Computational Geometry 42 (2009) 606–616

Contents lists available at ScienceDirect

Computational Geometry: Theory and Applications

www.elsevier.com/locate/comgeo

# Efficient algorithms for computing Reeb graphs

Harish Doraiswamy [a,*], Vijay Natarajan [b]

[a] Department of Computer Science and Automation, Indian Institute of Science, Bangalore 560012, India
[b] Department of Computer Science and Automation, Supercomputer Education and Research Centre, Indian Institute of Science, Bangalore 560012, India

ARTICLE INFO

*Article history:*
Received 4 January 2008
Received in revised form 24 July 2008
Accepted 4 December 2008
Available online 13 December 2008
Communicated by K. Fukuda

*Keywords:*
Computational topology
Algorithms
Dynamic graph
Level set
Manifold
Piecewise-linear function
Reeb graph

ABSTRACT

The Reeb graph tracks topology changes in level sets of a scalar function and finds applications in scientific visualization and geometric modeling. We describe an algorithm that constructs the Reeb graph of a Morse function defined on a 3-manifold. Our algorithm maintains connected components of the two dimensional levels sets as a dynamic graph and constructs the Reeb graph in $O(n \log n + n \log g (\log \log g)^3)$ time, where $n$ is the number of triangles in the tetrahedral mesh representing the 3-manifold and $g$ is the maximum genus over all level sets of the function. We extend this algorithm to construct Reeb graphs of $d$-manifolds in $O(n \log n (\log \log n)^3)$ time, where $n$ is the number of triangles in the simplicial complex that represents the $d$-manifold. Our result is a significant improvement over the previously known $O(n^2)$ algorithm. Finally, we present experimental results of our implementation and demonstrate that our algorithm for 3-manifolds performs efficiently in practice.

© 2008 Elsevier B.V. All rights reserved.

## 1. Introduction

The Reeb graph of a scalar function describes the connectivity of its level sets. Abstraction of the topology of the level sets in this graph enables the development of simple and efficient methods for modeling objects and visualizing scientific data. Reeb graphs and their loop-free version, called contour trees, have a wide variety of applications including computer aided geometric design [21,28], topology-based shape matching [14], topological simplification and cleaning [12,27], surface segmentation and parametrization [13,29], and efficient computation of level sets [25]. They serve as an effective user interface for selecting meaningful level sets [2,6] and transfer functions for volume rendering [26].

### 1.1. Related work

Several algorithms have been proposed for constructing Reeb graphs. However, only a few produce provably correct Reeb graphs: Shinagawa and Kunii proposed the first algorithm for constructing the Reeb graph of a scalar function defined on a triangulated 2-manifold [20]. Their algorithm explicitly tracked connected components of the level sets and has a running time of $O(n^2)$, where $n$ is the number of triangles in the triangulation. Cole-Mclaughlin et al. [7] improved the running time to $O(n \log n)$ by maintaining the level sets using dynamically balanced search trees. In a recent paper, Pascucci et al. [18] proposed an online algorithm that constructs the Reeb graph for streaming data. Their algorithm takes advantage of the coherency in the input to construct the Reeb graph efficiently. Though it performs well in practice, their algorithm has a worst case time complexity of $O(n^2)$. Other algorithms follow a sample based approach that produces potentially inaccurate results [14,24]. For the special case of loop-free Reeb graphs, Carr et al. [5] described an elegant $O(n \log n)$ algorithm that

* Corresponding author.
*E-mail addresses:* harishd@csa.iisc.ernet.in (H. Doraiswamy), vijayn@csa.iisc.ernet.in (V. Natarajan).

0925-7721/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.comgeo.2008.12.003

works in all dimensions. Besides the naïve $O(n^2)$ algorithm and the online algorithm, there is no known algorithm for computing Reeb graphs of 3-manifolds. Here, $n$ is the number of triangles in the tetrahedral mesh. The presence of loops in the Reeb graph implies that its decomposition into a join and split tree, which was crucial for the efficiency of the algorithm by Carr et al. [5], may not exist. Efficient storage and manipulation of connected components of level sets will result in fast construction of Reeb graphs. Cole-Mclaughlin et al. [7] adopt this approach to obtain an efficient algorithm for 2-manifolds. However they also exploit the unique property of one-dimensional level sets that their vertices can be ordered, and therefore, their algorithm does not extend to 3-manifolds.

### 1.2. Results

We utilize an efficient tree-cotree [10] decomposition-based representation of level sets to construct the Reeb graph in $O(ng\log n)$ time, where $n$ is the number of triangles in the tetrahedral mesh representation of the 3-manifold, and $g$ is the maximum genus over all level sets of the function. Efficient representation of the tree-cotree partition results in an improved $O(n\log n + n\log g(\log\log g)^3)$ time algorithm. We also extend our approach to construct Reeb graphs of $d$-manifolds in $O(n\log n(\log\log n)^3)$ time. Experimental results of our implementation of the tree-cotree based algorithm indicates that the algorithm is also efficient in practice.

### 1.3. Outline

The rest of the paper is organized as follows. Section 2 introduces the necessary definitions and describes the structure and behavior of level sets of a Morse function defined on a 3-manifold. Section 3 describes our algorithm to construct Reeb graphs of 3-manifolds and Section 4 presents experimental results of our implementation. Section 5 describes an extension of our algorithm to compute Reeb graphs for $d$-manifolds. Section 6 concludes the paper.

## 2. Background

Let $\mathbb{M}^d$ denote a $d$-manifold with or without boundary. A smooth, real-valued function $\{f : \mathbb{M}^d \to \mathbb{R}\}$ is called a *Morse function* if it satisfies the following conditions [7]:

(1) all critical points of $f$ are non-degenerate and lie in the interior of $\mathbb{M}^d$,
(2) all critical points of the restriction of $f$ to the boundary of $\mathbb{M}^d$ are non-degenerate, and
(3) for all pairs $(p, q)$ of distinct critical points of $f$ and its restriction to the boundary, $f(p) \neq f(q)$.

Critical points of a smooth function are exactly where the gradient becomes zero. In the following discussion, we assume that the given function is Morse. The above conditions may not hold in practice. However, a simulated perturbation ensures no two critical points share a common function value and multiple-saddles can be unfolded into simple saddles to ensure all critical points are non-degenerate [9].

The preimage of a real value is called a *level set*. The level set of a Morse function $f$ is a $(d-1)$-manifold with or without boundary, possibly containing multiple connected components. For the case when $d = 3$, a level set is called an *isosurface*. We are interested in the evolution of the isosurface as the function value increases. Significant changes occur at critical points. Specifically, the topology of the isosurface changes either by gaining/losing a component or by gaining/losing genus. No topological changes occur at regular points. Fig. 1 illustrates the various topology changes that occur at critical points. The isosurface gains a component when it evolves past a minimum and loses a component when it evolves past a maximum. At 2-saddles, the local pictures in Fig. 1 indicate an apparent splitting of a component into two. Global behavior of the isosurface component will determine if this is indeed a split or a reduction in genus.

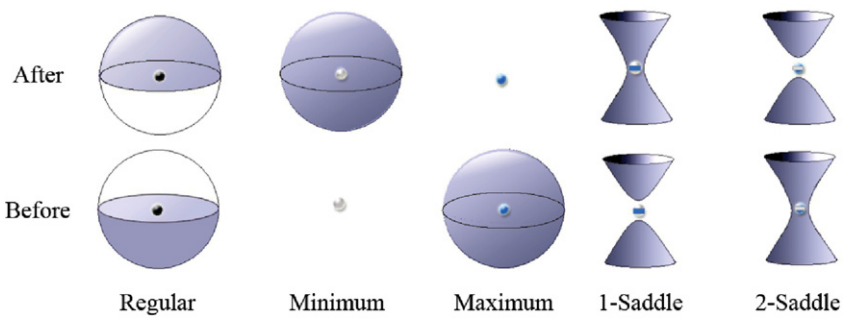



**Fig. 1.** Isosurfaces before and after passing through a point with function value $c$ ($f^{-1}(c-\epsilon)$ and $f^{-1}(c+\epsilon)$, respectively). Topology of the isosurface changes when it evolves past a critical point.

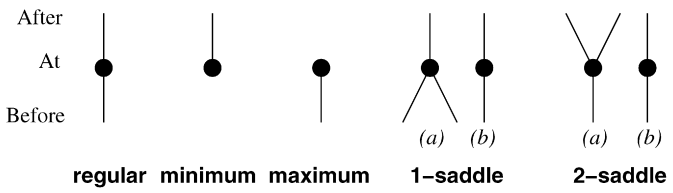



**Fig. 2.** Structure of Reeb graphs at regular and critical nodes.

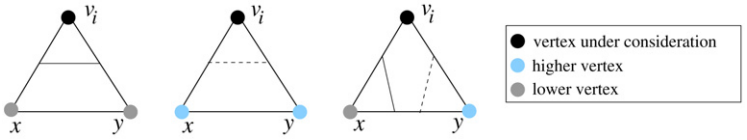



**Fig. 3.** Edges in the isosurface before (solid line) and after (dashed line) processing a vertex $v_i$.

The *Reeb graph* of $f$ is obtained by contracting each connected component of a level set to a point [19]. The Reeb graph expresses the evolution of connected components of level sets as a graph whose nodes correspond to critical points of the function. Fig. 2 illustrates the structure of the Reeb graph at various types of nodes. In the case of saddles, the corresponding node has degree 3 if the saddle merges/splits components, and degree 2 if it is a genus modifying saddle.

## 3. Reeb graphs of 3-manifolds

We now describe an algorithm to construct the Reeb graph of a Morse function defined on a piecewise-linear 3-manifold. We assume that the 3-manifold is represented by a tetrahedral mesh; the Morse function is specified as a sample at vertices of the mesh and linearly extended within each edge, triangle, and tetrahedron. We store the input using the triangle-edge data structure [16].

### 3.1. The sweep algorithm

The algorithm essentially follows from the definition of Reeb graphs. We track the evolution of isosurface components during a sweep of the 3-manifold parametrized by the function value. An isosurface of $f$ is a piecewise-linear surface that can be extracted as a triangle mesh. Vertices, edges, and triangles that constitute an isosurface lie within edges, triangles, and tetrahedra of the 3-manifold. Topology of the isosurface changes only when the sweep passes through critical points of $f$, which are restricted to vertices of the tetrahedral mesh [3,9].

Our algorithm proceeds by processing a sequence of events during the sweep. An event is triggered when the isosurface passes through a vertex. First, we sort the vertices on increasing function value and populate the event list. Processing the event includes updating the representation of the isosurface, its connected components and the Reeb graph. The algorithm maintains the isosurface at isovalue infinitesimally above the function value of the processed vertex. The Reeb graph is constructed incrementally based on the number of components of the isosurface.

End points of a single edge in the isosurface lie within two adjacent edges of a triangle in the tetrahedral mesh. Fig. 3 shows edges in the isosurface before and after processing a vertex event. The isosurface is updated locally depending on the relative function values at adjacent vertices of the triangle. Fig. 4 outlines the entire algorithm.

### 3.2. Dynamic maintenance of isosurfaces

A *map* $M$ is an embedding of a graph on a 2-manifold such that the two-dimensional cells of the embedding are disks. The *dual map* $M^*$ onto the same 2-manifold is constructed by creating a dual vertex $t^*$ within each face $t$ of the primal map $M$, and creating a dual edge $e^*$ for each primal edge $e$. If $e$ lies on the boundary of two faces $t_1$ and $t_2$, then $e^*$ connects $t_1^*$ and $t_2^*$ by a path that crosses $e$ exactly once and crosses no other primal or dual edge. The triangle mesh that represents an isosurface of $f$ is a map whose two-dimensional cells are triangles. Planar graphs can be embedded on the sphere, a 2-manifold whose genus equals zero. Non-planar graphs cannot be embedded on the sphere, but they can be embedded on a higher genus 2-manifold.

Let $T$ denote a spanning tree of $M$. If $C^*$ is a spanning tree of the dual map $M^*$, we call $C = \{e | e^* \in C^*\}$ a *spanning cotree* of $M$. Given a map $M$ with distinct edge weights, the minimum weight spanning tree and maximum weight spanning cotree of $M$ are disjoint [10]. Here, the weight of a dual edge is the same as that of the corresponding primal edge. If $M$ is a planar graph, the minimum spanning tree and maximum spanning cotree partition the edges in the graph [11].

A *tree-cotree partition* of $M$ is a triple $(T, C, X)$, where $T$ is the minimum spanning tree of $M$, $C$ is the maximum spanning cotree of $M$, and $X$ is the set of edges in $M$ that are neither in the tree $T$ nor in the cotree $C$. In the case of isosurfaces, since the edges of the mesh are unweighted, we can use any edge disjoint spanning tree and spanning cotree and maintain

**Algorithm** SweepComputeReebGraph
*Input :* Tetrahedral mesh $K$, Piecewise-linear function $f$

(1) Sort vertices of $K$ in increasing order of function value. Let $v_1, v_2, \ldots v_k$ be the sorted list of vertices.

(2) Initialize Reeb graph $R = \emptyset$

(3) **for** each $i = 1$ to $k$ **do**
  (a) **for** each triangle $(v_i, x, y)$ incident on $v_i$ **do**
    (i) **if** $f(v_i) > f(x), f(y)$
      (A) remove edge $((v_i, x), (v_i, y))$ from the isosurface
    (ii) **else if** $f(v_i) < f(x), f(y)$
      (A) insert edge $((v_i, x), (v_i, y))$ into the isosurface
    (iii) **else if** $f(x) < f(v_i) < f(y)$
      (A) remove edge $((v_i, x), (x, y))$ from the isosurface
      (B) insert edge $((v_i, y), (x, y))$ into the isosurface
    (iv) **else if** $f(y) < f(v_i) < f(x)$
      (A) remove edge $((v_i, y), (x, y))$ from the isosurface
      (B) insert edge $((v_i, x), (x, y))$ into the isosurface
  (b) Update $R$ to reflect change in the number of isosurface components.

(4) Return $R$

**Fig. 4.** Sweep algorithm that constructs the Reeb graph.

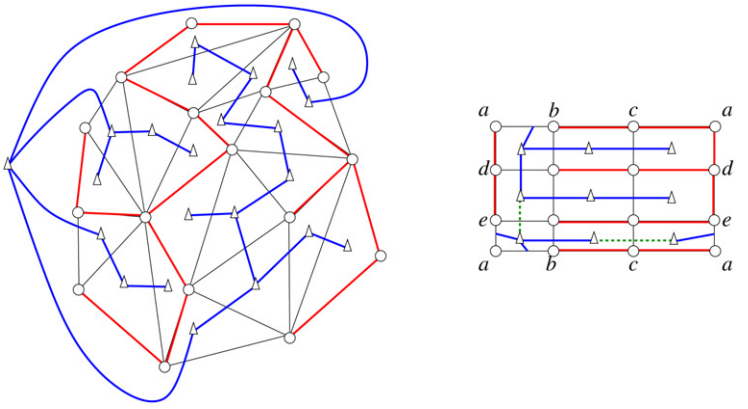



**Fig. 5.** A tree-cotree partition of a graph embedding on a sphere (left) and a torus (right). Tree edges are red, cotree edges are blue, and edges from $X$ are dotted and green in color. The 2-manifolds are "cut open" to make it easier to draw the graph embedding. The surface is obtained by gluing along the boundary. In the case of the sphere, the tree and cotree partition the edges of the graph. So, the set $X$ is empty for the sphere whereas it contains two edges for the torus. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

the updated tree-cotree partition during the sweep process. Fig. 5 shows a tree-cotree partition for a sphere and a torus. The cardinality of $X$, $|X|$, is equal to twice the *genus* of the 2-manifold. This follows from the fact that the Euler characteristic, $\chi = 2 - 2g$, of the 2-manifold can be expressed as the alternating sum of cells of $M$. If $\#v, \#e$, and $\#t$ denote the number of vertices, edges, and faces of $M$, then

$$\begin{aligned}
\chi &= 2 - 2g = \#v - \#e + \#t \\
&= \#v_T - \left(\#e_T + \#e_C + |X|\right) + \#v_C \\
&= \#v_T - \left(\#v_T - 1 + \#v_C - 1 + |X|\right) + \#v_C \\
&= 2 - |X| \\
\Rightarrow |X| &= 2g.
\end{aligned}$$

We store each isosurface component as a tree-cotree partition, as shown in Fig. 6. The set $X$ is stored using a simple list data structure. To store the tree $T$ and cotree $C$ individually, we use a dynamic tree data structure [22] as modified in [11], known as the *edge-ordered tree*. The edge-ordered tree imposes a total order on edges incident on a tree/cotree node $v$, referred to as the *edge list* of $v$. Each node $v$ is represented by a collection of *subnodes*, called a *node path*. A subnode $v_e$ in

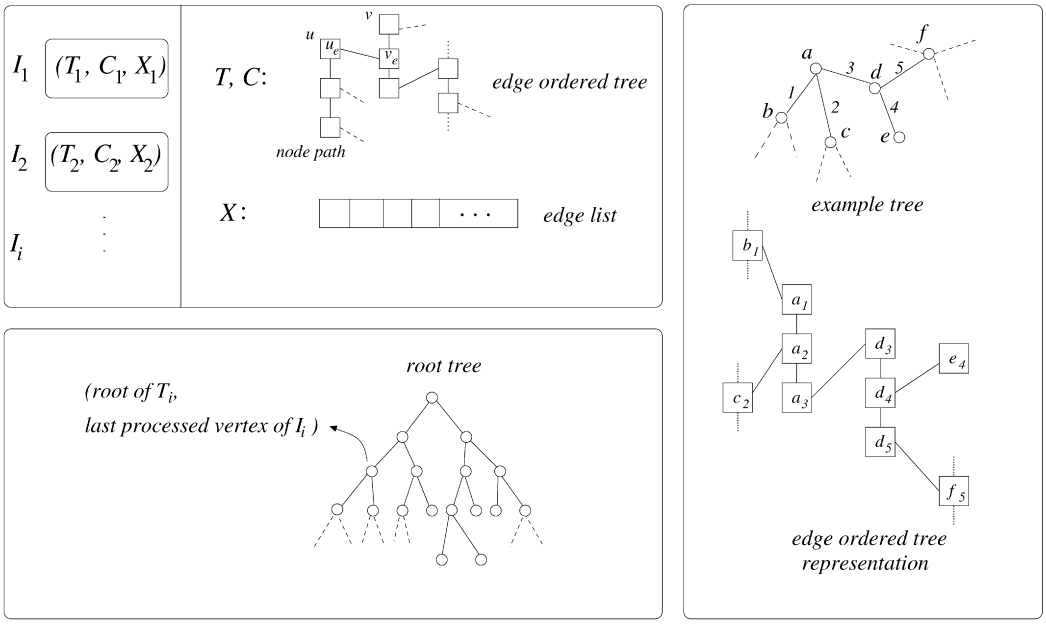



**Fig. 6.** Data structures used by the algorithm.

the node path of $v$ represents an edge $e$ in the edge list of $v$. Subnode $v_e$ is connected to the subnode of the predecessor and successor of $e$ in the edge list of $v$. Subnode $v_e$ is connected to subnode $u_e$ if the edge $e$ connects $u$ and $v$. The edge-ordered tree supports *InsertEdge* and *DeleteEdge* operations, both requiring $O(\log n_v)$ amortized time per operation, where $n_v$ is the number of nodes.

The ordering of edges around an isosurface vertex in this embedding is the same as the ordering of triangles around the corresponding edge in the input tetrahedral mesh. Given an edge to be inserted, its location with respect to the existing isosurface edges is determined by the corresponding triangle's position in the input mesh. The ordered ring of mesh triangles around a mesh edge is obtained directly from the triangle-edge data structure. For efficient determination of the isosurface edge location, we store the ordered set of mesh triangles around each isosurface vertex in a balanced search tree [8].

The *InsertEdge* and *DeleteEdge* operations are invoked to maintain the isosurface during the sweep algorithm as follows: To insert an edge $e$, check if the endpoints of the edge are in the same isosurface component. If not, connect the spanning trees of the two isosurface components using this edge, resulting in a spanning tree for the merged component. This also results in merging the two spanning cotrees: insertion of the edge merges two faces. So corresponding nodes in the cotree are merged. This is illustrated in Fig. 7(g), where an isolated node is connected with the spanning tree of an existing isosurface component.

If the end points of the edge do belong to the same isosurface component, then try inserting the edge into the cotree $C$. This is possible only if the pair of edges preceding $e$ in the edge lists of the endpoint vertices of $e$ share a common face. Else, the inserted edge will intersect with an edge in the tree $T$ or the cotree $C$. This insertion is illustrated in Figs. 7(h) and 7(i). Finally, if $e$ cannot be added to the cotree $C$, then it is added to $X$.

To delete an edge from the isosurface, we delete it from either the tree $T$, the cotree $C$, or the set $X$, as necessary. If the edge lies in the tree $T$, then the following two situations can occur:

(1) Deleting an edge merges two distinct faces of the isosurface into one: This causes a cycle in the cotree $C$. To handle this, remove any dual edge $e^*$ in the cotree $C$, from the path connecting the dual nodes corresponding to the two faces, and add the primal edge $e$ to the tree $T$. Fig. 7(c) shows the result of this operation after the dashed tree edge in Fig. 7(b) is removed.
(2) The edge is incident on a single face: In this case, removing the edge splits the corresponding isosurface component into two. This also results in the split of the dual node in the cotree corresponding to the split component. An example of such a removal is seen in Fig. 7(d)–(e).

If the edge lies in the cotree $C$, then removing this edge is equivalent to contracting the corresponding dual edge (Fig. 7(a)–(b)). If the edge was deleted from either the tree $T$ or the cotree $C$, then the genus of the surface may have decreased and hence an edge from $X$ can be inserted into the tree $T$ or the cotree $C$ without introducing a cycle. We exhaustively search the set $X$ to locate such an edge and move it to the tree $T$ or cotree $C$ as appropriate.

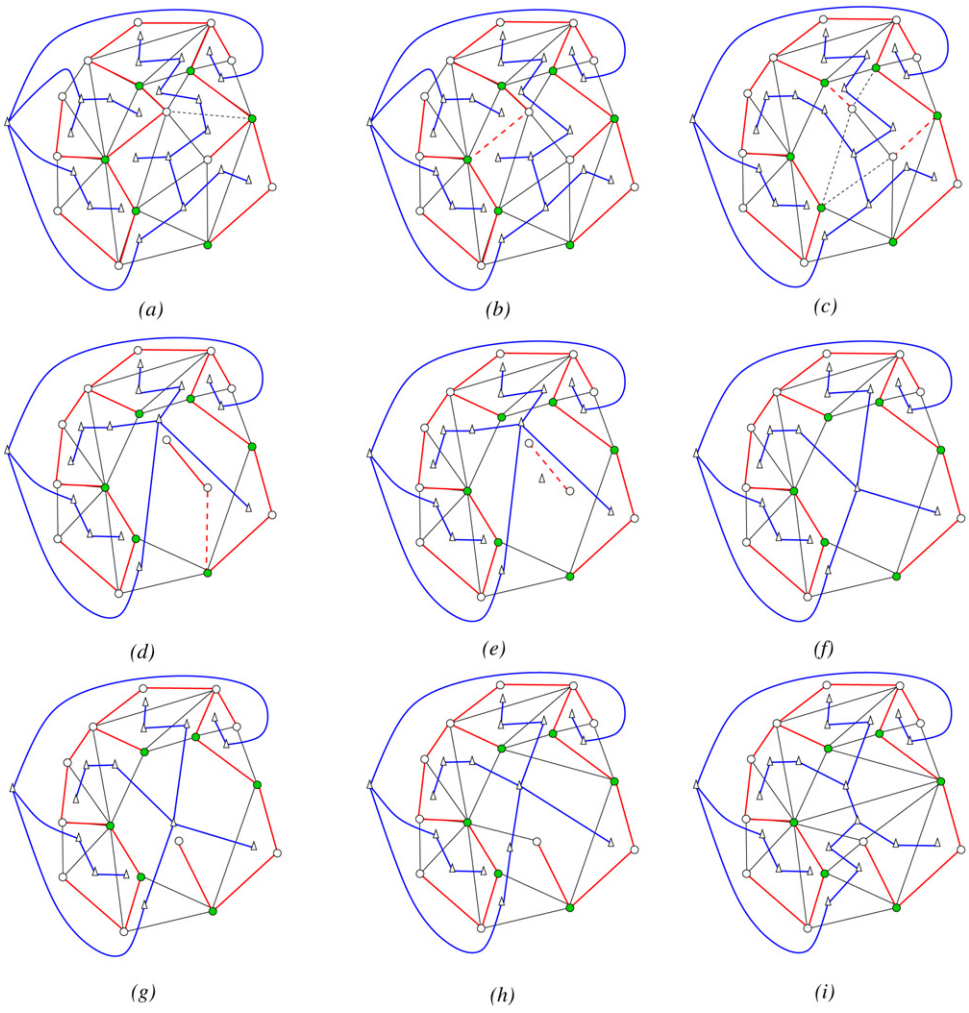



**Fig. 7.** Illustration of the update operations on a tree-cotree partition when a regular vertex is processed. The 1-skeleton of the isosurface does not change outside the ring of green nodes. The dashed edges are scheduled to be removed in the next step. (a) The initial tree-cotree partition. (b) After deleting a non-tree edge. (c) After deleting a tree edge. A cycle is created in the cotree $C$ when the dual nodes are merged. So, an edge from the path connecting the two dual nodes is transferred into the tree $T$. The dashed edges will be removed one after another, where each deletion will be of the type described in (a) or (b). (d) A tree edge will be removed next, resulting in a split in the component. (e) The dashed edge along with its dual node forms the tree-cotree partition for the newly created component. (f) Deleting the lone edge destroys the isosurface component. (g) Addition of an edge to the isosurface, thereby merging two components. The new node is considered as an individual component before addition of the edge. (h) Addition of a non-tree edge to the isosurface, which results in the modification of the cotree. (i) The tree-cotree partition after the regular vertex is processed. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

### 3.3. Dynamic maintenance of the Reeb graph

Each connected component of the isosurface is represented by the root of its tree $T$. Two nodes lie within the same component if their roots are equal. For fast access to the individual components, we store all roots in a balanced search tree, called the *root tree*. As we process a vertex $v_i$ from the tetrahedral mesh, we perform a set of edge insertions and/or deletions to the tree-cotree data structure. If a new component is created during this operation, then $v_i$ is a minimum. If an existing component is destroyed, then $v_i$ is a maximum. If either two components merge into one or a single component splits into two, then $v_i$ is a saddle. We compare the connectivity of end points of inserted/deleted edges before and after processing $v_i$ to identify the criticality of node $v_i$ and the components that were modified. We add a new node to the root tree if $v_i$ is a minimum or a saddle that splits a component. If $v_i$ is a maximum or a saddle that merges two components, an isosurface component is destroyed and we delete the corresponding node from the root tree. The sweep algorithm processes a vertex by modifying one or more isosurface components. We associate with each node in the root tree, the last processed vertex, $v_{last}$, that caused a modification of the corresponding isosurface component.

The Reeb graph is constructed incrementally by inserting a node after processing $v_i$. Assuming that edges in the Reeb graph are directed from a node with lower function value to a node with higher function value, each node can have at

most two predecessors. So, the Reeb graph can be stored using an adjacency list representation where each node has at most two adjacent nodes, namely the predecessors. Similarly, each node in the Reeb graph can have at most two successors. A Reeb graph node whose successors have been inserted is called a *stationary node*, else it is called a *growing node*. A node when inserted into the Reeb graph attaches to a growing node after which it becomes a growing node, unless it is a local maximum. The predecessor growing node becomes stationary if all of its successors have been inserted into the graph.

To insert a node into the Reeb graph, we first identify its predecessor from the list of growing nodes as the one representing the updated isosurface component. This is accomplished by querying the root tree for the updated component and obtaining the associated vertex. We then associate $v_i$ with this component in the root tree. If $v_i$ is a saddle that merges two components, then the corresponding node will have two predecessors, each of which can be identified by looking up the two modified components in the root tree. The vertex $v_i$ is then associated with the merged component. If the vertex $v_i$ is a component splitting saddle, it is associated with both components that are created. When all vertices are processed, we have an augmented Reeb graph. Each node in the augmented Reeb graph corresponds to a vertex, regular or critical, from the tetrahedral mesh. All regular nodes and genus-modifying saddle nodes are identified as degree-2 nodes and removed by merging their incident edges to obtain the Reeb graph.

### 3.4. Analysis

Let $n$ denote the number of triangles in the tetrahedral mesh of the 3-manifold. The number of vertices and edges in the input is less than $3n$. Let $g$ denote the maximum genus over all isosurfaces of the function. The number of saddles is a loose upper bound for $g$, since the genus of the isosurface can change only at a saddle. The maximum genus is typically a much smaller number.

The initial sorting of the tetrahedral mesh vertices takes $O(n \log n)$ time. To process each vertex, we perform a set of *InsertEdge* and *DeleteEdge* operations. Each *InsertEdge* and *DeleteEdge* operation takes $O(\log n)$ time. For deletion, if any edge is deleted from the tree $T$ or cotree $C$, then a replacement edge is identified from $X$. Since $|X| \leqslant 2g$, finding the replacement edge and updating the data structure takes $O(g \log n)$ time. In order to bound the number of insertions and deletions, consider the number of insertions into and deletions from each triangle. As shown in Fig. 3, there are exactly two insertions and two deletions per triangle to give a total of $2n$ insert/deletes. Nodes in the data structure correspond to edges in the 3-manifold. So, maintaining the tree-cotree partition requires $O(ng \log n)$ time using the edge-ordered tree.

Finding the replacement edge from $X$ is a costly operation. Selected edges from the tree $T$ and cotree $C$ can be contracted to derive a new tree $T'$ and cotree $C'$, each of which has $|X|$ edges, such that a replacement edge for $(T', C')$ is also the replacement edge for $(T, C)$. When $(T, C)$ changes, $(T', C')$ can be updated in $O(\log n)$ time [10]. The general dynamic graph connectivity algorithm of Holm et al. [15], as applied by Thorup [23] on smaller graphs $T' \cup X$ and $C' \cup X$ can find the replacement edges in $O(\log g(\log \log g)^3)$ time. The dynamic connectivity algorithm [15,23] is outlined in Section 5.

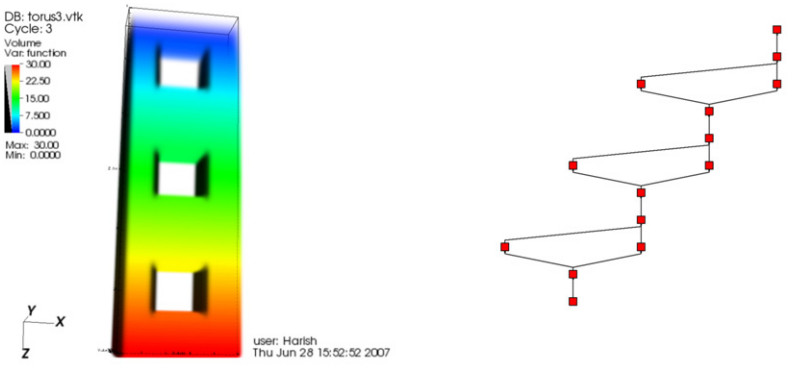
To identify the various isosurface components and to maintain the Reeb graph, we perform a constant number of insert, delete, or update operations on the root tree when a tetrahedral mesh vertex is processed. Inserting a node into the Reeb graph requires at most two $O(\log n)$ time queries on the root tree to identify the predecessor(s), and a constant time update of the adjacency list representation. Thus, the Reeb graph can be maintained in $O(n \log n)$ time. Putting the various steps together, the sweep algorithm constructs the Reeb graph in $O(n \log n + n \log g(\log \log g)^3)$ time.

## 4. Experiments

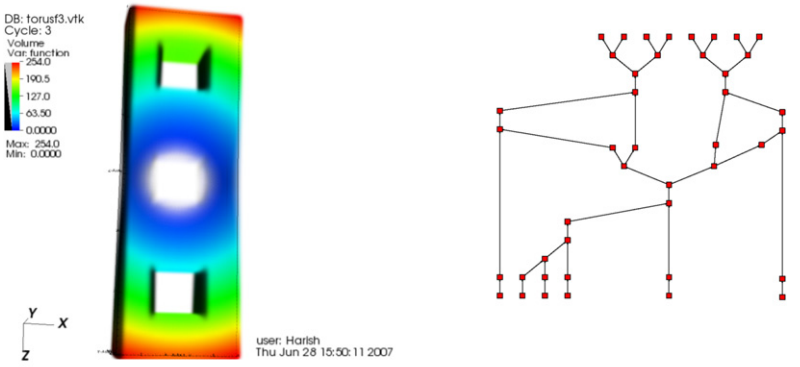
In this section, we discuss design choices made to simplify the implementation of the sweep algorithm and report experimental results. The edge-ordered tree data structure is implemented using the Sleator–Tarjan dynamic tree [22], which stores a given tree as a set of edge disjoint paths. The set of paths can be obtained using either a naïve partitioning strategy, where the set of paths is dependent on the sequence of tree operations, or the "partition by size" strategy, that is based on the size of paths. These paths can be stored using either a balanced search tree [8] or a biased search tree [4]. We chose an easier-to-implement balanced search tree to represent paths obtained using the naïve partitioning strategy, resulting in a $O(\log^2 n)$ amortized query time as opposed to the optimal biased binary tree and partition-by-size based implementation that provides a $O(\log n)$ worst case query time. We exhaustively search the set $X$ to find replacement edges after deleting an edge from the tree $T$ or cotree $C$. Thus our implementation has a running time of $O(ng \log^2 n)$, where $2g$ is the maximum value of $|X|$.

The code accepts a tetrahedral mesh and the function specified at vertices as input, computes the Reeb graph, and stores it as an edge list. A layout of the graph is generated using Tulip [1], an open source utility. Fig. 8 shows the Reeb graph computed for functions defined on a solid torus with three holes. Two functions were computed on the torus, a height function measuring the elevation of each vertex above a base plane (`TorusH`) and a distance function that measures the distance of each vertex from a single point in space (`TorusD`). Fig. 9 shows the Reeb graph computed for functions defined on two CAD models (`Engine` and `PMDC`).[1] Fig. 10 shows the running time for all four data sets—`TorusH, TorusD, Engine,` and `PMDC`. Each data set was available at multiple resolutions, coarse to fine. The results indicate that the algorithm

[1] `Engine` is a CAD model of an engine part and `PMDC` is available from the TetView distribution (http://tetgen.berlios.de/tetview.html).




(a) `TorusH`: Height function on a solid torus with three holes




(b) `TorusD`: Distance function on a solid torus with three holes

**Fig. 8.** Reeb graphs computed using our algorithm for various functions on a solid torus. Edges in the Reeb graph correspond to connected components of the isosurfaces.

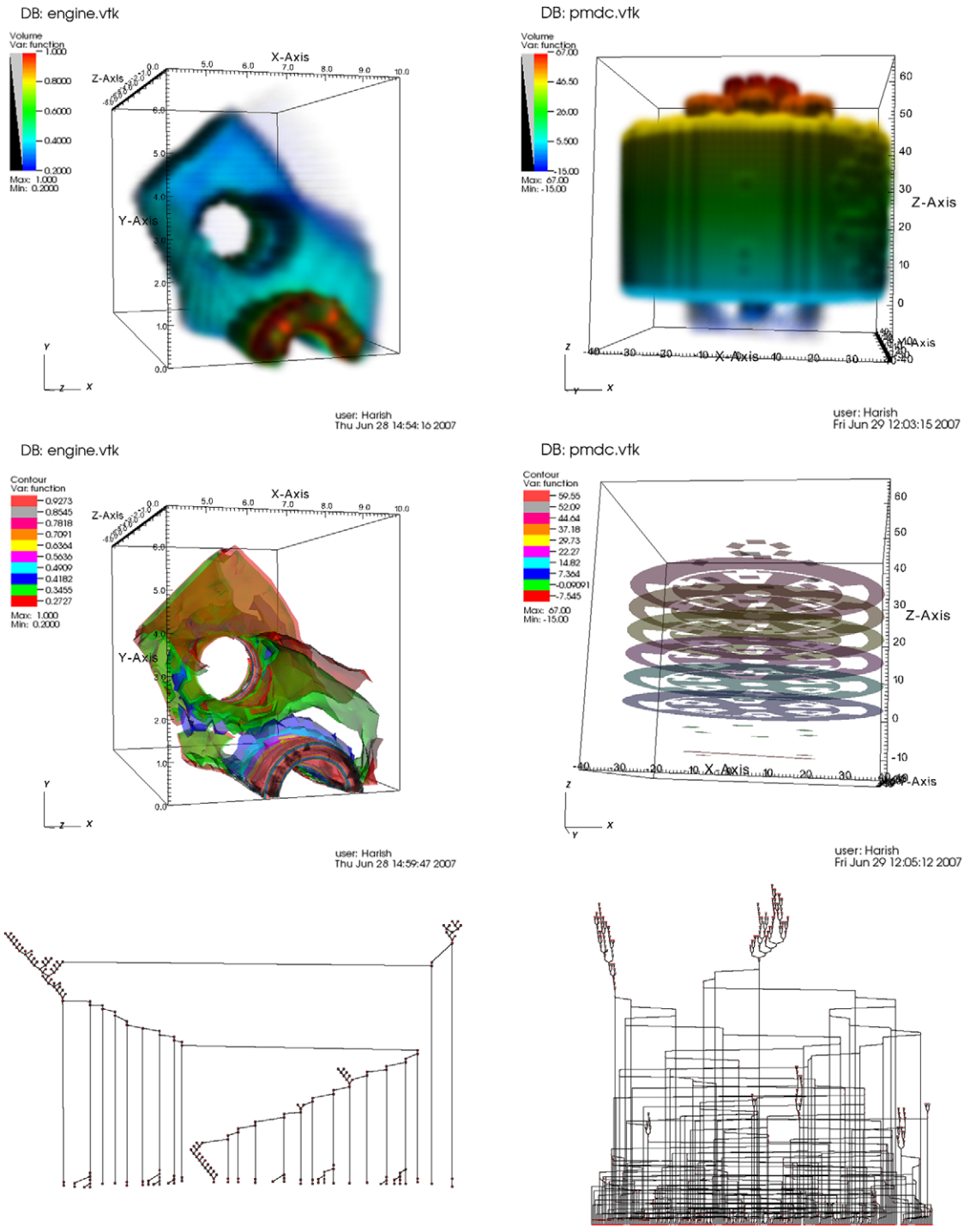
performs efficiently in practice. Our implementation is a prototype that has not been optimized. We expect the algorithm to yield better performance when appropriate code optimizations are added.

## 5. Reeb graphs of *d*-manifolds

We extend the sweep algorithm for 3-manifolds to construct Reeb graphs of $d$-manifolds. The level set of a Morse function defined on a $d$-manifold is a $(d-1)$-manifold. We are interested in tracking connected components of the level set. This is captured in the 1-skeleton of the level set. Therefore, it is sufficient to store edges and vertices of the level set. The 1-skeleton of the level set can be extracted from the 2-skeleton (triangles, edges, and vertices) of the $d$-manifold. Therefore, the sweep algorithm directly extends to higher dimension. However, the tree-cotree partition works only when the level sets are 2-manifolds because the 1-skeleton of the level set corresponds to a map $M$. This is not true in higher dimensions. So, we require a different data structure to store connected components of a level set.

### 5.1. Dynamic maintenance of level sets

Working with a graph representation of the 1-skeleton of the level set, we use the fully-dynamic connectivity algorithm described in [15] to track the evolution of level sets and answer connectivity queries. The dynamic connectivity algorithm stores the spanning forest $F$ of a graph $G$ for fast insertion of edges and quick response to connectivity queries. When an edge in $F$ is deleted, it causes a split in a tree in $F$, and if the corresponding component in $G$ is not split, then a replacement edge must be inserted into $F$. In order to find this replacement edge efficiently, each edge $e$ is associated with a level $l(e) \leqslant l_{max} = \lfloor \log n_v \rfloor$ for a graph with $n_v$ nodes. For each $i$, $F_i$, a sub-forest of $F$ induced by the edges of level at least $i$, is maintained. The replacement edge for a tree edge is now searched systematically in the set of sub-forests. The above replacement is carried out by a recursive *Replace*$((v, w), i)$ operation, which, assuming that there is no replacement edge on level $> i$, finds a replacement edge of the highest level $\leqslant i$, if any, such that $v$ and $w$ belong to the same component after adding the replacement edge.




(a) **Engine**: A CAD model of an engine part

(b) **PMDC**: A CAD model of a Permanent Magnet DC Motor

**Fig. 9.** Reeb graphs computed using our algorithm for functions on two CAD models. Edges in the Reeb graph correspond to connected components of the isosurfaces.

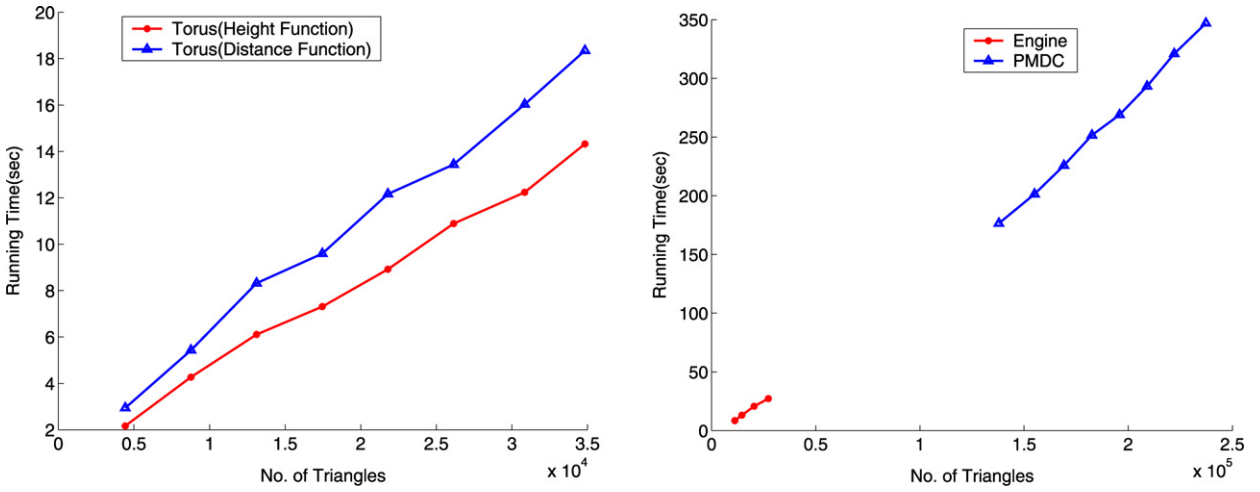



**Fig. 10.** Plot of running time of the sweep algorithm for various data sets.

### 5.2. Analysis

The fully-dynamic graph connectivity algorithm supports maintaining the spanning forest in $O(\log^2 n_v)$ amortized time per update and answering connectivity queries in $O(\log n_v / \log\log n_v)$ time for a graph with $n_v$ nodes. Since the rest of the implementation of the Reeb graph algorithm remains unchanged and the number of vertices in the level set is $O(n)$, we have an $O(n \log^2 n)$ time algorithm for constructing the Reeb graph. Again, $n$ is the number of triangles in the $d$-manifold.

In [23], the connectivity algorithm was modified to store an alternative rooted forest $S$, called the *structural forest* for a given graph $G$, instead of the spanning forest $F$. The leaves of $S$ correspond to vertices in $G$, and all of them have a depth equal to $l_{max}$. A level of a node in $S$ is its depth. For each $i$, $G_i$ denotes the subgraph induced by edges of level at least $i$. Nodes in $S$ at a level $i$ represents the components in $G_i$. This alternative representation was shown to support update operations in $O(\log n(\log\log n)^3)$ time and connectivity queries in $O(\log n/\log\log\log n)$ time [23]. Using this algorithm to store level sets will improve the time complexity of the Reeb graph construction to $O(n \log n(\log\log n)^3)$.

## 6. Conclusions

We have described an algorithm that constructs the Reeb graph of Morse functions defined on piecewise-linear 3-manifolds. Compared to prior known algorithms that run in $O(n^2)$ time, our algorithm has a running time of $O(n \log n + n \log g(\log\log g)^3)$, where $n$ is the number of triangles in the tetrahedral mesh representation of the 3-manifold and $g$ is the maximum genus over all isosurfaces of the Morse function. We have extended our algorithm to compute Reeb graphs of $d$-manifolds in $O(n \log n(\log\log n)^3)$ time for constant $d$.

The sweep algorithm works without any modifications both for closed manifolds and for manifolds with boundary. Further, no pre-processing of the data is required. The algorithm as described here, after minor changes, can also handle scalar functions with multiple saddles. Practical implementation of the algorithm necessitated a choice of simpler data structures with an increase in the worst case running time to $O(ng \log^2 n)$. Experimental results, however, indicate a better performance in practice. In future, we plan to extend our algorithm to work directly on voxel data and design parallel algorithms for computing Reeb graphs similar to existing contour tree algorithms [17].

## Acknowledgements

This work was supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/048/2007.

## References

[1] D. Auber, Tulip: A huge graph visualisation framework, in: P. Mutzel, M. Jünger (Eds.), Graph Drawing Softwares, in: Mathematics and Visualization, Springer-Verlag, 2003, pp. 105–126.
[2] C.L. Bajaj, V. Pascucci, D.R. Schikore, The contour spectrum, in: Proc. IEEE Conf. Visualization, 1997, pp. 167–173.
[3] T.F. Banchoff, Critical points and curvature for embedded polyhedral surfaces, Am. Math. Monthly 77 (1970) 475–485.
[4] S.W. Bent, D.D. Sleator, R.E. Tarjan, Biased search trees, SIAM J. Comput. 14 (3) (1985) 545–568.
[5] H. Carr, J. Snoeyink, U. Axen, Computing contour trees in all dimensions, Comput. Geom. Theory Appl. 24 (2) (2003) 75–94.
[6] H. Carr, J. Snoeyink, M. van de Panne, Simplifying flexible isosurfaces using local geometric measures, in: Proc. IEEE Conf. Visualization, 2004, pp. 497–504.
[7] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, V. Pascucci, Loops in Reeb graphs of 2-manifolds, Discrete Comput. Geom. 32 (2) (2004) 231–244.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, 2001.
[9] H. Edelsbrunner, J. Harer, V. Natarajan, V. Pascucci, Morse–Smale complexes for piecewise linear 3-manifolds, in: Proc. 19th Annual Symposium on Computational Geometry, 2003, pp. 361–370.
[10] D. Eppstein, Dynamic generators of topologically embedded graphs, in: SODA '03: Proc. Fourteenth Annual ACM–SIAM Symposium on Discrete Algorithms, 2003, pp. 599–608.
[11] D. Eppstein, G.F. Italiano, R. Tamassia, R.E. Tarjan, J. Westbrook, M. Yung, Maintenance of a minimum spanning forest in a dynamic plane graph, J. Algorithms 13 (1) (1992) 33–54.
[12] I. Guskov, Z. Wood, Topological noise removal, in: Proc. Graphics Interface, 2001, pp. 19–26.
[13] F. Hétroy, D. Attali, Topological quadrangulations of closed triangulated surfaces using the Reeb graph, Graph. Models 65 (1–3) (2003) 131–148.
[14] M. Hilaga, Y. Shinagawa, T. Kohmura, T.L. Kunii, Topology matching for fully automatic similarity estimation of 3d shapes, in: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, 2001, pp. 203–212.
[15] J. Holm, K. de Lichtenberg, M. Thorup, Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity, J. ACM 48 (4) (2001) 723–760.
[16] E.P. Mücke, Shapes and implementations in three-dimensional geometry, PhD thesis, Dept. Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1993.
[17] V. Pascucci, K. Cole-McLaughlin, Parallel computation of the topology of level sets, Algorithmica 38 (1) (2003) 249–268.
[18] V. Pascucci, G. Scorzelli, P.-T. Bremer, A. Mascarenhas, Robust on-line computation of reeb graphs: simplicity and speed, ACM Trans. Graph. 26 (3) (2007) 58.
[19] G. Reeb, Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique, C. R. Acad. Sci. Paris 222 (1946) 847–849.
[20] Y. Shinagawa, T.L. Kunii, Constructing a Reeb graph automatically from cross sections, IEEE Comput. Graph. Appl. 11 (6) (1991) 44–51.
[21] Y. Shinagawa, T.L. Kunii, Y.L. Kergosien, Surface coding based on Morse theory, IEEE Comput. Graph. Appl. 11 (5) (1991) 66–78.
[22] D.D. Sleator, R.E. Tarjan, A data structure for dynamic trees, J. Comput. Syst. Sci. 26 (3) (1983) 362–391.
[23] M. Thorup, Near-optimal fully-dynamic graph connectivity, in: STOC '00: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, 2000, pp. 343–350.
[24] T. Tung, F. Schmitt, Augmented Reeb graphs for content-based retrieval of 3d mesh models, in: SMI '04: Proceedings of the Shape Modeling International 2004 (SMI'04), 2004, pp. 157–166.
[25] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, D.R. Schikore, Contour trees and small seed sets for isosurface traversal, in: Proceedings of the 13th ACM Annual Symposium on Computational Geometry, 1997, pp. 212–220.
[26] G.H. Weber, S.E. Dillard, H. Carr, V. Pascucci, B. Hamann, Topology-controlled volume rendering, IEEE Trans. Vis. Comput. Graph. 13 (2) (2007) 330–341.
[27] Z. Wood, H. Hoppe, M. Desbrun, P. Schröder, Removing excess topology from isosurfaces, ACM Trans. Graph. 23 (2) (2004) 190–208.
[28] Y. Shinagawa, T.L. Kunii, H. Sato, M. Ibusuki, Modeling contact of two complex objects: With an application to characterizing dental articulations, Computers and Graphics 19 (1) (1995) 21–28.
[29] E. Zhang, K. Mischaikow, G. Turk, Feature-based surface parameterization and texture mapping, ACM Trans. Graph. 24 (1) (2005) 1–27.