involve

Comparing embedded graphs using
average branching distance

Levent Batakci, Abigail Branson, Bryan Castillo,
Candace Todd, Erin Wolf Chambers and Elizabeth Munch

msp

Comparing embedded graphs using average branching distance

Levent Batakci, Abigail Branson, Bryan Castillo,
Candace Todd, Erin Wolf Chambers and Elizabeth Munch

(Communicated by Kenneth S. Berenhaut)

Graphs drawn in the plane are ubiquitous, arising from data sets through a variety of methods ranging from GIS analysis to image classification to shape analysis. A fundamental problem in this type of data is comparison: given a set of such graphs, can we rank how similar they are in such a way that we capture their geometric "shape" in the plane?

We explore a method to compare two such embedded graphs, via a simplified combinatorial representation called a *tail-less merge tree* which encodes the structure based on a fixed direction. First, we examine the properties of a distance designed to compare merge trees called the *branching distance*, and show that the distance as defined in previous work fails to satisfy some of the requirements of a metric. We incorporate this into a new distance function called *average branching distance* to compare graphs by looking at the branching distance for merge trees defined over many directions. Despite the theoretical issues, we show that the definition is still quite useful in practice by using our open-source code to cluster data sets of embedded graphs.

1. Introduction

Embedded graphs appear in a wide variety of applications, including map reconstructions, GIS data, shape analysis, and medical imaging. Such graphs are more than simple abstract representations, since they have geometric information attached, usually in the form of coordinates and edge lengths. When faced with data of this type, an immediate question to ask is then how to compare them, so that we can apply techniques like clustering, image recognition, or machine learning. In essence, we seek a *distance measure*, which given two input graphs returns a number representing how similar they are. Ideally, this value should be zero if they are the same, and take increasing values for increasingly different embedded graphs.

While solving this problem perfectly is of course difficult in practice, many examples of graph distances do exist in the literature. One well-known example is the graph-edit distance [Gao et al. 2010], which gives a cost to inserting or removing vertices and edges; the distance is then the minimum possible sequence of edge and vertex operations when transforming one graph into the other. Unfortunately, this particular measure does not take any embedding information into account, so two graphs with identical vertices and weights which are embedded in a very different configuration will still have zero distance.

In this paper, we develop and implement a technique to compare 2-dimensional embedded graphs, such as those generated from GIS data or skeletonization of images. We note that there are available options which are close to our work, where distances are specifically built to take the embedding into account [Ahmed et al. 2014; Cheong et al. 2009; Biagioni and Eriksson 2012; Alt et al. 2003; Karagiorgou and Pfoser 2012; Buchin et al. 2017]. However, we seek to develop a faster pipeline which incorporates graph simplification while still retaining at least part of the geometric embedding information. More specifically, we compare embedded graphs by replacing them with a simpler graph which still encodes some aspect of the structure.

Our algorithm fixes a direction in the plane and computes the *merge tree* of the graph, which is a tree with a real-valued function representing how the connected components of sublevel sets of the graph change in that direction; see, e.g., Figure 1. To be able to compare merge trees in a meaningful way, we need a distance function that is computable, accurate, comprehensive, and versatile. There are many possible options for metrics to compare merge trees specifically, including the edit distance [Sridharamurthy et al. 2020] and interleaving distance [Morozov et al. 2013; Munch and Stefanou 2019; Gasparovic et al. 2019; Yan et al. 2020]. In this paper, we use the branching distance for merge trees [Beketayev et al. 2014]. Then, to avoid bias caused by fixing a single a direction, we rotate the graphs, calculating the merge trees at each rotation, and compare all of the resulting merge trees. We take the median of these distances, naming this distance as *average branching distance*.

The branching distance for merge trees [Beketayev et al. 2014] builds on the idea of a branch decomposition for a broader class of trees, introduced in [Pascucci et al. 2005]. However, in the course of our work, we have discovered an issue with the definition presented in [Beketayev et al. 2014], caused by ignoring the infinite tail present in merge trees. The result is that the distance as defined does not satisfy the properties of a metric, and we provide counterexamples to that effect. We then prove that this issue propagates into the average branching distance, so it also is not a metric, semimetric, or pseudometric. We also explore its theoretical properties on simple classes of graphs such as convex polygons.

Despite this, we show that the average branch decomposition is still a useful tool for distinguishing embedded graphs in practice. To test its utility on practical examples, we tested it with visualizations such as dendrograms and lower-dimensional embeddings, using a range of available data sets. Our code is posted publicly [Batakci et al. 2020]. We found that despite the theoretical issues, the distance still is potentially useful for understanding real data.

2. Background

In this section, we give the relevant background terminology and results in graph theory and merge trees, culminating in the definition of the branching distance for merge trees (Definition 2.4).

2A. *Graph theory.* Here, we provide a brief discussion of relevant terms from traditional graph theory, following [Bondy and Murty 2008].

A *graph* $G = (V, E)$ consists of a set of vertices $V = V(G)$ and a set of unordered pairs of vertices called edges $E = E(G)$. An edge $e = uv$ is *incident* to both of its endpoints u and v. An *edge deletion* operation removes an edge from $E(G)$. A *vertex deletion* operation removes a vertex v from $V(G)$ and removes all edges incident to v from $E(G)$

A *loop* is an edge such that both endpoints are the same vertex. Two edges are called *parallel* if their endpoints share the same vertices. A *simple graph* has no loops or parallel edges. The *degree* of a vertex v is the number of edges incident to v, with loops counting as two edges.

A *subgraph* H of a graph G, denoted by $H \subseteq G$, is a graph with $V(H) \subseteq V(G)$ and $E(H) \subseteq E(H)$. Note that a subgraph of G can be obtained by performing a combination of vertex deletions and/or edge deletions on G. An *induced subgraph* of G is a subgraph obtained solely by vertex deletions in G. In this case, we say that H is a *subgraph induced by* A if $V(H) = A \subseteq V(G)$. Two graphs G and H are *isomorphic* when there exists a bijective mapping between the vertices of G and H that preserves adjacency.

A *path* in a graph is a sequence of ordered vertices $v_1, \ldots, v_n$ with edges $v_i v_{i+1}$ for $i = 1, \ldots, n-1$. A graph is *connected* if every pair of vertices can be connected by a path. A *connected component* of a graph G is a connected subgraph $H \subseteq G$ which is not contained in a larger connected subgraph. A *cycle* is a path whose starting and ending vertices are the same vertex. A graph is *acyclic* if it has no subgraph which is a cycle graph. A *tree* is a connected acyclic graph, and a *forest* is a graph solely consisting of trees.

We assume we have a *topological graph*, where each edge can be considered as a copy of the unit interval. Let G be a topological graph and let f be a function on G, $f : G \to \mathbb{R}$, providing every vertex and point on each edge of the graph with
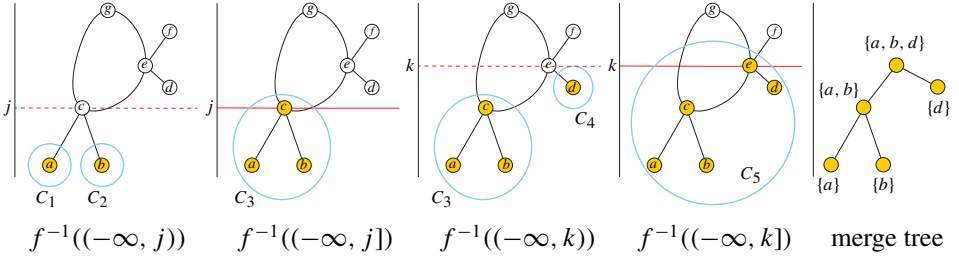
Figure 1. The sublevel sets of a graph below and at two values, $j, k \in \mathbb{R}$; and the tail-less merge tree of the graph.

a placement on the real number line. See an example in Figure 1. For a fixed $a \in \mathbb{R}$, the *sublevel set at a* is an induced subgraph H of G such that any vertex with a function value in (a, ∞) is deleted; that is, it is the subgraph induced by the set of vertices with function value $f(v) \leq a$. In this case, we write $H = f^{-1}((-\infty, a])$. The *sublevel set below a* is the subgraph induced by the set of vertices with function value $f(v) < a$. A component is a set of connected vertices in the sublevel set A.

2B. *Merge trees.* A merge tree is a representation of the connected components of sublevel sets of a graph with an $\mathbb{R}$-valued function. The vertices of a merge tree represent changes in connectedness of the input graph and its edges encode the relationships between these changes. In this way, the merge tree serves as a less complex representation of the graph. What follows is a formal definition of a merge tree closely related to the dendrogram definition of [Carlsson and Mémoli 2010].

A merge tree M is a structure defined on a graph G with a function $f : G \rightarrow \mathbb{R}$ which encodes the changing connected components of $f^{-1}(-\infty, a]$ for a shifting value a. For simplicity, we assume the function is entirely determined by values on the vertices $f : V(G) \rightarrow \mathbb{R}$ given by $v \mapsto f(v_i)$ and extended linearly to the edges. We use the following notation to define merge trees. Fix a connected set $A \subseteq \mathbb{R}$. Let $f^{-1}(A)$ denote the induced subgraph of G containing just the vertices $v \in G$ such that $f(v) \in A$. Fix a *connected subgraph* $C \subseteq G$. Define the *minima* $\mu(C)$ of C to be the set of vertices in C having no neighbors of lesser function value, that is,

$$\mu(C) = \{v \in V(C) \mid f(v) \leq f(w) \text{ for all } (vw) \in E(C)\}.$$

Let the *identified components* on A be defined as

$$\Gamma(A) = \{\mu(C) \mid C \text{ is a connected component of } f^{-1}(A)\}.$$

Note that $\Gamma(A)$ is a set of sets and that its elements are in one-to-one correspondence with the connected components. Let the *change in connectedness* $\Delta(a)$ at a be defined as $\Gamma((-\infty, a]) \setminus \Gamma((-\infty, a))$. Note that there is a change in connectedness in the sublevel sets of G at a if and only if $\Delta(a) \neq \varnothing$.

We note that we are defining merge trees as used in [Beketayev et al. 2014], which means that we inherited a bug in the definition from what they work with implicitly. Normally, a merge tree would have an infinite upwards tail representing the fact that a connected component is always visible in $f^{-1}(\infty, a]$ for a greater than the maximum value. However, [Beketayev et al. 2014] does not utilize this tail, cutting off the merge tree at the last merge. For this reason, we call this construction the *tail-less* merge tree to distinguish it from more standard mathematical constructions. For the remainder of the paper, we call this construction simply the "merge tree" unless there is potential for confusion.

Definition 2.1 (tail-less merge tree). The (tail-less) merge tree M of a graph G with function $f : G \to \mathbb{R}$ is a graph given by

$$V(M) = \{L \mid L \in \Delta(a), a \in \mathbb{R}\},$$

$$E(M) = \{L_1 L_2 \mid L_1 \in \Delta(a) \text{ for } a \in \mathbb{R}, \ L_2 \in \Gamma((-\infty, a)) \text{ and } L_2 \subset L_1\},$$

with function defined by

$$f_M : V(M) \to \mathbb{R}, \quad L \mapsto a \mid L \in \Delta(a).$$

To understand these definitions, consider the example of Figure 1. First, consider function value $j = f(c)$ for vertex c. The highlighted portion of the leftmost graph represents $f^{-1}((-\infty, j))$, which has only vertices a and b. The two connected components are identified as C_1 and C_2. Furthermore, $\mu(C_1) = \{a\}$ and $\mu(C_2) = \{b\}$. Thus, $\Gamma((-\infty, j)) = \{\{a\}, \{b\}\}$. The second graph represents $f^{-1}((-\infty, j])$, where the only connected component is identified as C_3. Here, $\mu(C_3) = \{a, b\}$ and $\Gamma((-\infty, j]) = \{\{a, b\}\}$. It follows that $\Delta(j) = \{\{a, b\}\}$.

Now we will compute $\Delta(k)$, where $k = f(e)$ for vertex e by considering the third and fourth graphs in Figure 1. We see that $\mu(C_3) = \{a, b\}$ and $\mu(C_4) = \{d\}$. Thus, $\Gamma((-\infty, k)) = \{\{a, b\}, \{d\}\}$ Furthermore, $\mu(C_5) = \{a, b, d\}$ and $\Gamma((-\infty, k]) = \{\{a, b, d\}\}$. It follows that $\Delta(k) = \{\{a, b, d\}\}$.

In the far right of Figure 1, we show the merge tree for the input graph and function. Each vertex v of the merge tree with $f(v) = a$ is labeled with its corresponding element $L \in \Delta(a)$. Edges are included between vertices L_1 and L_2 when $L_1 \subset L_2$.

2C. *Merge tree for a fixed direction.* In this paper, we will consider graphs with a map to $\mathbb{R}^2$. In the case that this map is injective, the result is an *embedded graph*.[1] We can then pick a direction in the plane by fixing an angle ω for the orientation and calculating function values for a vertex as the magnitude of the projection of the original position vectors onto ω. So, from the input $f : G \to \mathbb{R}^2$, we get a function

---

[1]While we will often be interested in embedded graphs in practice, much of the work in this paper does not require the assumptions of injectivity.
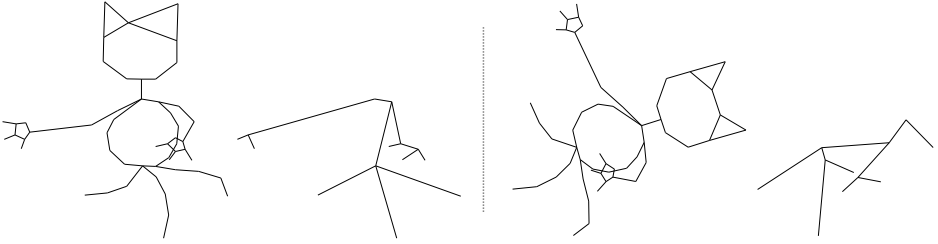
Figure 2. Two (nearly identical) embedded graphs at two different rotations with the associated merge trees.

$f_\omega : G \to \mathbb{R}$ for each angle $\omega \in [0, 2\pi)$. We normalize this function by shifting the function values to have median vertex value equal to 0. We get the same function if we think of rotating the graph in the plane and computing the height function in the vertical direction; i.e., $f_\omega(v)$ is given by the y-coordinate of $f(v)$. Then, we can compute the merge tree of the graph for each direction. See the example of Figure 2.

2D. *Branching distance.* In order to compare two merge trees, we use the following distance as defined in [Beketayev et al. 2014], which we call the *branching distance*. Note that this definition utilizes the tail-less merge tree construction as defined above.

In what follows, we assume the tail-less merge tree is a nonempty, connected graph. The root of the merge tree is the vertex with highest function value. A merge tree is trivial if it consists of a single vertex. For a nontrivial merge tree M, a *saddle* is any vertex with degree ≥ 2, and a *minimum* is any nonroot vertex with degree 1. For a trivial merge tree, the only available vertex is considered to be both the root and a minimum. A root branch is a pairing of some minimum m_r with the highest function value vertex s_r of the merge tree.

Definition 2.2 (branch decomposition). A branch decomposition B of a tail-less merge tree M is collection of pairs $B = \{(m_i, s_i)\}_i$ consisting of a minima m_i and either a saddle or the root s_i. Every vertex appears in at least one pair. Further, for each pair there is a descending path from the saddle s_i to the minimum m_i, no two such paths share an edge, and every edge appears in at least one such path.

We note that in the case of a trivial merge tree with a single vertex v, the only possible branch decomposition consists of the single degenerate branch $\{(v, v)\}$. For a nontrivial merge tree in general position (i.e., the number of edges adjacent and below any vertex is at most two), all saddles and minima (with the possible exception of the root) appear in exactly one pair of the branch decomposition. If a saddle vertex has k edges below it, then it will occur in $k - 1$ pairs. We can then represent these branches and the relationships between them in a graph of their own as follows.

Definition 2.3 (rooted tree representation). A rooted tree representation R of a branch decomposition B is a graph with vertex set given by $V(R) = B$. An edge

between (m, s) and (m', s') is included in $E(R)$ if and only if one of the saddles, s or s', is on the path between the other pair. That is, either s is on the path from m' to s', or s' is on the path from m to s. We denote the set of all rooted tree representations of a merge tree M by S_M.

See Figure 3 for an example of a merge tree with its possible branch decompositions and the associated rooted tree representations.

Some graphs may have only a single connected component across all nonempty sublevel sets, so we extend the definition of rooted tree representations to include trivial merge trees. A trivial merge tree is a single vertex with the function value of the lowest vertex of the original graph. The trivial merge tree's only rooted tree representation is a single vertex with both elements having the function value of the single vertex of the merge tree.

Branching distance utilizes comparisons between rooted tree representations in order to compare merge trees. To compare rooted tree representations R^X and R^Y of merge trees X and Y, we form a matching from one vertex set to the other. 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. Then a matching is an order-preserving isomorphism $\gamma : M^X \to M^Y$ for subsets of the vertices $M^X \subseteq V(R^X)$ and $M^Y \subseteq V(R^Y)$. The vertices in M^X and M^Y are called matched. The remaining vertices are said to be removed and are denoted by $E^X = V(R^X) \setminus M^X$ and $E^Y = V(R^Y) \setminus M^Y$. A matching is valid when the subgraphs of R^X and R^Y induced by M^X and M^Y respectively are trees and at least one root branch for each has not been removed.

Having fixed a matching $\gamma : M^X \to M^Y$, we now define the cost of the matching by incurring a cost for each vertex. The cost of matching two vertices $\gamma(u) = v$, where $u = (m_u, s_u) \in R^X$ and $v = (m_v, s_v) \in R^Y$, is the maximum of the absolute function value difference of their corresponding elements,

$$\mathrm{mc}(u, v) = \max \left(|m_u - m_v|, |s_u - s_v| \right).$$

The cost of removing a vertex $u \in R^X \cup R^Y$ is half the absolute function value difference of the elements of the vertex,
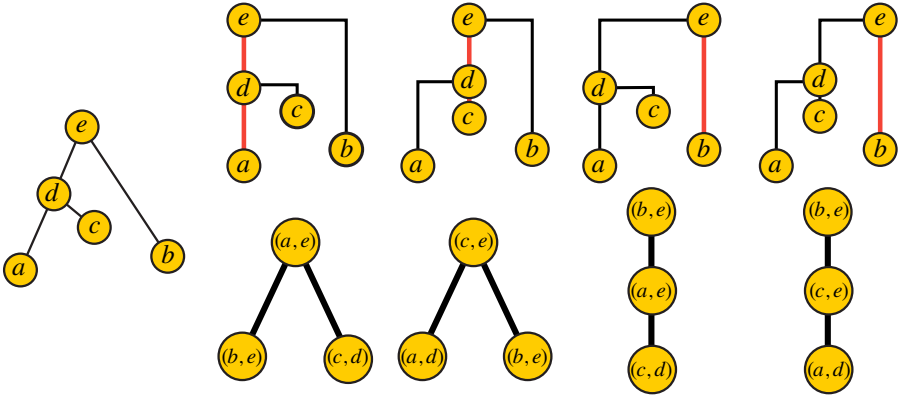
$$\mathrm{rc}(u) = \tfrac{1}{2} |m_u - s_u|.$$

We say that two rooted tree representations R^X and R^Y are ε-*similar* when we can find a valid matching $\gamma : M^X \to M^Y$ where the maximum cost does not exceed ε. That is,

$$\max_{u \in M^X} \mathrm{mc}(u, \gamma(u)) \leq \varepsilon \quad \text{and} \quad \max_{u \in E^X \cup E^Y} \mathrm{rc}(u) \leq \varepsilon.$$

Specifically, the smallest ε for which the above two inequalities hold is denoted by

$$\varepsilon_{\min}(R^X, R^Y) = \min_{\gamma : M^X \to M^Y} \max \left\{ \max_{u \in M^X} \mathrm{mc}(u, \gamma(u)), \max_{u \in E^X \cup E^Y} \mathrm{rc}(u) \right\}.$$

Figure 3. A merge tree M is shown at left. All of M's branch decompositions are shown in top right row, with each root highlighted red. The corresponding rooted tree representations are shown in the bottom right row.

Definition 2.4 (branching distance). The branching distance between two merge trees X and Y is the smallest $\varepsilon_{\min}$ out of every possible pair of rooted tree representations, namely

$$d_B(X, Y) = \min_{R^X \in S_X, R^Y \in S_Y} (\varepsilon_{\min}(R^X, R^Y)). \tag{1}$$

A naive approach to computing this distance would result in an exponential time complexity. Thus, we use the optimized algorithm described in [Beketayev et al. 2014]. The function ISEPSSIMILAR determines whether two merge trees can be matched within a given tolerance ε and is the core of the branching distance algorithm. The runtime complexity of ISEPSSIMILAR is $O(N^2 M^2 (N + M))$, where N and M are the number of leaves of the input trees. A binary search with a specified error tolerance is performed to determine $\varepsilon_{\min}$, the branching distance.

3. Properties of the branching distance

In this section, we consider and prove properties of the branching distance, showing that it is a semimetric but not a metric when defined on tail-less merge trees. We then define a new distance measure between two geometric graphs that is based on the branching distance, which we call the *average branching distance*. We prove that the average branching is a distance function but is not a metric, as two graphs can have average branching distance equal to 0 without being equivalent graphs.

Here, we define distance in the following way, as in [Conci and Kubrusly 2017].

Definition 3.1. Consider a function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ on a collection of objects $\mathcal{X}$. Some commonly used properties that such a function might satisfy are:

(i) (symmetry) For all $x, y \in \mathcal{X}$, $d(x, y) = d(y, x)$.

(ii) (nonnegativeness) For all $x, y \in \mathcal{X}$, $d(x, y) \geq 0$ and $d(x, x) = 0$.

(iii) (positiveness) For all $x, y \in \mathcal{X}$, $d(x, y) = 0$ implies $x = y$.

(iv) (triangle inequality) For all $x, y, z \in \mathcal{X}$, $d(x, y) \leq d(x, z) + d(z, y)$.

A function $d : \mathcal{X} \times \mathcal{X} \to \mathbb{R}_{\geq 0}$ is called a *distance* if it satisfies (i) and (ii). It is a *semimetric* if it satisfies (i)–(iii), but not necessarily (iv). It is a *pseudometric* if it satisfies (i), (ii), and (iv), but not necessarily (iii). It is a *metric* if it satisfies (i)–(iv).

3A. *Metric properties of branching distance.* In this section, we will show that the branching distance on tail-less merge trees satisfies properties (i)–(iii) above, making it a semimetric. However, we will also provide a counterexample to the triangle inequality, proving that the branching distance is not a metric. Throughout this section, let X, Y, and Z be merge trees. We will abuse notation by writing m and s for both the vertices in the merge tree that they represent, and for the function value of those vertices. We first show symmetry. To show that $d_B(X, Y) = d_B(Y, X)$, we will show that we obtain the same minimum $\varepsilon_{\min}$ when comparing X to Y as we obtain when comparing Y to X.

Lemma 3.2. $\qquad\qquad \varepsilon_{\min}(R^X, R^Y) = \varepsilon_{\min}(R^Y, R^X)$.

Proof. Let $u = (m_u, s_u) \in R^X$ and $v = (m_v, s_v) \in R^Y$ be vertices. Because of the symmetry of the absolute value expressions, we have

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

This means that $\max_{u \in M^X} \mathrm{mc}(u, \gamma(u)) = \max_{v \in M^Y} \mathrm{mc}(v, \gamma(v))$, so we obtain the same $\varepsilon_{\min}$ for the matching costs. By definition of union, $\max_{u \in E^X \cup E^Y} \mathrm{rc}(u) = \max_{u \in E^Y \cup E^X} \mathrm{rc}(u)$, so we obtain the same minimum ε for the removal costs. The $\varepsilon_{\min}$ are the same regardless of order, so the minimum ε that satisfies both will be the same regardless of the order in which we compare the rooted branchings. $\qquad\square$

Corollary 3.3. $\qquad\qquad d_B(X, Y) = d_B(Y, X)$.

Proof. Because our function considers all possible edits, and by Lemma 3.2, the minimum satisfying ε is the same regardless of the order of the rooted branchings, the cheapest edit from X to Y will be the cheapest edit from Y to X, and the minimum $\varepsilon_{\min}$ will be the same. $\qquad\square$

We next show nonnegativeness of the branching distance.

Lemma 3.4. $\qquad\qquad d_B(X, Y) \geq 0$.

Proof. The distance between any two graphs is nonnegative as all costs are nonnegative due to the absolute value expressions. $\qquad\square$

Lemma 3.5. $d_B(X, X) = 0.$

Proof. To show that $d_B(X, X) = 0$, we will show that if $X = Y$ then $d_B(X, Y) = 0$. Suppose $X = Y$. Then there is an isomorphism $\gamma : V(X) \to V(Y)$ which preserves function values. We select branches $(u, v) \in X$ and $(\gamma(u), \gamma(v)) \in Y$ for our branch decompositions such that all relations are preserved and corresponding vertices have equivalent function values. This means $\max \operatorname{mc}(u, \gamma(u)) = 0$ and we do not need to make any vertex deletions, so $\max rc(u) \leq 0$. Hence, $d_B(X, Y) = 0$.   □

Lemma 3.6. *If* $d_B(X, Y) = 0$ *then* $X = Y$.

Proof. Suppose $d_B(X, Y) = 0$. Then there exists a pair of rooted branchings $R^X \in B_X$ and $R^Y \in B_Y$ such that

$$\max_{u \in M^X} \operatorname{mc}(u, \gamma(u)) \leq 0 \quad \text{and} \quad \max_{u \in E^X \cup E^Y} rc(u) \leq 0.$$

If $\max_{u \in M^X} \operatorname{mc}(u, \gamma(u)) = 0$ then, for every pair of matching vertices u and v,

$$\max(|m_u - m_v|, |s_u - s_v|) = 0.$$

This means that all corresponding vertices in the merge trees must have equivalent function values.

The requirement that $\max rc(u) = 0$ is only satisfied when either all removals cost 0, or when we do not need to remove any vertices. A removal cost of 0 would mean that a minimum has the same function value as its saddle for some nonroot branch, which would not be included in a merge tree; hence the rooted tree representations must contain the same number of vertices.

We therefore have a bijective mapping between the vertices of X and Y that preserves function values and the relations between nodes. We consider isomorphic graphs with identical function values to be equivalent; hence $X = Y$.   □

We next give a counterexample to the triangle inequality.

Theorem 3.7. *The branching distance does not satisfy the triangle inequality, and therefore is not a metric.*

Proof. To prove this, we provide a counterexample to the triangle inequality, that is, a collection of three merge trees for which $d_B(X, Y) \geq d_B(X, Z) + d_B(Z, Y)$.

Consider the merge trees in Figure 4. When comparing X and Y we observe that the removal cost of any of the nodes will be less than the matching cost, so to determine the optimal edit from X to Y we need to select the root branches with the lowest comparison cost. We see that leaves b and f are the closest in function value, so our optimal edit will select (b, a) as our root branch for X, (f, d) as our root branch for Y, and delete (c, a) and (e, d). Our most expensive edit results from comparing the root branches and gives a minimum ε of 5; hence $d_B(X, Y) = 5$.
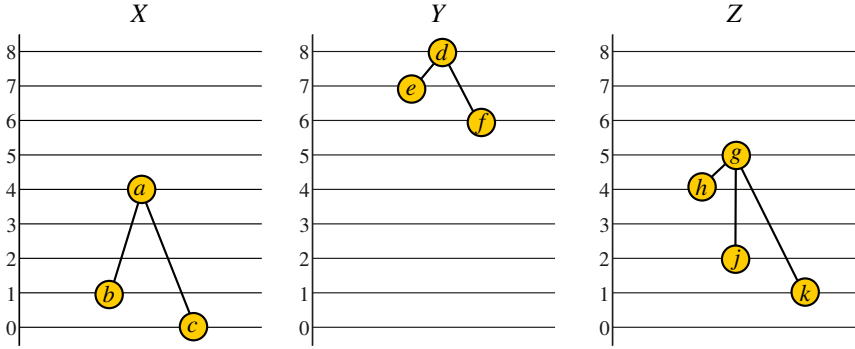
Figure 4. Merge trees for the counterexample to the triangle inequality.

When comparing Y and Z, we observe that the removal cost of the nodes will also be less than any matching cost, so to determine the optimal edit from Y into Z we need to select the root branches with the lowest comparison cost. We see that leaves f and h are closest in function value, so our optimal edit will select (f, d) as our root branch for Y, (h, g) as our root branch for Z, and delete (e, d), (j, g), and (k, g). Our most expensive edit results from comparing the root branches and gives a minimum ε of 3; hence $d_B(Y, Z) = 3$.

When comparing X and Z, we observe that removing (g, h) will always be cheaper than comparing it, so we have to select a different root branch for Z. We also observe that the removal cost of any remaining vertex will be more expensive than comparing it, so we can form an optimal edit by removing (h, g) and comparing the remaining vertices with the closest function values. We select either root branch for X and compare (b, a) to (j, g) and (c, a) to (k, g) to obtain an optimal edit with a minimum ε of 1; hence $d_B(X, Z) = 1$.

Putting this together, we have that $d_B(X, Y) = 5$, $d_B(Y, Z) = 3$, $d_B(X, Z) = 1$, and hence

$$d_B(X, Y) = 5 \geq 3 + 1 = d_B(X, Z) + d_B(Z, Y),$$

contradicting the triangle inequality. As a consequence of this, the branching distance is not a metric. □

3B. *Average branching distance.* Branching distance (Definition 2.4) is defined for comparing a pair of tail-less merge trees, but our goal is to compare graphs with a function to $\mathbb{R}^2$. We could take the merge tree of the graph for a single direction and use the branching distance, but this would result in a large amount of spatial information being be neglected, and there is no obvious "best" direction. Our solution is to compute the branching distance at some satisfactorily large amount of equally spaced angles and report the median of these measurements.

Since our rotation is about the origin, the range of function values is distorted in a way that disrupts branching distance's ability to compare merge trees. However, by shifting the function values when we calculate the merge tree, we are able to ensure that they stay centered about zero. In order to use branching distance, we first construct a merge tree corresponding to a subset of directions $\omega \in [0, \pi)$ as described in Section 2C. Note that as described in that section, we shift the merge tree function values to have an median value of 0 in order to compare the trees based on overall structure even when the functions are vastly different.

In our implementation, we begin with starting orientation $\omega = \frac{\pi}{2}$ and compute the merge tree as given by Definition 2.1. Then, to compare two graphs, we rotate both graphs simultaneously, and compute the merge tree for each in the given direction. The result is the average branching distance as follows.

Definition 3.8 (average branching distance (ABD)). Fix directions

$$\Omega = \{\omega_1, \ldots, \omega_n\} \subseteq [0, 2\pi)$$

and let $\mathrm{avg}(X)$ be fixed as either the mean or median of the set $X \subset \mathbb{R}$. Let M_{G,ω_i} be the merge tree of G rotated by angle ω_i. The average branching distance between two graphs G and H, $d_A(G, H)$, is defined to be

$$d_A(G, H) = \mathrm{avg}(\{d_B(M_{G,\omega_i}, M_{H,\omega_i} \mid i = 1, \ldots, n\}).$$

That is, $d_A(G, H)$ is either the median or the mean of the set of branching distances of their merge trees computed over a specified set of angles.

3C. *Metric properties of average branching distance.* In this section, we discuss properties of the average branching distance. In particular, we show that it is symmetric, and nonnegative as given in Definition 3.1. We provide a counterexample to positiveness, as well as to the triangle inequality, which shows that it is not a metric, semimetric, or pseudometric. For the rest of the section, let G, H, and J be arbitrary embedded graphs.

Lemma 3.9. $d_A(G, H) = d_A(H, G).$

Proof. We use the same set of angles to measure the distance and at each angle we are comparing the same pair of merge trees. This means we are always comparing the same pairs of merge trees for $d_A(G, H)$ and $d_A(H, G)$. By Corollary 3.3 we know $d_B(M_G, M_H) = d_B(M_H, M_G)$ for any merge trees M_G and M_H. This means the ordered set of distances for $d_A(G, H)$ is equivalent to the ordered set of distances for $d_A(H, G)$, so it follows that the medians of the sets will be equivalent.   □

Lemma 3.10. $d_A(G, H) \geq 0.$
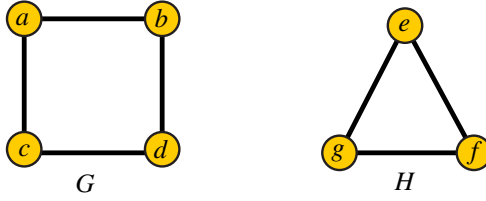
Figure 5. Graphs for the counterexample to $d_A(X, Y) = 0$ implies $X = Y$.

Proof. By Lemma 3.4 we know that $d_B(M_G, M_H) \geq 0$ for any merge trees M_G and M_H. This means every element of the ordered set of distances for two graphs is nonnegative; hence the median of the set must be nonnegative. $\qquad\square$

Lemma 3.11. $\qquad\qquad\qquad\qquad d_A(G, G) = 0.$

Proof. We are comparing the same orientations of identical graphs at each angle, so the merge trees being compared will be identical for any orientation. By Lemma 3.5, $d_B(M_G, M_G) = 0$ for any merge tree M_G; hence every element of any ordered set of distances for $d_A(G, G)$ is 0 and the median of the set will be 0. $\qquad\square$

We will now provide a counterexample to positiveness from Definition 3.1.

Theorem 3.12. *There is a pair of nonisomorphic graphs G and H for which* $d_A(G, H) = 0.$

Proof. Consider the nonisomorphic graphs G and H in Figure 5. We see that at any angle of comparison we are comparing two trivial merge trees which will always be shifted to function value 0 after their construction. The branching distance between any such pair of merge trees will be 0; hence every element of the ordered set of distances for $d_A(G, H)$ will be zero and the median of all these distances is 0. However, G is not equivalent to H as the graphs are nonisomorphic. As a consequence of this, average branching distance is neither a metric nor a semimetric. $\qquad\square$

We will now provide a counterexample to the triangle inequality, $d_A(G, H) \leq d_A(G, J) + d_A(J, H)$.

Theorem 3.13. *There is a collection of graphs G, H, and J for which $d_A(G, H) >$* $d_A(G, J) + d_A(J, H).$

Proof. Let the set of angles be $\{\frac{\pi}{2}\}$, and let G, H, and J be the graphs in Figure 6. The graphs' shifted merge trees M_G, M_H and M_J are identical to the original graphs at the standard orientation and we are only comparing at one angle; hence each pairwise ABD over $\{\frac{\pi}{2}\}$ will be the same as the respective branching distance.

When comparing M_G and M_H we observe that M_G is a trivial merge tree; hence the only rooted tree representation for M_G is a single root branch (a, a). Matching branch (c, b) is more expensive than removing it, so we remove (c, b) and match
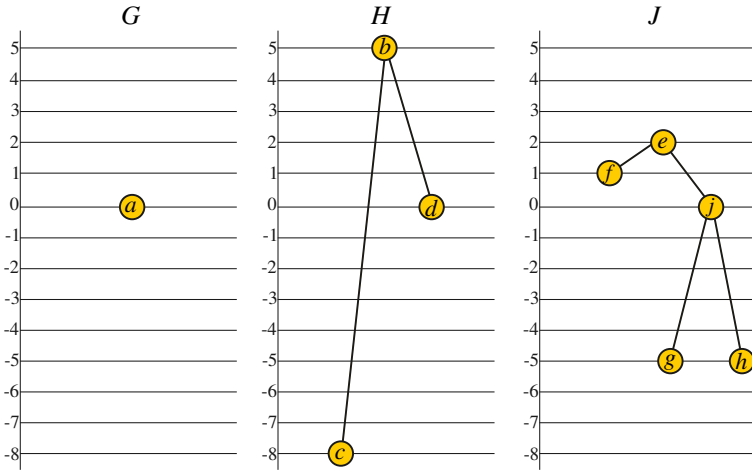
Figure 6. Graphs for the counterexample to the triangle inequality for ABD.

the single root branch of M_G to (d, b). The maximum cost for this optimal edit is 6.5 from removing (c, b); hence $d_A(G, H) = d_B(M_G, M_H) = 6.5$.

When comparing M_G to M_J, we observe that matching any branch containing g or h would be more expensive than removing it; hence we must choose either (f, e) or (j, e) as our root branch and remove all other branches. For either selection of root branch, the most expensive cost is 2.5 from removing (g, j) and (h, j); hence $d_A(G, J) = d_B(M_G, M_J) = 2.5$.

When comparing M_H to M_J, we observe that matching (c, b) to a branch containing g or h will be cheaper than removing it. We also see that we need to select (h, e) or (g, e) as our root branch in order to match (c, b) to a branch containing g or h. Removing (g, j) or (h, j) will always be cheaper than matching it to (b, d), so we remove the one we didn't select as our root branch. Matching or removing (f, e) and (d, b) does not effect our maximum cost; hence any optimal edit will have a maximum cost of 3 from matching the root branches. Therefore, $d_A(H, J) = d_M(M_H, M_J) = 3$.

We see that $d_A(G, H) = 6.5$, $d_A(G, J) = 2.5$, $d_A(H, J) = 3$, and $6.5 > 2.5 + 3$, contradicting $d_A(G, H) \leq d_A(G, J) + d_A(H, J)$.   □

Combining Theorems 3.12 and 3.13 gives us that ABD is neither a semimetric nor pseudometric, thus it is only a distance.

3D. *Average branching distance of convex polygons.* In the process of simplifying our data through merge tree construction, we lose information such as interior complexity and some shape data. We will show that average branching distance
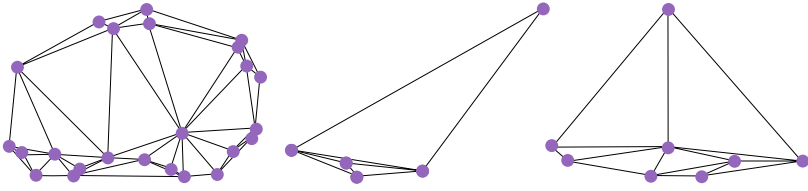
Figure 7. Graphs from the IAM database [Riesen and Bunke 2008] which have convex polygon borders.

will consider any convex polygons to be similar, and give an example of data not represented well by our function.

Lemma 3.14. *The merge tree from any direction of a convex polygon graph is trivial.*

Proof. For any orientation of a graph G, there will be some minimum function value k. Any two vertices with function value k must be connected by a horizontal path; otherwise we would need some higher function value reflex vertex connecting the two vertices, contradicting the fact that G is a convex polygon. Every other vertex must be degree 2 with a descending path to some bottom vertex with function value k; hence the entire graph is one connected component regardless of sublevel set. □

Corollary 3.15. $d_A(G, H) = 0$ *when G and H are graphs of convex polygons.*

Proof. By Lemma 3.14 we know that at any angle the entire merge tree for either graph is a trivial merge tree; hence we will always be comparing two trivial merge trees which will always be shifted to function value 0 after their construction. The branching distance between any such pair of merge trees will be 0; hence every element of the ordered set of distances for $d_A(G, H)$ is 0, and the median of this set is 0. □

This can be extended to graphs with convex polygon borders with no merges inside the polygon. For example, consider the graphs in Figure 7 from the IAM database [Riesen and Bunke 2008]: from a centered merge tree perspective, these graphs are identical from any angle, but we can clearly see that there are significant differences between them due to their shape and interior data.

4. Implementation, experimentation, and results

To explore the capabilities of our proposed distance function, we developed Python code (version 3.7.6) to compute the average branching distance. We implemented the algorithm presented in [Beketayev et al. 2014] to compute the branching distance. In order to use merge tree distances as a method for graph comparison, we also implemented an original method for merge tree construction, which we describe in

Section 4A. For tasks that required interactions with graphs, we used the NetworkX Python package [Hagberg et al. 2008]; we also utilized several other scientific computing packages [Hunter 2007; Pedregosa et al. 2011; Oliphant 2006; Virtanen et al. 2020]. Our code can be found in version 1.0 of our GitHub repository [Batakci et al. 2020].

4A. *Merge tree algorithm.* The design of our merge tree construction algorithm was motivated by the union-find data structure, which is commonly used to track elements of a set which are divided into disjoint subsets. Specifically, our algorithm keeps track of child and parent pointers to identify connected components at the graph's sublevel sets. Nodes in the original graph will be assigned child pointers, whereas nodes in the merge tree will be assigned parent pointers. We note that we use the term pointer here in the computer science sense, where, for each vertex, we are essentially associating a unique other vertex in the graph based on some desired property.

Fix a subgraph $H \subseteq G$, and note that this graph potentially has more than one connected component. We define the *representative* v_r of a vertex v to be the lowest function value vertex connected to v in the subgraph H. Assuming a generic function, i.e., all vertices have distinct function values, all nodes in the same component share the same, unique representative.[2] For this reason, we also call v_r the representative of the component containing v.

To find this representative, for each vertex $v \in V(H)$, we associate a *child pointer* which is in the same connected component as v but not necessarily adjacent. The representative is the only node in a component which is its own child pointer. The child pointers are set so that the representative of a vertex can be found by following child pointers until a node which is its own child pointer is found.

The *root* v_p of a vertex v is the node with the largest function value to which it is connected. Like the component representative, the root is the same for all vertices in a connected component. The *lower neighbors* of v are all vertices u adjacent to v such that $f(v) > f(u)$; *upper neighbors* are defined symmetrically. The *parent pointer* of a vertex in a merge tree is itself if it is the root, and its only upper neighbor otherwise. Similar to the representative, the root can be found by following the parent pointers of v until a vertex with is its own parent pointer is found.

We will use a sweep line algorithm, where we process the graph vertices in order of function value $a \in \mathbb{R}$. Setting $H_a = f^{-1}(-\infty, a]$ to be the sublevel set of the graph, we maintain the merge tree of H_a at each step as we increase a. As the merge tree is constructed, the child pointers of relevant nodes in H_a will be updated, and parent pointers will be maintained in the merge tree being constructed. The

---

[2]Note that our code does not require genericity, but the assumption simplifies the discussion of the algorithm.

Input: A graph $G = (V, E)$ with function $f : V \to \mathbb{R}$ such that $f(u) \neq f(v)$ for all $uv \in E$.

1: Initialize an empty graph $M = (V(M), E(M))$ with $V(M) = \varnothing$ and a function $f_M : V(M) \to \mathbb{R}$. (Note that each vertex in $V(M)$ will be associated to a unique vertex in $V(G)$ which caused its addition.)

2: Initialize all vertices in G with an empty child pointer, and an empty parent pointer.

3: Sort V ascending by function value:

4: **for** Vertex $v \in V$ with $f(v) = a$ **do**

5:      Let $R \subset V$ be the representatives of the components of v's lower neighbors in $H_a = f^{-1}(-\infty, a] \subseteq G$, found by recursively following children pointers until a vertex is found which is its own child.

6:      Let $R_M \subset V(M)$ be the vertices in M corresponding to the elements of R.

7:      **if** $|R| = 0$ **then**

8:          Add a new vertex v_M to $V(M)$.

9:          Set $f_M(v_M) = f(v)$.

10:         Set v_M as its own parent.

11:         Set v as its own child.

12:     **end if**

13:     **if** $|R| = 1$ **then**

14:         Set the one element in R as v's child.

15:     **end if**

16:     **if** $|R| \geq 2$ **then**

17:         Choose the vertex $c \in R$ with minimum function value, i.e., $f(c) \leq f(u)\ \forall u \in R$.

18:         Make a copy v_M of v in the merge tree M.

19:         Set $f_M(v_M) = f(v)$.

20:         For all $x \in R_M$, find the root of the connected component in M, x_p.

21:         Add the edge $x_p v_M$ to $E(M)$ for each x_p and set the parent of x_p to be v_M.

22:         Collapse all on-level neighbors of v_M.

23:         Set v_M as its own parent.

24:         Set c as the child of v and all $x \in R$.

25:     **end if**

26: **end for**

27: **return** M and f_M.

Algorithm 1. Create merge tree.

latter constitute the edge set of the newly constructed merge tree. This will be done to ensure that all parent-paths and child-paths in a connected component lead to the correct root and representative, respectively.

We assume that our input graph has been preprocessed as described in Algorithm 1. During this step, all adjacent nodes with the same function value will be collapsed into a single node; thus we can assume that our graph input has unique function value for adjacent vertices. This will not alter the merge tree and ensures that all edges have a specified upper and lower vertex.

Notice that by the definition of a merge tree, $\Delta(a)$ will only be nonempty when a minimum is encountered or components merge. Lines 7–12 of Algorithm 1 guarantee that all minima are detected and included in the merge tree. Furthermore,

lines 16–25 guarantee that all component merges are detected and included in the merge tree. Vertices in a merge tree are only connected when components merge. Line 21 accounts for all the connections between vertices. Trivially, f_M is correctly defined. Therefore, the proposed algorithm correctly constructs a merge tree based on Definition 2.1.

4B. *Experiments.* In this section, we describe our experiments to determine if ABD is a good measure of dissimilarity between graphs. We are looking to see whether graphs that appear similar are considered "close" to each other. To do this, we compute the ABD between several test data sets and use cluster analysis and dimensionality reduction to visualize whether our distance function upholds this idea. The plots shown in this section can be reproduced by running Plots.py from version 1.0 of our GitHub repository [Batakci et al. 2020].

We specify the orientations at which the branching distance between two graphs is calculated for ABD with the term *frames*. The *n* frames used to compute ABD are determined by the *n* evenly spaced orientations covering the interval $[0, 2\pi)$, where *n* is a positive integer parameter of the ABD function. For the following examples we use a small number of frames due to the simplicity of the input graphs. We do so because we do not anticipate significant changes between merge trees of the same graph if these graphs are rotated only slightly. For more intricate input graphs, such as maps, we would suggest a higher number of frames.

We consider graphs of different letters from the IAM graph database [Kersting et al. 2016; Riesen and Bunke 2008]. The graphs in this data set represent distorted letter drawings, so it was necessary for us to exclude graphs that we considered distorted beyond the point of recognition. Specifics of the process for identifying these outliers are documented in data/DataCleaning.py in version 1.0 of our repository. We use ABD on two sets of these graphs to compute pairwise distances. Because the letter graphs are so simple, we only compute the branching distance at 10 frames. We pass these values through a single linkage hierarchical clustering algorithm and construct a dendrogram to examine the results, as shown in Figure 8. This provides a visual representation of which graphs are most similar, which is more clear than looking at a matrix of values. As seen in the right of Figure 8, distinct graphs of the same letter have closer links between them than graphs of different letters. There are clear groupings of letters visible in the dendrogram, signified by color. Graphs that appear similar to human eyes to have smaller distances between them, which provides support for the validity of average branching distance as a measure of dissimilarity. We can see these result in another format if we pass the same distance matrix through a multidimensional scaling algorithm [Cox and Cox 1994], the results of which are shown in the bottom left of Figure 8. We see that letters whose appearances we can verify as similar are closer to each other on the scatter plot.
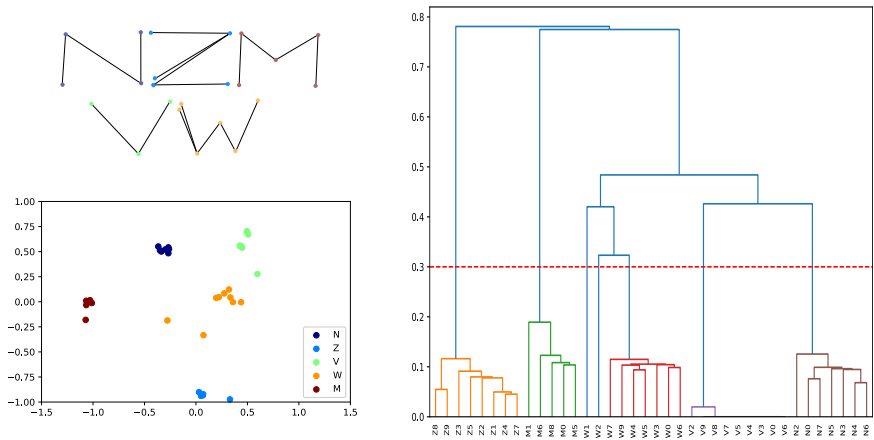
Figure 8. Clockwise from top left: A sample of letters from the IAM database; results of single-linkage hierarchical clustering on 38 letter graphs; MDS plot of the same graphs.

Next we consider graphs of skeletonized binary images from the ShapeMatcher5 model dataset [Macrini 2003]. ShapeMatcher is a program that can convert binary images to skeletons which can then be exported as graphs. The graphs given by this program had a noisy structure that resulted merge trees with many small branches, which increased our runtime greatly. In order for our program to finish running in a reasonable amount of time we smoothed out the structures, resulting in the graphs shown in Figure 9b. We use ABD on 20 skeletons of four different images to compute pairwise distances at different numbers of frames.

In Figure 10 we can see the results of MDS on four distance matrices with the same set of graphs but different matrix entries due to the number of frames used to compute each ABD. The only part of the distance matrix construction

(a) binary images   (b) graphs of skeletonized images
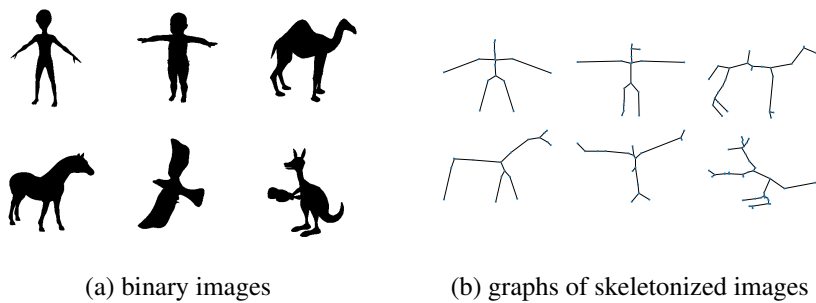
Figure 9. Clockwise from top left; alien, child, camel, kangaroo, eagle, horse (not to scale) [Macrini 2003].

(a) 5 frames

(b) 20 frames

(c) 50 frames

(d) 100 frames
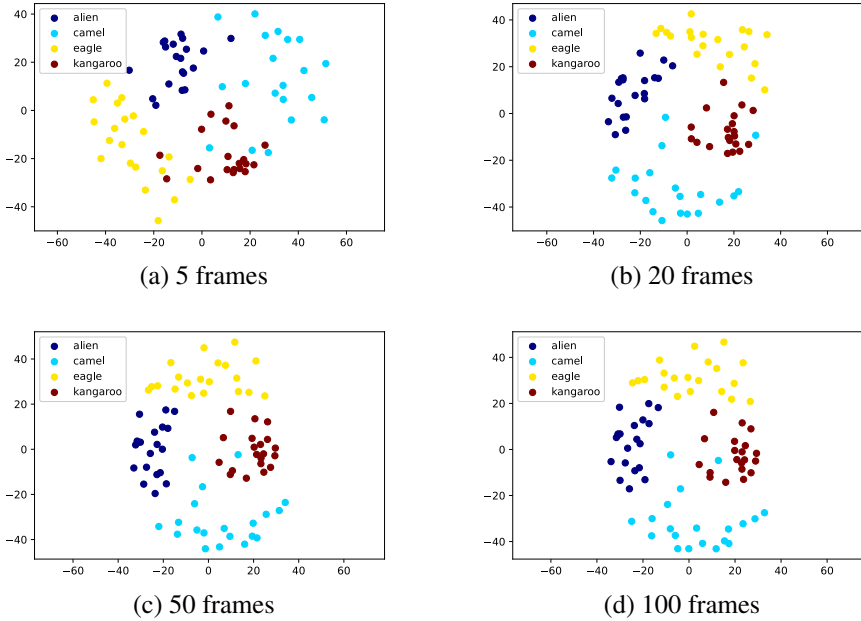
Figure 10. MDS using ABD on 20 skeletons for each of four categories of images shown in Figure 9.

that is dependent on frames is the ABD calculation, so the runtime for the matrix construction increases linearly as the number of frames increases.

In all four plots we see similar graphs clustering together. When we compute ABD at 20, 50, and 100 frames, our clusters seem to be more clearly defined than when we only use 5 frames. We can also see that there is not much difference between scatterplots 10(b), 10(c), and 10(d). This suggests that using 20 frames yields a good estimate for the exact average branching distance between these skeletons.

5. Discussion

We investigated the properties of the branching distance [Beketayev et al. 2014], showing that without inclusion of the infinite tail in the merge tree, the result is not a metric. The first interesting direction to go in with this work is to see if the definition of the branching distance can be modified to take the tail into account. Will fixing the distance in this way propagate into a distance measure as defined on the rotated, embedded graphs? For the average branching distance, we would also be interested to see if, rather than working with finitely many directions, defining the distance to be some sort of integral over the branching distance in all directions would give stronger metric properties.

The next direction for future work is to improve the computational techniques used in our code. Our implementation makes some trade-offs between runtime and accuracy. First, for a fixed direction, we implement a binary search for branching distance between the resulting merge trees which leaves behind a small error. While in theory this could be iterated enough to find an exact distance (since only one possible value would remain), that approach was not practical. In order to prevent an extremely large number of iterations, we choose a threshold for this error as a stopping criterion, which worked well enough in practice to give meaningful differentiation.

The second tradeoff between runtime and accuracy is caused by our choice to compute a specified set of angles rather than attempting to achieve an accurate representation of the entire range of angles $[0, 2\pi)$. We cannot test the entire interval as there would be an unbounded number of frames, and a mathematically accurate representation would require a determination of the cheapest rooted branching pairs and how they change due to rotations for the entire interval. It is possible that there is mathematical justification that can be undertaken for limiting the necessary number of directions, and indeed there is prior work on finding a rotation which would catch certain kinds of features in the data [Micka 2020] under various topological signatures (like the merge tree), although it mainly attempts to find a single direction to catch certain features. However, we leave this for future work as it is well beyond the scope of this project.

Another issue is that of normalization. We currently shift so that average vertex value is 0, which results in several instability issues. First off, this procedure is not immune to the addition of regular vertices and as such is not uniform when viewing the merge trees as a topological space. Second, we normalize each pair of merge trees at every direction choice, thus potentially leading to wildly different normalization shifts from one rotation to the next. In future work, we would seek to remedy these issues potentially by mean-shifting the original shapes.

Additionally, for this paper we have chosen to simultaneously rotate the graphs rather than attempt to compare each orientation of one to each orientation of the other. Note that rotating the graphs simultaneously assumes some sort of ideal starting alignment between the two graphs. For example, it would not make sense to simultaneously rotate two different orientations of the same graph. We also considered rotating only one graph, but concluded that this method would be inaccurate as it could consider irrelevant alignments. For example, two identical objects will seem very different if only one is being rotated. A consideration of all pairwise orientation matchings would increase the number of branching distance computations from n to n^2, where n is the number of specified frames. While this method may highlight different aspects of the compared graphs and, in particular, make the metric orientation invariant, it is too computationally expensive for our

current implementation to handle. However, it is possible that this modification would be of interest in applications such as shape comparison.

Finally, the data sets we considered involved only connected graphs, but in situations where we run into disconnected graphs, our implementation restricts to just the largest connected component. This of course will reduce the accuracy in practice, since parts of the data are ignored. Implementing an algorithm that alters ISEPSSIMILAR to account for disconnected components could improve our distance function. To do this, we would also have to define a new cost that basically accounts for deleting or adding an edge between two vertices such that the two vertices that were connected in one graph match two vertices that were not connected in the other graph, similar to edit distance calculations between graphs. This approach seems computationally prohibitive, but would likely lead to better comparisons if an approach can be made efficient.

Acknowledgements

We used the tools made available at https://csacademy.com/app/graph_editor, https://graphonline.ru/en/ and in [Macrini 2003] to draw and export graph data that could be read into Python as NetworkX graph objects. To examine the functionality of our code, we utilized graph data from several sources, including OpenStreetMap.org, the Stanford EarthWorks Library, and the Map Construction project [Ahmed et al. 2015]. These data do not appear in this paper but are included in our GitHub repository.

References

[Ahmed et al. 2014] M. Ahmed, B. T. Fasy, and C. Wenk, "Local persistent homology based distance between maps", pp. 43–52 in *Proceedings of the* 22*nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems* (Dallas, TX, 2014), edited by Y. Huang et al., ACM, New York, 2014.

[Ahmed et al. 2015] M. Ahmed, S. Karagiorgou, D. Pfoser, and C. Wenk, "A comparison and evaluation of map construction algorithms using vehicle tracking data", *GeoInformatica* **19** (2015), 601–632.

[Alt et al. 2003] H. Alt, A. Efrat, G. Rote, and C. Wenk, "Matching planar maps", *J. Algorithms* **49**:2 (2003), 262–283. MR Zbl

[Batakci et al. 2020] L. Batakci, A. Branson, E. Chambers, B. Castillo, L. Munch, and T. Todd, "Embedded graphs", GitHub repository, 2020, github.com/lizliz/SURIEM2020-EmbeddedGraphs.

[Beketayev et al. 2014] K. Beketayev, D. Yeliussizov, D. Morozov, G. H. Weber, and B. Hamann, "Measuring the distance between merge trees", pp. 151–165 in *Topological methods in data analysis and visualization*, *III*: *Theory*, *algorithms*, *and applications* (Davis, CA, 2013), edited by P. T. Bremer et al., Springer, 2014. Zbl

[Biagioni and Eriksson 2012] J. Biagioni and J. Eriksson, "Inferring road maps from global positioning system traces", *Transportation Res. Record* **2291**:1 (2012), 61–71.

[Bondy and Murty 2008] J. A. Bondy and U. S. R. Murty, *Graph theory*, Grad. Texts in Math. **244**, Springer, 2008. MR Zbl

[Buchin et al. 2017] M. Buchin, S. Sijben, and C. Wenk, "Distance measures for embedded graphs", preprint, 33rd European Workshop on Computational Geometry, 2017.

[Carlsson and Mémoli 2010] G. Carlsson and F. Mémoli, "Characterization, stability and convergence of hierarchical clustering methods", *J. Mach. Learn. Res.* **11**:47 (2010), 1425–1470. MR Zbl

[Cheong et al. 2009] O. Cheong, J. Gudmundsson, H.-S. Kim, D. Schymura, and F. Stehn, "Measuring the similarity of geometric graphs", pp. 101–112 in *Experimental algorithms* (Dortmund, Germany, 2009), edited by J. Vahrenhold, Lecture Notes in Comput. Sci. **5526**, Springer, 2009.

[Conci and Kubrusly 2017] A. Conci and C. Kubrusly, "Distances between sets: a survey", *Adv. Math. Sci. Appl.* **26**:1 (2017), 1–18. MR Zbl

[Cox and Cox 1994] T. F. Cox and M. A. A. Cox, *Multidimensional scaling*, Monogr. Stat. Appl. Probab. **59**, Chapman & Hall, London, 1994. MR Zbl

[Gao et al. 2010] X. Gao, B. Xiao, D. Tao, and X. Li, "A survey of graph edit distance", *PAA Pattern Anal. Appl.* **13**:1 (2010), 113–129. MR Zbl

[Gasparovic et al. 2019] E. Gasparovic, E. Munch, S. Oudot, K. Turner, B. Wang, and Y. Wang, "Intrinsic interleaving distance for merge trees", preprint, 2019. arXiv 1908.00063

[Hagberg et al. 2008] A. Hagberg, P. J. Swart, and D. A. Schult, "Exploring network structure, dynamics, and function using NetworkX", pp. 1–5 in *Proceedings of the* 7*th Python in Science Conference* (Pasadena, CA, 2008), edited by G. Varoquaux et al., DOE Office Scientific Technical Info., Oak Ridge, TN, 2008.

[Hunter 2007] J. D. Hunter, "Matplotlib: a 2D graphics environment", *Comput. Sci. Engrg.* **9**:3 (2007), 90–95.

[Karagiorgou and Pfoser 2012] S. Karagiorgou and D. Pfoser, "On vehicle tracking data-based road network generation", pp. 89–98 in *Proceedings of the* 20*th International Conference on Advances in Geographic Information Systems* (Redondo Beach, CA, 2012), edited by I. Cruz et al., ACM, New York, 2012.

[Kersting et al. 2016] K. Kersting, N. M. Kriege, C. Morris, P. Mutzel, and M. Neumann, Benchmark data sets for graph kernels, 2016, http://graphkernels.cs.tu-dortmund.de.

[Macrini 2003] D. A. Macrini, *Indexing and matching for view-based* 3*-D object recognition using shock graphs*, master's thesis, University of Toronto, 2003, https://hdl.handle.net/1807/118698.

[Micka 2020] S. A. Micka, *Searching and reconstruction: algorithms with topological descriptors*, Ph.D. thesis, Montana State University, 2020, https://scholarworks.montana.edu/xmlui/handle/1/15897.

[Morozov et al. 2013] D. Morozov, K. Beketayev, and G. Weber, "Interleaving distance between merge trees", preprint, 2013, https://mrzv.org/publications/interleaving-distance-merge-trees.

[Munch and Stefanou 2019] E. Munch and A. Stefanou, "The ℓ^∞-cophenetic metric for phylogenetic trees as an interleaving distance", pp. 109–127 in *Research in data science* (Providence, RI, 2017), edited by E. Gasparovic, Assoc. Women Math. Ser. **17**, Springer, 2019. MR Zbl

[Oliphant 2006] T. E. Oliphant, *Guide to NumPy*, Trelgol, Spanish Fork, UT, 2006.

[Pascucci et al. 2005] V. Pascucci, K. Cole-McLaughlin, and G. Scorzelli, "Multi-resolution computation and presentation of contour trees", LLNL Technical Report number UCRL-PROC-208680, 2005, http://www.pascucci.org/pdf-papers/orrery-2005.pdf.

[Pedregosa et al. 2011] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and É. Duchesnay, "Scikit-learn: machine learning in Python", *J. Mach. Learn. Res.* **12**:85 (2011), 2825–2830. MR

[Riesen and Bunke 2008] K. Riesen and H. Bunke, "IAM graph database repository for graph based pattern recognition and machine learning", pp. 287–297 in *Structural*, *syntactic*, *and statistical pattern recognition* (Orlando, FL, 2008), edited by N. V. Lobo et al., Lecture Notes in Comput. Sci. **5342**, Springer, 2008.

[Sridharamurthy et al. 2020] R. Sridharamurthy, T. B. Masood, A. Kamakshidasan, and V. Natarajan, "Edit distance between merge trees", *IEEE Trans. Vis. Comput. Graph.* **26**:3 (2020), 1518–1531.

[Virtanen et al. 2020] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, et al., "SciPy 1.0: fundamental algorithms for scientific computing in Python", *Nature Methods* **17** (2020), 261–272.

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

lab192@case.edu   *Case Western Reserve University, Hummelstown, PA, United States*

abby.branson@my.uu.edu   *Union University, Jackson, TN, United States*

bryancastillo@email.arizona.edu   *Department of Mathematics, The University of Arizona, Tucson, AZ, United States*

clt5441@psu.edu   *Department of Statistics, The Pennsylvania State University, University Park, PA, United States*

erin.chambers@slu.edu   *Department of Computer Science, Saint Louis University, St. Louis, MO, United States*

muncheli@msu.edu   *Department of Computational Mathematics, Science and Engineering, Department of Mathematics, Michigan State University, East Lansing, MI, United States*

involve

msp.org/involve

INVOLVE YOUR STUDENTS IN RESEARCH

Involve showcases and encourages high-quality mathematical research involving students from all academic levels. The editorial board consists of mathematical scientists committed to nurturing student participation in research. Bridging the gap between the extremes of purely undergraduate research journals and mainstream research journals, *Involve* provides a venue to mathematicians wishing to encourage the creative involvement of students.

Cover: Alex Scorpan

involve

2023   vol. 16   no. 3