f, v \in R_j^g$ exists if and only if they can be matched within given ε, and ISEPSSIMILAR returns true for their subtrees. We also add ghost vertices for each vertex in the rooted branch decomposition when it can be reduced within ε. When there exists a perfect matching in the bipartite graph, the function returns true; otherwise, it returns false. If one or both of the current pair of children has children of their own, we recursively call ISEPSSIMILAR. The matching is perfect when there exists an edge cover such that its edges are incident to all the non-ghost vertices and do not share any of them.

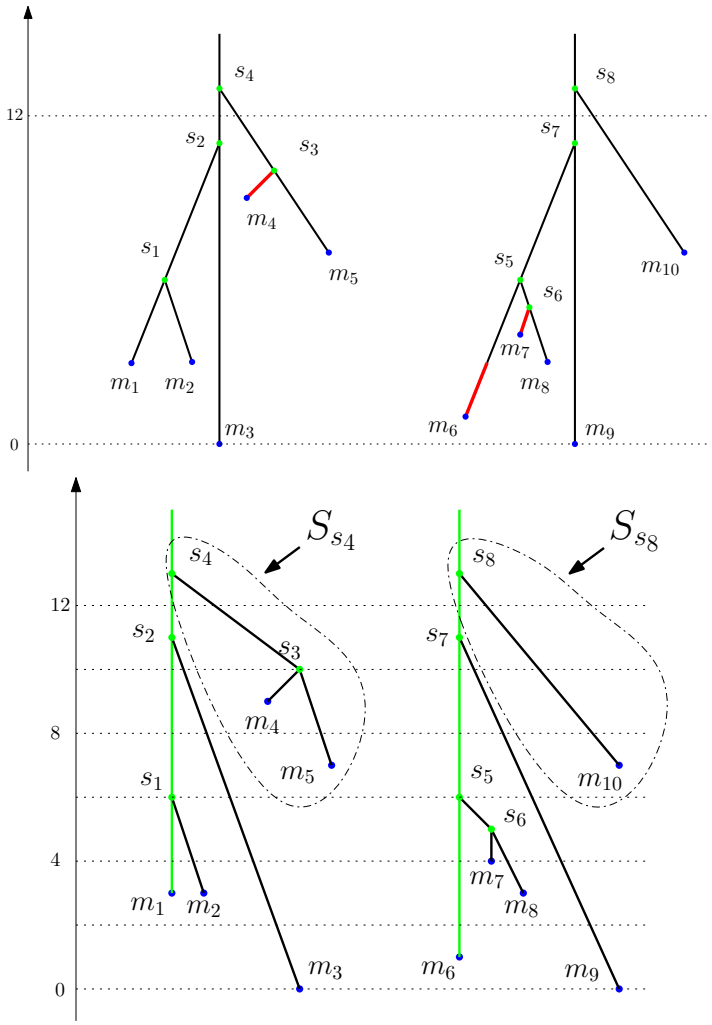

Fig. 5 Top: For the merge trees T_f, T_g, the smallest manually identifiable difference is shown as red segments. Bottom: The first iteration of the IsEpsSimilar function chooses (s_4, m_1) and (s_8, m_6) as root branches (depicted in green).

3.3 Optimized Algorithm with Memoization

The naive algorithm described above has exponential complexity. Indeed, there exist $O(2^{N-1})$ branch decompositions for a tree with N extrema (see Sect. 3.4 for details). Consequently, comparing all branch decompositions of two trees to each other would require a total of $O(2^{N+M-2})$ operations, where N, M are the numbers of extrema in each tree. This computational cost makes it infeasible to compare even small trees using this method. To alleviate this problem, we have designed an

optimization, which reduces the number of explicitly considered branch decompositions, thus improving the complexity of the function ISEPSSIMILAR from exponential to polynomial. (Details are given at the end of this section.)

We demonstrate the optimized version of the function ISEPSSIMILAR using the example in Fig. 5. The function starts by iterating over all possible root branches $(s_4, m_i), i \in 1 \dots 5$, and $(s_8, m_j), j \in 6 \dots 10$. Once the pair of root branches is fixed as (s_4, m_1) and (s_8, m_6), the function is called recursively for every possible pairing of child subtrees in each tree. Fixing the root branches leads to two sets of child subtrees, $\{S_{s_4-s_3}, S_{s_2-m_3}, S_{s_1-m_2}\}$ and $\{S_{s_8-m_{10}}, S_{s_7-m_9}, S_{s_5-s_6}\}$. A subtree (e.g., $S_{s_4-s_3}$) needs two vertices to be uniquely identified, a child saddle (e.g., s_4), and the immediate child vertex (e.g., s_3) that can be either a saddle or minimum.

A key observation allowing us to reduce cost is that each pair of subtrees, for which the function is called recursively, also appears in subsequent iterations over other root branches. For example, the pair $(S_{s_4-s_3}, S_{s_8-s_{10}})$ that appears in the first iteration that chooses (s_4, m_1) and (s_8, m_6) as root branches, also reappears in 11 subsequent iterations, e.g., in the iteration that chooses (s_4, m_2) and (s_8, m_7) as root branches. Therefore, it is sufficient to compute the matching for subtrees S_{s_4}, S_{s_8} once, and reuse the result in subsequent iterations. More generally, for any subtree pair $S_{s_i-s_{childof i}} \in T_f$ and $S_{s_j-s_{childof j}} \in T_g$, one of the three possibilities is recorded in the array $match[S_{s_i-s_{childof i}}][S_{s_j-s_{childof j}}]$: not yet compared – (0), comparison returned false – (1), or true – (2).

Furthermore, the same pair of subtrees also reappears in other binary search iterations as well, i.e., when the function ISEPSSIMILAR is called with other values of ε. However, the reuse of previous results in this case is selective. If two subtrees were matched for some ε, they would stay matched only if the value of ε stayed the same or gets larger. If it gets smaller, we will have to recompute the matching result. And correspondingly, if two subtrees were unmatchable for some ε, they would stay unmatchable, only if the value of ε was the same or lower. However, if the value gets higher, we will have to recompute the matching result.

Returning to the example in Fig. 5, consider the case where $\varepsilon = 5$. The first pair of root branches (s_4, m_1) and (s_8, m_6) (depicted in green in Fig. 5) do match, as $|f(s_4) - g(s_8)| = 0 < 5$ and $|f(m_1) - g(m_6)| = 2 < 5$, hence the function proceeds to their child subtrees. The first pair $S_{s_4-s_3}, S_{s_8-m_{10}}$ is matchable, thus there is an edge in the bipartite graph (similar to the naive algorithm) between the nodes that correspond to these subtrees. In fact, from nine pairs, only pairs $S_{s_2-m_3}, S_{s_8-m_{10}}$ and $S_{s_4-s_3}, S_{s_7-m_9}$ are unmatchable, thus there exists a perfect matching in the bipartite graph, and two merge trees are ε-similar for $\varepsilon = 5$. Consequently, we continue the search with decreasing value of ε, until it converges to $\varepsilon = 2$, in which case for the root branches (s_4, m_1) and (s_8, m_6), the only pairs of subtrees that match are (S_{s_4}, S_{s_8}), (S_{s_2}, S_{s_7}), (S_{s_1}, S_{s_5}). No lower value of ε would lead to the ε-similarity of the merge trees, making the value $\varepsilon_{min} = 2$ the distance between merge trees.

Such optimization reduces the run time complexity from exponential to polynomial. Indeed, the function ISEPSSIMILAR performs $N \cdot M$ iterations over the root branches, multiplied by the sum of processing $n_{c_f} \cdot n_{c_g}$ explicit pairs of subtrees, and the complexity of a maximal matching algorithm $(n_{c_f} + n_{c_g}) \cdot n_{c_f} \cdot n_{c_g}$. The latter

complexity dominates the former term. Hence, assuming that a look-up operation of previous results is done in a constant time via memoization, the resulting run time complexity of the function IsEpsSimilar is $O(N^2M^2(N+M))$. This complexity is multiplied by the number of iterations of the binary search algorithm, which we found to be moderate, given a reasonable selection of the search range and the precision. The worst-case memory complexity of the optimized algorithm is $O(N \cdot M)$, which is computationally less prohibitive than its run time complexity.

3.4 The Number of Branch Decompositions of a Merge Tree

We provide the details of computing the number of branch decompositions of the merge tree, used for the naive algorithm complexity analysis in the beginning of Sect. 3.3. We calculate the number of branch decompositions $P(N)$ for a merge tree with N minima in two steps. First, we compute the number $P(N)$ for the case when the merge tree is binary, in which case the tree has maximum possible number of saddles. Second, we show that for fewer saddles the number $P(N)$ decreases, leading to the worst case $P(N) = 2^{N-1}$ branch decompositions for any merge tree.

Theorem 1. *The number of branch decompositions of the binary merge tree with N minima equals* $P(N) = 2^{N-1}$.

Proof. For any saddle s, the number of branch decompositions in its subtree is $P_s = 2 \cdot P_{c_1} \cdot P_{c_2}$, where c_1 and c_2 are the children of the saddle s. Indeed, if the saddle s is paired with a minimum in a subtree of child c_1, then for each such pairing we have all the possible branch decompositions of a subtree of child c_2, resulting in $P_{c_1} \cdot P_{c_2}$ possibilities. Symmetrically, for the child c_2 we have $P_{c_2} \cdot P_{c_1}$ possibilities.

Using this fact we construct a proof by induction:

- For the base case of $N = 1$, the number of branch decompositions is one. On the other hand, $P(1) = 2^{1-1} = 1$. Hence, the formula holds.
- We assume that for all $N = 1, \ldots, k$ the formula $P(N) = 2^{N-1}$ holds true.
- Now let's consider the case with $N = k+1$, for which we have to prove that $P(k+1) = 2^k$. For the root saddle r of the tree with $k+1$ minima, we remember that $P_r = 2 \cdot P_{c_1} \cdot P_{c_2}$. If to denote the number of minima in the subtree of the child c_1 as $i \in [1,k]$, with $k-i+1$ denoting the number of minima in the subtree of the child c_2, we can expand $P(k+1) = 2 \cdot P(i) \cdot P(k-i+1)$. Since both i and $k-i+1$ are not greater than k, we can substitute $P(i)$ and $P(k-i+1)$ in accordance with assumptions for $N = 1, \ldots, k$:

$$\begin{aligned} P(k+1) &= 2 \cdot P(i) \cdot P(k-i+1) \\ &= 2 \cdot 2^{i-1} \cdot 2^{k-i+1-1} \\ &= 2 \cdot 2^{i-1+k-i+1-1} = 2^k. \end{aligned}$$

□

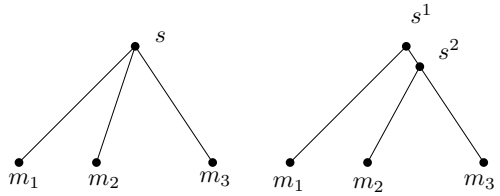

Fig. 6 Splitting the higher-degree saddle (with degree > 2) always creates more branch decompositions. The saddle s has three branch decompositions, while after splitting it into two saddles s^1 and s^2, we get four branch decompositions.

We consider the case with a number of saddles less than $N-1$, i.e., the merge tree has saddles with degree higher than two. The number of minima N remains the same, while some of the saddles have more than two children. Any saddle of degree $d > 2$, can be split into $d-1$ saddles of degree two, such that the structure of the tree changes only around the selected saddle, see Fig. 6. Such split leads to 2^{d-1} possible branch decompositions instead of the d for the selected saddle. Since $d > 2$, the inequality $2^{d-1} > d$ holds true, which means having degree-two saddles always leads to more branch decompositions. At the extreme, if all saddles become degree-two saddles, we obtain the binary merge tree, for which we already computed number of branch decompositions as 2^{N-1}.

4 Results

In this section we demonstrate the use of the proposed distance d_M. First, we apply it to simple data sets, observing its difference from the bottleneck distance and the L_∞ norm, as it captures additional information. We consider performance data sets obtained for a ray tracing program, and demonstrate how the proposed distance correctly captures the similarity of data sets.

4.1 Analytical Functions

We consider a set of simple functions that have a fixed number of maxima. Each function is constructed by creating a random set of maxima generators. For each maximum generator, function values decrease as the distance from the source grows, which results in a corresponding peak. The upper envelope of such peaks results in the required function.

We generate three bivariate (or 2D) functions f_1, f_2, f_3, and three trivariate (or 3D) functions f_4, f_5, f_6. We set the number of maxima to five, to keep them simple for visual exploration; see Fig. 7. The resulting distances, presented in Table 1, lead to two interesting observations.

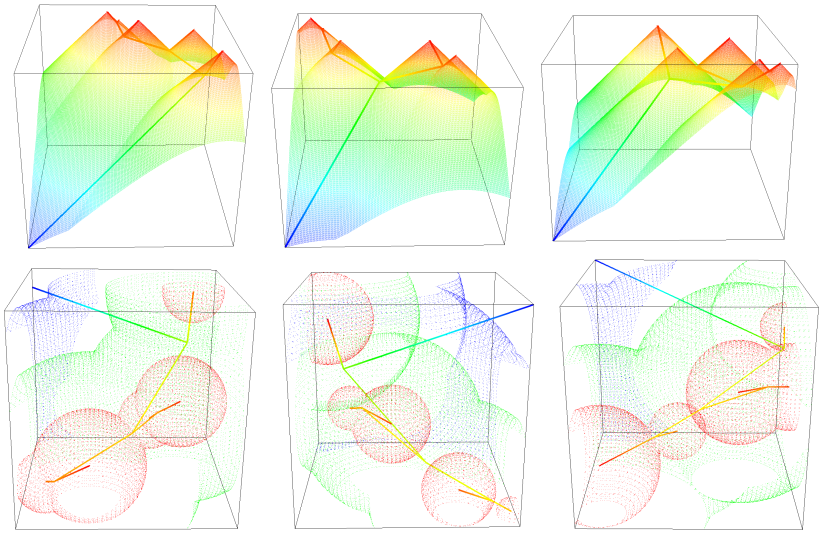

Fig. 7 Analytical functions. Rendering with embedded merge tree of three 2D functions f_1, f_2, f_3 (first row), and three 3D functions f_4, f_5, f_6 (second row).

Table 1 Resulting distances from Fig. 7.

Metric	f_1^{2D}	f_2^{2D}	f_3^{2D}	f_4^{3D}	f_5^{3D}	f_6^{3D}
d_B	4.525	8.647	7	3.011	2.598	4.031
d_M	8.398	8.664	7	5.031	2.604	4.833
L_∞	67.561	43.015	65.956	29.586	20.495	22.632

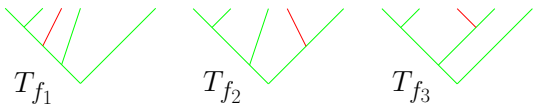

Fig. 8 Simplified view of nesting of merge trees for functions f_1, f_2, f_3. Unmatchable red edges cause the non-zero distance.

For the 2D functions, the bottleneck distance d_B for functions f_1, f_2 is about two times lower than for functions f_1, f_3 and f_2, f_3, suggesting relative closeness of the first pair. However, the distance between merge trees d_M suggests that all three functions are equally different. Closer investigation confirms this hypothesis. In Fig. 8, we see simplified depictions of merge trees having equally different nesting.

For the 3D functions, visually the function f_5 seems different from the other two. This fact is again captured by the distance between merge trees, as the resulting distance d_M is almost two times lower for functions f_4, f_6, than from them to function f_5. The bottleneck distance again fails to capture this distinction.

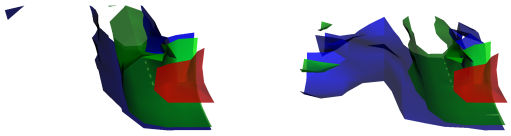

Fig. 9 Performance data. $d_B = 0.027, d_M = 0.027, L_\infty = 0.13$. The difference is small relative to the value range of functions (about 2.7%), implying little influence of the ray sampling option on the overall performance of the algorithm.

4.2 Tuning a Ray Tracing Algorithm

We consider the problem of tuning a ray tracing algorithm on a multicore shared-memory system from the study by Bethel and Howison [1]. The authors explored various tuning parameters and their effect on the performance of the algorithm, with a focus on three parameters: the work block width $\{1,2,\ldots,512\}$ and height $\{1,2,\ldots,512\}$, and the concurrency level $\{1,2,4,8\}$.

We generated two data sets with the same parameter space, but slightly different algorithm, based on the selection of a ray sampling method, which is either based on nearest neighbor or trilinear approximation. For each option, the performance of the algorithm (in terms of running time) was recorded.

In this example, one is interested in studying optimal run configurations that correspond to low run times of the algorithm. Fig. 9 shows two data sets using isosurfaces, such that isovalues are the same for both data sets. The similarity of data sets, implying that the selection of the chosen ray sampling method does not significantly influence the performance of the algorithm. This fact is confirmed by the resulting distance. The measured distances allow us to capture the similarity, regardless of shifted optimal configurations (minima) and the noise.

5 Conclusions

We presented a novel distance between merge trees, including the definition and the algorithm. We demonstrated the use of the proposed distance for several data sets.

We plan to perform a theoretical investigation of the proposed distance, including the concerns about its stability. We also plan to explore the use of the proposed distance for error analysis in the context of approximated scalar functions.

Acknowledgements The authors thank Aidos Abzhanov. This work was supported by the Director, Office of Advanced Scientific Computing Research, Office of Science, of the U.S. DOE under Contract No. DE-AC02-05CH11231 (Berkeley Lab), and the Program 055 of the Ministry of Edu. and Sci. of the Rep. of Kazakhstan under the contract with the CER, Nazarbayev University.

DISCLAIMER

References

1. Bethel, E.W., Howison, M.: Multi-core and many-core shared-memory parallel raycasting volume rendering optimization and tuning. Int. Journal of High Perf. Comput. Appl. (2012)
2. Biasotti, S., De Floriani, L., Falcidieno, B., Frosini, P., Giorgi, D., Landi, C., Papaleo, L., Spagnuolo, M.: Describing shapes by geometrical-topological properties of real functions. ACM Computing Surveys **40**, 12:1–12:87 (2008)
3. Biasotti, S., Marini, M., Spagnuolo, M., Falcidieno, B.: Sub-part correspondence by structural descriptors of 3D shapes. Computer Aided Design **38**(9), 1002–1019 (2006)
4. Bille, P.: A survey on tree edit distance and related problems. Journal of Theor. Comp. Sci. **337**, 217–239 (2005)
5. Boyell, R.L., Ruston, H.: Hybrid techniques for real-time radar simulation. In: Proc. of the Fall Joint Comput. Conf., pp. 445–458. IEEE (1963)
6. Bunke, H., Riesen, K.: Graph Edit Distance: Optimal and Suboptimal Algorithms with Applications, pp. 113–143. Wiley-VCH Verlag GmbH & Co. KGaA (2009)
7. Carr, H., Snoeyink, J., Axen, U.: Computing contour trees in all dimensions. Comp. Geom. – Theory and Appl. **24**(2), 75–94 (2003)
8. Cohen-Steiner, D., Edelsbrunner, H., Harer, J.: Stability of persistence diagrams. In: Proc. of 21st Annual Symp. on Comp. Geom., pp. 263–271. ACM (2005)
9. Edelsbrunner, H., Harer, J., Natarajan, V., Pascucci, V.: Morse-Smale complexes for piecewise linear 3-manifolds. In: Proc. of the 19th Symp. on Comp. Geom., pp. 361–370 (2003)
10. Edelsbrunner, H., Harer, J., Zomorodian, A.: Hierarchical Morse-Smale complexes for piecewise linear 2-manifold. Disc. & Comp. Geom. **30**, 87–107 (2003)
11. Flamm, C., Hofacker, I.L., Stadler, P., Wolfinger, M.: Barrier trees of degenerate landscapes. Physical Chemistry **216**, 155–173 (2002)
12. Gerber, S., Bremer, P.T., Pascucci, V., Whitaker, R.: Visual exploration of high dimensional scalar functions. IEEE Trans. Vis. Comput. Graph. **16**(6) (2010)
13. Heine, C., Scheuermann, G., Flamm, C., Hofacker, I.L., Stadler, P.F.: Visualization of barrier tree sequences. IEEE Trans. Vis. Comp. Graph. **12**(5), 781–788 (2006)
14. Hilaga, M., Shinagawa, Y., Kohmura, T., Kunii, T.L.: Topology matching for fully automatic similarity estimation of 3d shapes. In: SIGGRAPH'01, pp. 203–212. ACM (2001)
15. Jiang, T., Lawler, E., Wang, L.: Aligning sequences via an evolutionary tree: complexity and approximation. In: Symp. on Theory of Computing, pp. 760–769 (1994)

16. Milnor, J.W.: Morse Theory. Princeton University Press, Princeton, New Jersey (1963)
17. Munkres, J.R.: Elements of Algebraic Topology. Addison-Wesley, Redwood City, CA (1984)
18. Oesterling, P., Heine, C., Janicke, H., Scheuermann, G., Heyer, G.: Visualization of high-dimensional point clouds using their density distribution's topology. IEEE Trans. Vis. Comput. Graph. **17**, 1547–1559 (2011)
19. Pascucci, V., Cole-McLaughlin, K., Scorzelli, G.: Multi-resolution computation and presentation of contour trees. Tech. Rep. UCRL-PROC-208680, LLNL (2005)
20. Pascucci, V., Scorzelli, G., Bremer, P.T., Mascarenhas, A.: Robust on-line computation of Reeb graphs: Simplicity and speed. ACM Trans. on Graph. **26**(3), 58.1–58.9 (2007)
21. Reeb, G.: Sur les points singuliers d'une forme de pfaff complement intergrable ou d'une fonction numerique. Comptes Rendus Acad. Science Paris **222**, 847–849 (1946)
22. Thomas, D.M., Natarajan, V.: Symmetry in scalar field topology. IEEE Trans. Vis. Comput. Graph. **17**(12), 2035–2044 (2011)
23. Weber, G.H., Bremer, P.T., Pascucci, V.: Topological landscapes: A terrain metaphor for scientific data. IEEE Trans. Vis. Comput. Graph. **13**(6), 1416–1423 (2007)