Comparing Embedded Graphs Using Average Branching Distance

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

Abstract

Geospatial data is readily available but often noisy and inaccurate. In the field of map reconstruction, there is a need to be able to quantitatively compare maps in order to evaluate specific methods and data sources for reconstruction. In our work, we aim to do so by comparing embedded graphs representing road networks. The graphs we are interested in are too computationally expensive to compare directly, so we opt to instead compare their *merge trees*, a simplified representation of their structures.

We introduce a new, rigorous definition of merge trees and provide an original algorithm for constructing a merge tree. We examine the properties of a semi-metric distance designed to compare merge trees and incorporate this into a new distance function called *average branching distance* designed to compare graphs. We use clustering techniques to confirm the efficacy of average branching distance and consider its properties as distance function.

Levent Batakci
Case Western Reserve University
lab192@case.edu

Abigail Branson
Union University
abby.branson@my.uu.edu

Bryan Castillo
Mesa Community College
bryancastillo98@gmail.com

Candace Todd
The Pennsylvania State University
clt5441@psu.edu

Erin Chambers
Saint Louis University
erin.chambers@slu.edu

Elizabeth Munch
Michigan State University
muncheli@msu.edu

Contents

1 Introduction

Geospatial data is readily available but often noisy and inaccurate. In the field of map reconstruction, there is a need to be able to quantitatively compare maps in order to evaluate specific methods and data sources for reconstruction. In our work, we aim to do so by comparing embedded graphs representing road networks. The graphs we are interested in are too computationally expensive to compare directly, so we opt to compare a simplified representation of their structures instead. *Merge trees* are structures that represent the evolution of the connectedness of a graph as we look at expanding sub-level sets of the graph. To compute the merge trees of an embedded graph, we assign function values to the graphs vertices based on the their positions.

To be able to compare merge trees in a meaningful way, we need a distance function that is computable, accurate, comprehensive, and versatile. We considered interleaving distance, but it is not versatile because it does not allow us to compare graphs with a different number of vertices. After considering other distances, we choose branching distance [3], because it distinguishes noise from the data while respecting the relationship

between sub-level sets, due to the branch decompositions. This seems to meet most of our criteria, but some improvements are necessary, because branching distance is designed to compare merge trees, and we wish to compare graphs. Our solution is to rotate the graphs, calculating the merge trees at each rotation, then comparing those merge trees. We take the average of these distances, naming this distance as average branching distance. Some other improvements might be necessary as we continue, such as extending the branching distance to comparing disconnected merge forests instead of just connected merge trees.

To verify the accuracy of our distance function, we tested it with visualizations such as dendrograms and scatter-plots, using our preferred data sets. Our algorithm's ability to compare these in an intuitive way helps to verify the accuracy of our distance. We also discovered a few characteristics that our algorithm is not effective at comparing such as convex polygons and large graphs. So currently we avoid those characteristics in our testing until we can find a solution. This includes avoiding large maps, so a solution could be comparing small pieces of maps.

2 Background

2.1 Definitions

Here, we provide a brief discussion of relevant terms for our paper, following [4].

We define a *graph* G to consist of a set of vertices $V(G)$ and edges $E(G)$. 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 $E(G)$

One can obtain a *subgraph* F of a given graph G by performing vertex deletion or edge deletion on G. Then a subgraph is considered a graph, and we write $F \subseteq G$. An *induced subgraph* of G is a subgraph obtained solely by vertex deletions in G.

Let G be a graph and let f be a function on G such that $f : G \to \mathbb{R}$ providing every vertex and point on each edge of the graph with a placement on the real number line. For a fixed $a \in \mathbb{R}$, the *sub-level set* is an induced subgraph A of G such that any vertex with a function value in (a, ∞) is deleted and $A = f^{-1}((-\infty, a])$. A component is a set of connected vertices in the sub-level set A.

Two graphs G and H are *isomorphic* when there exists a bijective mapping between the vertices of G and H that preserves adjacency.

An *embedding* on some surface is a drawing of a graph on that surface such that edges only intersect at their endpoints. We will consider that surface to be $\mathbb{R}^2$.

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.

So a *path* is a simple graph such that the vertex set $V(G)$ is arranged into a consecutive linear sequence and no edges are repeated.A *cycle* is a special path whose start and end vertices are the same vertex. A *tree* T is a connected acyclic graph, and a *forest* is a graph solely consisting of trees.

The *degree* of a vertex v is the number of edges incident to v, with loops counting as two edges.

2.2 Merge Trees

A merge tree is a topological representation of a graph with a real-numbered function on its vertices on its vertices extending linearly to the edges.The graph experiences changes in connectedness across its sublevel sets determined by function value. The vertices of a merge tree represent changes in connectedness of the input graph and its edges encode the relationships between these changes. Merge trees serve as a less complex representation of a graph. What follows is a formal definition of a merge tree.

A merge tree M is a structure defined on a graph G with a function $f : V(G) \to \mathbb{R}$ given by $v \mapsto f(v_i)$. We use the following notation to define merge trees.

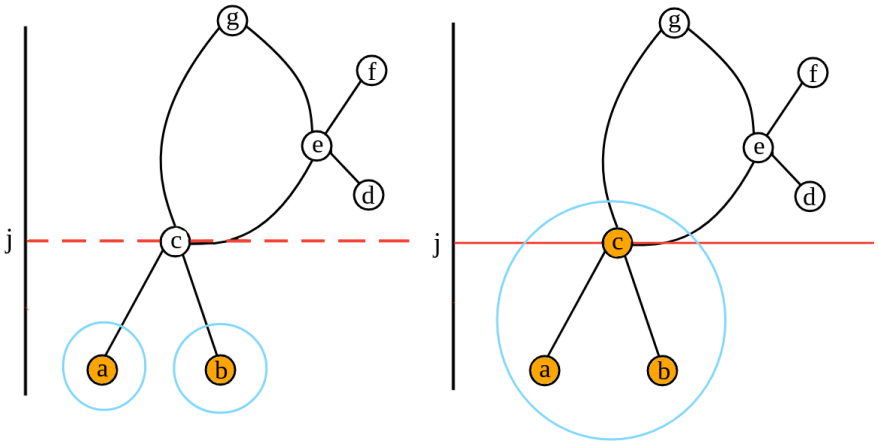
- Fix a connected set $A \subseteq \mathbb{R}$. Let $f^{-1}(A)$ denote the induced subgraph of G containing just the vertices $v \in G$ such that $f(v) \in A$.
- Fix a *connected subgraph* $C \subseteq G$. Define the *minima* $\mu(C)$ of C to be the set of vertices in C having no neighbors of lesser function value; that is
$$\mu(C) = \{v \in V(C) \mid f(v) \leq f(w) \forall (vw) \in E(C)\}. \tag{1}$$
- 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 identify 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 $\Delta(a) \neq \emptyset$.

Definition 1 (Merge Tree)**.** *Let the merge tree M of a graph G with function $f \to \mathbb{R}$ be defined by a set of vertices $V(M)$, edges $E(M)$, and function f_M such that*

$$\begin{aligned} V(M) &= \{L \mid L \in \Delta(a), a \in \mathbb{R}\}, \\ E(M) &= \{L_1 L_2 \mid \exists a \in \mathbb{R}, L_1 \in \Delta(a) \text{ and } L_2 \in \Gamma((-\infty, a)) \text{ and } L_2 \subset L_1\}, \end{aligned} \tag{2}$$

$$\begin{aligned} f_M : V(M) &\to \mathbb{R} \\ L &\mapsto a \mid L \in \Delta(a). \end{aligned} \tag{3}$$

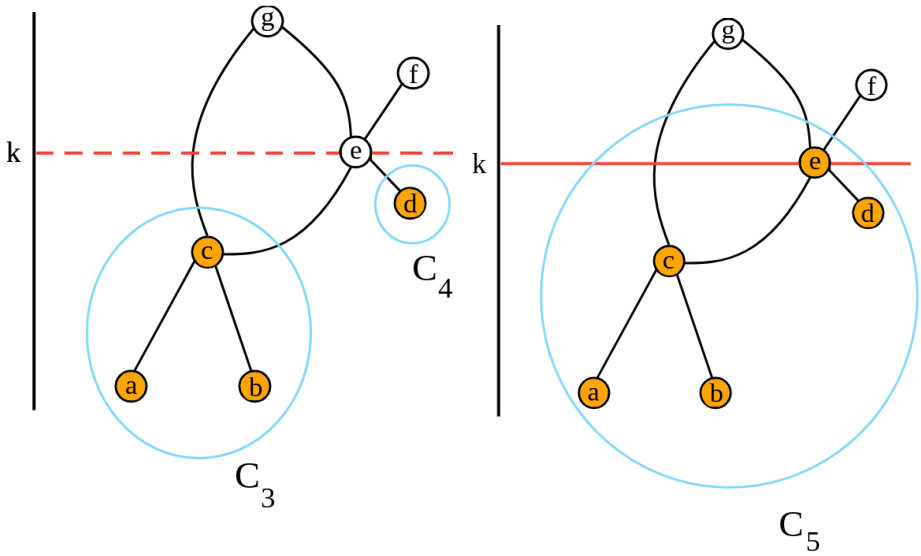
For illustrations of this notation and definition, see Figures 1-3 below. The j and k values are chosen since there is a change in connectedness at these values.



(a)$f^{-1}((-\infty, j))$ (b)$f^{-1}((-\infty, j])$

Figure 1: The sublevel sets of a graph below and at j

The highlighted portion of Figure 1a represents $f^{-1}((-\infty, j))$. Here, $j = f(c)$ for vertex c. Vertices a and b are the only vertices included in $f^{-1}((-\infty, j))$. The two connected components are identified as C_1 and C_2. Furthermore, $\mu(C_1) = \{a\}$ and $\mu(C_2) = \{b\}$. Thus, $\Gamma((-\infty, j)) = \{\{a\}, \{b\}\}$. Figure 1b represents $f^{-1}((-\infty, j])$. The only connected component is identified as C_3. Furthermore, $\mu(C_3) = \{a, b\}$ and $\Gamma((-\infty, j]) = \{\{a, b\}\}$. It follows that $\Delta(j) = \{\{a, b\}\}$.

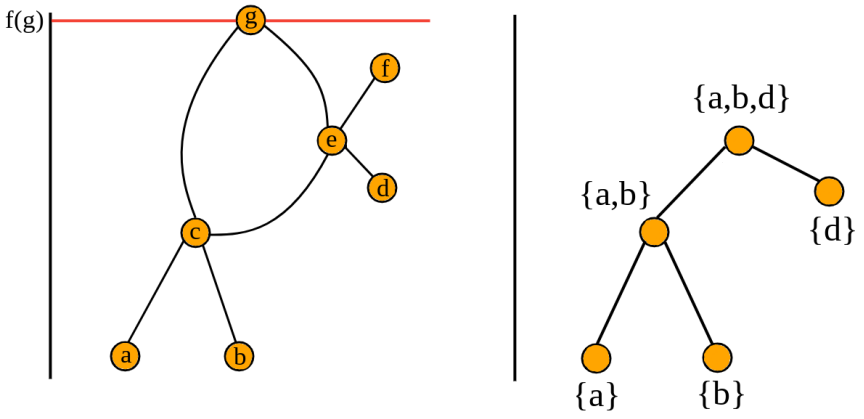


(a)$f^{-1}((-\infty, k))$ (b)$f^{-1}((-\infty, k])$

Figure 2: The sublevel sets of a graph below and at k

Now we will compute $\Delta(k)$, where $k = f(e)$ for vertex e. 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 $\Delta(k) = \{\{a, b, d\}\}$.

Figure 2 depicts the input graph and the corresponding merge tree on the right. Note that each vertex of the merge tree is labeled with the element of Δ that it corresponds to.



(a)$f^{-1}((-\infty, f(g)])$ (b)Merge Tree

Figure 3: The sublevel sets of a graph at f(g) and the completed merge tree

2.3 Branching Distance

An effective method for comparing two graphs' merge trees will be both computable (not NP-hard) and versatile. With these two considerations in mind, we use the following distance between merge trees as defined by Beketayev et al [3] which we call the *branching distance*.

For a merge tree M, a *saddle* is any vertex with degree ≥ 2, and a *minimum* is any vertex with degree 1. A root branch is a pairing of some minimum m_r with the highest function value vertex s_r of the merge tree.

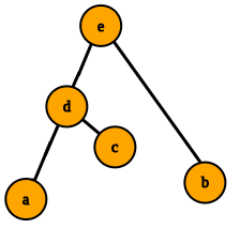


Figure 4: A merge tree M

Definition 2 (Branch Decomposition)**.** *A branch decomposition B represents the relations between a specified root branch (m_r, s_r) and all other minimum and saddle pairs such that for any pair, there is one descending path from a saddle to its minimum.*

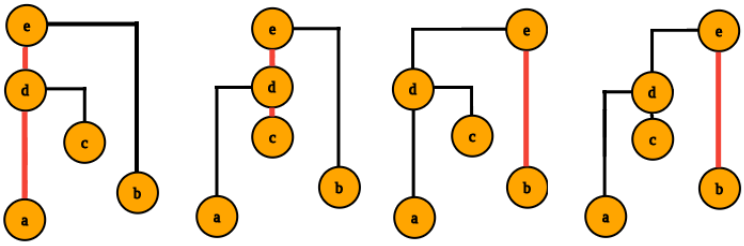


Figure 5: All of M from Figure 4's branch decompositions.

Definition 3 (Rooted Tree Representation)**.** *A rooted tree representation R of some branch decomposition B is a graph consisting of a set of ordered pair vertices with each ordered pair $(m, s) \in V(R)$ representing a saddle and minimum pair of B, and a set of edges $E(R)$ which describe the relations between branches. We denote the set of all rooted tree representations of a merge tree M to be S_M.*

Certain graphs may be one connected component across all non-empty sublevel sets, so we extend the definition of rooted tree representations to include trivial merge trees. We consider a *trivial* merge tree to be a single vertex with the function value of the lowest vertex of the original graph. We define the trivial merge tree's only rooted tree representation to be a single vertex with both elements having the function value of the single vertex of the merge tree.

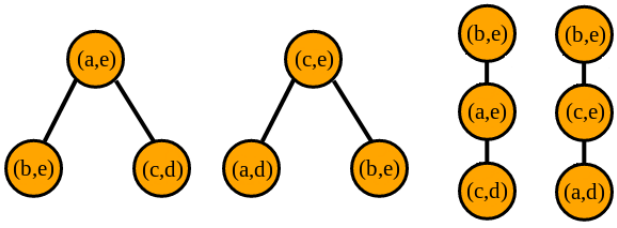


Figure 6: The rooted tree representations of the merge tree M shown in Figure 4.

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 try to form a matching from one to the other such that any vertex is either matched or removed. We denote the set of removed vertices for a rooted tree representation R^X to be E^X, and the set of matched vertices for R^X to be M^X. A comparison is valid when the induced subgraphs R^X/E^X and R^Y/E^Y are both trees and neither root branch has been removed.

The *matching cost* for two vertices $u \in R^X$ and $v \in R^Y$ is the maximum of the absolute function value difference of their corresponding elements,

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

and the *removal cost* for a vertex $u \in R^X \cup R^Y$ is half the absolute function value difference of the elements of the vertex,

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

Definition 4 (ε-Similarity)**.** *We say that two rooted tree representations R^X and R^Y are ε-similar when we can find two valid partitions of matched vertices M^X and M^Y and removed vertices E^X and E^Y, together with an order preserving isomorphism $\gamma : M^X \to M^Y$ where the maximum cost does not exceed ϵ. That is,*

$$\max_{u \in M^X} mc(u, \gamma(u)) \leq \varepsilon \tag{4}$$

and

$$\max_{u \in E^X \cup E^Y} rc(u) \leq \varepsilon. \tag{5}$$

The smallest ε for which the above two inequalities hold is denoted $\varepsilon_{min}(R^X, R^Y)$.

Definition 5 (Branching Distance)**.** *The branching distance between two merge trees X and Y is the smallest ε_{min} out of every possible pair of rooted tree representations, namely*

$$d_B(X, Y) = \min_{R^X \in S_X, R^Y \in S_Y} (\varepsilon_{min}(R^X, R^Y)). \tag{6}$$

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. [3]. The function `ISEPSSIMILAR` determines whether two merge trees are matchable within a given ε and is the core of the branching distance algorithm. The runtime complexity of `ISEPSSIMILAR` is $O(N^2M^2(N+M))$, 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 ε_{min}, the branching distance.

3 Further Examination of Branching Distance

3.1 Metric Properties of Branching Distance

Let X, Y, and Z be merge trees. We will show that branching distance satisfies the following properties making it a semi-metric [5]:

(i) $d_B(X, Y) = d_B(Y, X)$

(ii) $d_B(X, Y) \geq 0$

(iii) $d_B(X, X) = 0$

(iv) If $d_B(X,Y) = 0$ then $X = Y$

We will also provide a counterexample to the triangle inequality $d_B(X,Y) \leq d_B(X,Z) + d_B(Z,Y)$ in order to show that branching distance is not a metric.

To show that $d_B(X,Y) = d_B(Y,X)$, we will show that we obtain the same minimum ε_{min} when comparing X to Y as we obtain when comparing Y to X.

Lemma 3.1. $\varepsilon_{min}(R^X, R^Y) = \varepsilon_{min}(R^Y, R^X)$

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

$$\begin{aligned} mc(u,v) &= \max(|m_u - m_v|, |s_u - s_v|) \\ &= \max(|m_v - m_u|, |s_v - s_u|) \\ &= mc(v,u). \end{aligned} \tag{7}$$

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

Corollary 3.1.1. $d_B(X,Y) = d_B(Y,X)$

Proof. Because our function considers all possible edits, and 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 ε_{min} will be the same. □

Lemma 3.2. $d_B(X,Y) \geq 0$

Proof. The distance between any two graphs is non-negative as all costs are non-negative due to the absolute value expressions. □

Lemma 3.3. $d_B(X,X) = 0$

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 $\max mc(u, \gamma(u)) = 0$ and we do not need to make any vertex deletions, so $\max rc(u) \leq 0$. Hence, $d_B(X,Y) = 0$. □

Lemma 3.4. *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

$$\max_{u \in M^X} mc(u, \gamma(u)) \leq 0 \tag{8}$$

and

$$\max_{u \in E^X \cup E^Y} rc(u) \leq 0. \tag{9}$$

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

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

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 non-root 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 will now provide a counterexample to $d_B(X, Y) \leq d_B(X, Z) + d_B(Z, Y)$. Consider the merge trees in Figure 7.

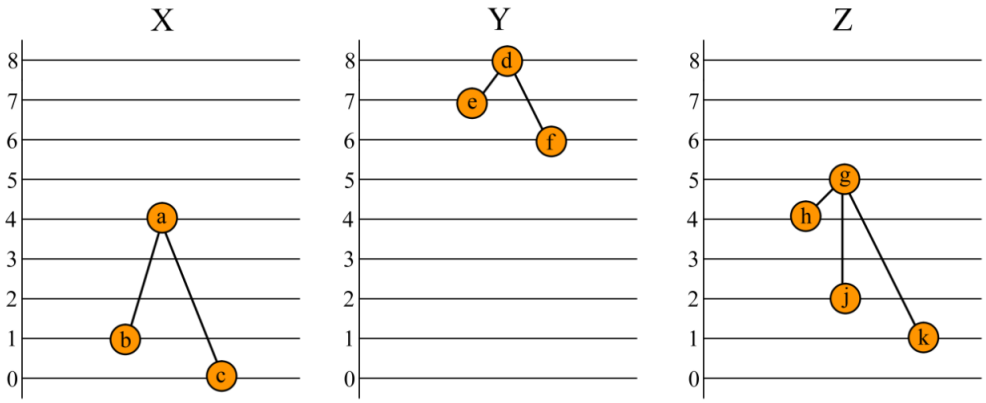


Figure 7: Merge Trees for the counterexample to the triangle inequality

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 $d_B(X, Y) = 5$.

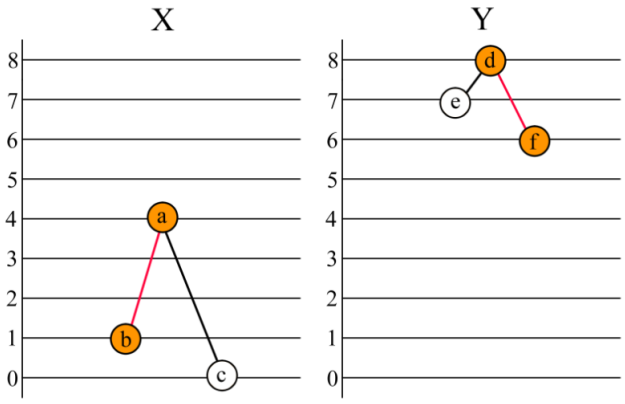


Figure 8: Optimal edit from X to Y

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 $d_B(Y, Z) = 3$.

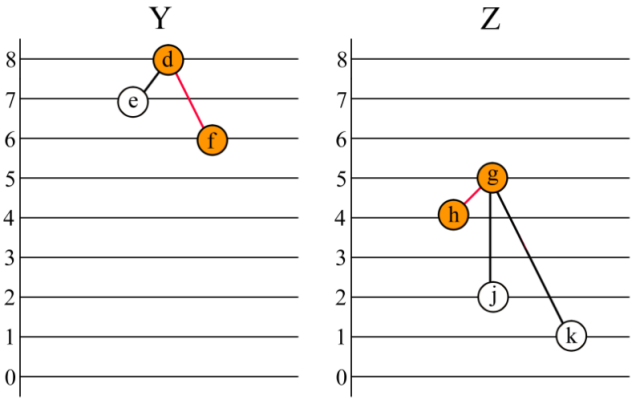


Figure 9: Optimal edit from Y to Z

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 $d_B(X, Z) = 1$.

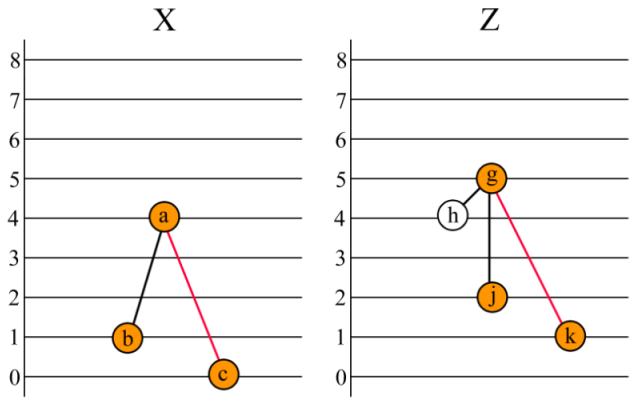


Figure 10: Optimal edit from X to Z

We see that $d_B(X,Y) = 5$, $d_B(Y,Z) = 3$, $d_B(X,Z) = 1$, and $5 > 3+1$ contradicting $d_B(X,Y) \leq d_B(X,Z) + d_B(Z,Y)$. As a consequence of this, branching distance is not a metric.

3.2 Average Branching Distance

Branching distance is effective for comparing merge trees, but our goal is to compare embedded graphs in $\mathbb{R}^2$. In order to use branching distance we must first construct a merge tree corresponding to each graph.

To construct these merge trees we specify an angle for the orientation and calculate function values by projecting the original position vectors of the graph's vertices onto the specified direction vector ω. The function values for the specified orientation will be the magnitudes of the projections onto ω. A direction of $\frac{\pi}{2}$ will give the standard orientation with the function values of the vertices representing their corresponding y values.

After a merge tree is constructed, we shift the function values on the vertices by setting the average equal to zero and shifting the rest of the values accordingly. This is a necessary step when comparing graphs with function values that are inherently different such as latitude and longitude. Without shifting, branching distance may return high values when comparing two maps from different areas of the globe no matter how similar their structure.

Comparing merge trees at only one angle 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 average 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 and are validly comparable.

We found rotating both graphs simultaneously to be reasonably effective and efficient. This method 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. Rotating the input graphs independently was another option, but this method proved too computationally expensive. This method require us to compute n^2 comparisons instead of n, where n is the number of angles. This subject will be further addressed in the discussion section.

Definition 6 (Average branching distance (ABD))**.** *The average branching distance between two graphs G and H, $d_A(G, H)$, is either the median or the mean of the ordered set of branching distances of their shifted merge trees computed over a specified set of angles.*

3.3 Metric Properties of Average Branching Distance

Let G, H, and J be graphs. We will show that average branching distance satisfies the following properties making it a distance function:

(i) $d_A(G, H) = d_A(H, G)$

(ii) $d_A(G, H) \geq 0$

(iii) $d_A(G, G) = 0$

We will also provide a counterexample to $d_A(G, H) = 0$ implies $G = H$ and to $d(G, H) \leq d(G, J) + d(J, H)$ in order to show that average branching distance is not a metric, semi-metric, or pseudo-metric.

Lemma 3.5. $d_A(G, H) = d_A(H, G)$

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 3.1.1 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 averages of the sets will be equivalent. □

Lemma 3.6. $d_A(G, H) \geq 0$

Proof. By lemma 3.2 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 non-negative, hence the average of the set must be non-negative. □

Lemma 3.7. $d_A(G, G) = 0$

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 3.3 $d_B(M_G, M_G) = 0$ for any merge tree M_G, hence every element of any ordered set of distances for $d_A(G, G)$ is 0 and the average of the set will be 0. □

We will now provide a counter-example to $d_A(G, H) = 0$ implies $G = H$. Consider the following non-isomorphic graphs G and H in Figure 11.

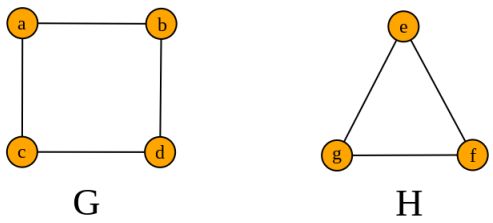


Figure 11: Graphs for the counterexample to $d_A(X, Y) = 0$ implies $X = Y$

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 average of all these distances is 0. However, G is not equivalent to H as the graphs are non-isomorphic. As a consequence of this, average branching distance is neither a metric nor a semi-metric.

We will now provide a counter example to the triangle inequality, $d_A(G, H) \leq d_A(G, J) + d_A(J, H)$. Let the set of angles be $\{\frac{\pi}{2}\}$, and let G,H, and J be the graphs in Figure 12. 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.

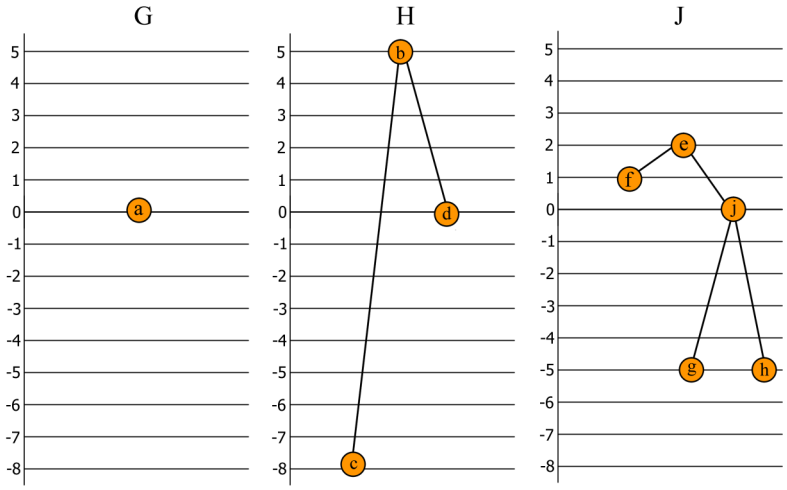


Figure 12: Graphs for the counterexample to the triangle inequality

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 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 $d_A(G, H) = d_B(M_G, M_H) = 6.5$.

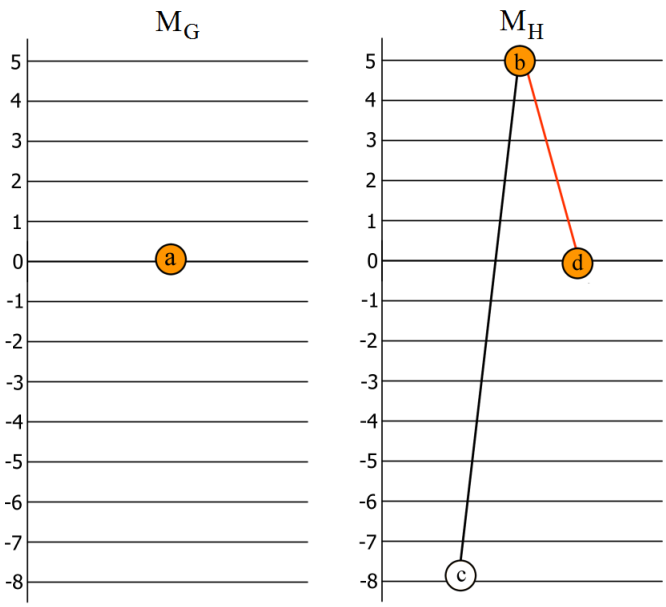


Figure 13: Optimal edit from M_G to M_H

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 $d_A(G, J) = d_B(M_G, M_J) = 2.5$.

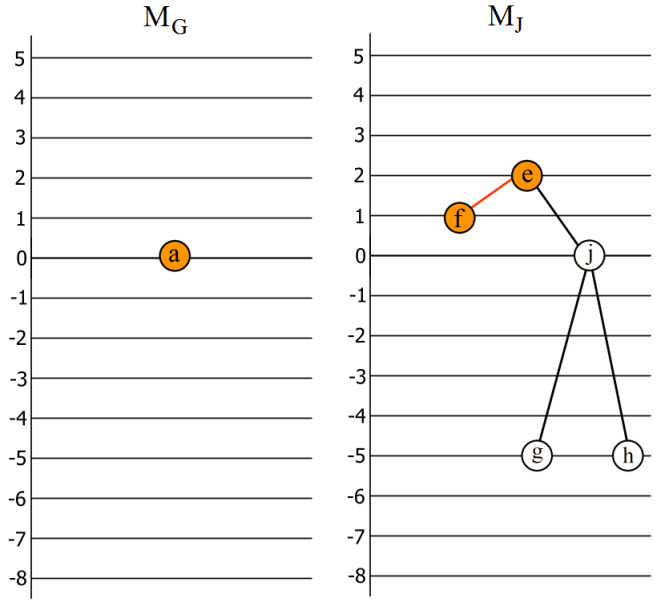


Figure 14: Optimal edit from M_G to M_J

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, $d_A(H, J) = d_M(M_H, M_J) = 3$.

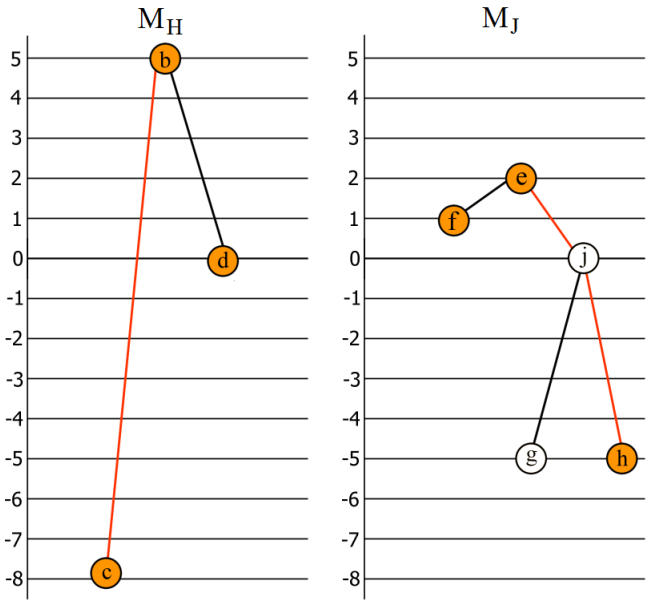


Figure 15: Optimal edit from M_H to M_J

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 $d_A(G, H) \leq d_A(G, J) + d_A(H, J)$. As a consequence of this, average branching distance is not a pseudo-metric.

4 Implementation, Experimentation, and Results

To explore the capabilities of our proposed distance function, we implemented the algorithms suggested by Beketayev et al. [3] in Python 3.7.6 with an extended functionality to act as the average branching distance mentioned before. In order to use merge tree distances as a method for graph comparison, an original method for merge tree construction was also implemented.

For tasks that required interactions with graphs, the NetworkX [6] Python package was used. We also utilized several other scientific computing packages [7, 11, 10, 17]. The work mentioned can be found in version 1.0 of our GitHub repository at `https://github.com/lizliz/SURIEM2020-EmbeddedGraphs` [2].

Due to the size of geospatial data, graphs of smaller objects like letters [8, 13] and skeletonized binary images [15, 12, 9, 9] were used to explore the capabilities of our distance function. Using these simpler graphs not only resulted in shorter run times for our algorithms, they also allowed us to visually confirm that our distance function yielded results that were compatible with our notion of apparent similarity. We expand upon these results in Section 4.2

4.1 Merge Tree Algorithm

The design of our merge tree construction algorithm was motivated by the union-find data structure. 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 define the *representative* v_r of a vertex v to be the lowest function valued vertex connected to v. All nodes in the same component share a unique representative. Also, the representative is the only node in a component which is its own child pointer. The representative of a vertex can be found by following child pointers until a node which is its own child pointer is found. The way that child pointers are set and updated ensures that this can always be done. The *lower neighbors* of v are all vertices u adjacent to v such that $f(v) > f(u)$.

The *parent pointer* of a vertex in a merge tree is its only neighbor of higher function value. Nodes which have the highest function value in their component are their own parent pointers. The *root* v_p of a vertex v is the node with the largest function value to which it is connected. Similar to the representative, the root can be found by following the parent pointers of v until a vertex which is its own parent pointer is found. When components merge, we connect their parent representatives to the new node.

As the merge tree is constructed, the child and parent pointers of relevant nodes will be updated at each step. 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.

Furthermore, we have chosen to include a pre-processing step before constructing the merge tree. During this step, all adjacent nodes with the same function value will be collapsed into a single node. This won't alter the merge tree and will ensure that all edges indicate a lower neighbor relationship.

Algorithm 1 Create merge tree

Pre-process the graph collapse all adjacent nodes with the same function value.

Create an empty graph M
Create a function $f_M : V(M) \to \mathbb{R}$
Create a list V of the vertices in the graph
Sort V ascending by function value
for Every vertex v in V **do**
    Create a set R of all roots of v's lower neighbors
    Create a set R_M of all vertices in M corresponding to the elements of R

    if $|R| = 0$ **then**
        Create a copy v_M of v in the merge tree M
        Set $f_M(v_M) = f(v)$
        Set v_M as its own parent
        Set v as its own child
    end if

    if $|R| = 1$ **then**
        Set the one element in R as v's child
    end if

    if $|R| \geq 2$ **then**
        Choose a vertex $c \in R$ such that $\forall u \in R, f(c) \leq f(u)$
        Make a copy v_M of v in the merge tree M
        Set $f_M(v_M) = f(v)$

26: For all $x \in R_M$, connect x_p to v_M
27: Update the parents of all x_p
28: Collapse all on-level neighbors of v_M
29: Set v_M as its own parent
30: Set c as the child of v and all $x \in R$
31: **end if**
32: **end for**
33: Return M and f_M

After a merge tree is constructed, we shift the function values as described in Section 3.2.

Notice that by the definition of a merge tree, $\Delta(a)$ will only be non-empty when a minimum is encountered or components merge. Lines 9-13 guarantee that all minima are detected and included in the merge tree. Furthermore, lines 19-31 guarantee that all component merges are detected and included in the merge tree. Vertices in a merge tree are only connected when components merge. Lines 22 and 26 account for all the connections between vertices. Trivially, f_M is correctly defined. Therefore, the proposed algorithm correctly constructs a merge tree based on Definition 1.

4.2 Experimentation

We describe our next proof-of-concept experiment to determine if ABD is a good measure of dissimilarity between graphs. This would mean graphs that appear similar are considered "close" to each other.

We begin with a simpler graph dataset 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 our version 1.0 of GitHub repository.

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 is 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.

In our implementation, *average* is assumed to refer to median rather than mean unless stated otherwise [1]. In this section, all examples shown used median function value to shift merge tree values and median Branching distance for ABD calculations. We favor the median over the mean due to its resistance against outliers.

We consider graphs of different letters from the IAM Graph Database [8, 13]. 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 [2]. 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 [17] to examine the results, as shown in Figure 16. This provides a visual representation of which graphs are most similar, which is more clear than looking at a matrix of values.

[1] "*In our implementation... unless stated otherwise*", meaning `average` is parameter in our code with a default value of `''median''` and an optional value of `''mean''`. See the `average_distance()` function in DataCalculations.py and `shift_f()` in Merge.py in version 1.0 of our repository

[2] The graphs we considered bias due to distortion "beyond recognition" as well as our process for identifying these outliers are documented in data/DataCleaning.py in version 1.0 of our repository

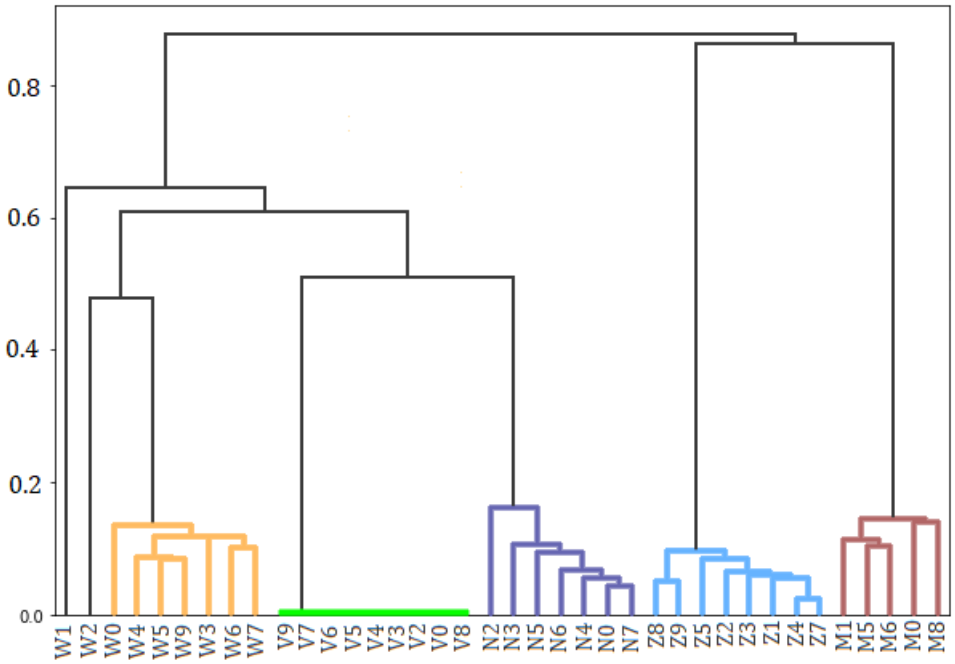
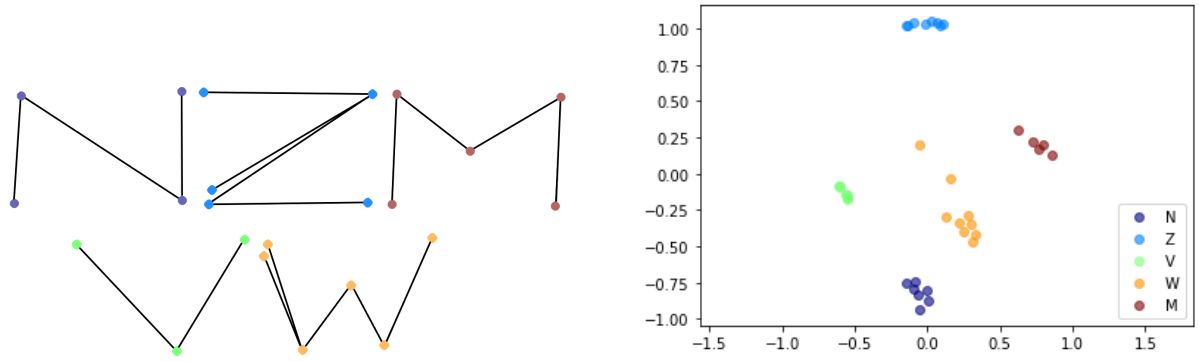


Figure 16: Results of Single-Linkage Hierarchical Clustering on 38 letter graphs

As seen in Figure 16, 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.

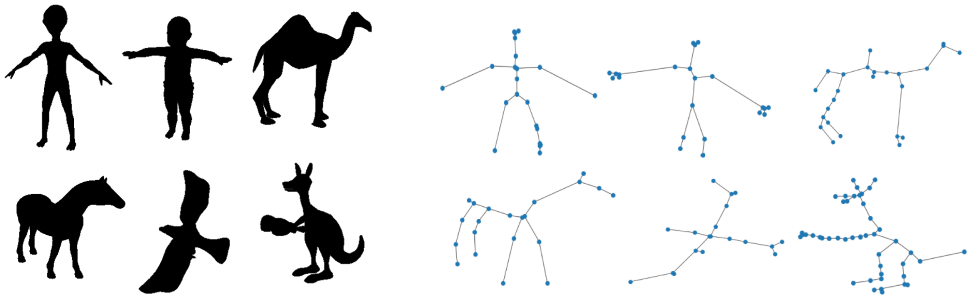


(a) Sample of letter graphs from IAM Database

(b) MDS on 38 letter graphs

Figure 17: Results of MDS on an average branching distance matrix

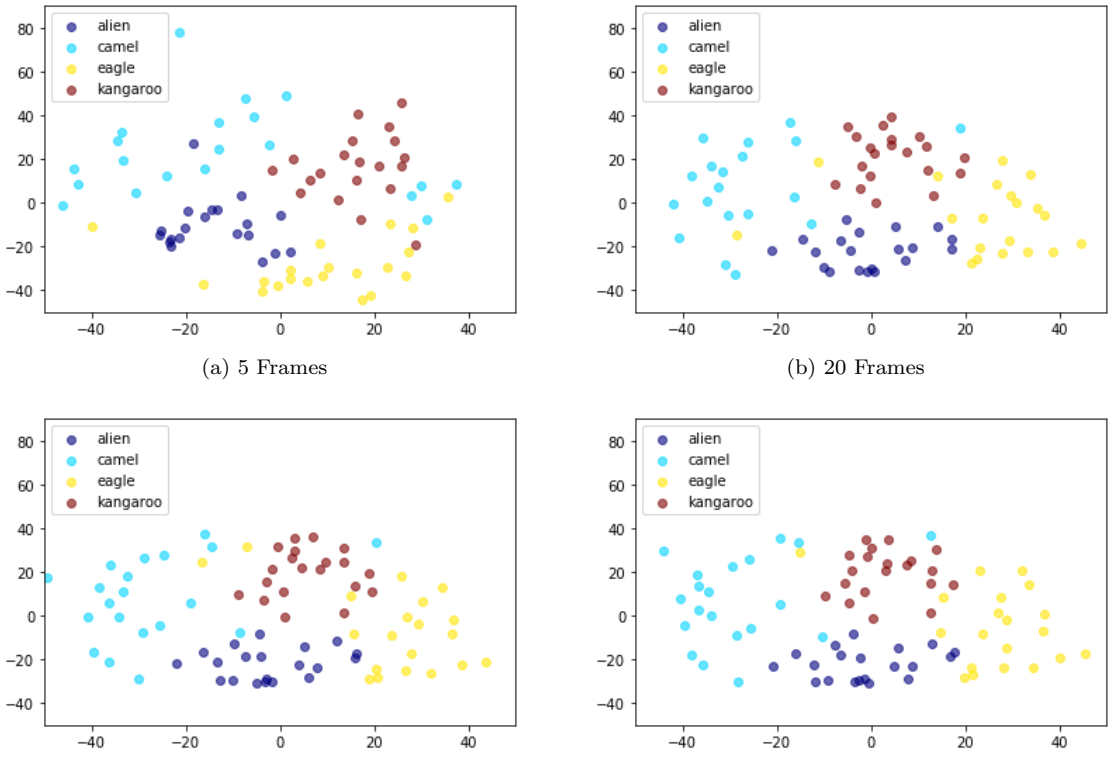
We can see these result in another format if we pass the same distance matrix through a multidimensional scaling algorithm [17], the results of which are shown in Figure 17b. We see that letters whose appearances we can verify as similar are closer to each other on the scatter plot.



(a) Binary images (b) Graphs of skeletonized images

Figure 18: Clockwise from top left; alien, child, camel, kangaroo, eagle, horse (not to scale) [9]

Next we consider graphs of skeletonized binary images from the ShapeMatcher5 model dataset [9]. 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 18b. We use ABD on 20 skeletons of 4 different images to compute pairwise distances at different numbers of frames.



(a) 5 Frames (b) 20 Frames

(c) 50 Frames (d) 100 Frames

Figure 19: MDS using ABD on 20 skeletons for each of 4 images

In Figure 19 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 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 19(b), 19(c), 19(b), and 19(d). This suggests that using 20 frames yields a good estimate for the exact average branching distance between these skeletons.

5 Discussion and Future Work

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

Lemma 5.1. *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 two 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 5.1.1. *$d_A(G, H) = 0$ when G and H are graphs of convex polygons.*

Proof. By lemme 5.1 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 average 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 following graphs taken from the IAM database[13]:

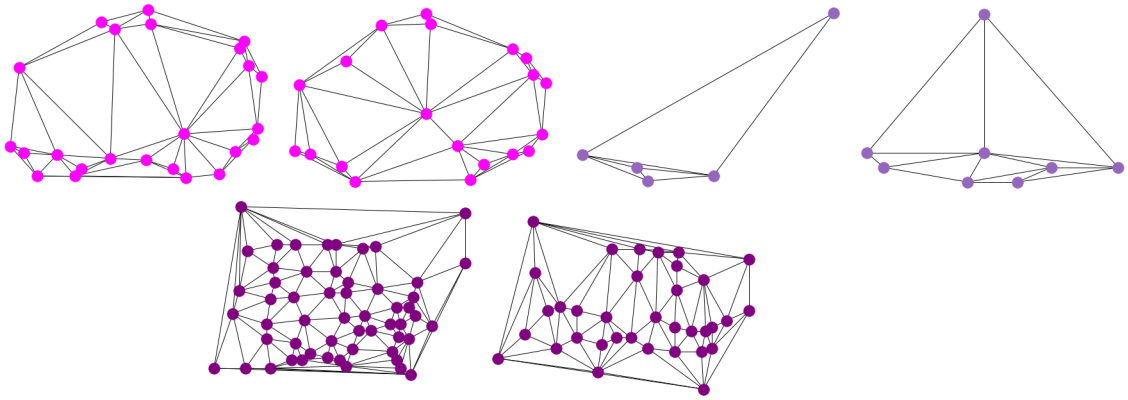


Figure 20: Graphs from the IAM database [13] which have convex polygon borders

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.

5.2 Comparing Disconnected Graphs

Generally, maps are connected, but in situations where we run into disconnected graphs, we only use the largest connected component. This can reduce the accuracy since parts of the data are not calculated. 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.

5.3 Accuracy and Runtime

Our implementation makes some trade-offs between runtime and accuracy. We implement a binary search for branching distance which generally has a margin of error when we rotate our graphs. We designed a method for calculating the exact branching distance. The core idea is to choose a smart margin of error based on the domain of the inputs' function value. By doing so, one can perform a binary search until only one possible distance is in within the margin of error. However, we found that this method is not generally applicable. A smart margin of error can only be chosen when function values of the inputs are all terminating decimals. This is almost never the case when rotating graphs.

We also choose to simultaneously rotate the graphs rather than attempt to compare each orientation of one to each orientation of the other. A consideration of all 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, it is too computationally expensive for most data.

Furthermore, we choose 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. We know that the rooted branchings change whenever there is a change in the structure of the merge tree. This will occur whenever some minimum would take on the same function value as its parent. However, it is difficult to determine when the cheapest rooted tree representation pair will change due to a rotation. Due to this, implementing a mathematically accurate comparison of all angles of the given graphs would be computationally prohibitive and beyond the scope of our research.

6 Acknowledgements

The funding for this project was supported by the National Science Foundation (NSF Award No. 1852066), the National Security Agency (NSA Grant No. H98230-20-1-0006), and Michigan State University.

The work of Erin Chambers was supported in part by NSF grants CCF-1614562, CCF-1907612, and DBI-1759807. The work of Elizabeth Munch was supported in part by NSF grants CCF-1907591 and DEB-1904267.

We used the tools made available by [14, 16, 9] to draw and export graph data that could be read into Python as NetworkX Graph objects. Some of these graphs apeared in this paper. To examine the functionality of our code, we imported graph data from several sources. In spirit of our original motivation, many of the graphs imported were in the form of map data. Sources for these geospatial data include OpenStreetMap.org, the Stanford EarthWorks Library, and the Map Construction[1] project. These data do not appear in this paper but there are present in our GitHub repository.

References

[1] M. Ahmed, S. Karagiorgou, D. Pfoser, and C. Wenk. A comparison and evaluation of map construction algorithms., 2015.

[2] Levent Batakci, Abigail Branson, Bryan Castillo, Erin Chambers, Liz Munch, and Candace Todd. Embedded graphs. `https://github.com/lizliz/SURIEM2020-EmbeddedGraphs`, 2020.

[3] Kenes Beketayev, Damir Yeliussizov, Dmitriy Morozov, Gunther Weber, and Bernd Hamann. *Measuring the Distance Between Merge Trees*, pages 151–165. Springer, 01 2014.

[4] A. Bondy and U.S.R. Murty. *Graph Theory: Graduate Texts in Mathematics*. Springer, 2008.

[5] AURA Conci and CS Kubrusly. Distance between sets-a survey. *arXiv preprint arXiv:1808.02574*, 2018.

[6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[7] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[8] Kristian Kersting, Nils M. Kriege, Christopher Morris, Petra Mutzel, and Marion Neumann. Benchmark data sets for graph kernels, 2020.

[9] Diego Alejandro Macrini. *Indexing and matching for view-based 3-d object recognition using shock graphs*. Citeseer, 2003.

[10] Travis E Oliphant. *A guide to NumPy*, volume 1. Trelgol Publishing USA, 2006.

[11] 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 E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[12] Frank Ratchye. Skeleton tracing. `https://skeleton-tracing.netlify.app`.

[13] K. Riesen and H. Bunke. IAM graph database repository for graph based pattern recognition and machine learning. accepted for publication in SSPR 2008.

[14] Unknown. Cs academy. `https://csacademy.com/app/graph_editor`.

[15] Unknown. dcode. `https://www.dcode.fr/binary-image`.

[16] Unknown. Graphonline. `https://graphonline.ru/en/`.

[17] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, CJ Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake Vand erPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1. 0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.