

Polytechnic
UNIVERSITY

Brooklyn · Long Island · Westchester

Simple and Optimal Output-Sensitive Computation of Contour Trees

Yi-Jen Chiang

Xiang Lu



Department of Computer and Information Science

Technical Report
TR-CIS-2003-02
06/20/2003

Simple and Optimal Output-Sensitive Computation of Contour Trees

Yi-Jen Chiang* Xiang Lu†

Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201 USA

Abstract

Isosurface extraction is one of the most powerful techniques in the investigation of volume datasets in scientific visualization. The *contour tree* is a fundamental data structure for fast isosurface extraction, and has also been used to build user interfaces to report the complete topological characterization of the isosurfaces embedded in the volume data, as well as to simplify the volume data to build a multi-resolution hierarchy while preserving the isosurface topologies.

In this paper, we present a new *output-sensitive* algorithm for computing the contour tree. Our algorithm is simple, and achieves the optimal bound of $\Theta(m + t \log t)$ in running time for both *structured-* and *unstructured-grid* volume datasets, where m is the number of cells of the input volume, and t is the number of *critical points* in the input volume, which bounds the size of the contour tree. Our algorithm improves the previous best running time of $O(m\alpha(n) + n \log n)$ given in [5] for unstructured grids (where n is the number of vertices of the input volume), and as the algorithm of [5], works in all dimensions as well. The experiments show that typically t is less than 5% of the overall number n of the input vertices, and that our algorithm is 2 to 3 times as fast as the previous best algorithm [5].

Keywords: Contour tree, output-sensitive algorithms, Morse theory, isosurface topologies, isosurface extraction, design and analysis of algorithms, algorithm experimentation, computer graphics and scientific visualization.

*yjc@poly.edu. Research supported in part by NSF CAREER Grant CCR-0093373, NSF Grant ACI-0118915, and NSF ITR Grant CCR-0081964.

†xlu@photon.poly.edu. Research supported by NSF Grant CCR-0093373.

1 Introduction

Isosurface extraction is one of the most effective and powerful techniques for the investigation of volume datasets. It has been used extensively in scientific visualization applications such as biology, medicine, chemistry, computational fluid dynamics, and so on. It has also found extensive applications in computer graphics such as simplification [10] and implicit modeling [16]. The widespread use of isosurface extraction has made it a focus of intensive research for many years.

Specifically, given tuples $(v, F(v))$ in the volume dataset, where v is a 3D sample point and F is a scalar function defined over 3D points, the isosurface at isovalue q is a set of points (a surface) in the volume whose scalar value is q . Some representative isosurfaces (generated from our experiments) are shown in Figure 4. We are mainly interested in *unstructured-grid* volume data represented as a tetrahedral mesh (i.e., a simplicial complex in 3D). This is the most general class of volumetric data and has been proposed as an effective means of representing disparate field data that arises in a broad spectrum of scientific applications including structural mechanics, computational fluid dynamics, partial differential equation solvers, shock physics, and so on. We remark that our results can be easily applied to *structured grids* (where the underlying cell topology of the volume is a cube or a box) as well, by first tetrahedralizing each cube/box into 5 or 6 tetrahedra as is done in [5, 9].

The *contour tree* is a fundamental data structure that represents the relations between connected components of the isosurfaces embedded in a volume dataset. Two connected components that merge together (as one continuously changes the isovalue) are represented as two edges that join at a node of the tree. This structure was used by van Kreveld *et. al.* [20] to speed up isosurface extraction in the *seed-cell propagation* paradigm: given an isovalue, find a *seed cell* at *each* connected component of the isosurface, and generate the isosurface by exploring the neighboring cells in the volume starting from the seed cells. (Note that a cell in the volume data is either a tetrahedron (a simplex in 3D) if the volume is an unstructured grid, or a cube/box if the volume is a structured grid). To save the space complexity, one wants to keep the set of seed cells small, while guaranteeing that for every connected component of every isosurface there is at least one seed cell in the set so that no component of the isosurface would be missed. Van Kreveld *et. al.* [20] proposed the use of the contour tree to compute such seed set.

The succinct encoding of the isosurface topologies in the contour tree also leads to other important applications. Bajaj *et. al.* [2] proposed the display of the contour tree to provide the user with insights into the topological structures of the isosurfaces embedded in the volume data. Recently, Pascucci and Cole-McLaughlin [15] gave an elegant algorithm of computing and associating the Betti numbers $(\beta_0, \beta_1, \beta_2)$ with the contour tree so that the isosurface topologies, including the number of connected components and the genus number, can be completely determined and displayed. Very recently, we developed a volume simplification technique that preserves all isosurface topologies [6], by making use of the contour tree.

Given the versatile and important applications of the contour tree, in this paper we focus on the efficient computation of the contour tree. Throughout the paper, we assume that the input volume has n vertices and m cells (equivalently, $O(m)$ edges), and t *critical points* (to be defined in Section 2), where the size of the contour tree is bounded by $O(t)$. Note that $n = O(m)$.

Previous Work

The first efficient algorithm for computing the contour tree in 2D was given by de Berg and van Kreveld [8], with running time $O(m \log m)$. Later, van Kreveld *et. al.* [20] developed algorithms for computing the contour tree in 2D with the same running time (but simpler) and in higher dimension in $O(m^2)$ time. They also showed how to compute a small seed-cell set for fast isosurface extraction using the contour tree [20]. Tarasov and Vyalys [19] improved the complexity of computing the contour tree in 3D to $O(m \log m)$ time. Carr *et. al.* [5] simplified and extended the method of Tarasov and Vyalys [19] so that the contour tree can be computed in any dimension in $O(m\alpha(n) + n \log n)$ time, where $\alpha(n)$ is a variant of the extremely slowly

growing inverse of the Ackermann’s function. This algorithm [5] is simple and elegant, and is the basis for proving the correctness of our new algorithm presented in this paper. We review the algorithm [5] in Section 3.

The first *output-sensitive* algorithm for computing the contour tree was given by Pascucci and Cole-McLaughlin [15], with running time $O(n + t \log n)$, for *structured grids*. Notice that for structured grids, $m = O(n)$ and thus m does not appear in the bound. Recall that t is the number of *critical points* in the volume data and is the upper bound on the size of the contour tree. As mentioned above, they also gave a very nice method to associate the Betti numbers to the contour tree to enable the computation of isosurface topologies.

The $O(n + t \log n)$ -time contour tree algorithm [15] is based on the following nice idea. At each step, the volume is recursively subdivided into two halves of roughly equal number of vertices, with the common boundary (the *separator*) having $O(n^{2/3})$ vertices (and same order of edges). This splitting step can be performed trivially on a *structured grid* in $O(1)$ time—just take a plane cutting through the median of the volume. In each of the half volume, the corresponding contour tree is built recursively, and then the two contour trees, one on each side, are merged to form the final contour tree. The process is similar to merge sort: when the sub-volume has only $O(1)$ vertices, computing the contour tree takes $O(1)$ time. The trick is in the merge step: the two contour trees from the two sub-volumes can possibly interact with each other only at the $O(n^{2/3})$ vertices on the separator; for the rest of the two contour trees we just need to copy them, which takes $O(t)$ time at each recursive level. Therefore, the merging step, rather than taking $O(n)$ time as in the merge sort, takes $O(n^{2/3}\alpha(n) + t)$ time (by merging the separator vertices from both sides into a combined sorted list and then performing the sweeping algorithm of Carr *et. al.* [5] with union-find operations [7], plus copying the unchanged portions of the two contour trees). Since the merge step (as well as the splitting step) takes less than linear time in n , the overall complexity is $O(n + t \log n)$.

In [15] the authors also mentioned that their method might be extended to *unstructured grids*, by pointing out the $O(n)$ -time algorithms [12, 13] for finding a separator with $O(n^{2/3})$ vertices in unstructured grids. However, even with these separator algorithms, the output-sensitive bound no longer holds: The volume splitting step, originally taking $O(1)$ time in each recursion for structured grids, now takes $O(n)$ time, which is the same as the merge-step complexity in merge sort—the overall complexity for all splitting steps is $O(n \log n)$ already, and thus it does not pay off to use the more complicated separator algorithms [12, 13], compared to the simple sorting step used in the algorithm of Carr *et. al.* [5].

Our Results

In this paper, we present a new *output-sensitive* algorithm for computing the contour tree. Our algorithm is simple, and achieves the optimal bound of $\Theta(m + t \log t)$ in running time, for both *structured* and *unstructured* grids. Since there is a lower bound of $\Omega(m + t \log t)$ by the lower bound construction of Bajaj *et al.* [3] (see Section 2), our algorithm is *optimal*. Our algorithm improves the previous best running time of $O(m\alpha(n) + n \log n)$ given by Carr *et. al.* [5] for unstructured grids, and as the algorithm of [5], works in all dimensions as well.

The key idea of our algorithm is to avoid sorting all vertices, but rather identify all critical points first, sort them, and then connect them using a clever sweeping. An interesting aspect of our sweeping is that we do not need to explore the entire set of the vertices, but overall only explore a very small portion of the vertex set, via so called *monotone paths*. Moreover, we traverse the monotone paths along the direction that is *opposite* to the sweeping direction.

We have implemented our algorithm and performed experiments on datasets from real-world scientific visualization applications. The experiments show that typically t is less than 5% of the overall number n of the input vertices, and that our algorithm is 2 to 3 times as fast as the previous best algorithm [5]. We refer to Section 5 for the details.

We remark that concurrent to our work, Lenz and Rote [11] independently developed an output-sensitive

algorithm for computing contour trees, achieving the same optimal bound of $\Theta(m + t \log t)$ as ours. Their technique is also based on the similar idea of exploring *monotone paths* and is essentially equivalent to ours, with different algorithm details and proof methods.

2 Preliminaries

Our results extend to any dimensions as the results of Carr *et. al.* [5]. However, motivated by real-world applications we will mainly focus our discussions on the case of 3D.

A *scalar volume dataset* consists of tuples $(v, F(v))$, where v is a 3D sample point and F is a scalar function defined over 3D points. Our main interest is on *unstructured-grid* volume data represented as a tetrahedral mesh (i.e., a simplicial complex in 3D) since this is the most general class of volume data and has found extensive applications. For a simplicial complex, we naturally choose F to be a *piecewise-linear* function whose value at a sample point v (which is a vertex of the mesh) is the scalar data value $F(v)$, and whose value at a point within a simplex is obtained by linear interpolation from the scalar values of the simplex vertices. Moreover, we assume that the scalar values at vertices are distinct (achieved by symbolic perturbation). These are the same assumptions used in Carr *et. al.* [5].

Given an isovalue q , the *isosurface* $C(q)$ at value q is defined as the set $C(q) = \{p | F(p) = q\}$. In a more general term, it is called the *level set* at q , where a level set in a 2D mesh is an *isoline* and in a 3D mesh is called an *isosurface*. In general, one can consider level sets in a higher dimensional mesh. We will use the term *isosurface* throughout the paper (for our focus on the 3D case), though the concepts apply to the more general *level set*.

The field of Morse theory [4,14,18] studies the changes in topology of isosurface as the isovalue changes. One important notion is that of *critical points*, which are the points where the isosurface topology changes (i.e., an isosurface connected component appears/disappears, changes genus, splits to more components, or components merge) as we continuously change the isovalue. Morse theory requires that the critical points be isolated, i.e., that they occur at distinct points and values. A function satisfying this condition is called a *Morse function*. All points other than critical points are called *regular points* and do not affect the number or genus of components of the isosurface. Note that our assumptions on the scalar function F that (1) it is a linear interpolant over a simplicial complex, and (2) the scalar values at vertices are distinct, ensure that F is a Morse function, and that the critical points occur at *vertices* of the mesh [4].

There are three types of critical points: minimum, maximum, and saddle points. A vertex with scalar value *smaller* (resp., *larger*) than those of all its neighboring vertices is called a *minimum* (resp. *maximum*), and corresponds to the *appearing* resp., *disappearing*) of an isosurface connected component if we change the isovalue continuously from $-\infty$ to ∞ . A *saddle point* is the remaining type of critical point, and corresponds to an isosurface connected component split/merge or a change of genus number.

The *contour tree* of a Morse function is a graph in which (1) each leaf node represents a minimum or a maximum critical point, at which an isosurface connected component appears or disappears, (2) each interior node represents the joining and/or splitting of two or more isosurface connected components at a critical point, which is necessarily a saddle point, and (3) each edge represents a connected component in the isosurfaces at all isovalues between the scalar values of the two endpoint nodes of the edge. This graph is shown to be a tree [20], hence called a *contour tree*. In addition, the nodes in the contour tree are naturally sorted by the scalar values in increasing order from bottom to top [20] (see Fig. 1(a) for an example).

Notice that all nodes in the contour tree are critical points, so the size of the contour tree is bounded by $O(t)$, where t is the number of critical point. However, *not all* critical points are necessarily present in the contour tree—those saddle points that correspond *only to the genus change* of an isosurface connected component are missing from the tree, and thus such genus-change events can *not* be captured by the contour tree. If we augment an ordinary contour tree with such genus-change-only critical points (so that the tree contains all critical points), again with all nodes in the tree sorted by the scalar values in increasing order

from bottom to top, the resulting contour tree is called an *augmented contour tree* (see Fig. 1(b) for an example). In such tree, an edge representing an isosurface connected component is further subdivided into *segments* by the genus-change-only saddle points in that component (see Fig. 1(b)); each *segment* now defines a *topological-equivalence region*, meaning that between the scalar values of the two endpoints of each segment, there is no isosurface topology change. In this way, the augmented contour tree captures all isosurface topology changes. Finally, if a contour tree is augmented with all vertices of the mesh, including all critical points and regular points, we call it a *fully augmented contour tree*¹ (see Fig. 1(c) for an example).

It is worth noting that the nodes in the contour tree are sorted along each monotone path in the tree. Bajaj *et al.* [3] have used this property to prove a lower bound of $\Omega(t \log t)$ on the worst-case time to construct the contour tree by reduction from sorting. For any given set of t numbers, they showed how to construct a bivariate piecewise linear function on a triangulated domain of size $O(t)$, so that the contour tree contains a long monotone path from which the sorted sequence of t numbers can be read off. Together with the $\Omega(m + n) = \Omega(m)$ time for reading the input, this gives a lower bound of $\Omega(m + t \log t)$ on the running time for computing the contour tree.

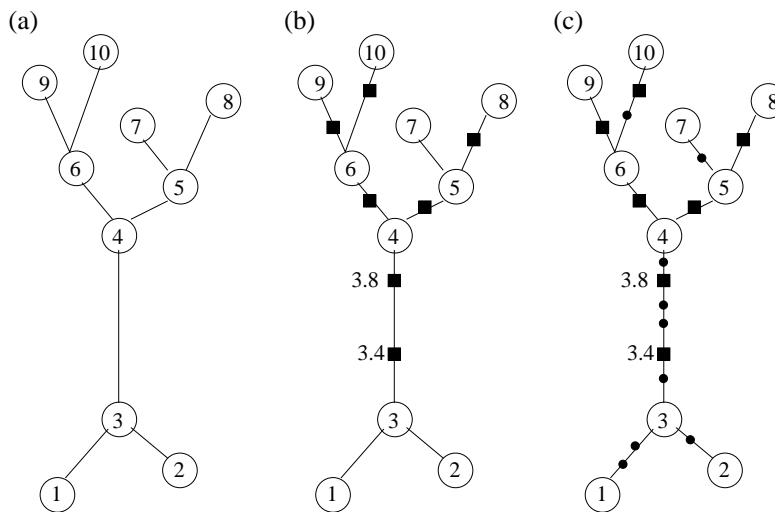


Figure 1: Schematic examples of an ordinary contour tree, an *augmented contour tree*, and a *fully augmented contour tree* (the label of a node denotes its scalar value): (a) ordinary contour tree; (b) augmented contour tree; (c) fully augmented contour tree. In (b), the square nodes are genus-change-only saddle points. Each *segment* defines a *topological-equivalence region*, where an endpoint of a segment can either be a square node or a non-filled circle node. The edge (3, 4) is divided into three segments (3, 3.4), (3.4, 3.8) and (3.8, 4). In (c), the square nodes are genus-change-only saddle points as in (b), and the filled circle nodes are vertices of the mesh that are regular points.

3 Review: Sweep Algorithm for Contour Trees

In this section, we review the algorithm of Carr *et al.* [5] for computing the *fully augmented contour tree*, i.e., the contour tree with all vertices of the mesh present in the tree; see Section 2. (The ordinary contour tree can be obtained by removing nodes from the fully augmented contour tree.) This algorithm will be used in Section 4 to prove the correctness of our new algorithm.

¹In [5], the term *augmented contour tree* means our *fully augmented contour tree* here, and there is no concept of a contour tree containing all critical points but no regular points, i.e., there is no concept of our *augmented contour tree* here.

The algorithm of Carr *et. al.* [5] consists of the following stages.

1. Sort all n vertices of the mesh by the scalar values.
2. Perform a sweep of the n vertices from the smallest scalar value to the largest scalar value, and build the *join tree* (defined below).
3. Perform another sweep of the n vertices, now from the largest scalar value to the smallest scalar value, and build the *split tree* (defined below).
4. Merge the join tree and split tree together to obtain the fully augmented contour tree. Perform an optional step of removing nodes to obtain the ordinary contour tree if needed.

The formal definitions of *join tree* and *split tree* are given in [5], but the concept is best understood by a sweeping process (see Fig. 2). Given a contour tree, if we sweep the contour tree nodes from bottom to top, and ignore the splitting events, namely, components can only merge (i.e., *join*) but never split, then we get the *join tree*. Observe that the leaves correspond to the vertices of local minimum and to the creation of isosurface components, and that eventually all components merge into one component, for which the highest node, namely the root, corresponds to a vertex of local maximum (see Fig. 2(b)). Similarly, if we sweep the contour tree nodes from top to bottom and only allow components to merge, then we obtain the *split tree* (see Fig. 2(c)). Note that if we fix the sweeping direction to be always from bottom to top, then the split tree always splits and never merge.

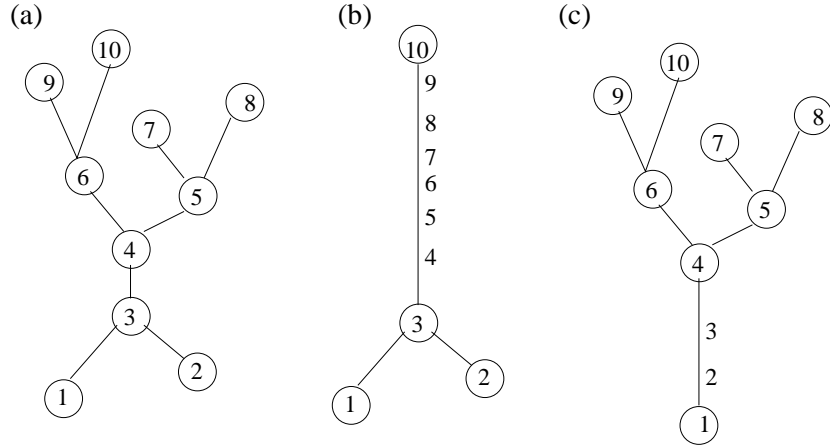


Figure 2: An example of a contour tree and its corresponding join tree and split tree: (a) contour tree; (b) join tree; (c) split tree.

It is shown in [5] that Stage 4 can be performed in time linear in the total size of the join and split trees. Stage 2 and Stage 3 are completely symmetric, and are essentially the same process performed in the opposite sweeping directions. To understand our new algorithm, it suffices to describe Stage 2, the process of computing the join tree.

Now we describe the algorithm of [5] for computing the join tree. During the sweep process, the union-find data structure [7] is used to maintain the connectivity information for the isosurface connected components. Initially, each vertex is added to its own singleton set. As we sweep from the vertex of the smallest scalar value to the vertex of the largest scalar value (viewed as changing the isovalue of the isosurface

continuously), the vertices corresponding to the same connected component of the isosurface are unioned together into the same set, which corresponds to an edge in the join tree.

For the current vertex v during the sweep, we look at its neighboring vertices (neighboring in the mesh) with scalar value less than v . If there is no such neighbor, then v is a minimum critical point corresponding to the appearing of a new isosurface connected component—we create a new set containing v and also a new leaf of the join tree. If v has some neighbors with smaller scalar values, then each such neighbor has been processed (since the sweep is in the order of increasing scalar values) and put to the correct set. We perform the find-set operation on each of such neighbors. If these neighbors are all in the same set, i.e., in the same connected component, then we just add v to this set, effectively adding v to this connected component. If some of these neighbors belong to different sets, then it means that these different connected components, each from a distinct set, now merge together at v to become a single connected component— v is a saddle point for components merge. We perform union operations on these sets to union them together, as well as putting v to this newly unioned set, and grow the join tree accordingly.

It is easy to see that the entire sweeping process involves $O(n)$ union operations and $O(m)$ find operations (there are $O(m)$ edges), so it takes $O(m\alpha(m, n))$ time for sweeping. Including the sorting time in Stage 1, the overall time is $O(m\alpha(m, n) + n \log n)$.

4 Output-Sensitive Algorithm for Contour Trees

In this section we present our new output-sensitive algorithm for computing contour trees. Our main algorithm computes the *augmented contour tree*, i.e., the contour tree augmented with all genus-change-only critical points (and hence the contour tree containing all critical points; see Section 2). Note that augmented contour tree *captures all* events of isosurface topological changes, including *genus changes*. If we only need the ordinary contour tree, then it can be obtained by removing nodes from the augmented contour tree.

Main Algorithm

Now we present our main contour tree algorithm, which computes the *augmented contour tree*, i.e., the contour tree containing all critical points (and hence capturing all events of isosurface topological changes). Recall from Section 3 that the algorithm of Carr *et. al.* [5] cannot distinguish between a regular point and a genus-change-only critical point. In fact, the criticality of a vertex is not known until the entire contour tree is built, and thus we do not know which vertices are more important than the others.

The key idea of our algorithm is to avoid sorting all vertices, but rather identify all critical points first, sort them, and then connect them using a clever sweeping. An interesting aspect of our sweeping is that we do not need to explore the entire set of the vertices, but overall only explore a very small portion of the vertex set, via so called *monotone paths*. Moreover, we traverse the monotone paths along the direction that is *opposite* to the sweeping direction.

Our algorithm consists of the following stages.

1. Scan through the n vertices, classify each vertex into either a critical point (local minimum, local maximum, or saddle point) or a regular point. In the process, also collect all critical points. Suppose there are t critical points.
2. Sort the t critical points by their scalar values.
3. Perform a sweep of the t critical points from the smallest scalar value to the largest, and build the join tree.
4. Perform another sweep of the t critical points, now from the largest scalar value to the smallest, and build the split tree.

5. Merge the join tree and the split tree to obtain the contour tree.

Again, Stage 3 and Stage 4 are completely symmetric, and can be carried out by the same procedure with the opposite sweeping directions. Also, Stage 5 can be performed by the tree-merging method of [5]. Therefore we only need to describe Stage 1 and Stage 3.

We first describe Stage 1: classification of the vertices into critical or regular points. We can classify a vertex v for its criticality by locally checking its neighboring vertices [1,9], summarized as follows. We take all the tetrahedral cells sharing v as one of their cell vertices. For each such cell, there is one triangle without using v as a vertex (i.e., the triangle facing v). We take all such triangles, whose vertices are exactly the neighbors of v and whose edges connect these neighbors together. These triangles then form a graph G with nodes and edges being the vertices and the edges of the triangles. For each node p in G , we classify p as “+” if its scalar value is larger than the scalar value of v , and “-” otherwise. Recall that all the scalar values at vertices are distinct. Now in the graph G , we remove edges connecting two nodes of opposite signs (a “+” and a “-”) and obtain a new graph G' . Now, we compute the connected components in G' via depth-first search or bread-first search, and count how many connected components there are. Note that the nodes in the same connected component all have the same sign. If there are exactly two components (necessarily one “+” and one “-”) then v is a regular point. Otherwise v is a critical point, with the following cases: one component—a minimum (for a “+” component) or a maximum (for a “-” component), more than two components—a saddle point. This process works for higher dimensional simplicial meshes too, with an obvious analogy from a 3D simplex to higher dimensional simplex.

Clearly, the above process takes time proportional to the number of neighbors of v , which can be $O(n)$, and thus overall it can be $O(n^2)$ time to classify all vertices. However, taking a closer look, we see that the number of neighbors of v is its vertex degree in the original mesh, so the overall work is bounded by the total number of edges in the mesh, i.e., $O(m)$. We remark that usually a tetrahedral mesh is given in the form of a vertex list and a cell list, where each cell has indices to its four vertices in the vertex list. By scanning through the cell list and putting each cell to its four vertices in the vertex list, we obtain the vertex-neighbor information as needed, in $O(m)$ time. In the process, we also record the neighbors of v with the largest and the smallest scalar values, among these neighbors. This information will be used in the sweeping process of Stages 3 and 4.

Lemma 1 *We can classify all n vertices of a simplicial mesh in any dimensions into regular points or minimum, maximum, or saddle critical points, in a total of $O(m)$ time, where m is the number of cells in the mesh.*

Now we describe Stage 3, the construction of the join tree. Recall that we sweep the critical points in the order of increasing scalar values. During the sweeping process, we also use the union-find data structure [7] to maintain the isosurface connected components information similar to the algorithm of [5].

First, we define the concept of a *monotone path*. A *monotone path* in the mesh is a directed path consisting of vertices and edges of the mesh such that traversing along the path the scalar values of the vertices on the path is monotonically decreasing or increasing; we call such a path *monotone decreasing path* if the values decrease and *monotone increasing path* if the values increase.

We are ready to describe our sweep algorithm for constructing the join tree in Stage 3. For a current critical point c , we distinguish the following cases.

1. c is a minimum critical point. Put c into a new set, and create a new leaf in the join tree.
2. c is a saddle point.

Consider the neighbors of c that have scalar values *smaller* than c and that are *not* visited before. (Such neighbors must be *regular points*, since any critical point with a smaller scalar value than c has

been processed and visited.) For each such neighbor v , we want to put v to the correct set “below v ”, that is, the correct set containing some critical points that have been processed before (equivalently, the critical points with scalar values *smaller* than v).

The task is carried out as follows. For each such v , explore a *monotone decreasing path* starting from v , i.e., starting from v , continuously visit a neighbor of the current vertex with a smaller scalar value than the current vertex (recall that in Stage 1 we record for each vertex its largest and smallest scalar-value neighbors), until a vertex x visited before is reached. By Lemma 2 below, there always exists a monotone decreasing path from v before reaching a vertex x visited before. Put v and all vertices on the monotone decreasing path to the set $S(x)$ of x , effectively putting these vertices to the same connected component of $S(x)$.

When all such neighbors v of c are put to some existing sets, look at *all* neighbors of c with scalar values smaller than c , including those visited before. These neighbors have all been assigned to some sets. Perform the find operation to see these sets. If they are all the same set, then c is a “genus-change-only saddle point for joint tree”, meaning that c does not cause connected-component number to change in the join tree (but it may cause connected-component number to change in the split tree)—put c to this set and continue to grow the join tree. Otherwise, when there are some different sets, then c causes components to merge—union these sets into one set, add c to the set, and join these components at c in the join tree.

3. c is a maximum point. This case is similar to that of a saddle point, but we only need to go from *one* neighbor v of c (since all other neighbors lead to the same set), and proceed as before.

Lemma 2 *When we explore from v , there is always a monotone decreasing path from v before we stop exploring.*

Proof. The only case that we cannot find a neighbor with smaller scalar value is when the current vertex is a minimum critical point, which must have been visited before. ■

Lemma 3 *The algorithm correctly computes the join tree.*

Proof. We want to show that every vertex visited by the algorithm is put to the correct set as in the original sweep algorithm [5] described in Section 3. Then the correctness of our algorithm follows from the correctness of the original algorithm [5].

We prove the above statement by induction on the number of critical points processed. The base case is trivially true, as we process the first minimum critical point and put it to a singleton set. For the induction step, consider the current critical point c being processed. Our goal is to show that every neighbor v of c with a smaller scalar value is put to the correct set (so that c correctly joins different connected components and/or is put to the correct set), and that all vertices in each monotone decreasing path from such an unvisited v are put to the correct set as well. Note that if such v has been visited before, then v was visited when processing some critical point c' prior to processing c (possibly $v = c'$), and hence v has been put to the correct set by induction hypothesis. Now it remains to show the following claim.

Claim 4 *Every neighbor v of c that has a smaller scalar value than c and that is not visited before is put to the correct set. Moreover, all vertices in the monotone decreasing path from v to x are put to the correct set as well, where x is the first encountered vertex visited before when exploring the monotone decreasing path from v .*

Proof of Claim. Recall that v must be a regular point, since a critical point with a smaller scalar value than c must have been visited before. Consider the correct join tree T_1 (with all vertices present) produced by the original algorithm [5] as described in Section 3, at the snap shot of the sweeping at c . Then v has been correctly put into some set, say S_1 . Consider the segment of T_1 containing v , and the highest critical point a on this segment that is below v . (Here a segment in the join tree contains only regular vertices in the interior, plus two critical endpoints, in the join tree.) Then a is the highest-valued critical point in the set S_1 containing v , and there is a path P_1 in T_1 linking a to v through *regular vertices only* (see Fig. 3(a)).

Now, suppose our algorithm produces a join tree T_2 , and at the point of processing v as described in our algorithm, v is put to a set S_2 . This means that as we explore the monotone decreasing path P (in the *mesh*) from v to x , we stop at the previously visited vertex x and put all the vertices on P , including v , to the set containing x , which is S_2 . Let x' be the highest-valued critical point in S_2 with scalar value smaller than v ; in our constructed join tree T_2 , v and x' are on the same segment, and x' is the highest critical point on this segment that is below v . We want to show that $a = x'$ (and hence $S_1 = S_2$). Assume for the purpose of contradiction that this is not true. Since there is a monotone decreasing path P (in the *mesh*) from v to x , as we sweep from x' to v using the original sweeping algorithm [5], the vertices on P with scalar values larger than x' will be swept, in the reverse order of P , leading all such vertices as well as v to be included into S_2 (note that x and x' are in the same set S_2 , and hence there is a path in the *mesh* connecting x' and x). Therefore, in the correct join tree T_1 , there is a path P_2 from x' to v (see Fig. 3(b)). This means that using the original sweeping algorithm [5], the two distinct sets S_1 and S_2 in the correct join tree T_1 will become a single set at the time v is reached. Let v' be the point at which S_1 and S_2 merge (possibly $v' = v$, or otherwise v' has a smaller scalar value than v). Note that the scalar value of v' must be larger than *both* a and x' , or otherwise S_1 would be part of S_2 or vice versa and the two sets would not be distinct. Observe that v' is a *saddle point* merging S_1 and S_2 in T_1 (see Fig. 3(b)). However, v' is on P_1 with scalar value larger than a , meaning that v' can only be a *regular point* (this is true for $v' = v$ too—recall that v is a regular point), a contradiction. Therefore $a = x'$ and $S_1 = S_2$. Since the vertices in the monotone decreasing path P are put to $S_2 = S_1$, they are put to the correct set as well. \square

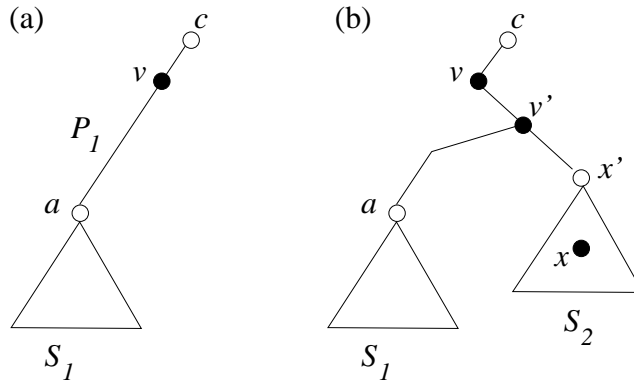


Figure 3: Proof of Claim 4. (a) Correct join tree T_1 . The path P_1 in T_1 links a to v through *regular vertices only*. (b) Correct join tree T_1 , assuming that $a \neq x'$ (and hence $S_1 \neq S_2$). There is a path P_2 in T_1 from x' to v .

The proof of Claim 4 completes the induction, and the lemma follows. \blacksquare

Additional Algorithm Details and Analysis

Now we analyze the running time of our algorithm. Suppose there are n' vertices visited, where $n' = O(n)$. If all these n' vertices participate in the union-find data structure, then we need to perform n' union operations. However, we can do better than that. Observe that a monotone decreasing path from v to x (excluding x) contains regular points only; these regular points cannot cause any components merge and hence do not actually need the support of the union operation—we only need to *assign* them to the destination set. Therefore, we make the following simple improvement of the algorithm: Only the *critical points* participate in the union-find data structure, i.e., we start the union-find structure with each critical point in a singleton set. When we assign a regular point w to a set, we put a pointer from w to some point p in the set, where p is necessarily a critical point. When we explore a monotone decreasing path P from v to x , where x is the first encountered vertex visited before, we have that x is either a critical point or a regular point with a pointer to some critical point p . Then, for each vertex w on P , we either put a pointer from w to x if x is a critical point, or otherwise we put a pointer from w to p . To perform a find operation on a regular point, we just follow the pointer to the critical point in $O(1)$ time, and then perform a find operation on that critical point. In this way, the total number of union operations is t .

It is easy to see that exploring the monotone decreasing paths overall takes $O(n')$ time ($O(n') = O(n) = O(m)$), but in practice n' is very small compared to n ; see Section 5). Each such path P stops at some vertex x visited before, so a vertex might be visited more than one time. However, each P contains at least one *unvisited* vertex v , and hence the cost of looking at the stopping point x can be charged to v .

To analyze the number of find operations performed, observe that for each critical point c , we try to find the set of each neighbor v of c with a smaller scalar value than c . Clearly, we perform e_c find operations, where e_c is the total number of edges in the mesh with at least one critical endpoint (note that $e_c \geq t$ but $e_c = O(m)$); typically e_c is much smaller than the total number of edges, and also much smaller than the total number of cells, m ; see Section 5). Overall, there are t union operations and e_c find operations, and hence the total time for the union-find operations is $O(e_c \cdot \alpha(e_c, t))$, where $\alpha(e_c, t)$ is the extremely slowly growing inverse of the Ackermann’s function. Summing over the time for all stages, the overall time complexity is $O(m + t \log t + e_c \cdot \alpha(e_c, t))$.

We can make the above bound optimal ($O(m + t \log t)$) by using the same analysis technique of Lenz and Rote [11] as follows: For $e_c > t \log \log t$, we have $\alpha(e_c, t) \leq 1$, and $e_c \cdot \alpha(e_c, t) = O(e_c) = O(m)$. For $e_c \leq t \log \log t$, we have $e_c \cdot \alpha(e_c, t) \leq t \log \log t \cdot \alpha(e_c, t) \leq t \log \log t \cdot \alpha(t, t) = O(t \log t)$.

Theorem 5 *For a mesh in any dimension with m cells and t critical points, there exists an algorithm that computes the contour tree in optimal $\Theta(m + t \log t)$ time.*

5 Experimental Results

We have implemented both our new algorithm presented in Section 4 and the sweep algorithm [5] described in Section 3 in C++/C, and ran our experiments on a Sun Blade 1000 workstation with 750MHz UltraSPARC III CPU and 4GB of main memory. The datasets we used for the experiments, as listed in Table 1, are all from real-world scientific visualization applications: The Blunt Fin (blunt, blunt2), the Liquid Oxygen Post (post, post2), and the Delta Wing (delta, delta2) datasets are from NASA, and the Combustion Chamber (comb, comb2) datasets are from Vtk [17] (generated from a combustion simulation). These datasets are all given as tetrahedral volume data, and each pair of datasets (e.g., blunt and blunt2) are at different sampling rates and hence their input sizes are different. Some representative isosurfaces generated from our experiments are given in Figures 4.

For each of the datasets, we ran our algorithm and the sweep algorithm [5] to build the contour tree. Detailed statistics of the experiments are given in Table 1. It is very interesting to see that for all the datasets tested, the number t of critical points is typically fairly small, ranging from 0.152% to 4.44% of the overall

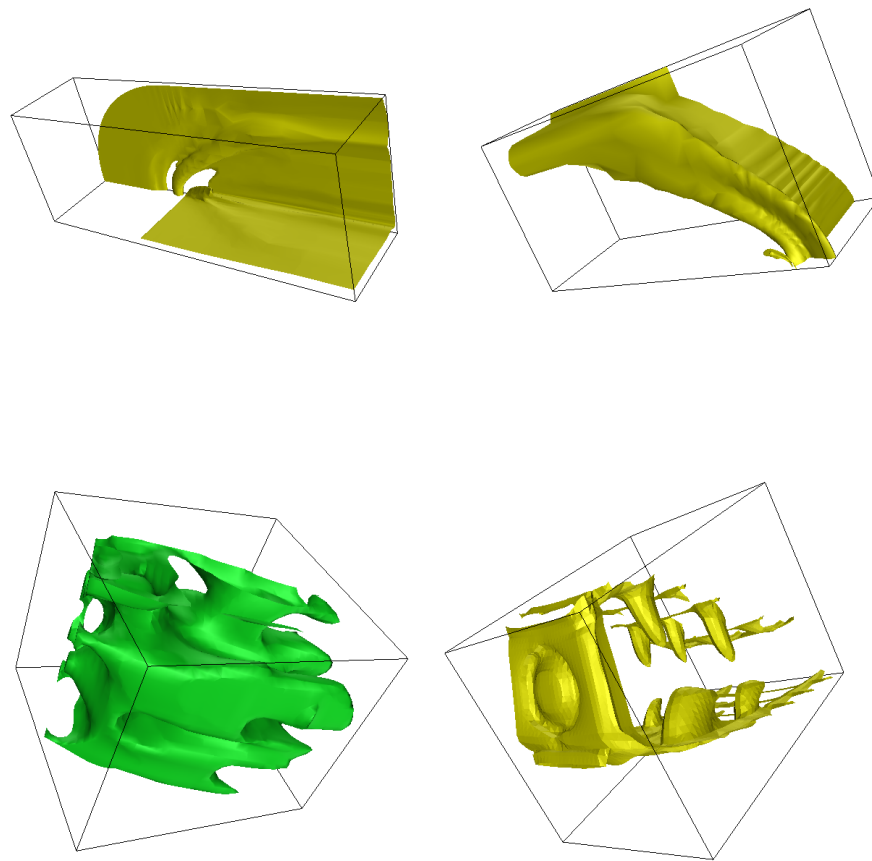


Figure 4: Typical isosurfaces. The upper two are from the Blunt Fin dataset. The ones in the bottom are from the Combustion Chamber dataset.

Dataset	# Cells	# Vert.	# Crit. Pts (%)	e_c (%)
blunt	187395	40960	1820 (4.44%)	20072 (10.7%)
comb	215040	47025	524 (1.11%)	5791 (2.7%)
post	513375	109744	927 (0.84%)	10349 (2.0%)
delta	1005675	211680	1462 (0.70%)	17307 (1.7%)
blunt2	749580	228355	1833 (0.80%)	58159 (7.8%)
comb2	860160	262065	533 (0.20%)	13615 (1.6%)
post2	2053500	623119	947 (0.152%)	24241 (1.2%)
delta2	4022700	1217355	2042 (0.17%)	52913 (1.3%)
Dataset	n' -Join (%)	n' -Split (%)	New (s)	Sweep (s)
blunt	9958 (24.3%)	8345 (20.4%)	2.04	4.66
comb	4953 (10.5%)	3257 (6.9%)	2.34	2.95
post	12163 (11.1%)	4544 (4.1%)	5.58	9.24
delta	13669 (6.5%)	10157 (4.8%)	10.94	24.36
blunt2	9894 (4.3%)	8310 (3.6%)	8.40	25.35
comb2	4802 (1.8%)	3180 (1.2%)	9.49	13.53
post2	11927 (1.9%)	3906 (0.6%)	22.70	44.77
delta2	16875 (1.4%)	12028 (1.0%)	43.94	146.72

Table 1: Experimental results. For each dataset, we list the number of cells, the number of vertices, the number of critical points (and the percentage), the number e_c of edges in the mesh with at least one critical endpoint (and the percentage of the ratio over the number of cells), the number n' of vertices (and the percentage) visited by our algorithm in computing the join tree, the number n' of vertices (and the percentage) visited by our algorithm in computing the split tree, the total running time (in seconds) of our algorithm in computing the join tree and the split tree, and finally the total running time (in seconds) of the sweep algorithm [5] in computing the join tree and the split tree. The total running time does not include the time for reading the input file from disk.

number n of the input vertices. This certainly shows the importance of having an *output-sensitive* contour tree algorithm, where the size of the output contour tree is $O(t)$ rather than $O(n)$. We also show the number e_c of the edges with at least one critical endpoint, which is less than 3% of the number m of cells for all but two datasets tested. Recall that our union-find operations takes time $O(e_c \cdot \alpha(e_c, t))$; with $\alpha(e_c, t)$ no more than 4 in practice, we see that $e_c \cdot \alpha(e_c, t)$ is much smaller than m , meaning that the time for performing union-find operations is dominated by $O(m)$ in practice.

We also show in Table 1 the number n' of the vertices visited by our algorithm during the construction of the join tree and the split tree. Recall from Section 4 that our algorithm does *not* explore all the input vertices, but rather only explores a monotone path, to connect a neighboring vertex of the current critical point to some existing set. Here we see that n' is indeed very small, ranging from as low as 0.6% to 24.3% of n , showing that our algorithm only explores a very small portion of the input. Recall that exploring all monotone paths takes $O(n')$ time in total; with both n' and t much smaller than n in practice, our running time is essentially the $O(m)$ time for processing the input in the initial stage.

Finally, we compare the overall running time of our algorithm with the sweep algorithm of [5] in computing the join tree and the split tree. These running times include those for all the steps in computing the contour tree, except for the time for reading the input from disk and the time for merging the join tree and the split tree (since these times are the same for both algorithms). The advantage of our algorithm is clear,

and typically our algorithm is about 2 to 3 times as fast. As the dataset size increases, the savings in running time by our algorithm also increases; for the largest dataset in Table 1, we reduce the running time from 146.72 seconds to 43.94 seconds.

6 Conclusions

We have presented a new output-sensitive algorithm for computing the contour tree, which is simple and achieves the optimal bound of $\Theta(m + t \log t)$ in running time, for both *structured* and *unstructured* grids. Our algorithm improves the bound of $O(m\alpha(n) + n \log n)$ given by the previous best algorithm [5] for the unstructured grids represented as simplicial complexes, and, as that of [5], works in all dimensions as well. Our experiments show that typically our algorithm is 2 to 3 times as fast as the previous best algorithm [5], and effectively achieves the running time of $O(m)$ in practice.

Acknowledgements

We thank Boris Aronov for useful discussions. The Blunt Fin, the Liquid Oxygen Post, and the Delta Wing datasets are courtesy of NASA. The Combustion Chamber dataset is from Vtk [17]. Work on this paper has been supported by NSF CAREER Grant CCR-0093373. Research of Yi-Jen Chiang has also been supported in part by NSF Grant ACI-0118915 and NSF ITR Grant CCR-0081964.

References

- [1] B. Aronov. Personal communications, 2002.
- [2] C. Bajaj, V. Pascucci, and D. Schikore. The contour spectrum. In *Proc. IEEE Visualization*, pages 167–175, 1997.
- [3] C. Bajaj, M. van Kreveld, R. van Oostrum, V. Pascucci, and D. R. Schikore. Contour trees and small seed sets for isosurface traversal. Technical Report UU-CS-1998-25, Department of Computer Science, Utrecht University, 1998.
- [4] T. F. Banchoff. Critical points and curvature for embedded polyhedra. *J. Diff. Geom.*, 1:245–256, 1967.
- [5] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. In *Proc. ACM-SIAM Sympos. Discrete Algorithms*, pages 918–926, 2000.
- [6] Y.-J. Chiang and X. Lu. Progressive simplification of tetrahedral meshes preserving all isosurface topologies. *Computer Graphics Forum (Special Issue for Eurographics '03)*, 22(3), 2003 (to appear).
- [7] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, 2nd edition, 2001.
- [8] M. de Berg and M. van Kreveld. Trekking in the Alps without freezing or getting tired. *Algorithmica*, 18:306–323, 1997.
- [9] T. Gerstner and R. Pajarola. Topology preserving and controlled topology simplifying multiresolution isosurface extraction. In *Proc. IEEE Visualization*, pages 259–266, 2000.
- [10] T. He, L. Hong, A. Varshney, and S. Wang. Controlled topology simplification. *IEEE Transactions on Visualization and Computer Graphics*, 2(2):171–184, June 1996.
- [11] T. Lenz and G. Rote. Efficient and simple construction of contour trees using monotone paths, Manuscript, 2003.

- [12] G.L. Miller, S.-H. Teng, W. Thurston, and S.A. Vavasis. Automatic mesh partitioning. *Graph Theory and Sparse Matrix Computation*, (56):57–84, 1993.
- [13] G.L. Miller, S.-H. Teng, W. Thurston, and S.A. Vavasis. Geometric separators for finite element meshes. *SIAM J. Sci. Comput.*, 19(2), 1995. Submitted.
- [14] J. W. Milnor. *Morse Theory*. Princeton University Press, Princeton, NJ, 1963.
- [15] V. Pascucci and K. Cole-McLaughlin. Efficient computation of the topology of level sets. In *Proc. IEEE Visualization*, pages 187–194, 2002.
- [16] W. Schroeder, W. Lorensen, and C. Linthicum. Implicit modeling of swept surfaces and volumes. In *IEEE Visualization '94*, pages 40–45, October 1994.
- [17] W. Schroeder, K. Martin, and W. Lorensen. *The Visualization Toolkit*. Prentice-Hall, 1996.
- [18] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien. Surface coding based on Morse theory. *IEEE Comput. Graph. Appl.*, 11:66–78, September 1991.
- [19] S. Tarasov and M. Vyalyi. Construction of contour trees in 3D in $O(n \log n)$ steps. In *Proc. ACM Sympos. Comput. Geom.*, pages 68–75, 1998.
- [20] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. ACM Sympos. Comput. Geom.*, pages 212–220, 1997.