EUROGRAPHICS 2014 / B. Lévy and J. Kautz
(Guest Editors)

Volume 33 (2014), Number 2

Finding Feature Similarities Between Geometric Trees

Submission id: 1055

Abstract

We define geometric trees as graphs with no cycles in which nodes have spatial co-ordinates and edges are geometric curves. Temporal morphing of geometric trees through changing positions and geometry of the nodes and edges are common in many applications, physical and simulated data. Examples include geological structures such as changing patterns of river beds and biological structures such as respiratory tracks and deforming patterns of neurons. Given two snapshots of such trees at different times, or two trees representing similar data sets, or trees generated by two different methods, comparison between these trees will provide a wealth of information for interpreting the data or the methods that produced that data. We propose an algorithm to compare geometric trees by detecting feature similarities between the trees, wherein the features are geometric as well as topological.

1. Introduction

Geometric trees as defined as graphs having no cycles in which the nodes have spatial co-ordinates and the edges are geometric curves. Special cases of geometric trees are well known embedded trees in a metric space in which the edges do not self-intersect. Examples of geometric trees include visualization of river's tributaries and distributaries, animal neurons, and embedding and visualization of software (non-recursive) function call/usage data.

In many applications, a geometric tree may structurally morph by altering the positional co-ordinates of the nodes, the geometry of the edge curves, and by adding or deleting a few nodes and edges. For example, a neuron can be represented as a geometric tree which changes structurally under the influence of a chemical agent (Figure 1), the course of a river can change over time, the respiratory tracks of mammals can grow with age or the execution path of a software can change for different data sets (Figure 14). Geometric tree morphings are also quite common in animation applications. Given two snapshots of such morphing geometric trees, an appropriate matching between them provides valuable insight about the process of metamorphosis or the attributes of the external stimuli causing it.

A matching between two geometric trees entails determining similarities between different parts or subtrees of the two trees. In other words, the output of the geometric tree matching algorithm would be a list of matched node pairs, one from each tree, and ensuring that each node has at most one matched node in the other tree.

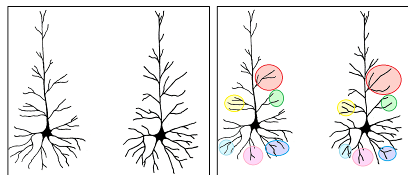

Figure 1: *Finding similarities between two geometric trees representing a regular animal neuron and a post ischemic neuron. Some of the relevant matches in the trees identified by our algorithm are shown, color coded with similar colors.*

1.1. Main Contributions

The challenges in finding similarities in geometric trees mainly arise due to allowed and possible differences from one tree to the other. Many graph similarity algorithms assume that the graphs have same number of nodes, or assume that one graph is a subgraph of the other. We present a tree matching algorithm that allows for addition and deletion of nodes and edges, and vast changes in both topology and geometry. Further, in spite of such large differences between the trees, our algorithm can take into account node adjacency and edge coherency in the matching i.e. two adjacent nodes in one tree is highly likely to be matched with two adjacent nodes in the other.

In order to ensure node adjacency and edge coherency, one has to consider exponential number of sets of edge disjoint paths in each tree for matching. Most of the earlier works reduce the solution space by limiting the problem to labeled, rooted, topological trees maintaining strict ancestor

descendant relationship which implicitly defines the sets of edge disjoint paths. In contrast, we consider completely arbitrary geometric trees that may be rooted or unrooted, labeled or unlabeled, topological or geometric. Instead of finding or restrictively pre-defining the sets of edge disjoint paths, we build these paths using salient geometric and topological features of the trees and thus reduce the solution space.

In summary, we present a novel, one of the most generic algorithm to find a matching between two unlabeled, geometric and/or topological, rooted or unrooted trees. The trees need not have equal number of nodes or edges and can have arbitrary geometry in the plane. Our algorithm is capable of handling even approximate matches between trees. The matches are invariant to affine transformation of the trees. The average run time of our algorithm is $O(n^3 log(n))$, where n is the number of leaf nodes in each tree.

The rest of the paper is organized as follows. Following the discussion on related work in Section 2, we formally introduce the problem of tree matching in Section 3, followed by Section 4 which describes in detail the process of geometric tree matching. We present the results of our matching algorithm in Section 5 and finally conclude in Section 6.

2. Related Work

Tree structures have been well studied in diverse fields of computer science ranging from computer vision, natural language processing, computational biology, web applications, compiler optimization and many others. Most of these applications consider *labeled* trees, wherein, the nodes of the tree are assigned symbols taken from a fixed finite alphabet. Some applications consider *ordered* trees where the sibling nodes are assigned a specific left-right order. The general structure for finding matches in labeled trees is to first define a set of basic operations on the trees such as relabeling nodes or inserting/deleting nodes. Each such operation is associated with a cost, based on which a tree edit distance is calculated which is a sequence of operations with a minimum total cost. The edit distance notion for ordered trees was introduced by Tai [Tai79], which was further modified by Kosaraju [Kos89], Zhang and Shasha [ZS89], Klein [Kle98] and recently by Chen [Che01] to imrove the complexity. A survey of tree edit distane and related problems can be found in [Bil05]. The subtree isomorphism problem for ordered trees have been extensively studied by Makinen [M89̈].

Tree matching is a well studied topic in the fields of computer vision and pattern recognition. Liu et.al. [LG99] provides a framework for 2D shape comparison by representing shapes with their skeletons and matching such skeletons for different shapes under rigid transformations and occlusions. The basic operations for matching are merging and cutting branches of a tree and are limited in applications. Tree matching has been used by Cantoni et.al. [CCG*98] to match two planar objects by decomposing them into trees having nodes representing several levels of resolution and then finding a one to one mapping between paths in both the trees. A video comparison framework was proposed by Wing Ng et.al. [NKL01], wherein each video sequence is hierarchically decomposed into scenes and shots and two such sequences are matched at each hierarchical level.

Tree matching has applications in natural language processing in searching and retrieving complex feature structures from lexical databases. Kilpelainen et.al. [KM92] and Oflazer [Ofl96] apply this concept to search for trees which are ‘close’ to a given query tree in a database of labeled trees, where closeness is defined in terms of an error metric. Wang et.al. [WMC09] employs a syntactic tree matching approach to identify ‘similar’ questions in a question-answer (QA) database. Syntactic tree matching is also used to identify differences between two programs of the same programming language, as shown by Yang [Yan91]. Pattern matching, as employed to programming languages have been extensively studied by Hoffmann et. al. [HO82] and Ramesh [RR92].

Tree matching finds its place in web applications and design as well. The most common application is comparing similar web pages represented as labeled document object tree. In this context Kumar et.al. [KTA*11] introduce the concept of flexible tree matching and prove that such matching is NP complete and provide approximation algorithms to solve for the same. Jindal et.al. [JL10] uses tree matching for web data extraction from webpages having repeated embedded patterns.

Computational biology is another field where tree matching finds wide applications. Aoki et.al. [AYO*03] use tree matching for finding and aligning maximally matching subtrees in two glycan (carbohydrate sugar chains) trees. Luccio [LP91] considers h-ary trees commonly found in biochemical structures such as glycogen, and computes the occurrence of such trees in larger trees. Jiang et.al. [JWZ95] provides an efficient tree alignment algorithm for comparing ordered trees as observed in RNA secondary structures.

The fundamental difference between all of the above mentioned methods and our method is that, our method considers geometric trees which are not labeled. Since geometry is involved, the tree nodes and edges can have arbitrary configuration and the tree can be rooted as well. We do not assume any specific kind of tree, and this makes our algorithm more general in nature and not confined to topology. The work of Pisupati, et.al. [PWMZ96] is somewhat similar to ours as they consider geometric tree matching as applied to 3D lung structures. However, they consider only rooted binary trees with one-to-one mapping between nodes, as is found in their application, and have developed an algorithm to determine whether two such trees are isomorphic under the operation of graph minors. In contrast, we consider arbitrary trees which may not have a one-to-one matching and assume that the matching parts may exist anywhere in both the trees, with no strict ancestor-descendant relationship.

3. Problem Definition

A geometric tree is a graph without any cycles where each *node* is associated with co-ordinate values and each *edge*, joining two nodes, is a geometric curve. The number of edges incident on a node is defined as the degree of the node. The nodes with degree one are defined as *leaf nodes*, those with degree two as *path nodes*, and nodes with degree more than two are defined as *internal nodes* (Figure 2). Our geometric tree may allow intersections of tree edges in plane but such intersections are not considered as nodes in the graph and hence this graph is still a tree.

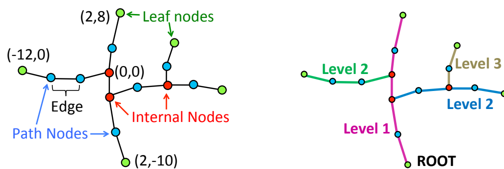

Figure 2: *Left: A geometric tree with co-ordinates of some of the nodes. Right: A branch decomposition is the segmentation of the tree into edge disjoint paths. The root node and the branches with their hierarchical levels are highlighted.*

Matching two geometric trees is equivalent to finding matches between their node sets. However, a naïve matching between nodes does not take into account spatial coherency, potentially generating matches in drastically different tree locations. A match between the different paths of the two trees avoids this problem but fails to produce a one-to-one mapping between tree nodes and edges due to possible path overlaps. In order to avoid such overlaps, it is necessary to decompose a tree into a set of edge-disjoint paths, unless such a decomposition is already provided, and then find matches between such decompositions of two trees. This process is defined as a *branch decomposition* of the tree (Figure 2, Section 4.1), and an element or *branch* in this decomposition is defined as the path from a leaf node to an internal node or another leaf node. In this process, if the tree is unrooted, we compute one geometrically dominant branch between two leaf nodes one of which is considered as the root. If the input is a rooted tree, we find branches that start from an internal or root node and end in a leaf node such that they are pairwise edge-disjoint and the union of these branches is the given tree. Each branch thus contains exactly one leaf node and one internal or root node.

Given the branch decompositions of the two trees, we match these trees by finding a mapping between their decompositions such that a branch in one tree has either zero or one matched branch in the other tree. Using branch decomposition, which by definition is edge-disjoint, tree matching ensures that no edge has more than one match in the other tree. Since we consider trees that may not have the same number of nodes, edges, or branches, there may be unmatched branches in either or both the trees.

Consider two sets, A and B, of mutually disjoint branches in the two trees. Each pair of branches (a_i, b_i), where $a_i \in A$ and $b_i \in B$ has a branch matching cost which takes into account their geometric and topological features. Each branch in either set also has a cost for not matching it with any branch. Let Φ be the set of 'dummy' branches such that the cost of matching any branch p in either set to a branch in Φ is the cost of not matching p to any branch in the other tree. We define the problem of similarity matching as one in which every branch is matched to no more than one branch in the other tree, and the total cost of this matching is minimum. In other words, our problem is to find the set of matching branches S, $S \subseteq (A \cup \Phi) \times (B \cup \Phi)$ such that *if* $(a_i, b_i) \in S$, $(a_j, b_j) \in S$, $a_i, a_j \in A$, $b_i, b_j \in B$, *then* $a_i \neq a_j$ and $b_i \neq b_j$; and the sum of the cost of the matches in S is minimum.

4. Geometric Tree Matching

The source and target trees are decomposed into a set of edge disjoint branches prior to matching. The branches are then compared to obtain an optimum set of matched branches.

4.1. Branch Decomposition of a Tree

Let Ω be the set of all possible paths of the tree, wherein a path originates in an internal node or a leaf node and terminates in a leaf node. Each path in Ω is characterized by a set of features based on its geometric properties. In a vector space of features, which we call the feature space, each path is represented by a feature vector given by $u = (u_1, u_2, ..., u_n)$. A scalar function $f : R^n \to R$ from the feature space is defined which assigns a scalar value, called a feature value to each path. We model f as a linear combination of the features.

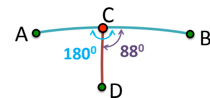

Figure 3: *The feature value of the path ACB is more than that of path ACD even if the lengths are same, as the former has a turning angle of* 180° *at C, making ACB more preferred as a branch over ACD.*

The geometric features that comprise the feature vector of a path are its length, algebraic and absolute sums of the turning angles at each intermediate node of the path (assuming the path to be piece wise linear), and the number of self-intersections along the path (Figure 4). The function f is evaluated and a feature value obtained for every path in Ω. The definition of f for this purpose has higher contribution from the turning angle feature, e.g., for an internal node of degree three, there are two potential paths through the node, out of which the one making an angle closer to 180° has a higher feature value contribution at the internal node implying a possible path continuity (Figure 3). All other features

have equal contribution in evaluating f. The path p in Ω with the highest feature value is selected as the geometrically dominant branch b, and the process ensures that this branch has two leaf nodes, one of which is considered a root node in case of an unrooted tree, and all the paths having edge overlaps with this branch are not considered further. Next, for each internal node i of b, the path originating from i with maximum feature value is selected as a second level branch with b as its parent. The process continues for each of the second level branches to obtain a set of third level branches, and so on (Figure 2). The selection of branches in this manner ensures that there is no edge-overlap between two branches. Note that in case of a rooted tree, there can be multiple disjoint branches originating from the root, each of which is a first level branch. The above process gives a branch decomposition of the tree.

The branch decomposition of a tree allows us to augment the feature vector of a branch with topological features like ancestor-descendant relationship between branches and the hierarchical position of a branch in the tree from a specified root (Figure 4). This *enhanced feature vector* of the branch is later used to calculate its matching cost with another branch. For the purpose of calculating the matching cost between branches we also add the geometric feature of histogram of turning angles at internal branch nodes to the feature vector.

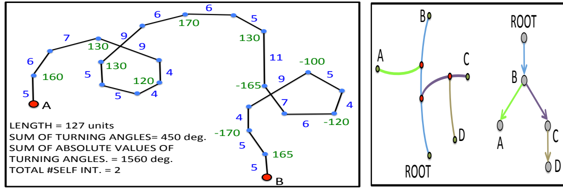

Figure 4: *Left: The geometric properties in the feature vector of a branch. Right: The hierarchical decomposition of a rooted tree into different levels. branch B is in level one and has two children, A and C, each in level two. Branch D is a child of C and is in level 3.*

4.2. Cost of Matching Individual Branches

The cost, C_{ij} of matching two individual branches i and j, in the source and target trees respectively can be obtained by computing the weighted distance between their respective enhanced feature vectors in the feature space (Equation 1):

$$C_{ij} = \sum_{k=1}^{n} w_k * d_k(u_k, v_k) \tag{1}$$

where u_k and v_k are the kth feature in the feature vectors, $u = (u_1, u_2, u_3, ..., u_n)$ and $v = (v_1, v_2, v_3, ..., v_n)$ of the branches i and j respectively, d_k and w_k are the distance function evaluating the feature similarity and the weight associated with the kth feature respectively. The distance function we consider is the chi-squared metric for comparing the histograms of turning angles (Equation 2 where u_k and v_k are the histograms of turning angles of branches in the source and target trees, and $u_k(i)$ and $v_k(i)$ are the values in the ith bin of u_k and v_k respectively) and the Manhattan distance (Equation 3) for comparing all other features. Note that any other distance function can be used, e.g. Bhattacharya distance [Bha43] or Kullback-Liebler divergence [KL51] for comparing histogram of turning angles.

$$d(u_k, v_k) = \sum_i \frac{(u_k(i) - v_k(i))^2}{u_k(i)} \tag{2}$$

$$d(u_k, v_k) = |u_k - v_k| \tag{3}$$

Figure 5 shows the five closest matches for a source tree branch with target tree branches, where the matches are evaluated in terms of the cost function described above.

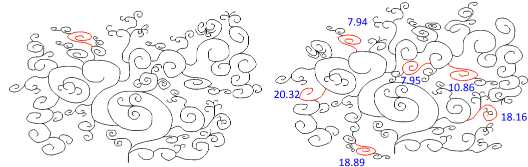

Figure 5: *The match costs of a source tree (left) branch highlighted in red with every target tree (right) branch are evaluated and the five closest matches are highlighted in red. The costs are shown in blue alongside.*

4.3. Cost of not matching a tree branch

Since we consider trees which may not have equal number of nodes, not all branches may have a match. Even if there are equal number of elements, a few matches may not be appropriate when considering node adjacency and edge coherency, and may be better to leave such branches unmatched. The cost $C_{i\phi}$, of not matching a branch i with any branch is calculated as a weighted combination of its features (Equation 4).

$$C_{i\phi} = \sum_k w_k * d_k(u_k, 0) \tag{4}$$

where u_k is the kth feature in the feature vector of branch i and w_k and d_k have the same meaning as before.

With this framework, the overall tree matching cost can be computed as follows. Let A and B be the sets of branches in the source and target trees respectively. Let $S \subseteq (A \cup \Phi) \times (B \cup \Phi)$ be a set of matched branches where Φ is the set of dummy branches, such that *if* $(a_i, b_i) \in S$, $(a_j, b_j) \in S$, $a_i, a_j \in A$, $b_i, b_j \in B$, *then* $a_i \neq a_j$ and $b_i \neq b_j$. The cost, C_S of such a set S is defined as the sum of the costs of

the matched branches and the costs of not matching the unmatched ones. Let Ψ be the collection of all such sets S. The similarity match, $S_m \in \Psi$ is given by Equation 5.

$$S_m = \arg\min_{C_S}(S \in \Psi) \tag{5}$$

4.4. Tree Matching as Minimum Weight Perfect Matching Problem

The tree matching problem to calculate the matching cost as described above, can be modeled as a bipartite graph matching problem, where G is a graph, with two disjoint node sets, P and Q, corresponding to the branch sets A and B of the source and target trees respectively. Null sets of branches, which we call dummy nodes are added to both P and Q to make them equal in size. An edge E_{ij} in G connects a node $i \in P$ to a node $j \in Q$. Each edge E_{ij} is associated with a cost C_{ij} which is the branch matching cost if i and j are regular branches, is the cost of not matching a branch if either i or j is a dummy node, and is equal to zero if both i and j are dummy nodes. With this framework, tree matching is equivalent to finding a perfect matching of G with a minimum sum of edge costs (Figure 6). This is known as the 'Minimum weight perfect matching problem', formally stated below:

Minimum weight perfect matching problem: In a complete bipartite graph, $G(P \cup Q, E)$, with node sets P and Q, $|P| = |Q|$, weight C_{ij} for all the edges E_{ij} connecting nodes $i \in P$ and $j \in Q$, a minimum weight perfect matching M is a 1-factor (spanning subgraph where every vertex is a degree 1 vertex) of G where the sum of the weights of the edges in M is minimum.

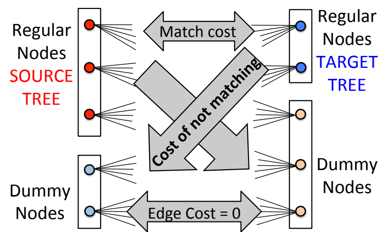

Figure 6: *Modeling the tree matching as a minimum weight perfect matching problem. An edge between regular nodes is the match cost between them, whereas that between a regular and a dummy node represents the cost of not matching the branch. Dummy nodes have zero edge cost between them.*

We use Kuhn-Munkres algorithm [Kuh55, Mun57], also known as the Hungarian algorithm which has a running time of $O(n^3)$, where n is the number of nodes in each set, to solve the minimum weight perfect matching problem.

Computing minimum weight perfect matching with all branches of all hierarchy of both the trees as the node set, will produce completely arbitrary results with matches scattered throughout the trees as shown in Figure 7. Restricting the matches to the same hierarchical level produces an intuitive matching in this case. However, this restriction is not sufficient, as Figure 8 shows. Even if the matched branches in the same level are geometrically very similar, they may have completely different subtrees. A better match can be obtained by not limiting the matching to individual branches, but also by considering and matching the subtrees originating from the two branches, thereby taking into account the spatial coherency.

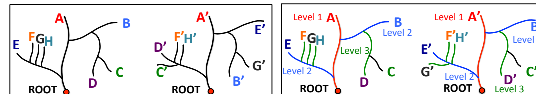

Figure 7: *Matching (shown by same alphabets) all the branches of the two trees (left) does not produce the intuitive match obtained by restricting the matches to the same hierarchical levels (right).*

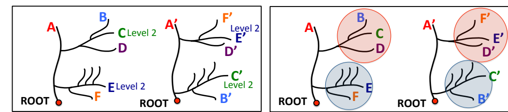

Figure 8: *Restricting branch matches to the same level does not always produce intuitive match. Although the matched branches, C-C' and E-E' are geometrically similar, they have structurally different subtrees. A more intuitive match obtained with the modified branch match cost is shown on right with red and blue circles.*

In order to identify such deeper matchings, the cost basis for matching individual branches have to be modified to include their descendants as well. The match cost between two branches a and b is thus computed recursively in terms of their children branches as follows (Algorithm 1). A minimum weight perfect matching is computed for the graph with node sets representing the children branches of a and b. Let the cost of this matching be C_1. Let C_2 be the cost of matching a and b individually (as obtained in Section 4.2). Then the cost of matching a and b is given by $C_1 + C_2$. This modification of the branch cost gives an intuitive matching for the trees in Figure 8.

The restrictive tree matching with the modified branch cost is also not sufficient for producing intuitive matches as Figure 9 shows. The exposure of the branches level by level fails to match near identical sub trees at different levels. A better approach is to expose the branches that can be potentially matched, in clusters, over multiple iterations. In each iteration, only a few branches are exposed and thus the options for finding a match for a branch within the exposed set is limited, and if no acceptable matching branch is available

Algorithm 1 Calculate Matching Cost (a, b)

1: **Input:** a in source tree T_1 and b in target tree T_2
2: **Output:** The cost C of matching a and b
3: **if** a = NULL and b = NULL **then**
4:     **return** 0
5: **end if**
6: $S_1 \leftarrow$ all children of a, $S_2 \leftarrow$ all children of b
7: $\forall i \in S_1, j \in S_2, C(i,j)$=Calculate Matching Cost (i,j)
8: $C_1 \leftarrow$the cost of minimum weight perfect matching of a graph with node sets S_1 and S_2 and edge costs $C(i,j)$
9: $C_2 \leftarrow$ individual cost of matching a and b
10: **return** $C_1 + C_2$

from that set, then in the next iteration of perfect matching, more potentially matchable branches are exposed to increase the chances of a good match. We call this approach *sliding window matching* which is described in detail in the next section. Sliding window matching produces a desirable match for the trees shown in Figure 9.

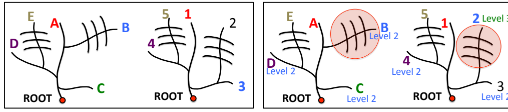

Figure 9: *Left: Restrictive matching with modified branch cost produces a matching of B-3,D-4 and E-5. The subtrees at B and 2 are identical and should be matched ideally. Right: Sliding window matching exposes branches B,C and D from source tree and branches 3 and 4 from target tree, all in level two. A perfect match between these two sets produces B-3 and D-4 (and E-5), out of which D-4 has a low match cost and considered a good match, while B-3 is not. In the next step, branch 2 from target tree is exposed and a new perfect match is computed between sets* $\{B,C\}$ *and* $\{2,3\}$ *from which a desirable B-2 match with low match cost is obtained. Note that a match between all branches with modified cost could have produced a B-5 match, which is not desirable.*

4.5. Sliding Window Matching

The sliding window tree matching algorithm is presented in Algorithm 2). Let σ_1 and π_1 denote the set of first level branches in the source and target trees respectively. Let S be the set of matched branches, which is the output of the algorithm. A minimum weight perfect matching, P is computed with the bipartite graph with node sets as σ_1 and π_1. The weight of the edge between the nodes is the match cost that is computed by Algorithm 1. Among the chosen edges in the perfect matching, if an edge cost, say between a_1 and b_1, is less than a specified threshold η, then (a_1, b_1) is added to the solution set S of matched branches. Further, since, according to Algorithm 1, the matching cost between a_1 and b_1 includes the perfect matching cost between the descendent branches of a_1 and b_1, the matched descendent branches that are responsible for this minimum match cost between a_1 and b_1 are also included in the solution set (Step 10, Algorithm 2). Adding matches between the subtrees into the final set also improves spatial and topological coherency of matches through the levels of hierarchy. Any unmatched descendent branches are no longer considered in the subsequent matching process and are left unmatched, since their 'non-matching' cost is also included and considered while matching a_1 and b_1 in the recursive Algorithm 1. Alternatively, since the cost of an unmatched branch in the subtree is usually higher than the cost of matching, such unmatched branches in the subtree can be used in the subsequent iterations of the algorithm to find a possible match. In such a case, which is dependent on the application requirement, Algorithm 2 is modified appropriately to include unmatched branches in subsequent iterations.

Let $\sigma_1\prime$ and $\pi_1\prime$ be the set of branches which have remained unmatched in the source and target trees respectively. Also let σ_2 and π_2 be the set of second level branches which are the children of those in $\sigma_1\prime$ and $\pi_1\prime$. A new instance of bipartite graph matching is created using $\sigma_1\prime \cup \sigma_2$ and $\pi_1\prime \cup \pi_2$ as the node sets which is solved to obtain a new set of matches in a similar manner as described before. The process is continued by repeatedly adding new branches from lower hierarchical levels to the unmatched ones from the previous level in each successive iteration until both the sets remain unchanged from the previous iteration. In each iteration the sets σ and π denote the sliding window to which elements are added and removed between iterations.

Algorithm 2 Tree matching algorithm

1: **Input:** The sets of edge disjoint branches A from source Tree, T_1 and B from target tree, T_2
2: **Output:** A match, S, of the form (a_i, b_i), where $a_i \in A, b_i \in B$, and if $(a_i,b_i),(a_j,b_j) \in S$, $a_i \neq a_j, b_i \neq b_j$
3: $\sigma\prime \leftarrow$ first level branches of T_1
4: $\pi\prime \leftarrow$ first level branches of T_2
5: **repeat**
6:     $P \leftarrow$ a minimum weight perfect matching of $\sigma\prime$, $\pi\prime$
7:     **for each** $(a,b) \in P$ **do**
8:         **if** Cost of edge $(a,b) < \eta$, **then**
9:             $Q \leftarrow$ perfect matching of descendants of a,b
10:             $S = S \cup \{(a,b)\} \cup Q$
11:             $\sigma\prime \leftarrow (\sigma\prime - a)$, $\pi\prime \leftarrow (\pi\prime - b)$
12:         **end if**
13:     **end for**
14:     $\sigma \leftarrow$ children branches of $\sigma\prime$
15:     $\pi \leftarrow$ children branches of $\pi\prime$
16:     $\sigma\prime = \sigma \cup \sigma\prime$, $\pi\prime = \pi \cup \pi\prime$
17: **until** $\sigma = \sigma\prime$ and $\pi = \pi\prime$
18: $S = S \cup P$

4.6. Analysis of The Sliding Window Matching

In our matching algorithm, if two branches are matched at any iteration, their descendants are also subsequently matched. Suppose at any iteration the branches to be matched are $\sigma_i = \{a_1, a_{11}\}$, where a_{11} is a child branch of a_1, and $\pi_i = \{b_1, b_{11}\}$, where b_{11} is a child branch of b_1. Now suppose the matching algorithm matches a_1 to b_{11} and a_{11} to b_1. Although a_{11} is matched to b_1, it may be possible that due to recursive matching of a_1 and b_{11}, as a child of a_1, a_{11} is matched to a child of b_{11}, at the same time. It is highly unlikely due to the cost structure of the edges that a_{11} has low cost both with b_1 and b_1's grandchild, unless the given tree is an infinite recursive tree with self-similarity. If such matches happen, the match with the lower cost is accepted.

Memoization: The match cost between two branches is calculated recursively, which allows us to memoize the solution, e.g. if two branches a_1 and b_1 and all of their descendants remain unmatched in a particular iteration, the match cost between them remain unchanged, and can be used in the next iteration. If, on the other hand, any of the descendants, say a_{11} of a_1 is matched to some other branch, then a_{11} is not considered in the subsequent iterations. This changes the subtree originating from a_1, thereby calling for a match cost recalculation between a_1 and any branch in the other tree.

The sliding window paradigm introduces branches one level at a time, in every iteration. Alternatively, branches from multiple levels can be introduced in the matching at each stage which may open up the possibility of different matches across hierarchical levels at the expense of solving a matching problem with higher number of nodes in each set of the bipartite graph. However, this may lead to less intuitive matches as is explained in Figure 10.

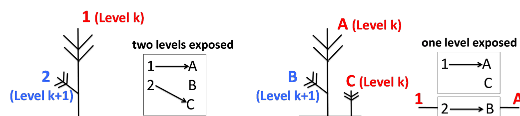

Figure 10: *When two levels are exposed for matching, branch 2 in source tree is matched to branch C in target tree. A more intuitive match is obtained by exposing one branch at a time that matches branches 1 and A and their respective descendants 2 and B.*

Our matching algorithm exposes branches from the first level onwards. It is also possible to expose branches starting from the lowest level and gradually add higher level ones. However, this has the risk of generating far less intuitive mapping as the spatial coherency is often lost and subtrees from drastically different tree locations can be potential matches (Figure 11).

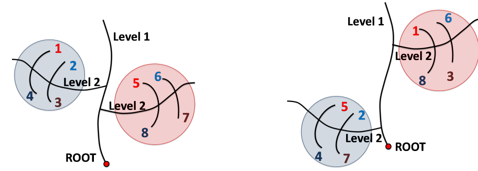

Figure 11: *A potential matching produced between two trees when the branches are exposed from the leaf are shown with similar numbering. A much more intuitive matching shown by similarly colored circles is obtained if branches are exposed from higher to lower hierarchical levels.*

4.7. Choice of Weight Vector and Threshold

The choice of the weight vector and threshold value used for matching is often dictated by the application under consideration (Figure 12). We provide an automatic way of computing the weight vectors by comparing the standard deviation of the features in either tree. Each feature is normalized to lie within the same range and the standard deviation of the feature is calculated for each tree. Let the minimum of the two standard deviations (one from each tree) be ζ_i for the ith feature. The proprotion of the weight vectors used is same as the proportion of the ζ's, rounded to the nearest integer, e.g. if there are two features f_1 and f_2 with corresponding proportions of minimum standard deviations $\zeta_1 : \zeta_2 = 1 : 2$, then the respective weight vectors, w_1 and w_2 are also in the proportion $1 : 2$. This ensures that a feature having a greater variation is a greater deciding factor for matching. The weight for the histogram of turning angles feature is taken to be the same as the weight of the sum of turning angles. The weight vectors for topological features is typically chosen as the average of the weights of the geometric features. Our system is also flexible in allowing the user to provide weights of his or her choice prior to matching.

Typically the threshold chosen at the start of the algorithm is gradually reduced over subsequent iterations thereby ensuring a lower threshold for matching smaller subtrees. We choose a starting threshold of 40% of the highest matching cost between branches of the two trees, although the user can manually set this value.

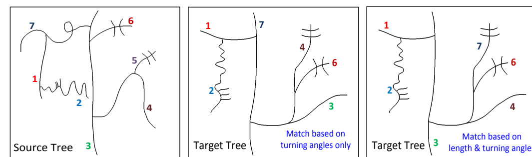

Figure 12: *Different matches obtained between source and target tree with varying weight vectors. Center: Feature vector consists of turning angles only. Right: Feature vector consists of equally weighted length and turning angles.*

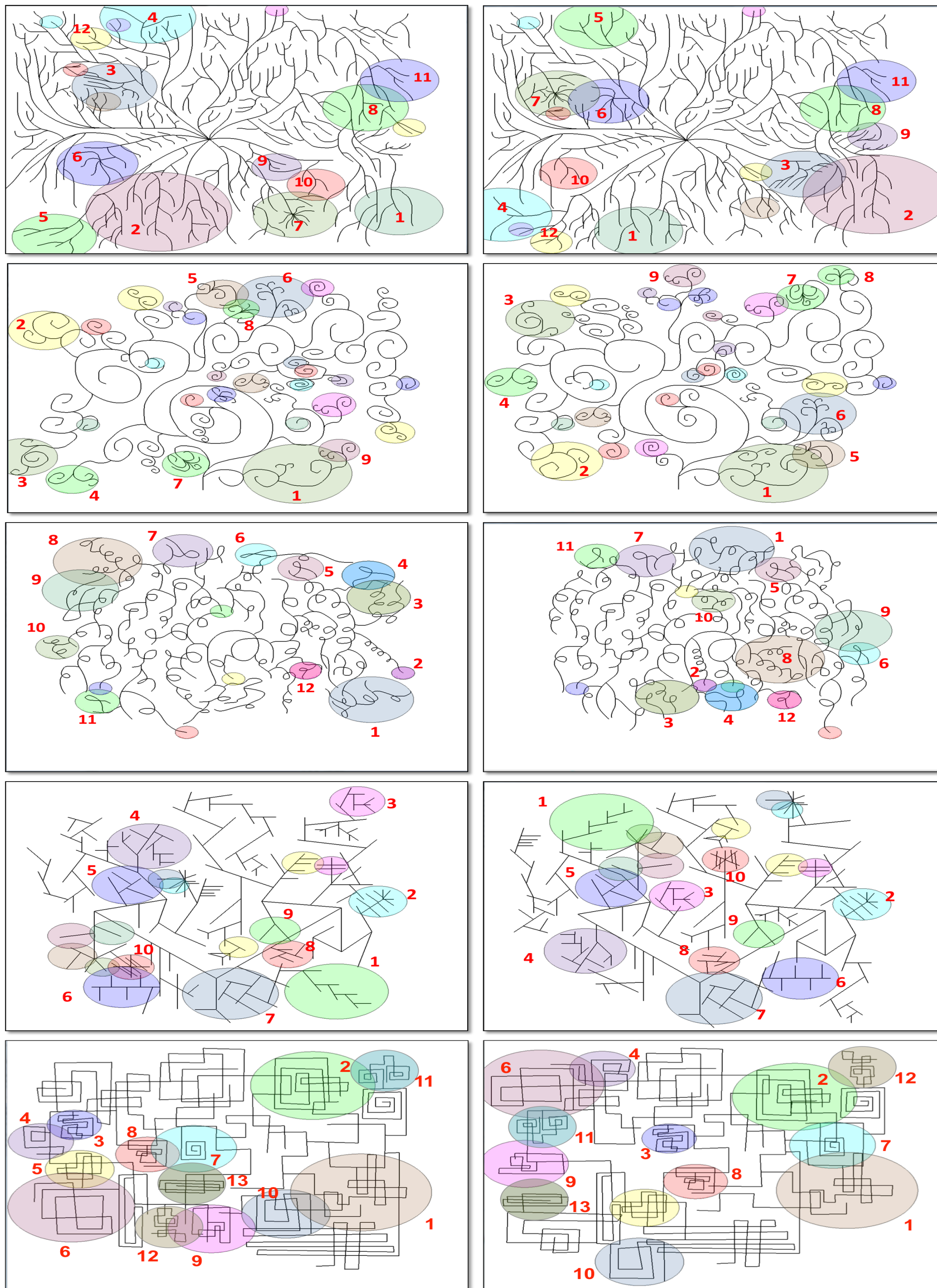

Figure 13: *Examples of trees matched by our algorithm. Each row shows a set of source and target trees. The matched parts in each set is highlighted with the same color and also numbered with same numbers. Note that only a few relevant matches are shown and a lot of trivial matches occupying the same relative position on either tree have not been highlighted for aesthetic reasons, although such matches are detected by our algorithm.*

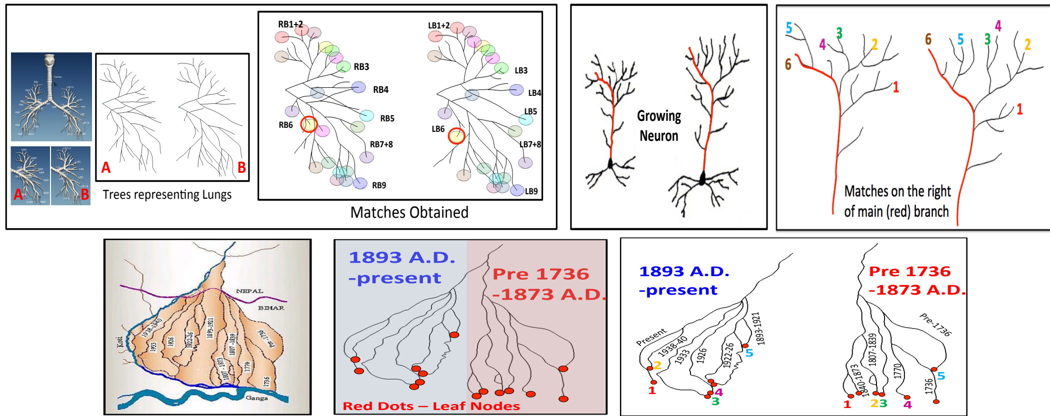

Figure 14: *Top Row Left: Matching both sides of a human respiratory tree (picture: 3D Bronch ipad/iphone app). The matches obtained between the right and left (shown reflected) lungs brings out the symmetry between them. A slight mismatch is shown by the red circle. Top Row Right: Aging process of cortical pyramidal neurons ([PM01]). A fairly good match between the first level neuron branches illustrates the growing process. Bottom Row: Matching the shifting pattern of Kosi river over two time spans. Matches obtained reveal the similar nature of river flow over these time spans, which is of high geographical importance.*

5. Results

Figure 13 shows the results of our matching algorithm. We have shown five characteristically different types of geometric trees to be matched. Note that only a few relevant matches found by our algorithm are highlighted for aesthetic reasons, although our algorithm detects all of them. For each row in Figure 13, the weights of the features are calculated automatically and considering the features in the following order: length of branches, sum of turning angles, absolute value of turning angles, number of self-intersections and histogram of turning angles, the weight proportions from top to bottom rows are 13:26:22:0:26, 21:22:22:0:22, 20:16:17:20:16, 83:0:0:0:0 and 11:22:20:14:22 respectively. Our algorithm also produces intuitive matches in characteristically different trees as shown in Figure 15. The matching is predominantly in terms of number of intersections and the weight vectors are manually set to 5:1:1:5:1.

Our algorithm works well when applied to geometric trees obtained from real world physical systems. Figure 14 shows the results of our algorithm when applied to a human respiratory tree, changing river bed pattern and a growing neuron.

The entire system is built in C++ using Visual Studio 2012 in an Intel Xeon CPU (2.00 GHz, 2 GB RAM, 64-bit OS). The run times of our algorithm applied on the examples shown in Figure 13, in order, are given in Table 1. It can be observed that the run time depends upon the depth of the tree as well the number of tree branches.

Table 1: *Run times of our algorithm on Figure 13 trees.*

Depth of Branches	Time (sec)	No. of Branches	No. of Nodes
15	12.35	286	9500
11	4.2	101	10265
12	4.37	76	6475
21	28.58	202	550
7	3.75	28	400

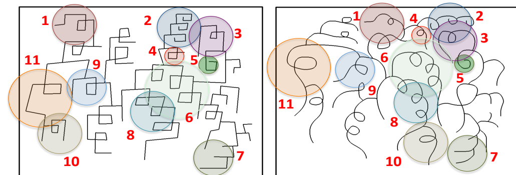

Figure 15: *Matching produced by our algorithm in two completely different geometric trees. The dominant geometric fetures used for matching are length and number of self-intersections of branches.*

6. Summary

We have introduced a variant of the minimum weight perfect matching called the sliding window matching in which the elements to be matched are exposed in clusters over multiple iterations. The sliding window matching is used to design a new algorithm for finding similarities between geometric trees in terms of their geometric as well as topological features. Our algorithm has direct application to actual physical data having tree structure. When the tree structures representing the physical data undergo metamorphosis, our matching algorithm can be used to compare the trees to pro-

vide useful insight about the actual morphology of the physical system.

References

[AYO*03] AOKI K. F., YAMAGUCHI A., OKUNO Y., AKUTSU T., UEDA N., KANEHISA M., MAMITSUKA H.: Efficient tree-matching methods for accurate carbohydrate database queries. *Genome Informatics 14* (2003), 134–143. 2

[Bha43] BHATTACHARYYA A.: On a measure of divergence between two statistical populations defined by their probability distributions. *Bulletin of the Calcutta Mathematical Society 35* (1943), 99–109. 4

[Bil05] BILLE P.: A survey on tree edit distance and related problems. *Theoretical Comp. Science 337*, 1-3 (2005), 217–239. 2

[CCG*98] CANTONI V., CINQUE L., GUERRA C., LEVIALDI S., LOMBARDI L.: 2-d object recognition by multiscale tree matching. *Pattern Recognition 31*, 10 (1998), 1443 – 1454. 2

[Che01] CHEN W.: New algorithm for ordered tree-to-tree correction problem. *Jnl. of Algorithms 40*, 2 (2001), 135–158. 2

[HO82] HOFFMANN C. M., O'DONNELL M. J.: Pattern matching in trees. *Jornal of the ACM 29*, 1 (Jan. 1982), 68–95. 2

[JL10] JINDAL N., LIU B.: A generalized tree matching algorithm considering nested lists for web data extraction. In *Proc. of the SIAM Int. Conf. on Data Mining* (2010), pp. 930–941. 2

[JWZ95] JIANG T., WANG L., ZHANG K.: Alignment of trees - an alternative to tree edit. *Theoretical Computer Science 143*, 1 (1995), 137 – 148. 2

[KL51] KULLBACK S., LEIBLER R. A.: On information and sufficiency. *Ann. Math. Stat. 22* (1951), 79–86. 4

[Kle98] KLEIN P. N.: Computing the edit-distance between unrooted ordered trees. In *Proc. of the 6th Annual European Symp. on Algorithms* (1998), pp. 91–102. 2

[KM92] KILPELÃĎINEN P., MANNILA H.: Grammatical tree matching. In *Combinatorial Pattern Matching*, vol. 644 of *Lecture Notes in Computer Science*. 1992, pp. 162–174. 2

[Kos89] KOSARAJU S.: Efficient tree pattern matching. In *Proc. on Foundations of Computer Science* (1989), pp. 178–183. 2

[KTA*11] KUMAR R., TALTON J. O., AHMAD S., ROUGHGARDEN T., KLEMMER S. R.: Flexible tree matching. In *Proc. Artificial Intelligence* (2011), pp. 2674–2679. 2

[Kuh55] KUHN H. W.: The hungarian method for the assignment problem. *Naval Research Logistics Qtrly 2*, 1-2 (1955), 83–97. 5

[LG99] LIU T. L., GEIGER D.: Approximate tree matching and shape similarity. In *Proc. of IEEE Int. Conference on Computer Vision* (1999), vol. 1, pp. 456–462 vol.1. 2

[LP91] LUCCIO F., PAGLI L.: Simple solutions for approximate tree matching problems. vol. 493 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1991, pp. 193–201. 2

[M89̈] MÄKINEN E.: On the subtree isomorphism problem for ordered trees. *Inf. Proc. Letters 32*, 5 (1989), 271–273. 2

[Mun57] MUNKRES J.: Algorithms for the assignment and transportation problems. *Journal of the Society for Industrial and Applied Mathematics 5*, 1 (1957), pp. 32–38. 5

[NKL01] NG C. W., KING I., LYU M. R.: Video comparison using tree matching algorithm. In *Proc. of The Int. Conf. on Imaging Science, Systems, and Technology* (2001), pp. 184–190. 2

[Ofl96] OFLAZER K.: Error-tolerant tree matching. In *Proc. of 16th Conf. on Comp. Linguistics - Vol 2* (1996), pp. 860–864. 2

[PM01] PROLLA T. A., MATTSON M. P.: Molecular mechanisms of brain aging and neurodegenerative disorders: lessons from dietary restriction. *Trends in Neurosciences 24, Supplement 1*, 0 (2001), 21 – 31. 9

[PWMZ96] PISUPATI C., WOLFF L., MITZNER W., ZERHOUNI E.: Geometric tree matching with applications to 3d lung structures. In *Proc. of the 12th ACM symposium on Computational Geometry* (1996), pp. 419–420. 2

[RR92] RAMESH R., RAMAKRISHNAN I. V.: Nonlinear pattern matching in trees. *Jornal of the ACM 39*, 2 (1992), 295–316. 2

[Tai79] TAI K. C.: The tree-to-tree correction problem. *Journal of the ACM 26*, 3 (July 1979), 422–433. 2

[WMC09] WANG K., MING Z., CHUA T.-S.: A syntactic tree matching approach to finding similar questions in community-based qa services. In *Proc. of the 32nd Int. ACM SIGIR Conf. on Research and Dev. in Inf. Retrieval* (2009), pp. 187–194. 2

[Yan91] YANG W.: Identifying syntactic differences between two programs. *Software Practice and Experience 21*, 7 (June 1991), 739–755. 2

[ZS89] ZHANG K., SHASHA D.: Simple fast algorithms for the editing distance between trees and related problems. *SIAM Journal on Computing 18*, 6 (Dec. 1989), 1245–1262. 2

Appendix A: Complexity Analysis of the Sliding Window Matching Algorithm

A branch decomposition of the source and target trees, which is a pre-processing step, has a time complexity of $O(N^2)$, where N is number of vertices in each tree, since there are N^2 possible paths.

A minimum weight perfect matching P is computed for every iteration of the algorithm. Let n be the average number of branches in each tree, and d be the average number of children of each branch. Since each leaf node is represented in exactly one branch, n is also the average number of leaf nodes in each tree. On an average P is computed the same number of times as the number of hierarchical levels of branches, which is $log(n)$. Since the Hungarian Algorithm has a run time of $O(n^3)$, the overall runtime for perfect match calculation is $O(n^3 log(n))$.

The match cost between a pair of branches is recalculated only when one or more descendants of either branch has been added to the final match list in the latest iteration. Since there can be at most n elements in the final match list, each match cost can be recalculated at most n times. The time required for calculating each match cost using Hungarian algorithm is $O(d^3)$, as each branch has d children on an average. So, the recalculate cost for each pair of branches is $O(nd^3)$, and since there are n^2 possible matches, the overall runtime for this part of the algorithm is $O(n^3d^3)$. Assuming the average degree of each node is constant, this can be approximated to $O(n^3)$.

Thus, the overall runtime of the algorithm is $O(n^3 log(n)) + O(n^3)$, which is $O(n^3 log(n))$. Hence, the overall time complexity including pre-processing is $O(n^3 log(n)) + O(N^2)$. The space complexity is $O(n^2)$, to store the memoization table used in recursion.