A tree-edit-distance algorithm for comparing simple, closed shapes

Philip Klein [*] Srikanta Tirthapura [*] Daniel Sharvit Ben Kimia[†]

Abstract
We discuss a graph-algorithmic approach to comparing *shapes*. We focus in this paper on comparing simple closed curves in the plane. Our approach is to (1) represent such a shape by its skeleton, which is a tree embedded in the plane, and (2) compare two shapes by comparing their skeletons via *tree edit-distance*.

In this paper, we define our version of tree edit-distance (it differs from that previously described in the literature), and give a polynomial-time algorithm to compute the distance between two trees.

1 Introduction

This paper arose out of a collaboration between a computer-vision researcher and an algorithms researcher. Kimia et al. [4] had previously compared shapes by comparing their graphs using a heuristic for general graph-comparison. The heuristic, due to Gold and Rangarajan [3], is based on finding a local minimum to a quadratic program. This approach had several disadvantages, however, and Kimia was searching for another approach. Klein suggested that the notion of edit-distance might be appropriate.

The notion of edit-distance originated in a paper by Wagner and Fischer [10] on comparing two character strings. There are three kinds of edit operations: deleting a character, inserting a character, and changing one character into another. For a given assignment of costs to all such edit operations, one can use dynamic programming to compute the minimum-cost sequence of operations required to convert one given string to another.

This notion has been generalized in several ways to apply to trees. For example, instead of characters, we have edge-labels. There are three edit operations: contracting an edge with a particular label, "uncontracting" an edge and giving it a particular label, and changing the label. There are polynomial-time algorithms to compute the minimum-cost sequence of operations required to convert one given tree to another.

The vision researchers (Kimia and Sharvit) concluded that these basic operations were not sufficient for the computer-vision application, and proposed an alternative set of edit operations. The algorithms researchers (Klein and Tirthapura) developed an algorithm for computing the edit distance of trees under the alternative set of edit operations.

Klein has implemented a version of this algorithm, and Kimia is experimenting with its application to shape comparison.

2 Edit Operations for Shape Comparison

Object recognition is an important task in computer vision. While many approaches have been proposed for shape comparison for the purpose of object recognition, we focus here on a particular approach which does not consider the two shapes to be matched *statically*, but rather considers them in the context of an embedding in a shape space with a dense collection of deformation paths, *i.e.,* morphing sequences taking one shape to another.

In this shape from deformation framework, similarity between two shapes is related to the best deformation path between them: if there exists a short and simple path, then the two shapes are similar. Clearly, a continuum of deformation paths connect any two shapes, which renders the task of characterizing them in a computational framework difficult. This problem can be resolved by creating an equivalence class of deformations which are then represented in the discrete domain. Specifically, the key to establishing this equivalence class is to understand how the representation of shape changes as the shape itself changes. We digress to describe the representation first.

We represent shape by a *shock graph*, which is constructed from the locus of the centers of maximal circles which are at least bitangent to the boundary. This locus is divided into branches which are the largest segments with monotonically increasing or decreasing radii. Note that radius is given the interpretation of time in a scheme where waves are propagated from the boundary, so that the shock locus is actually given a dynamic interpretation of the path of a moving particle. The nodes of the graph are the end points of these shock

[*]research supported by NSF Grant CCR-9700146. Department of Computer Science, Brown University, Providence, RI. Email addresses are {klein,snt}@cs.brown.edu.

[†]Department of Engineering, Brown University, Providence, RI. kimia@lems.brown.edu

branches. The links of the graph contain the length of the branch, the initial and final radii, the curvature and acceleration (the second derivative of arc-length with radius) functions along the branch. It is shown in [14] that this information is sufficient to completely reconstruct the shape.

Returning to the issue of how the representation of shape changes with the shape itself, note that in a majority of cases as a shape changes slightly, its shock graph representation does not *typically* change topology. Rather, only link attributes change slightly and continuously with the deformation. However, along certain points along the deformation path, there are instabilities in the representation, referred to as *transitions*, where the topology of the shock graph changes abruptly with a slight change in shape. Giblin and Kimia [13] have classified these transitions of shape into six types using results from singularity theory, as summarized in Figure 1. These are the *only* generic transitions or instabilities when deforming a smooth closed curve under a one-parameter family of deformations. These instabilities require that the representations on the either side of the transition must be identified as equivalent. The edit operations achieve this goal by explicitly transforming the discrete structure of the shock graph across each transition. From a graph-theoretic perspective and disregarding the "shock" semantics, these six shock transitions can be divided into three groups of edit operations: (*i*) *splice* or *prune* removes a shock branch and then merges the remaining two branches; (*ii*) *contract* removes the branch connecting two degree-three nodes and comes in two flavors; and, (*iii*) *merge* combines two branches at a degree two node and arises in three distinct cases. While these operation are complete for a closed shape, other transitions (and thus new edit operations) are necessary for deformations which create or close gaps, or which split a shape into the other shapes. We do not consider such cases in this paper and plan to address them in future work.

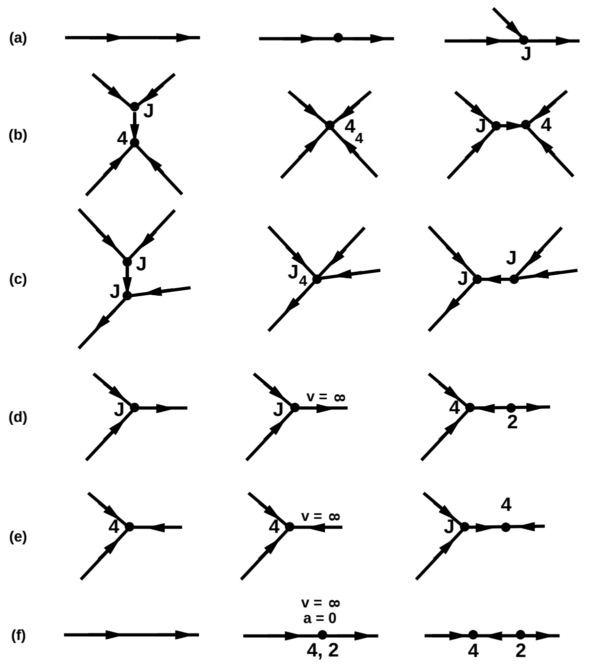

Figure 1: This figure from [13] shows a classification of all generic transitions of the shock structure for closed shapes under a one-parameter sequence of deformations which is proven using results from singularity theory. The intuitive description of these transition is as follows: normally, each point of the shock branch is the center of a bitangent circles, except at the initial point of the branch where these two points of tangency collapse onto each other, and at junctions where three branches come together representing tri-tangency. In a one-parameter family of deformation, additional constraints can be placed on the points which are: *(a)* tri-tangency with two of the points collapsing onto each other, *(b)* and *(c)* four points of tangency, *(d)* and *(e)* tri-tangency with infinite velocity along one of the branches, and *(f)* bitangency with infinite velocity and zero acceleration. The edit operations to make the right and left columns equivalent are: splice, contract (two types), and merge, respectively.

3 Previous work in tree edit distance

An ordered, rooted tree is a tree in which every node's children are ordered. Most of the work on edit distance in trees addresses the computation of distance between ordered, rooted trees. The comparison of rooted trees arises in applications where hierarchies must be represented, e.g. parse trees and image decomposition. Tai [8] gave the first algorithm for edit distance in such trees. Zhang and Shasha [11] gave a faster and more space-efficient algorithm. The space required by the latter is $O(n_1 n_2)$ where n_1 and n_2 denote the number of nodes in the two trees being compared. The time required is $O(n_1 n_2 \beta_1 \beta_2)$ where β_i is a parameter of a tree, called the *collapsed depth* of the tree by Zhang and Shasha, which is bounded by the depth and by the number of leaves. The value of β_i is likely to be much smaller than n_i, but is $\Omega(n_i)$ in the worst case. Hence in the worst case the running time is $O(n^4)$, where n is the input size. Klein [5] gave an algorithm for edit distance between rooted trees with complexity $O(n_1^2 n_2 \log n_2)$ which is $O(n^3 \log n)$. The same paper also considers the problem of comparing unrooted trees, i.e when there is no natural root in the trees. The time complexity of Klein's algorithm for the unrooted version is also $O(n^3 \log n)$.

One can also consider the comparison of unordered trees; however, the edit distance between such trees is NP-hard to compute [12]. Also, in view of the interpretation of ordered trees as embedded in the plane, one might consider the comparison of planar graphs; again, the edit-distance problem is NP-hard for such graphs [5].

Variations on edit distance have also been considered, e.g. minimizing average cost per operation [7].

4 Previous work using graphs in vision

Graphs have been used in image understanding in several ways. One can use a graph to represent a complex object or scene by capturing the relationships between its parts, as represented by the work of Eshera and Fu [2]. They proposed a distance between graphs and a method to compute it; the method, however, requires exponential time in the worst case. A similar approach has been used for recognition of Chinese ideographs [1].

As mentioned in the introduction, Kimia et al. [4] used a graph representation of a shape, a "shock graph," essentially the medial axis of the shape's boundary, augmented with some additional information. They compared shock graphs using a graph-comparison heuristic of Gold and Rangarajan.

The idea of using edit distance for comparison of shapes was introduced in [9], and the new edit operations were described in terms of "shocks." This paper contained nothing about the algorithm aside from stating that the problem could be solved in polynomial time. At that time, we had an implementation of an algorithm that was not capable of solving the new edit distance problem in full generality.

5 Tree edit distance

basic edit distance

As mentioned in the introduction, a basic definition of tree edit distance involves three edit operations: contract an edge, "uncontract" an edge, and change the label of an edge. Given an assignment of costs to these operations (where the cost may depend on the labels of the edges involved), the edit-distance between two trees is the minimum cost of a sequence of edit operations taking one input tree to another.

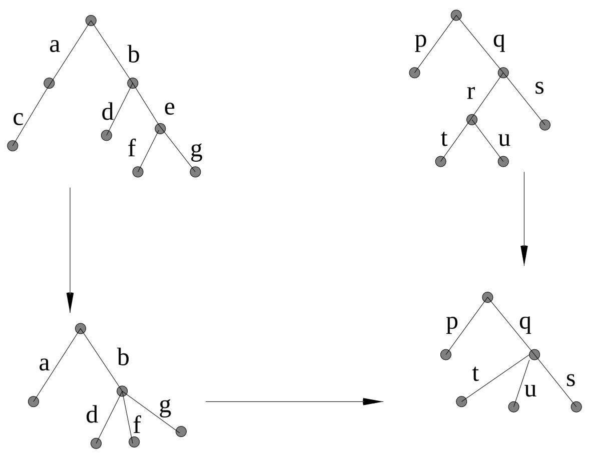

Figure 2: The comparison between two planar-embedded trees. They can be transformed into the same tree as follows. In the tree on the left contract the edges c, e. In the tree on the right, contract the edge r, and then change labels a, b, d, f, g to p, q, t, u, s.

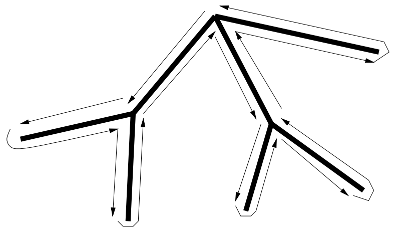

Figure 3: A rooted tree (in bold) and the corresponding Euler string of darts (indicated by arrows).

One can alternatively define the edit distance to be the minimum cost of separately transforming the two input trees to a common tree. That is, the edit distance is the minimum sum of costs of operations in two edit-operation sequences S_1 and S_2, where applying S_1 to input tree T_1 yields the same tree as does applying S_2 to the input tree T_2. There are several advantages to this formulation. First, there is no need for the "uncontract" operation. Second, the order within a sequence S_i is irrelevant; it can be viewed as a set.[1] An example is shown in the figure 2.

[1]There are two much less important advantages. The symmetric formulation better reflects the symmetry of the algorithm. Also, in some applications (our vision application in particular), the common tree has some meaning. Note that the labeling of the common tree is not uniquely determined.

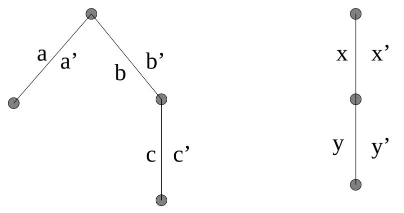

Figure 4: The Euler tour of the left tree is $b'c'cba'a$ and that of the second is $x'y'yx$.

modified edit distance

The problem with the basic definition of tree edit is that it lacks a way to combine two edges into one without losing information. In our vision application, a pair of edges with a common endpoint might represent shape information visually similar to that represented by a single edge. This is true even if a third edge is incident to the same node, as long as that third edge's other endpoint is a leaf. Consequently, it is natural to introduce a new edit operation, which we call "merge".

Merging two edges with a common endpoint results in a single edge whose label is derived from (conceptually, is the concatenation of) the labels of the two edges. The merge is permitted if there is at most one other edge incident to the common endpoint and if that edge's other endpoint is a leaf. If there is such an edge, the merge results in that edge being removed. We call this removing *pruning the edge.*[2]

We also make a restriction on the applicability of the contract operation. For reasons related to the vision application, an edge may be contracted only if its two endpoints each have degree three or higher.

Finally, the trees arising in the vision application have the property that the node degrees are no more than three. Essentially, a node of degree higher than three represents a degeneracy in the shape; degeneracies essentially never occur in real input data—though the common tree T (to which the input trees are transformed) is almost always degenerate.

6 Preliminaries

6.1 Euler strings

Given an ordered, rooted tree T, replace each edge $\{x, y\}$ of T by two oppositely directed arcs (x, y) and (y, x), called *darts*. The depth-first search traversal of T (visiting each node's children according to their order) defines an Euler tour of the darts of T. Each dart appears exactly once. (See Figure 4.) We interpret the tour as a string, the *Euler string* of T, and we denote this string by $E(T)$. The first dart of the string goes from the root to the rightmost child of the root. For a dart a, the oppositely directed arc corresponding to the same edge will be called a's *mate* and denoted mate(a). A *substring* of a string is defined to be a consecutive subsequence of the string.

[2]There are restrictions imposed by the algorithm on how the cost of a merge operation may depend on the edges involved; this will be discussed later.

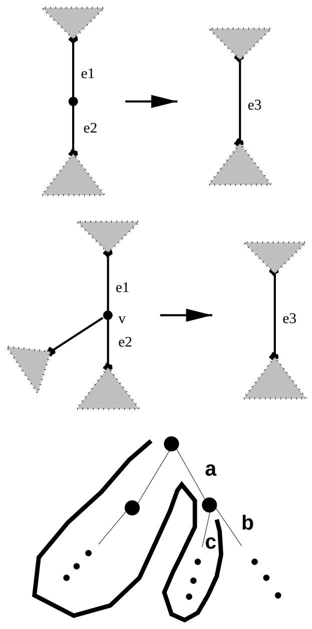

Figure 5: On the top is depicted a merge operation. In the middle is a merge with a pruning. At the bottom is a tree and Euler string used in explaining *left-pruning*.

6.2 Intuition

One can compute the basic edit-distance between two rooted trees T_1 and T_2 as follows. A *half-problem* is a pair $H_i = (T_i, s_i)$ where s_i is an Euler string for T_i. Essentially, the Euler string s_i specifies a portion of T_i. A *subproblem* is a pair (H_1, H_2) of half-problems, one for each tree. For each subproblem in some set, the dynamic program computes a kind of edit-distance between the indicated portions of the two trees; the value of this distance can be computed in constant time from the values for a subproblem having shorter strings. In the end, the dynamic program computes an edit-distance for a subproblem where both strings completely cover their trees; this edit-distance is the edit-distance between the original trees.

More specifically, to compute the edit distance for $((T_1, s_1), (T_2, s_2))$, one considers the rightmost darts of s_1 and s_2. In the optimal edit sequence, there are only three (non-exclusive) possibilities concerning these darts: (i) the rightmost dart in s_1 represents an edge of

T_1 that gets contracted, (ii) the edge corresponding to rightmost dart in s_2 is similarly contracted, (iii) the two edges are matched together, and one's label is changed to the others. For each of these cases, the edit distance can be computed from the edit distance of half-problems in which one or the other of the rightmost darts do not appear.

To extend this algorithm to handle the modified edit-distance requires an enhanced dynamic program that embodies several tricks. The first trick enables us to handle the merges. The difficulty presented by merges is representing the edge resulting from merges (a "merged edge"). Note, however, that the edges merged to form such an edge are consecutive edges on a root to leaf path P. In the dynamic program, each half-problem needs to specify at most one merged edge, namely the rightmost one. We can represent such a half-problem using an additional parameter, a node v, giving us the form (T, s, v) where s is an Euler string of T.

Incorporating this idea into the dynamic program is straightforward. As before, the three possibilities are (i) contract the merged edge in T_1, (ii) contract the merged edge in T_2, and (iii) match the merged edge in T_1 to the merged edge in T_2.

The second trick addresses the pruning of subtrees. In the definition of pruning, the edge being pruned must have a leaf as one endpoint. However, it is easy to see that by a series of merging and pruning steps, an entire subtree may be pruned away. Thus for each subtree in each tree, we can precompute the cost of pruning away that whole subtree.

The third trick addresses a rather technical problem. Consider the half-problem consisting of the tree and associated Euler string at the bottom in Figure 5. The rightmost dart is c. Recall that one possibility for the optimal edit sequence is that a and b are merged, and c is pruned off. In considering the current half-problem, however, we cannot anticipate whether a and b will be merged or not, and so it is not clear whether pruning off c should be allowed.

There is a solution to this problem. Assume that a and b are merged in the optimal edit sequence, and consider what happens to the edge d resulting from the merge (and possibly some other merges). The edge d can either be matched to some edge in the other tree, or can be contracted out. Assume first that it is matched. In this case, the cost of the corresponding subproblems in the dynamic program don't depend on a half-problem like that on the right in Figure 5, where c is the rightmost dart. Thus when we consider such a half-problem, we can assume that if a and b are to be merged then the resulting edge is to be contracted. The solution, then is this: the decision as to whether a and b are to be merged then contracted is not made when a or b is considered (i.e. when either is the rightmost dart); it is made when c is considered. That is, the cost is the minimum of (i) the cost of merging a and b and pruning out c and (ii) the cost of not doing so, *assuming* in both cases that a and b are contracted in the end (whether as a single, merged edge or as separate edges). This approach can be made to work under a reasonable assumption about the costs of contraction and merging, as described in Condition 8.2 below. This is called *left pruning* because the edge being pruned is to the left of the edges being merged.

We say a dart is *downward* if its head is farther from the root than its tail, and otherwise upward. We assume in our description that of the two darts associated with an edge, the downward dart is to the right of the upward dart in the Euler tour.

For technical reasons, the left pruning is considered not when the dart to be pruned is the rightmost dart in the Euler string, but when it is the second-from-rightmost dart, and the rightmost dart points up (towards the root). This is necessary to ensure that the case of pruning does not enter into the cost when the edge b is matched.

7 The algorithm

Here we neglect spelling out the base cases. For a dart d , we use $d + 1$ to denote the dart immediately to the left of d in the Euler tour.

The dynamic program builds a table indexed by two darts from T_1 (a_1 and d_1) and two darts from T_2 (a_2 and d_2). The procedure `lookup(a1, d1, a2, d2)` gives the value stored in the appropriate entry.

For some of the procedures below, there are versions for T_1 and T_2. For brevity, we give only the versions for T_1.

```
findCost(a1, d1, a2, d2) =
 if d1 is upward-pointing then
  maybeLeftPruneOut1(a1, d1, a2, d2)
 else
  if d2 is upward-pointing then
    maybeLeftPruneOut2(a1, d1, a2, d2)
  else (* both d1 and d2 are
       down-pointing darts *)
    findCost'(a1, d1, a2, d2,
            tail(T1, d1), tail(T2, d2))

findCost'(a1, d1, a2, d2, v1, v2) =
 min (match(a1, a2, d1, d2, v1, v2),
  contract1(a1, d1, a2, d2, v1, v2),
  contract2(a1, d1, a2, d2, v1, v2),
  mergeDown1(a1, d1, a2, d2, v1, v2),
  mergeDown2(a1, d1, a2, d2, v1, v2))

mergedown1(a1, d1, a2, d2, v1, v2) =
 for each child dart d of d1, return
  the cost of splicing out the
  subtrees rooted at all other
  children of d1,
 plus
  findCost'(a1, d, a2, d2, v1, v2)

contract1(a1, d1, a2, d2, v1, v2) =
 if the endpoints of the path from v1 to
 the head of d1 have degree > 2 and each
 node on this path is the leftmost child
 of its parent, then return the cost of
 contracting out the path,
  plus
   lookup(a1, d1 + 1, a2, d2)
 else infinity

match(a1, d1, a2, d2, v1, v2) =
 return the cost of matching
  the path in T1 from v1 to head of d1
  vs. the path in T2 from v2 to head of d2,
   plus
 lookup(substrings corresponding to the
  subtrees of T1 and T2 rooted as the heads
  of d1 and d2)
   plus
    lookup(
    a1,
    mate(ancestor dart of d1 having tail=t1)+1,
    a2,
    mate(ancestor dart of d2 having tail=t2)+1)

maybeLeftPruneOut1(a1, d1, a2, d2) =
 let d = d1 + 1
 if d is upward then lookup(a1, d, a2, d2)
 else (* d is the left sibling of d1 *)
  if (a1 is not the mate of d) then
    min (lookup(a1, d, a2, d2),
      cost of pruning out subtree
        rooted at d
      plus
        lookup(a1, parent of d1, a2, d2))
```

It is easy to order the computation of subproblems so that the cost of a subproblem has already been computed when it is needed. We have sketched the algorithm for comparing rooted trees.

7.1 Analysis

Since this is a dynamic program where the cost of a subproblem can be computed in constant time from the cost of "smaller" subproblems, the time complexity is of the order of the number of subproblems. Let n_1, d_1 be the number of edges and the depth of the tree T_1 respectively. Similarly n_2 and d_2.

A subproblem has three parts for each tree, the left dart l_i, the right dart r_i and the vertex v_i that lies between the top of l_i and top of r_i. l_i and r_i could take $2n_i$ values each and for a given l_i and r_i, v_i could take atmost d_i values. Hence, a straightforward analysis yields a time complexity of $O(n_1^2 n_2^2 d_1 d_2)$.

For the space complexity we observe (as in [11]) that we do not have to store the solution to every subproblem all the time and can reuse the space. This leads to a space requirement of $O(n_1 n_2)$ for the "permanent array" (i.e for the subproblems we need to store throughout the computation) and the rest of the computation can also be done so that its space requirement is $O(n_1 n_2)$. Hence the space complexity is $O(n_1 n_2)$.

8 Correctness

The *upper endpoint* of an edge is the endpoint of the edge that is closer to the root. The other endpoint is called the *lower endpoint.*

A node has *high degree* if its degree is greater than or equal to three. Otherwise it is said to have *low degree.*

For a pair of rooted trees A, B containing edges named e and e' respectively, the edit operations are as follows.

- merge(e) merges e with its parent edge, splicing off its left or right sibling edge if it has one. This operation is legal if the degree of the common node is at most three. The name of the merged edge is e.

- *contract*(e) contracts the edge e. This operation is legal if the endpoints of e have high degree.

- match(e, e') establishes a correspondence between the edge e of one tree and the edge e' of the other.

A sequence of edit operations $o_1 \dots o_n$ is a legal edit sequence for a pair of trees A_0, B_0 if

- o_i is a legal operation on the pair of trees A_{i-1}, B_{i-1}, and

- the trees A_i, B_i are obtained from A_{i-1}, B_{i-1} by performing o_i

We also require that, for each edge name e, there is at most one match operation involving e.

For two trees, we say an edit sequence is in *normal order* if, for each of the two trees, the order in which its edges appear in the sequence is consistent with the order in which they appear in its Euler sequence.

We assume that the following property holds for the merges.

CONDITION 8.1. *Associativity of merging: For edges a, b, c the result of merging a with b and then the result with c is the same as the result of merging b and c first and then merging a with the result. The costs of the two sequences of edits are also the same.*

LEMMA 8.1. *Any edit sequence can be reordered so that it is in normal order, while preserving legality.*

We say an edge named e *survives* an edit sequence if an edge with that name exists in one of the two trees resulting from the edit sequence.

We say a sequence is *complete* if the sequence is legal and every surviving edge participates in a match operation.

For an edit sequence, we inductively define the *consumption* relation between edges of the same tree, as follows. An edge e *consumes* an edge e' if $e = e'$ or e consumes an edge with which e' was merged.

For an edit sequence $o_1 \dots o_n$ on trees A_0, B_0, we say operation o_i is a *contracting right merge* if $o_i = \text{merge}(e)$ where e is the right child edge of its parent in A_{i-1} or B_{i-1}, and e does not survive the whole edit sequence.

We now introduce an auxiliary operation, leftsplice(e), that truncates the subtree rooted at e (including e). Given an edit sequence $o_1 \dots o_n$ in normal order, for each contracting right merge $o_i = \text{merge}(b)$, we replace o_i with the two operations contract(a), leftsplice(c), where (as in Figure 5) a and c are the parent and left sibling edge of b. For each replacement, define the cost of the leftsplice operation to be whatever value preserves the overall cost of the sequence.

LEMMA 8.2. *The leftsplice transformation preserves legality.*

Proof. Consider a particular contracting right merge $o_i = \text{merge}(b)$, and let a and c be the parent and sibling of b. We will consider the legality of the sequence after the replacement

$$\text{merge}(b) \longrightarrow \text{contract}(a)\text{leftsplice}(c)$$

By definition of contracting right merge, the edge b is eventually consumed by some edge b' that gets contracted. By legality, the upper and lower endpoints of b' at time of contraction must have high degree.

By normal order, the upper endpoint of b' just before contraction is the upper endpoint of a in A_i or B_i. By legality of the contraction, this endpoint has high degree. Since b has a left sibling in A_i or B_i, the lower endpoint of a has high degree as well. Hence contracting a is legal. After contracting a, since a's upper endpoint had high degree, the upper endpoint of b has degree four or more (three from the upper endpoint of a, plus one for c). Hence after leftsplice(c) the upper endpoint of b still has high degree. Hence the contraction of b' remains legal.

The leftsplice operations violate normal order of the edit sequence. Reorder the edit sequence to be in normal order. Since not doing the left splice now keeps the degree of the upper endpoint of b at four until later in the edit sequence, whichever operations are possible currently are also possible if the sequence were transformed so that the left splice is done later.

We make assumptions about the cost functions for edit operations so that the cost of a leftsplice operation on an edge e can be derived *a priori* from the tree containing e.

CONDITION 8.2. *We assume that the cost of merge(e) is the sum of two parts, (1) the cost of merging e with its parent and (2) the cost of pruning out the subtree rooted at it's left or right sibling (f), if it has one, denoted by $C_s(f)$. In the problem as posed by the vision application, there is a restriction that only leaf edges can be pruned out. So, the cost of pruning a subtree is precomputed as the least cost required to convert the subtree into a single edge and then truncate it.*

Suppose the sequence of edges $e_1, e_2 \dots e_k$ got merged into a single edge and the result was contracted. Let $C_c(e_1 \dots e_k)$ denote the cost of the above sequence of operations excluding the cost of the prunes (part (2) above). We require that $C_c(e_1 \dots e_k) - C_c(e_2 \dots e_k) - C_c(e_1)$ is a function of e_1 and e_2, say $f(e_1, e_2)$.

We now assign the cost of leftsplice(c) to be $C_s(c) + f(a, b)$ where a and b are the parent and the right sibling of c in the tree containing c. This can be computed *a priori* by looking at the tree containing c.

Let's say that $a, b \dots x$ are all merged into a single edge before getting contracted. The difference in cost between the sequence before and after the transformation is seen to be $[C_c(a, b \dots x) + C_s(c) - C_c(b \dots x) - C_c(a) - \text{leftsplicecost}(c)]$, (where leftsplicecost($c$) is the cost of the operation leftsplice(c)) which is zero. This assignment of cost to the leftsplice operation is thus seen to work.

Now, for any legal, normal-order sequence of edit operations including the auxiliary operation leftsplice, we say the sequence is *feasible* if

- there are no contracting right merges, and

- for each leftsplice(c), the sequence contains contract(a) and contract(b') where a is the parent of c in one of the original trees, and b' is a consumer of the right sibling of c in that tree.

We have proved the following lemma.

LEMMA 8.3. *For any complete edit sequence (not containing a leftsplice operation), there is a corresponding feasible normal-order complete edit sequence (using leftsplice operations) of the same cost.*

By applying the reverse of the transformations that introduced the leftsplice operations, we can show the following.

LEMMA 8.4. *For any feasible normal-order complete edit sequence, there is a corresponding complete edit sequence not containing a leftsplice operation.*

LEMMA 8.5. *The dynamic program computes the minimum cost of a feasible normal-order complete edit sequence with leftsplice operations.*

To prove this lemma, first we prove that given a feasible normal-order complete edit sequence $o_1 \ldots o_n$ on the trees T_1 and T_2, the algorithm computes a cost less than or equal to the cost of the edit sequence.

A subproblem of the dynamic program is specified by a tuple $(a_1, d_1, a_2, d_2, v_1, v_2)$ where a_i and d_i are darts in respective T_i's and v_i is a node, where d_i comes before a_i in the Euler tour of T_i, where the upper endpoint of a_i is an ancestor of the upper endpoint of d_i, and where v_i lies between these endpoints.

Given such a subproblem and an edit operation contract(d_1), there is a corresponding modified subproblem $(a_1, d_1 + 1, a_2, d_2, v_1, v_2)$. For the edit operation match(d_1, d_2), the subproblem is $(a_1, d'_1, a_2, d'_2, v'_1, v'_2)$ where d'_i is the dart following the mate of d_i, and v_i is the upper endpoint of d_i. Similarly, for each other edit operation there is a corresponding modified subproblem.

The effect of applying a sequence of operations $o_1 \ldots o_k$ on a subproblem is defined to be the effect of applying o_1, then o_2 and so on in sequence.

The proof proceeds as follows. Assume inductively that the lemma holds for any rooted subtrees of T_1 and T_2 where at least one of the subtrees is a strict subtree (i.e. is smaller than the tree). We prove by reverse induction on i that for the subproblem s resulting from applying $o_1 \ldots o_{i-1}$, the value of `lookup(`s`)` is at most the cost of the operations $o_i \ldots o_n$. The induction step is based on a case-analysis of the operation o_{i-1} and the corresponding steps in the algorithm. In the case of match, we use the induction hypothesis on subtrees.

For the other direction in the proof (to show that the dynamic program computes a cost not greater than the best feasible normal-order edit sequence), the interesting aspect is to prove that the algorithm considers a leftsplice on an edge c only if its right sibling and parent (b and a) have both been contracted earlier. In the following discussion, by the parent or sibling of a dart, we mean the parent or sibling of the edge corresponding to the dart.

The algorithm considers a leftsplice operation on an edge in tree T_1 in the subroutine `maybeLeftPruneOut1(a1,d1,a2,d2)`. This subroutine considers a leftsplice of the tree rooted at dart `d1+1`, rather than the tree rooted at dart `d1`. This way, it is ensured that the right sibling of `d1+1` has been contracted (since `d1` is upward pointing).

Since `a1` occurs to the left of the mate of `d1+1` (from the condition in the subroutine), the parent of `d1+1` has one dart between `a1` and `d1+1` and has only one dart in the Euler string. This ensures that the parent of `d1+1` has been contracted too.

We obtain

THEOREM 8.1. *The dynamic program computes the minimum cost of a complete legal edit-sequence without leftsplices.*

References

[1] H. Dong, Y. Wu, and X. Ding, "an ARG representation for Chinese characters and a radical extraction based on the representation," 9th IEEE International Conference on Pattern Recognition, vol. 2 (1988), pp. 920-922

[2] M. Eshera and K. Fu, "An image understanding system using Attributed Symbolic Representation," *IEEE Transactions on Pattern Analysis and Machine Intelligence 14* (1986), pp. 604-618.

[3] S. Gold and A. Rangarajan, "A graduated assignment algorithm for graph matching," *IEEE Transactions on Pattern Analysis and Machine Intelligence 18* (1996), pp. 377-388.

[4] B. Kimia, J. Chan, D. Bertrand, S. Coe, Z. Roadhouse, and H. Tek, "A shock-based approach for indexing of image databases using shape," *Proceedings of the SPIE's Multimedia Storage and Archiving Systems II* (1997), pp. 288-302.

[5] P. Klein, "Computing the edit distance between unrooted ordered trees", *Proceedings, 6th European Symposium on Algorithms* (1998), pp. 91–102.

[6] M. Maes, "On a cyclic string-to-string correction problem," *Information Processing Letters* 35 (1990), pp. 73-78.

[7] A. Marzal and E. Vidal, "Computation of normalized edit distance and applications", IEEE Transactions

on Pattern Analysis and Machine Intelligence, vol 15, (1993), pp. 926-932.

[8] K.-C. Tai, "The tree-to-tree correction problem", *Journal of the Association for Computing Machinery* 26 (1979), pp. 422-433.

[9] Srikanta Tirthapura, Daniel Sharvit, Philip Klein, and Benjamin Kimia, "Indexing based on edit-distance matching of shape graphs,"*Proceedings, SPIE's Multimedia Storage and Archiving Systems II* (1998)

[10] R. A. Wagner and M. J. Fischer, "The string-to-string correction problem," *Journal of the Association for Computing Machinery* 21, (1974), pp. 168-173.

[11] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems, *SIAM Journal on Computing* 18 (1989), pp. 1245-1262.

[12] K. Zhang, R. Statman and D. Shasha, "On the editing distance between unordered labeled trees," *Information Processing Letters* 42 (1992), pp. 133-139

[13] P. J. Giblin and B. B. Kimia, "On the Local Form and Transitions of Symmetry Sets, and Medial Axes, and Shocks in 2D", *Proceedings of the Seventh International Conference on Computer Vision, KerKyra, Greece, September 20-25, 1999*, IEEE Computer Society Press, pp. 385-391.

[14] P. J. Giblin and B. B. Kimia, "On the Intrinsic Reconstruction of Shape from its Symmetries", *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition, Fort Collins, Colorado, USA, June 23-25, 1999*, IEEE Computer Society Press, pp. 79-84.