Pacific Graphics (2014)
J. Keyser, Y. J. Kim, and P. Wonka (Editors)

Finding Feature Similarities Between Geometric Trees

Uddipan Mukherjee and M. Gopi

University of California, Irvine, USA

Abstract

Geometric trees are graphs with no cycles in which the nodes have spatial co-ordinates and the edges are geometric curves. Many physical systems can be represented effectively using geometric trees, e.g. river beds, animal neurons, respiratory tracks of mammals etc. As these systems undergo structural metamorphosis, temporally or under the effect of some external stimulus, the underlying tree structures also change. Given two snapshots of structurally morphed trees, an algorithm for comparing them based on geometric and topological tree features is presented. Such comparison provides a wealth of information for interpreting the metamorphosis.

1. Introduction

A geometric tree is defined as a graph with no cycles in which the nodes have spatial co-ordinates from the embedded metric space and the edges are geometric curves, e.g. visualization of a river bed pattern, animal neurons, and genetic trees of animals or plants (Figure 6).

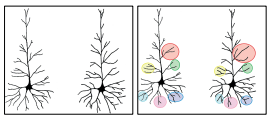

Figure 1: *A regular neuron (left) and a post ischemic one (right) with computed matches color coded with same colors.*

In many applications, a geometric tree may structurally morph by altering the positional node co-ordinates, the geometry of the edges, and by adding or deleting a few nodes and edges, e.g., a neuron can be represented as a geometric tree which changes structurally under the influence of a chemical agent, a river can change course over time or the respiratory tracks of mammals can grow with age. Given snapshots of two morphing geometric trees, an appropriate matching between different subtrees provides valuable insight about the process of metamorphosis or the attributes of the external stimuli causing it (Figure 1).

The challenges in finding matches in geometric trees mainly arise due to allowed and possible differences between trees. Many graph similarity algorithms assume that the graphs have same number of nodes, or one graph is a subgraph of the other. The matching algorithm presented here allows inserting and deleting nodes and edges, and vast changes in both topology and geometry. Further, in spite of such large differences between the trees, the algorithm can take into account node adjacency and edge coherency, 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. 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, completely arbitrary geometric trees are considered here that may have unequal number of nodes and edges, may be rooted or unrooted, labeled or unlabeled, topological or geometric. Instead of restrictively predefining the sets of edge disjoint paths, these paths are built using salient geometric and topological tree features. In summary, the tree matching algorithm is a novel and one of the most generic algorithms and is capable of handling even approximate tree matches.

2. Related Work

Tree structures have been well studied in diverse fields including computer vision, computational biology, natural language processing among others. Most applications consider *labeled* or *ordered* trees where the sibling nodes maintain a strict left-right order. The basic idea of matching such trees is to compute a minimized sequence of edit distances required to relabel or add/delete certain nodes. The edit distance notion for ordered trees was introduced by Tai [Tai79] and modified by Kosaraju [Kos89], Makinen [M̈89], Zhang and Shasha [ZS89], Klein [Kle98] and Chen [Che01]. In computer vision Liu et.al. [LG99] matches

DOI: 10.2312/pgs.20141255

2D shapes through their skeletal tree representation and Cantoni et.al. [CCG*98] uses trees to match two planar objects at multiple resolution levels. A similar approach was used for video comparison by Wing Ng et.al. [NKL01]. Tree matching is used in natural language processing in searching and retrieving complex feature structures from lexical databases as shown by Kilpelainen et.al. [KM92], Oflazer [Ofl96] and Wang et.al. [WMC09]. Syntactic tree matching is used to identify similarities between computer programs by Yang [Yan91], Hoffmann et. al. [HO82] and Ramesh [RR92]. Tree matching is also used for comparing similar web pages represented as labeled document object tree by Kumar et.al. [KTA*11] and Jindal et.al. [JL10]. In computational biology, tree matching is used by Aoki et.al. [AYO*03] for aligning maximally matching glycan subtrees, Luccio [LP91] for comparing glycogen trees, Jiang et.al. [JWZ95] for matching RNA structures, and Pisupati et.al. [PWMZ96] for matching 3D human lung structures.

The fundamental difference between all of the above methods and the one presented here is that, the latter considers arbitrary unordered and unlabeled geometric trees without any node correspondence or ancestor-descendant relationship. Since there is no prior assumption about the type of tree considered, the matching is more general in nature and the matching parts may exist anywhere in both the trees.

3. Problem Definition

A geometric tree is a graph with no cycles where each *node* has co-ordinate values and each *edge* is a geometric curve. The number of edges incident on a node is defined as its degree. A node is a *leaf*, *path* or *internal* when there are one, two or more than two incident edges respectively (Figure 2).

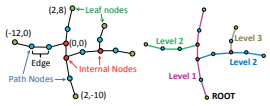

Figure 2: *Left: A geometric tree with node co-ordinates. Right: Branch decomposition with the root and different levels highlighted.*

Matching two geometric trees is equivalent to matching their node sets. However, a naïve node matching does not take into account spatial coherency, potentially generating matches in drastically different tree locations. Although this problem can be avoided by computing a match between the paths of the two trees, such an approach is computationally expensive due to the exponential number of paths (a path can be practically formed between any two tree nodes). Also, the matched paths will most likely have an undesirably large number of edge overlaps. A better approach is to decompose each tree into a set of edge-disjoint paths, unless such a decomposition is already provided, and then find matches between those decompositions. This process is called a *branch decomposition* (Figure 2, Appendix A), and an element or *branch* in this decomposition is a path from a leaf node to an internal node or another leaf node. In this process, if the tree is unrooted, one geometrically dominant branch is computed between two leaf nodes one of which is considered as the root. If the tree is rooted, edge-disjoint branches that start from an internal or root node and end in a leaf node are computed such that their union is the given tree. Each branch thus contains exactly one leaf and one internal or root nodes.

Given the branch decompositions of the two trees, a match is computed by finding a mapping between their decompositions such that a branch in one tree has at most one match in the other. Tree matching in this way ensures that no edge has more than one match in the other tree. Since the trees 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. The problem of similarity matching is defined as one in which every branch is matched to at most one branch, and the total match cost is minimum, i.e. the problem is to find the set of matching branches S, $S \subseteq A \times B$ 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 total cost $C = C_1 + C_2$ is minimum, where C_1 is the sum of the cost of the matches in S and C_2 is the sum of the cost of not matching the unmatched branches in either set.

4. Geometric Tree Matching

The problem of matching two geometric trees is reduced to that of finding an optimum match between their edge-disjoint branches. The cost, or measure of similarity between two branches is evaluated from their geometric and topological features. The geometric features used are branch length, algebraic and absolute sums of the turning angles at each intermediate branch node, difference in turning angles, and the number of self-intersections. The considered topological features are the ancestor-descendant relationship between branches and the hierarchical branch position from a specified root (Figure 3). Without loss of generality, trees considered here have linear edges (piece wise linear branches) and the branches may have self-intersections in general positions, although the intersections are not considered as nodes.

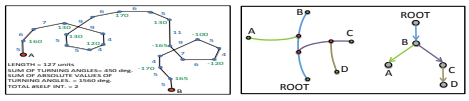

Figure 3: *Left: The geometric properties in the feature vector of a branch. Right: Hierarchical decomposition of a rooted tree. Branch B in level one has two children, A and C, each in level two. Branch D, a child of C is in level 3.*

The branch features form a vector space, called the feature space, in which each branch is represented by a feature vector. The cost, C_{ij} of matching two individual branches i and j, in the source and target trees respectively, is a scalar value obtained by computing the weighted distance between their respective feature vectors, i.e. $C_{ij} = \Sigma w_k * d_k(u_k, v_k)$, where u_k and v_k are the k^{th} 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 k^{th} feature respectively. The distance function considered is the Manhattan distance i.e. $d(u_k, v_k) = |u_k - v_k|$, although any other distance function can be used, e.g. Bhattacharya distance [Bha43] or Kullback-Liebler divergence [KL51].

Since the two trees to be matched may not have equal number of nodes, not all branches may have a match. Even if there are equal number of branches, a few matches may not be appropriate when considering node adjacency and edge coherency, and hence such branches are left unmatched. The cost $C_{i\phi}$, of not matching a branch i is calculated as a weighted combination of its features, i.e. $C_{i\phi} = \Sigma w_k * d_k(u_k, 0)$, where u_k is the k^{th} feature in the feature vector of branch i and w_k and d_k have the same meaning as before.

4.1. Tree Matching as Minimum Weight Perfect Matching Problem

Tree matching 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, termed as 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 $j \in Q$, and 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 one of i or j is a dummy node, and is zero otherwise. Finding an optimal match is thus reduced to finding a minimum weight (minimized sum of edge costs) perfect matching of G. A well known technique to find this match is the Hungarian algorithm [Kuh55, Mun57]

Minimum weight perfect matching problem: In a complete bipartite graph, G, 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 edge weights in M is minimum.

The above described formulation naïvely matches tree branches without considering spatial coherency, and being completely dependent on individual branch features can produce matches in unlikely tree locations (Figure 4).

In most practical cases, the two trees being matched are not completely different, rather they have patches of similar subtrees widely scattered, which a 'good' matching should identify. Therefore, the matching criterion should not be limited to the matching individual branches, but should also consider matching the subtrees originating from them. One way to identify such deeper matching is to enhance the cost basis for matching individual branches by including their descendants. The match cost between two branches a and b is thus computed recursively in terms of their children as follows (Algorithm 1). A minimum weight perfect matching, C_1 is computed for the graph with node sets representing the children branches of a and b. If C_2 is the cost of matching a and b individually, the overall match cost is given by $C_1 + C_2$. The new branch cost gives an intuitive matching in Figure 4.

Algorithm 1 Calculate Matching Cost (a, b)

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

Tree matching with the modified branch cost is also not sufficient for producing intuitive matches as the matching is still performed over the entire branch sets of both the trees, thereby keeping the risk of arbitrary matching still exposed. One way to avoid this is to restrict the matching between same hierarchical branch levels. However, there is no guarantee that similar subtrees in the two trees will be in the same level, and the number of levels in the trees can also be different (Figure 5). A more logical approach is to expose the branches in clusters, over multiple iterations of matching. In each iteration, only a few branches are exposed and thus the options for finding a match is limited, and if no acceptable match is available 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. This approach is termed as *sliding window matching* (Algorithm 2).

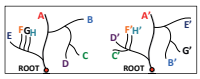

Figure 4: *Naïve match (same letters) differs from intuitive BE',DB',CG',ED'.*

Let S be the final set of matched branches, and σ_1 and π_1 denote the set of first level branches in the two trees. A minimum weight perfect matching, P is computed with the bipartite graph with node sets as σ_1 and π_1. The edge weights are 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. Further, since the matching cost between a_1 and b_1 includes the perfect matching cost between the their descendents, the matched descendent branches that are responsible for this minimum match cost between a_1 and

b_1 are also included in S. 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 branch is left unmatched in the subsequent matching process, since their 'non-matching' cost is also included and considered while matching a_1 and b_1 in the recursive Algorithm 1. Let $\sigma_1\prime$ and $\pi_1\prime$ be the set of branches which have remained unmatched in the two trees. 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. The process is continued by repeatedly adding new branches from lower hierarchical levels to the unmatched ones 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. If in a particular iteration, a branch b is not matched, but one of its descendants, say b_1 gets matched, then in the next iteration the branch b is modified by removing b_1 and all its descendents from its descendant list. This is especially helpful in situations shown in the right hand side of Figure 5.

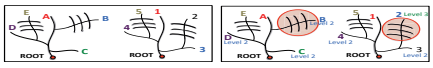

Figure 5: *Left: Restrictive match with modified branch cost produces a B-3,D-4 and E-5 match although identical subtrees B and 2 should be matched ideally. Right: Sliding window matching exposes branches B,C,D from source and 3,4 from target trees, all in level two. A perfect match produces B-3 and D-4 (and E-5), out of which D-4 has a low cost and considered a good match, while B-3 is not. In the next step, branch 2 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. A match between all branches could have produced an undesirable B-5 match.*

The sliding window technique allows matching of branches (and hence subtrees) scattered throughout the trees, minimizing the risk of producing arbitrary matches. Since the branch match cost is calculated recursively, it allows memoization of 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 need not be recalculated for the next iteration. The sliding window paradigm allows for flexible expansion of window size wherein branches from multiple levels may be exposed simultaneously, a feature which may be useful in some applications.

4.2. Choice of Weight Vector and Threshold

Typically, the more important tree features should be assigned higher weights, e.g. if the trees to be matched consist of straight line branches only, the turning angles of the branches practically play no role in matching and should have zero weights. An automatic way of weight assignment based on feature importance is described next.

Algorithm 2 Tree matching algorithm

Input: Branch sets A and B of source and target trees
Output: A match, S, of the form (a_i,b_i), where $a_i \in A, b_i \in B$, and if $(a_i,b_i),(a_j,b_j) \in S$, $a_i \neq a_j, b_i \neq b_j$
$\sigma\prime,\pi\prime \leftarrow$ First level branches of source and target trees
repeat
  $P \leftarrow$ a minimum weight perfect matching of $\sigma\prime$, $\pi\prime$
  for each Edge $(a,b) \in P$ **do**
    if Cost of edge $(a,b) < \eta$, **then**
      $Q \leftarrow$ perfect matching of descendants of a,b
      $S = S \cup \{(a,b)\} \cup Q$
      $\sigma\prime \leftarrow (\sigma\prime - a)$, $\pi\prime \leftarrow (\pi\prime - b)$
    end if
  end for
  $\sigma,\pi \leftarrow$ children branches of $\sigma\prime$ and $\pi\prime$ respectively
  $\sigma\prime = \sigma \cup \sigma\prime$, $\pi\prime = \pi \cup \pi\prime$
until $\sigma = \sigma\prime$ and $\pi = \pi\prime$
$S = S \cup P$

The feature vector f for each branch is normalized to be zero mean and unit variance. The normalized feature vectors for each tree is represented by an n-by-n matrix, $F = (f_1, f_2, ..., f_n)$, where $f_i (i = 1, 2, ..., n)$ is the feature vector of the i^{th} branch. The relative importance of the different features can be estimated from the eigen decomposition of the covariance matrix, $C = \frac{1}{n} F F^T$, which can be represented as $C = Q \Delta Q^T = \sum_{i=1}^{n} \sigma_i q_i q_i^T$, where $Q = (q_1, q_2, ..., q_n)$ is an orthogonal matrix of eigenvectors q_i of C, and Δ is a diagonal matrix of corresponding eigenvalues σ_i. The elements of the eigenvectors or principal components reflects the loading of each feature, e.g. if a feature heavily loads the first principal component (with highest eigenvalue), its variance is high along this component and hence it plays an important role in matching. The overall contribution or load, λ of a feature i is calculated by the weighted sum of the i^{th} element of each eigenvector with the corresponding eigenvalue. This ensures that the features loading the important principal components (associated with higher eigenvalues) are assigned high weights. The process is carried out for both the trees and the resultant weight for a feature is: $w_i = \frac{\lambda_{1i}}{\sum_{i=1}^{n} \lambda_{1i}^2} \frac{\lambda_{2i}}{\sum_{i=1}^{n} \lambda_{2i}^2}$, where λ_{1i}s and λ_{2i}s are the i^{th} feature loads for the two trees respectively, and n is the number of features. Thus the weights are proportional to their importance and the sum of weights equal one when the two trees are identical.

The threshold for selecting good matches is typically chosen before the first iteration as 10% of the cost of matching the larger tree to a null tree, and is gradually reduced in proportion to the iteration level, thereby ensuring a lower threshold for matching smaller subtrees. Although the

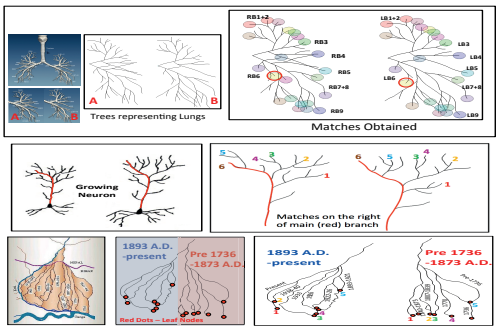

Figure 6: *Top Row Left: Matching both sides of a human respiratory tree (picture: 3D Bronch ipad/iphone app). The matches obtained between the right and left (shown reflected) lungs shows symmetry. A slight mismatch is shown by the red circle. Center Row: Aging process of cortical neurons ([PM01]). A fairly good match between the first level 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, which is of high geographical importance.*

weights and threshold are automatically calculated, the user can manually control them as well.

5. Results

Figure 6 shows the results of the matching algorithm when applied to real world physical systems and a series of synthetically generated geometric tree pairs (Figure 7). In each of the synthetic examples, the target tree is generated by arbitrarily editing the source tree by means of transforming and migrating different subtree parts, which conforms to almost all physically changing geometric trees. It can observed that the matching algorithm correctly identifies the similar tree branches across the two trees even when they undergo drastic transformations and migrations. Affine transformations like rotation and translation do not affect the geometric branch features, and can easily be detected by the algorithm, whereas scaling and shear will affect a few features and unless subtrees transform drastically, can be detected.

6. Summary

A variant of the minimum weight perfect matching, called the sliding window matching is introduced for finding similarities between geometric trees in terms of their geometric as well as topological features. The technique is the first algorithm to match geometric trees and has direct application to actual physical data having tree structure. When the tree structures representing the physical data undergo metamorphosis, the matching algorithm can be used to compare the trees to provide useful insight about the actual morphology of the physical system.

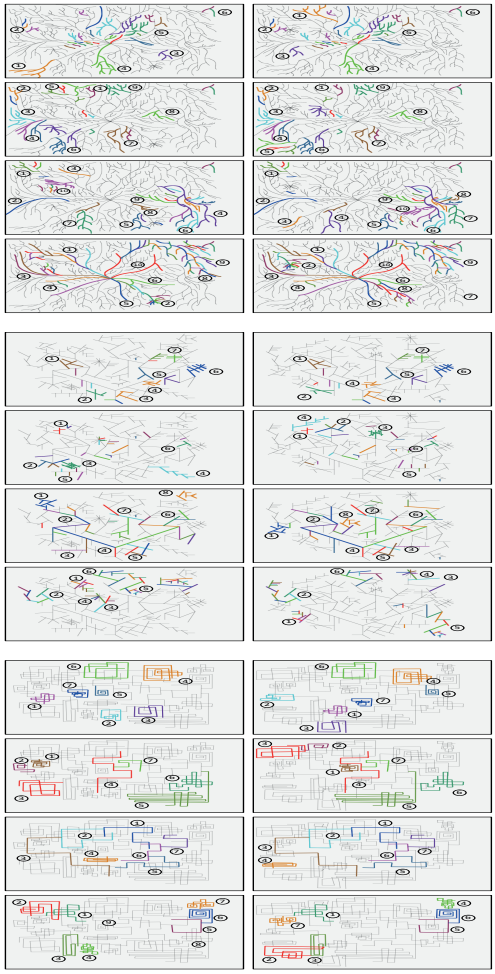

Figure 7: *Synthetic tree matching. Each example is split in four rows. Each row of each set shows the source and target trees. Partial matches are shown in each row for clarity. The matches in each set is highlighted with the same color and numbers. A zoom of 4x provides optimum clarity.*

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. *Bul. of Calcutta Math. Society 35* (1943), 99–109. 3

[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. 1

[GLC*05] GU J., LU L., CAI R., ZHANG H.-J., YANG J.: Dom-

inant feature vectors based audio similarity measure. In *Adv. in Multimedia Inf. Proc.*, vol. 3332. 2005, pp. 890–897.

[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. *Th. Comp.Sc. 143*, 1 (1995), 137. 2

[KL51] KULLBACK S., LEIBLER R. A.: On information and sufficiency. *Ann. Math. Stat. 22* (1951), 79–86. 3

[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. 1

[KM92] KILPELÃĐINEN 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. 1

[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. 3

[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. 1

[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

[M̈89] MÄKINEN E.: On the subtree isomorphism problem for ordered trees. *Inf. Proc. Letters 32*, 5 (1989), 271–273. 1

[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. 3

[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* (2001), 21–31. 5

[PWMZ96] PISUPATI C., WOLFF L., MITZNER W., ZERHOUNI E.: Geometric tree matching with applications to 3d lung structures. In *Proc. of ACM Comp. Geom.* (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. 1

[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 Prac. and Exp. 21*, 7 (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. 1

Appendix A: Branch Decomposition of a Tree

Let Ω be the set of all possible unique paths originating in an internal node or a leaf node and terminating in a leaf node in a geometric tree. The most dominant path p in Ω forms the main (first level) branch b. Ideally, this path should have two leaf nodes as its terminal nodes and at any internal node the turning angle should be close to 180°, implying path continuity (Figure 8). In general, this criterion is enough to compute the main branch. In some cases, if the application demands a main branch to be characterized by specific geometric or topological features, e.g. length, a weighted measure of the desired features is also taken into account (Figure 8).

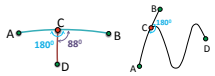

Figure 8: *Left: Path ACB having turning angle closer to 180° at C is preferred over ACD as main branch. Right: If ACD is the desired main branch, length is also considered.*

The main branch b 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 b are discarded. Next, for each internal node i of b, the most dominant path (determined similarly as the main branch) originating from i 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 first level branches originating from the root.

Appendix B: Complexity Analysis of Matching Algorithm

Let N be the average number of tree nodes, n be the average number of branches (hence leaf nodes), and d be the average number of children of each branch. A minimum weight perfect matching P is computed $log(n)$ times which is the number of levels of branches. Since the Hungarian Algorithm has a cubic run time, the overall runtime for perfect match calculation is $O(n^3 log(n))$. The match cost between a branch pair is recalculated only when one or more descendants of either has been matched 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 $O(d^3)$. 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 $O(n^3 d^3)$ which can be approximated to $O(n^3)$ assuming the average degree of each node is constant. 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 branch decomposition step is $O(n^3 log(n)) + O(N^2)$.