Report Number: WUCS-94-30

1994-01-01

Morphing Binary Trees

John Hershberger and Subhash Suri

We investigate the problem of transforming one binary tree into another by rotatoins, subject to certain weight ocnstraints on the nodes of the trees. These constraints arise in the problem of "morphing" one simple polygon to another simple polygon by continuous deformatinos (translations and scalings) that preserve the turn angles and the simplicity of the polygon; the two polygons must have the same sequence of turn angles. Our main theorem is that two arbitrary n-leaf binary trees satisfying our weight constraint can be morphed into each other with O(n log n) rotations. Furthermore, we also present an O(n log... **Read complete abstract on page 2.**

Morphing Binary Trees

John Hershberger and Subhash Suri

Complete Abstract:

We investigate the problem of transforming one binary tree into another by rotatoins, subject to certain weight ocnstraints on the nodes of the trees. These constraints arise in the problem of "morphing" one simple polygon to another simple polygon by continuous deformatinos (translations and scalings) that preserve the turn angles and the simplicity of the polygon; the two polygons must have the same sequence of turn angles. Our main theorem is that two arbitrary n-leaf binary trees satisfying our weight constraint can be morphed into each other with O(n log n) rotations. Furthermore, we also present an O(n log n) time algorithm to determine these rotations. The previous best algorithm for this problem used O(n4/3 + ε) rotations.

Morphing Binary Trees

John Hershberger and Subhash Suri

WUCS-94-30

November 1994

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130-4899

Morphing Binary Trees

John Hershberger
Mentor Graphics
San Jose, CA 95131

Subhash Suri
Washington University
St. Louis, MO 63130

November 1, 1994

Abstract

We investigate the problem of transforming one binary tree into another by rotations, subject to certain weight constraints on the nodes of the trees. These constraints arise in the problem of "morphing" one simple polygon to another simple polygon by continuous deformations (translations and scalings) that preserve the turn angles and the simplicity of the polygon; the two polygons must have the same sequence of turn angles. Our main theorem is that two arbitrary n-leaf binary trees satisfying our weight constraint can be morphed into each other with $O(n \log n)$ rotations. Furthermore, we also present an $O(n \log n)$ time algorithm to determine these rotations. The previous best algorithm for this problem used $O(n^{4/3+\epsilon})$ rotations.

1 Introduction

"Morphing," the continuous deformation of one shape to another, is a popular theme in computer graphics [1, 4, 5, 6]. A recent paper by Guibas and Hershberger [3] considers the problem of morphing a simple polygon P to another simple polygon Q whose edges, taken in counterclockwise order, are parallel to the corresponding edges of P and oriented the same way (we say that the two polygons are *parallel*). The transformation preserves simplicity and parallelism: every intermediate polygon during the deformation is simple and parallel to Q.

1.1 Node Weights and Valid Trees

The algorithm of Guibas and Hershberger reduces the geometric problem of deforming polygons to a combinatorial problem involving rotations on binary trees. In particular, the problem of morphing polygons is first reduced to the slightly simpler problem of morphing bi-infinite polygonal chains. A bi-infinite polygonal chain is represented by a binary tree—the leaves represent the turn angles of the chain in order, and the tree structure represents a hierarchical grouping of the angles of the chain. Each node of the tree has a real-valued weight that corresponds to $1/\pi$ times the total turn angle in its group. The weight of a leaf is an arbitrary real number in the range $(-1, +1)$; the weight of an internal node is the sum of the leaf weights in its subtree. We say that the weight at a node is *valid* if it lies in the open interval $(-1, +1)$, and a tree is valid if all its nodes are valid. Thus, a valid tree is one all of whose nodes correspond to realizable turn angles, namely ones in the range $(-\pi, +\pi)$.

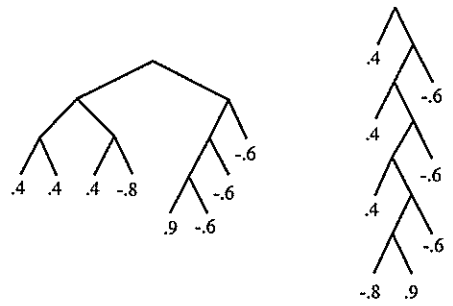

Figure 1: Two valid trees with the same weight sequence at their leaves.

Let T_1 and T_2 be two valid n-leaf binary trees whose leaves have the *same* weight sequence $w_1, \ldots, w_n$. These trees correspond to two *parallel* bi-infinite chains, that is, chains with the same turn-angle sequences. The structures of T_1 and T_2 above the leaves, however, are in general not the same. See Figure 1. These differing structures correspond to two different conformations of polygonal chains with the same underlying turn sequence. In particular, the two trees may have vastly different heights, ranging from $\Theta(\log n)$ to $\Theta(n)$.

The *tree-morphing* problem is to transform T_1 to T_2 by rotations that preserve validity, called *valid rotations*. In the polygon-morphing setting, a valid rotation corresponds to a deformation (translation and/or scaling) of a sub-chain that preserves parallelism and simplicity. Transforming T_1 to T_2 by a sequence of m valid rotations corresponds directly to morphing one bi-infinite chain into another with m elementary morphing steps. In the remainder of this paper, we will focus exclusively on the tree-morphing problem, ignoring any relationship to the polygon morphing problem.

1.2 Our Results

The problem of transforming one tree to another *without any weight constraint* has been studied before. Sleator, Tarjan, and Thurston [7] show that an n-node tree can be made isomorphic to any other n-node tree by at most $2n - 6$ rotations, slightly improving an earlier result by Culik and Wood [2].

Our tree-morphing problem, however, is more difficult because of the weight constraint on the nodes, which restricts the choice of rotations available to an algorithm. The best previous algorithm for tree morphing requires $O(n^{4/3+\epsilon})$ rotations [3]. Our main result is the following theorem: $O(n \log n)$ rotations suffice to morph any valid binary tree into another such tree. We can also compute these rotations in the same time bound.

This paper is organized in seven sections. In Section 2, we review some of the relevant material from the paper of Guibas and Hershberger, and introduce the basic terminology. In Section 3, we introduce the concept of node inclinations and a key invariant maintained by our algorithm. Sections 4 and 5 provide the details of the proof of our main theorem, and Section 6 describes our algorithm for finding the rotations. We conclude with some remarks and open problems in Section 7.

2 The Basics of Tree Morphing

Let T_1 and T_2 be two valid n-leaf trees with the same weight sequence at their leaves. Tree T_1 is called the *source tree* and T_2 the *target tree*. The weight at each internal node in either tree is the sum of the weights at its leaf descendants; because T_1 and T_2 are in general not isomorphic, the internal weights are not the same in the two trees. Because the trees are valid, the weight of every node is strictly between -1 and $+1$.

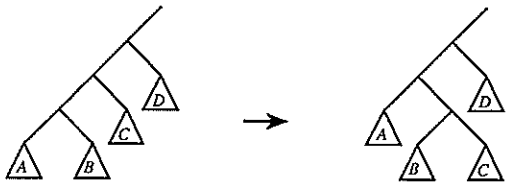

Figure 2: A rotation is valid if all resulting node weights are valid.

A tree rotation creates one new node whose weight is not present in the tree before the rotation. See Figure 2; the weight of the new parent of B and C does not exist in the tree before the rotation. A *valid rotation* is one in which the new weight lies in the open interval $(-1, +1)$.

The algorithm of Guibas and Hershberger [3] transforms T_1 to T_2 by a sequence of valid rotations. The following high-level algorithm shows how the sequence is computed:

ALGORITHM BASICMORPH

while $|T_1| > 1$ **do**

1. Let a' and b' be a pair of sibling leaves in the target tree T_2, and let a and b be the corresponding leaves in the source tree T_1.
2. Perform valid rotations on T_1 to make a and b into siblings in T_1.
3. Collapse a and b into their parent in T_1, and assign the parent node (a new leaf) the sum of the weights of a and b. Similarly collapse a' and b' into their parent in T_2. This produces two smaller valid trees with the same weight sequence at the leaves.

endwhile.

2.1 Spines and Sibling Operations

The core of the algorithm is Step 2. We call one execution of this step a *sibling operation*, because it makes two consecutive leaves into siblings. The two consecutive leaves of T_1 that are made into siblings are joined by a path whose structure, shown in Figure 3, is best described in terms of *spines*. (The significance of the arrows in the figure will be explained later.)

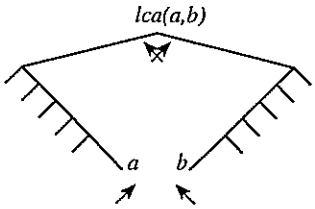

Figure 3: Leaves a and b in the source tree are ready for a sibling operation.

A *right spine* is a maximal sequence of right children in a tree; the top of a right spine is either the root of the tree or the left child of its parent. A left spine is defined symmetrically. Each leaf is the bottom of exactly one spine, either left or right. The total length of all the spines in an n-leaf tree is $2n-2$, since every edge belongs to exactly one spine. We denote by $lca(a,b)$ the lowest common ancestor of two leaves a and b. In Figure 3, the path from the left leaf a to $lca(a,b)$ consists of a right spine followed by a single edge from the top of the spine to $lca(a,b)$. The path from b to $lca(a,b)$ consists of a left spine followed by a single edge.

In general, the combined spine length for two adjacent leaves can be $\Omega(n)$, implying a similar lower bound on each execution of Step 2 in the worst case. To beat the naïve quadratic bound for the entire algorithm, one must take additional steps to reduce the spine lengths for the merging siblings.

2.2 Spine Contraction and Normalization

The following three lemmas are from [3].

Lemma 2.1 *Among any four consecutive internal nodes along a spine in a valid tree, at least one can be rotated out of the spine by a valid rotation.*

Lemma 2.2 *Let a and b be two consecutive leaves whose weights have a valid sum, and suppose that the spines above a and b have been contracted to length at most four by valid rotations. Then a and b can be made into siblings by at most eight valid rotations involving only nodes on the path from a to b.*

Lemma 2.3 (Normalization Lemma) *With $O(n \log^* n)$ valid rotations, we can transform one valid n-leaf tree into another valid tree in which the maximum spine length is $O(1)$.*

Given two consecutive leaves a and b as in Lemma 2.2, we can shorten the spines above them to at most four edges apiece by repeated applications of Lemma 2.1. These *spine-contracting* rotations affect only internal nodes of the spines, and not the nodes at the tops. Once the spines have been contracted, we can make a and b into siblings by at most eight more rotations, possibly involving the tops of the spines.

If the path length between a and b is k at the start of Step 2, then $k-2$ valid rotations suffice to make a and b into siblings. The number of rotations is equal to the sum of the lengths of the spines based at a and b. To make the sibling operation efficient, we want to

ensure that these spines are short. The Normalization Lemma above shows how to make *all* spines short initially. The constant in Lemma 2.3 is modest: the maximum spine length can be reduced to 8 by $O(n \log^* n)$ valid rotations (this is not stated in [3], but is not difficult to prove). We call the operation of shortening the maximum spine length to eight a *normalization* of the tree.

The three lemmas given above are the basic tools used by Guibas and Hershberger [3] to morph T_1 into T_2 by $O(n^{4/3+\epsilon})$ valid rotations. Specifically, they use Lemma 2.1 to argue that if all spines are of constant length initially, then only $O(n^{1+\epsilon})$ rotations are needed during $n^{2/3}$ iterations of the algorithm's main loop. Their algorithm runs in $n^{1/3}$ phases, each consisting of a normalization and $n^{2/3}$ iterations of the main loop. The running time of a single phase is $O(n^{1+\epsilon})$, and the overall running time of the algorithm is therefore $O(n^{4/3+\epsilon})$.

In the next three sections, we develop an improved algorithm that uses only $O(n \log n)$ rotations to morph one tree into another. The number of rotations needed to perform a sibling operation in Step 2 of BasicMorph is the sum of the spine lengths above the two consecutive leaves. The $O(n^{4/3+\epsilon})$ algorithm achieves its efficiency by shortening *all* spines periodically and observing that spines cannot grow too rapidly during a sequence of sibling operations. That algorithm's insistence on keeping all spines short is overly aggressive. We improve the rotation bound by focusing only on the spines that participate in sibling operations. We identify those spines *a priori* and maintain them at constant length. The cost of a single sibling operation is therefore constant; however, we need $O(\log n)$ rotations to restore the spine length bound after each sibling operation.

3 Node Inclinations and the Inclination Invariant

The pattern of sibling operations (though not the exact sequence) is completely determined by the target tree T_2. Each sibling operation collapses a pair of sibling leaves of T_2 into their parent node, while keeping the rest of the target tree unchanged.

At some time during the run of the algorithm, each leaf of the target tree collapses into its parent. Internal nodes of the target tree become leaves when their children collapse into them, and then they collapse into their parents. For each node, the direction of the edge to its parent is fixed. If a target tree node is a left child, then it will be paired with a leaf to its right when it eventually collapses into its parent. We say that a leaf that is a left child in the target tree is *right-inclined*, because it will merge with a leaf to its right.

Although leaf inclinations are defined on the target tree, they are useful because of what they tell us about the source tree. If a' is a right-inclined leaf in the target tree, then the corresponding leaf a in the source tree *is also right-inclined*: it will merge with a leaf to its right when it participates in a sibling operation.

3.1 Merge Centers

Consider the sequence formed by taking the leaf inclinations in left-to-right order. The sequence is composed of alternating runs of right and left inclination. Each alternation from right inclination to left inclination is associated with a pair of sibling leaves in the target tree. Each such pair is a possible site for a sibling operation, and we call the gap between the two such leaves a *merge center*. Because the leaf inclinations are the same in

the two trees, the merge centers are also the same.

The number of merge centers is non-increasing as the algorithm runs: Each sibling operation removes the two leaves adjacent to a merge center, replacing them by a single leaf whose inclination is determined by the target tree. The new leaf may or may not belong to a merge center. No other leaves change inclination. See Figure 4.

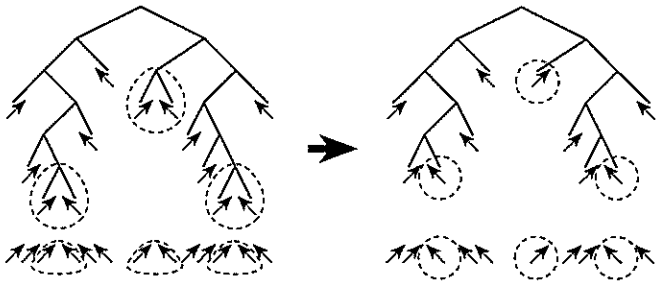

Figure 4: The target tree: leaf inclinations, merge centers, and the results of performing sibling operations at the merge centers.

The inclination of nodes in the target tree is determined by that tree's structure, but the inclination of nodes in the source tree is less obvious, since the structure of the source tree differs from that of the target tree. We define the inclination of a source tree node to be the union of the inclinations of its leaf descendants. Thus every node has some inclination, and some are inclined both left and right (such nodes are *both-inclined*; nodes inclined only one way are *singly-inclined*). The labels in Figure 3 show the inclinations of a, b, and $lca(a, b)$.

3.2 The Invariant

If leaves a and b participate in a sibling operation, the cost of the operation depends on the lengths of the right spine above a and the left spine above b. Any *left spine* involving a or *right spine* involving b has no effect on the cost of this sibling operation. For this reason, we maintain the following invariant:

> **Inclination Invariant:** If a node in the source tree is right inclined (left inclined), then the portion of the right (left) spine above it to which it belongs (if any) has constant length.

The Inclination Invariant is established in the beginning by the normalization operation (Lemma 2.3), which ensures that all spines have constant length (≤ 8).

Lemma 3.1 *If the Inclination Invariant holds, then the two leaves at any merge center are joined by a constant length path in the source tree.*

> **Proof:** The path consists of two spines, which have constant length by the Inclination Invariant, plus two edges linking the spines to the lowest common ancestor of the two leaves. ∎

Before describing how to restore the Inclination Invariant after a sibling operation, we must understand the effect of rotations on spine lengths. The following lemma summarizes those effects.

Lemma 3.2 *Let subtrees A and B be two children of a spine, consecutive in left-to-right order. If a rotation is applied, making A and B into sibling grandchildren of the spine, then the left spine of A and the right spine of B are lengthened by one. No other spine gets longer.*

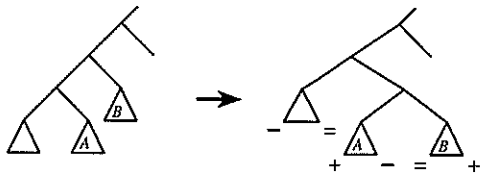

Figure 5: $+/-/=$ indicate changes in spine lengths due to the rotation.

Proof: See Figure 5. ∎

4 Restoring the Invariant

In this section we describe how to restore the Inclination Invariant after it has been invalidated by rotations. Although this may in general require many rotations, by restricting the invalidating rotations to subtrees containing only a few merge centers, we can apply some of the ideas from balanced tree algorithms to restore the invariant with only $O(\log n)$ rotations.

We choose the constant c implied by the Inclination Invariant to be 9 (required by Lemma 4.2 below). We first consider restoring the invariant for a subtree in which all nodes have the same inclination.

Lemma 4.1 *Let $T' \subseteq T_1$ be a subtree of the source tree whose nodes are all left (right) inclined. Suppose that all the left (right) spines in T' have length at most c, for some constant $c \geq 8$, except for one spine longer than c that is incident to the root of T'. Then, with $O(\log |T'|)$ rotations, we can shorten the long spine by one edge, while keeping all other left (right) spines of T' no longer than c. No spine in $T_1 - T'$ becomes longer.*

Proof: We prove the lemma for left spines; the other case is symmetric. Lemma 2.1 implies that there are at least two possible rotation sites on the spine that do not involve the top node of the spine. (Because the top node is not involved, neither of the possible rotations lengthens the single spine that extends from T' up into the rest of the tree.) Each rotation takes two children of the spine and makes them into sibling grandchildren of the spine (Lemma 3.2). The two rotations can be chosen to involve two disjoint pairs of children. Let the two pairs be (A, B) and (C, D), each pair ordered left-to-right, and denote by AB and CD the two possible subtrees, children of the spine, produced by the two possible rotations.

We shorten the long spine of T' by performing the rotation that involves the smaller of the subtrees AB and CD: if $|AB| \leq |CD|$, the first rotation is chosen; otherwise the second rotation is chosen. Because AB and CD are disjoint, it follows that $|AB| \leq \frac{1}{2}|T'|$. The rotation lengthens exactly one left spine: the left spine of AB is one edge longer

than the left spine of A before the rotation. The length of the left spine of AB may increase to $c+1$, in which case we recursively apply the same algorithm to shorten it. Because $|AB| \leq \frac{1}{2}|T'|$, the number of rotations needed to restore the Inclination Invariant obeys the recurrence $t(m) \leq t(m/2)+1$, where m is the subtree size. This well-known recurrence solves to $t(m) = O(\log m)$, completing the proof. ∎

We subdivide the leaf-sequence of the source tree into blocks of leaves with the same inclination. The breakpoints between consecutive blocks are called *inclination reversals*. Each inclination reversal where the inclination changes from right to left is a merge center. Thus the total number of inclination reversals in any subsequence of the leaves containing k merge centers is at most $2k+1$.

Lemma 4.1 is not directly applicable to the general case of sibling operations, because we cannot limit invariant violations to subtrees with only a single inclination. To remedy this problem, we prove in the next section that we can always restrict the invariant violations to subtrees containing only a constant number of inclination reversals. Therefore, the following lemma lets us restore the invariant with only $O(\log n)$ rotations in the general case as well.

Lemma 4.2 *Let $T' \subseteq T_1$ be a subtree of the source tree in which the Inclination Invariant holds (with constant $c \geq 9$), except for one left (right) spine of length greater than c incident to the root of T'. Suppose that there are at most k inclination reversals among the leaves in the left (right) subtree of T'. Then, using $O(\sqrt{k}\log|T'|)$ valid rotations, we can shorten the long spine by one edge, and ensure that the Inclination Invariant holds for all other spines in T'. No spine in $T_1 - T'$ becomes longer.*

Proof: As in the proof of Lemma 4.1 above, the long spine has at least two disjoint rotation sites not involving the top node of the spine. Let the two pairs of subtrees involved in these rotations be (A,B) and (C,D), and denote by AB and CD the subtrees that result from the rotations. Let k_{AB} and k_{CD}, respectively, denote the number of inclination reversals among the leaves of the subtrees AB and CD.

We shorten the long spine of T' by performing the rotation that involves the subtree with the smaller number of inclination reversals; suppose that $k_{AB} \leq k_{CD}$, and so the first rotation is chosen. Because AB and CD are disjoint, it follows that $k_{AB} \leq k/2$. The rotation lengthens two spines: the left spine of A and the right spine of B are lengthened by one. This increase in length may result in a violation of the Inclination Invariant. We restore the invariant recursively on the left and right spines of AB separately, choosing rotations strictly below the root of AB. (Because $c \geq 9$, the spine length below the root of AB is at least nine, and two independent rotations are always available below the root.) Since the rotations are below the root of AB, the two recursive invariant restorations are independent. If either A or B is singly-inclined, then we apply Lemma 4.1 to restore the Inclination Invariant in that subtree.

The inclination reversals in AB are divided between the subtrees A and B, meaning that $k_A + k_B \leq k_{AB}$. (Equality holds unless an inclination reversal falls between A and B.) The number of rotations needed to restore the invariant obeys the following recurrence:

$$\begin{aligned} g(k,m) &\leq g(k_1,m_1)+g(k_2,m_2)+1, \\ &\qquad \text{with } k_1+k_2 \leq \lfloor k/2 \rfloor \text{ and } m_1+m_2 \leq m \\ g(0,m) &= \lfloor \log m \rfloor, \end{aligned}$$

where k is the number of inclination reversals in a subtree, and m is the subtree size. To remove the dependence on two variables, we replace the first inequality by

$$g(k, m) \leq g(k_1, m) + g(k_2, m) + 1.$$

The resulting system of inequalities is satisfied by

$$g(k, m) \leq \sqrt{k}\ \lfloor \log m \rfloor - 1,$$

as can easily be verified. This completes the proof. ∎

5 Choosing a Sibling Operation

Lemma 4.2 shows that correcting violations of the Inclination Invariant is relatively inexpensive, so long as the number of inclination reversals in the subtree below the violating spine is small. This section shows how to choose sibling operations that are not too destructive: the invariant-violating spines they induce have only $O(1)$ inclination reversals below them.

Let x be a merge center, and let v_x be the lowest common ancestor (LCA) of x in the source tree; a merge center corresponds to two consecutive leaves, and the LCA of these leaves is defined as the LCA of the merge center. Node v_x is both-inclined, since its subtree includes two leaves that are inclined toward each other. By the Inclination Invariant, the top of the spine that extends up from v_x is a constant number of edges away. Let this spine top, which is a strict ancestor of v_x unless the latter is the root, be called the *ancestral-spine top* of x, and denoted w_x.

At each execution of Step 2 of the main algorithm, we select a merge center at which to perform a sibling operation using the following criterion:

Merging Criterion:

1. Among all ancestral-spine tops defined by merge centers, choose a spine-top w that is the ancestor of no other ancestral-spine top.
2. Among the merge centers with ancestral-spine top w, choose one whose LCA v is the ancestor of no other LCA.

Algorithmically, one easy way to accomplish this selection is to sort all merge centers lexicographically on the key (*height*(w), *height*(v)) and select a minimal element from the sorted list.

This selection criterion is well-defined: The lowest common ancestor function is a one-to-one mapping between pairs of consecutive leaves along the bottom of the source tree and internal nodes of the tree. Thus part 2 of the Merging Criterion selects a single merge center, since there is only one merge center associated with each lowest common ancestor. Furthermore, since any internal node w (except the root) is the top of only one spine, all the LCAs with w as their spine top lie along one spine (or at most two spines). There is exactly one LCA (or at most two, if w is the root) that is the ancestor of no other—it is the lowest on its spine.

The following lemmas show that a merge center selected according to the Merging Criterion creates violations of the invariant that are easy to repair.

Lemma 5.1 *The rotations needed to process a sibling operation selected according to the Merging Criterion affect only the spine from the LCA v to its spine top w and the spines below v.*

Proof: The rotations needed to process the sibling operation involve no nodes above v (Lemmas 2.1 and 2.2). The only spine above v that may be affected is the spine from v to its spine top w. ▮

Lemma 5.2 *Let v be the LCA of a merge center selected according to the Merging Criterion. Then there is at most one inclination reversal in each of the left and right subtrees of v.*

Proof: There is only one merge center below v, namely the one with v as its LCA: if there were another merge center, then either the spine top of that merge center would lie below w, contradicting part 1 of the Merging Criterion, or the LCA would lie below v, contradicting part 2. A subtree containing no merge centers, such as the left or right subtree of v, can have at most one inclination reversal among its leaves. ▮

Lemma 5.3 *Let v be the LCA and w the ancestral-spine top of a merge center selected according to the Merging Criterion. Let z be the child of w that is an ancestor of v. (If v is the root, then let $w = z = v$.) Then there are $O(1)$ inclination reversals among the leaf descendants of z.*

Proof: Let us assume for convenience that w is not the root of the source tree; the proof is similar when w is the root. CLAIM 1: $O(1)$ merge centers have w as their spine top. Each such merge center must have its lowest common ancestor on the unique spine from v to w. This spine has constant length, by the Inclination Invariant. Since each LCA corresponds to a unique merge center, the claim follows. CLAIM 2: All merge centers in the subtree rooted at z have w as their spine top. The spine of such a merge center cannot end above w, because z and the parent of w are on the same side (left or right) of w. The spine of such a merge center cannot end below w, or else w would not have been selected by part 1 of the Merging Criterion. The two claims together prove the lemma. ▮

Lemma 5.4 *Suppose that a merge center is chosen by the Merging Criterion. If the Inclination Invariant holds before the corresponding sibling operation is performed, then the invariant can be restored after the sibling operation using only $O(\log n)$ rotations.*

Proof: A sibling operation performs k rotations in the subtree below the LCA v, for some $k = O(1)$, and affects only spines below the spine top w (Lemmas 3.1 and 5.1). These rotations add a total of at most $2k$ to the lengths of spines below v and extending up from v to w (Lemma 3.2).

We restore the Inclination Invariant from the bottom up. We repeatedly pick a spine that violates the invariant and has no violations below it. If the spine lies below v, then it has at most three inclination reversals below it, by Lemma 5.2. If the length of the spine is $c+j$, where c is the constant in the invariant, then the invariant can be restored on and below the spine with $O(j \log n)$ rotations, by Lemma 4.1. If the spine has w as

its top, then it has $O(1)$ inclination reversals below it, by Lemma 5.3. Its length can also be reduced from $c+j$ to c by $O(j \log n)$ rotations (Lemma 4.2). (The constant of proportionality is worse for the spine that reaches up to w, but the asymptotic rotation bound is the same as for the spines below v.) ∎

The preceding lemmas yield our main theorem on tree morphing.

Theorem 5.5 *Let T_1 and T_2 be two valid n-leaf binary trees with the same weight sequence at the leaves. Then there is a sequence of $O(n \log n)$ valid rotations that transforms T_1 into T_2.*

Proof: We establish the Inclination Invariant initially by normalizing the source tree, then perform $n-1$ sibling operations, each chosen according to the Merging Criterion. Each sibling operation requires a constant number of rotations (Lemmas 2.1, 2.2, and 3.1), but it may introduce violations of the invariant. After each sibling operation, we restore the invariant with an additional $O(\log n)$ rotations, using Lemma 5.4. ∎

6 A Tree-Morphing Algorithm

The previous section shows that only $O(n \log n)$ valid rotations are needed to transform T_1 to T_2, but does not give an algorithm to identify those rotations in less than quadratic time. In this section we show how to compute the sequence of rotations in $O(n \log n)$ time. To compute the rotations, we maintain a linear amount of auxiliary data at the nodes of the source tree and outside the tree. In all of these structures, we represent positions in the tree as indices into the original list of leaves. Even though the source and target trees change as leaves are merged into their parents, the leaf indices are computed as though the leaves were still present.

We maintain the following auxiliary data, identified below with the notation "(DSx)":

1. An ordered list of inclination reversals, represented as positions in the original list of leaves.
2. For each merge center x
   (a) Its lowest common ancestor v_x and its ancestral-spine top w_x.
   (b) A pointer to its position in (DS1), the list of inclination reversals.
3. For each node of the source tree
   (a) Indices of the first and last leaves below the node.
   (b) A list of the merge centers for which the node is a lowest common ancestor or an ancestral-spine top.
   (c) A boolean flag that is TRUE iff any descendant of this node is the ancestral-spine top of a merge center.
4. A set of merge centers that satisfy the Merging Criterion.

After the initial normalization of the source tree, all of these data can be computed in linear time by traversing the source and target trees.

We use these data to perform three functions critical to the algorithm: (1) identify merge centers that satisfy the Merging Criterion, (2) compute subtree sizes as required by Lemma 4.1, and (3) count inclination reversals in subtrees, as required by Lemma 4.2. (In fact, we do not compute the actual subtree sizes, but only an approximation that is sufficient to give an $O(\log n)$ performance bound in Lemma 4.1.) As the algorithm runs, we must maintain the data as (1) the source tree is modified by rotations, and (2) merge centers move or disappear because of sibling operations.

Lemma 6.1 *The data described above are sufficient to perform each of the following algorithm operations in constant time: (1) identifying merge centers that satisfy the Merging Criterion, (2) computing subtree sizes, and (3) counting inclination reversals.*

Proof: Identifying merge centers that satisfy the Merging Criterion is easy, since they are maintained as part of the data (DS4).

Lemma 4.1 requires us to compute subtree sizes, which our data structures do not record. However, we can get an asymptotic bound similar to that of Lemma 4.1 by using approximate sizes. When Lemma 4.1 requests the size of the subtree at a node v, we return one more than the difference of its first and last leaf indices, using the data (DS3a) stored at v. This counts the "original" size of the subtree, namely the size it would have if no leaves had collapsed into their parents. This quantity has the crucial property that the value at a node is at most the sum of the values at the node's children; its maximum value is n. The analysis in the proof of Lemma 4.1 carries through if we use original sizes instead of current sizes for the subtrees, and so we get an $O(\log n)$ bound for restoring the Inclination Invariant when there are no inclination reversals in the subtree.

Lemma 4.2 requires us to count inclination reversals in subtrees, but Lemma 5.3 ensures that there will be only $O(1)$ inclination reversals in the subtrees for which the queries are posed. Once a merge center has been chosen from (DS4), we locate it in the list (DS1) by following the pointer (DS2b). The inclination reversals that will be counted during the application of Lemma 4.2 are all within constant distance of this position in the list. We can answer the counting queries in constant time by examining $O(1)$ list elements in the vicinity of the merge center. ∎

Lemma 6.2 *The data described above can be maintained in $O(1)$ time per source tree rotation, and in $O(1)$ amortized time per movement or disappearance of a merge center because of a sibling operation.*

Proof: We consider rotations first. The list of inclination reversals (DS1) is unaffected by rotations. Likewise the pointers (DS2b) are unaffected. The leaf indices (DS3a) change only at the rotated nodes. A rotation affects a constant number of nodes in the tree; in Figure 2 these nodes are B, C, and their parents. Before the rotation, we identify all the merge centers whose lowest common ancestors or spine tops are at one of these nodes, using (DS3b). After the rotation, some of these LCAs and spine tops may have moved. We compute their new locations by visiting $O(1)$ nodes in the vicinity of the rotation (because the tree satisfies the Inclination Invariant, all spines

above LCAs have constant length). We change the affected node variables (DS3b) and likewise change the merge center variables pointing to those locations (DS2a). The flags (DS3c) may change within a constant-sized neighborhood of moved spine tops. Spine tops whose flag value is FALSE correspond to merge centers that satisfy the Merging Criterion. We update the set (DS4) with the merge centers corresponding to any spine tops whose flag value has changed.

When a merge center moves or disappears, there is no change to the structure of the source tree. The leaf indices (DS3a) do not change, and the LCAs and spine tops corresponding to unchanged merge centers likewise remain unchanged. We update the list of inclination reversals (DS1) at the position of the changed merge center. If the merge center moves, it is because two sibling leaves collapse into their parent, and the new merge center is defined by the parent and its sibling in the target tree. The position of the new merge center in the list (DS1) is the same as that of the one from which it is derived, so (DS1) and (DS2b) are trivial to update. Because of the Inclination Invariant, the LCA and spine top for the new merge center can be computed in constant time by walks on the source tree, and variables (DS2a) and (DS3b) are easily updated.

If the merge center simply moves, then flag values (DS3c) and the set (DS4) change only for spine tops in the vicinity of the old spine top. When the merge center disappears, because the parent of the two collapsing leaves has no sibling leaf in the target tree, the flag values may be affected non-locally. Flag values change from TRUE to FALSE on a path of arbitrary length above the disappearing merge center. We argue that the total number of flag changes is nevertheless $O(n+m)$, where m is the number of rotations or merge center movements. There are initially $O(n)$ flags set to TRUE. The total number of changes from FALSE to TRUE is $O(m)$, because each rotation or merge center movement affects only $O(1)$ flags. It follows that the number of changes from TRUE to FALSE is $O(n+m)$, which proves the claim. The amortized cost of updating flag values (DS3c) is $O(1)$. Updating the set (DS4) after a flag change takes at most $O(1)$ time per flag change. ∎

By combining Theorem 5.5 and Lemmas 6.1 and 6.2, we obtain our main theorem.

Theorem 6.3 *Let T_1 and T_2 be two valid n-leaf binary trees with the same weight sequence at the leaves. Then we can transform T_1 into T_2 using $O(n \log n)$ valid rotations. Furthermore, the sequence of rotations can be computed in $O(n \log n)$ time.*

7 Conclusion

In the few years since its introduction, morphing has become a hugely popular technique in computer graphics. Although its most prominent applications are still in the entertainment industry, there are many serious applications in industrial design [6], medical imaging [4], and data visualization. While morphing is useful in many applications, many of the techniques employed to morph images are not fully understood. In fact, most of the commercially available morphing software packages require a nontrivial amount of human interaction, especially in choosing correspondence points between the source and target images. Furthermore, the intermediate stages of a morphing transformation often are *illegal* images, with no corresponding valid three-dimensional picture.

Our investigation is motivated by the need to provide some theoretical underpinnings for morphing. The work of Guibas and Hershberger [3] is a first step in this direction: it formulates a rigorous model for morphing polygons using "parallel" moves and proposes an algorithm to compute the moves required to morph one polygon to another. In the present paper, we have focused on the *combinatorial* problem at the heart of the polygon-morphing algorithm, and have obtained a significantly better bound than was previously known.

Although the weight constraint on our trees originated in the polygon-morphing problem, it appears to be a rather weak condition, and we suspect our theorem on tree-morphing may have other applications outside the morphing context. The main open problem suggested by our paper is to determine the true complexity of tree morphing. The number of valid rotations needed to morph between two valid trees is $\Omega(n)$ and $O(n \log n)$; either a tighter lower bound or a better algorithm is needed.

References

[1] T. Beier and S. Neely. Feature-based image metamorphosis. *Proc. of ACM Conference on Computer Graphics*, pages 35–42, 1992.

[2] K. Culik and D. Wood. A note on some tree similarity measures. *Information Processing Letters* 15:39–42, 1982.

[3] L. Guibas and J. Hershberger. Morphing simple polygons. In *Proceedings of the 10th ACM Symposium on Computational Geometry*, pages 267–276, 1994.

[4] J. F. Hughes. Scheduled Fourier volume morphing. *Proc. of ACM Conference on Computer Graphics*, pages 43–46, 1992.

[5] J. R. Kent, W. E. Carlson, and R. E. Parent. Shape transformation for polyhedral objects. *Proc. of ACM Conference on Computer Graphics*, pages 47–54, 1992.

[6] T. Sederberg and E. Greenwood. A physically based approach to 2-D shape blending. *Proc. of ACM Conference on Computer Graphics*, pages 25–34, 1992.

[7] D. Sleator, R. E. Tarjan, and W. Thurston. Rotation distance, triangulations, and hyperbolic geometry. *J. AMS*, 1:647–682, 1988.