# Efficient Output-Sensitive Construction of Reeb Graphs[⋆]

Harish Doraiswamy[1] and Vijay Natarajan[1,2]

[1] Department of Computer Science and Automation
[2] Supercomputer Education and Research Centre
Indian Institute of Science, Bangalore 560012, India.
{harishd,vijayn}@csa.iisc.ernet.in

**Abstract.** The Reeb graph tracks topology changes in level sets of a scalar function and finds applications in scientific visualization and geometric modeling. This paper describes a near-optimal two-step algorithm that constructs the Reeb graph of a Morse function defined over manifolds in any dimension. The algorithm first identifies the critical points of the input manifold, and then connects these critical points in the second step to obtain the Reeb graph. A simplification mechanism based on topological persistence aids in the removal of noise and unimportant features. A radial layout scheme results in a feature-directed drawing of the Reeb graph. Experimental results demonstrate the efficiency of the Reeb graph construction in practice and its applications.

## 1 Introduction

The Reeb graph of a scalar function describes the connectivity of its level sets. The abstract representation of level-set topology within the Reeb graph enables 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 [20, 25], topology-based shape matching [13], topological simplification and cleaning [11, 24], surface segmentation and parametrization [12, 26], and efficient computation of level sets [22]. They serve as an effective user interface for selecting meaningful level sets [2, 5] and transfer functions for volume rendering [23].

### 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 [19]. 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 input. Cole-Mclaughlin et al. [7] improved the running time to $O(n \log n)$ by maintaining the

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

level sets using dynamically balanced search trees. In a recent paper, Pascucci et al. [17] 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. In the case of streaming data, where triangles are processed one after another, the algorithm essentially attaches the straight line Reeb graph corresponding to the current triangle with the Reeb graph computed so far. Even though the algorithm has a $O(n^2)$ behavior in the worst case, it performs very well in practice for 2-manifolds. However, the optimizations that result in fast incremental construction of Reeb graphs for 2-manifolds do not provide a performance benefit in higher dimensions. We adopt a simple but different approach to compute Reeb graphs that traces connected components of interval volumes (the volume between two level sets). This approach results in an algorithm that exhibits good worst-case behavior and works well in practice - while we obtain good running times for 2-manifolds, our algorithm performs better than the online algorithm for 3-manifolds.

For the special case of loop-free Reeb graphs, Carr et al. [4] described an elegant $O(v \log v)$ algorithm that works in all dimensions, where $v$ is the number of vertices in the input. Besides the naïve $O(n^2)$ algorithm and the online algorithm, there is no known algorithm for computing Reeb graphs of manifolds in higher dimension. 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., may not exist. Efficient storage and manipulation of connected components of level sets will lead to fast construction of Reeb graphs. Cole-Mclaughlin et al. [7] adopt this approach to obtain an efficient algorithm for 2-manifolds. However they exploit the unique property of one-dimensional level sets that their vertices can be ordered, and hence, their algorithm does not directly extend to higher dimension manifolds. Other algorithms for computing Reeb graphs follow a sample based approach that produces potentially inaccurate results [13, 21].

## 1.2 Results

We present an efficient two-step algorithm for computing the Reeb graph of a piecewise-linear (PL) function in $O(n + l + t \log t)$ time, where $n$ is the number of triangles in the input mesh, $t$ is the number of critical points of the function, and $l$ is the size (number of edges) of all critical level sets. The algorithm has various desirable properties. It is

- *output-sensitive*: the running time depends of the number of critical points of the function, which is equal to the number of nodes in the Reeb graph, and the size of critical level sets, which is indicative of the importance of features in the data.
- *near-optimal*: the size of critical level sets is usually $O(n)$ in practice. So, the worst-case running time is close to the optimal bound of $(n + t \log t)$ [22].
- *generic*: the algorithm works, without any modifications, for functions defined on $d$-manifolds and for non-manifolds.
- *simple*: the algorithm is simple to implement.

We also describe a method to simplify the Reeb graph based on an extended notion of persistence [1] that removes short leaves and cycles in the graph. Finally, we describe a feature-directed layout of the Reeb graph that serves as a useful interface for exploring and understanding three-dimensional scalar fields. We also present experimental results that demonstrate the efficiency of our algorithm.

## 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.
3. All critical values are distinct *i.e.*, $f(p) \neq f(q)$ for all critical points $p \neq q$.

The above conditions typically do not hold in practice for PL functions. However, simulated perturbation of the function [8, Section 1.4] ensures that no two critical values are equal. All multiple saddles (degenerate) can be unfolded into simple (non-degenerate) saddles by repeatedly splitting the link of the multiple saddle [9]. A total order on the vertices helps in consistently identifying the vertex with the higher function value between a pair of vertices.

### 2.1 Critical points and level sets

Critical points of a smooth function are exactly where the gradient becomes zero. Banchoff [3] and later Edelsbrunner et al. [9] describe a combinatorial characterization for critical points of a PL function, which are always located at vertices of the mesh. The *link* of a vertex consists of all vertices adjacent to it and the induced edges, triangles, and higher-order simplices. Adjacent vertices with lower function value and their induced simplices constitute the *lower link*, whereas the adjacent vertices with higher function value and their induced simplices constitute the *upper link*. For functions defined on 2- and 3-manifolds, the critical points are classified based on the number of connected components (denoted as $\beta_0$, the zeroth Betti number) of the lower and upper link. Classification of all critical points in higher dimensions requires the computation of higher order Betti numbers.

The preimage of a real value is called a *level set*. The level set of a regular value is a $(d-1)$-manifold with or without boundary, possibly containing multiple connected components. We are interested in the evolution of level sets against increasing function value. Significant topological changes occur at critical points, whereas topology of the level set is preserved across regular points [14].

In the context of Reeb graphs, we are only interested in critical points that modify the number of level-set components. So, it is sufficient to count the number of connected components of the lower / upper link for identifying these critical points. Given a critical point $c_i$ with function value $f_i$, we define a *critical level set* as the level set at a function value infinitesimally below / above $f_i$ (i.e. $f^{-1}(f_i \pm \varepsilon)$).

For example, consider the case when $d = 3$. A level set of a 3-manifold is called an *isosurface*. Figure 1 illustrates the topology changes that occur at critical points of a 3-manifold. Specifically, the level set topology changes either by gaining/losing

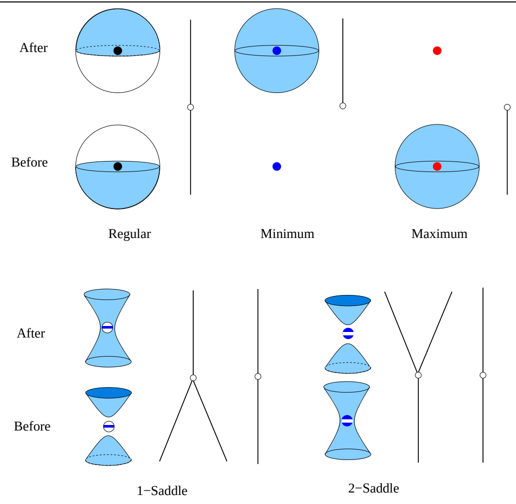


**Fig. 1.** Figures above show the isosurfaces before $(f^{-1}(c-\varepsilon))$ and after $(f^{-1}(c+\varepsilon))$ passing through a point with function value $c$ and the structure of the Reeb graph at the corresponding node. Topology of the isosurface changes when it evolves past a critical point.

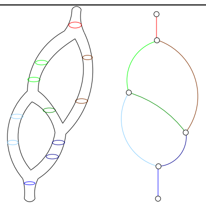


**Fig. 2.** Reeb graph of the height function defined on a surface with two tunnels. Reeb graph tracks the topology of level sets.

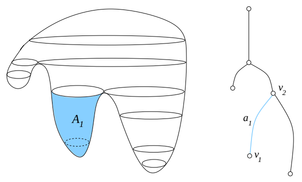


**Fig. 3.** Mapping between arcs in the Reeb graph and cylinders in the input.

a component or by gaining/losing genus. 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 Figure 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.

## 2.2 Reeb graph

The *Reeb graph* of $f$ is obtained by contracting each connected component of a level set to a point [18]. The Reeb graph expresses the evolution of connected components of level sets as a graph whose nodes correspond to critical points of the function, see Figure 2. Figure 1 illustrates the structure of the Reeb graph for 3-manifolds 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.

This view of the Reeb graph focuses on the mapping between components of individual level sets and nodes or points within arcs of the graph. We propose the use of an alternate mapping between nodes / arcs of the graph and components of critical level sets / equivalence classes of regular level set components. The two mappings are consistent with each other. The advantage of the alternate view is that it leads to a simple and efficient algorithm to compute the Reeb graph. For example, in Figure 3, the arc $a_1$ is mapped to *cylinder* $A_1$, a collection of regular level set components that are topologically equivalent to each other. The boundary of $A_1$ consists of two critical level set components. The end point $v_2$ of the arc originating at $v_1$ can be computed by tracing the cylinder from the lower boundary component to the upper component.

### 2.3 Input

We assume that the input manifold is represented by a triangulated mesh, the function is sampled at vertices, and linearly interpolated within each simplex. In the case of higher dimensional manifolds ($d \geq 3$), the algorithm requires only the 2-skeleton (vertices, edges, and triangles) of the mesh.

## 3 The Reeb graph algorithm

We now describe an algorithm that computes the Reeb graph of a PL function $f$ defined on a 3-manifold. The algorithm directly extends to $d$-manifolds ($d \geq 2$) and non-manifolds but in order to simplify the description, we will consider the case of $d = 3$ in this section. The algorithm proceeds in two phases:

1. Locate critical points of the input and sort them based on function value.
2. Connect critical point pairs to obtain arcs of the Reeb graph.

A vertex is *regular* if it has one lower link component and one upper link component. All other vertices are *critical*. A critical point is a *maximum* if the upper link is empty and a *minimum* if the lower link is empty. Number of components of the upper and lower links are computed using a breadth first traversal of the link. We only need to locate the critical points and classify them as either a minimum, maximum or saddle.

### 3.1 $ls$-graph

Tracking components of the level set requires only a 1-skeleton (vertices and edges) representation of the level set. Edges in a level set of $f$ will pass through a set of triangles in the input mesh. We track components of level sets between two function values $f_1$ and $f_2$ ($f_1 < f_2$) by traversing triangles through which each component of the level set passes as function values are varied from $f_1$ to $f_2$. We introduce a dual graph that stores triangle adjacencies and helps track level set components. This graph $G_{ls}(V,E)$, called the *ls-graph* , is a directed graph whose nodes $V = \{t_1, t_2, \ldots, t_n\}$ corresponds to the $n$ triangles $\{T_1, T_2, \ldots, T_n\}$ in the input mesh. Each node $t_i$ is assigned a cost equal to the maximum over function values at the vertices of the triangle $T_i$. Let $v_0, v_1$, and $v_2$ $(f(v_0) < f(v_1) < f(v_2))$ be vertices of a triangle $T_i$. $G_{ls}$ contains an edge between vertices $t_i$ and $t_j$ if triangles $T_i$ and $T_j$ are adjacent, unless $T_i$ and $T_j$ share the edge $(v_0, v_1)$ and the cost of $t_j$ is greater than $f(v_1)$, see Figure 4. The dotted, solid and dashed lines within the triangles indicate the edges of the level set when the function value becomes greater than $f(v_0)$, $f(v_1)$ and $f(v_2)$ respectively. An edge is directed towards the node with higher cost. Traversing an edge in $G_{ls}$ implicitly tracks a component of a level set as function value increases. If this edge does not cross a critical value, then the traversal is equivalent to tracing a path within a cylinder. Figure 4(f) shows a configuration where the level set potentially splits and hence no edge is inserted between $t_i$ and $t_j$.

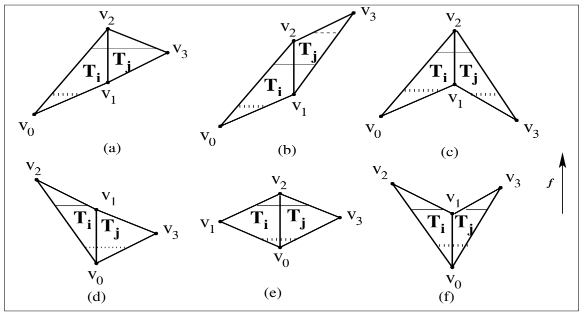


**Fig. 4.** The $ls$-graph contains an edge between $t_i$ and $t_j$ in all cases except the forbidden configuration in (f).

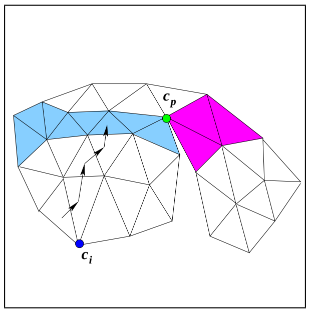


**Fig. 5.** Connecting critical points in the algorithm

## 3.2 Connecting the critical points

Our iterative algorithm uses the $ls$-graph to compute arcs in the Reeb graph. Let $\{c_1, c_2, \ldots, c_t\}$ be the ordered set of critical points with function values $\{f_1, f_2, \ldots, f_t\}$ ($f_x < f_y$ whenever $x < y$). Let $L_i$ denote the set of triangles that contain the components of the critical level set $f^{-1}(f_i - \varepsilon)$ that are modified by $c_i$. The $i$th iteration of the algorithm connects $c_i$ with a set of critical points $c_p$ ($f_p > f_i$).

The *star* of a vertex consists of all simplices incident on the vertex. All simplices in the star where the function value is greater than the vertex constitute the *upper star*. Let the upper star of $c_i$ contain $k$ connected components ($k \leq 2$ for a Morse function). Each component of the star corresponds to a possible arc in the Reeb graph starting from $c_i$. Initiate a search in the $ls$-graph beginning with a node $t_{ij}$ corresponding to a triangle $T_{ij}$ in the $j$th component ($j = 1 \ldots k$) of the upper star of $c_i$. We move to a higher cost node in $G_{ls}$ at each step of the search. The search terminates when we find a node $t'_{ij}$ with cost greater or equal to $f_{i+1}$. An arc connects the nodes corresponding to $c_i$ and $c_{i+1}$ in the Reeb graph iff the triangle $T'_{ij}$ belongs to the set $L_{i+1}$. This search represents a monotone ascent through the cylinder. The search procedure for the $i$th iteration is illustrated in Figure 5, which shows a slice of the input mesh with the relevant triangles. Triangles in $L_p$ are shaded, different colors indicating disjoint components of the level set.

We use a triangle-edge data structure [15] to store the input triangulation. The $ls$-graph is implicitly stored in this data structure because each triangle-edge pair stores a reference to neighboring triangle-edge pairs. During the search, we tag a visited node with a label $[i, j]$ if its cost is lesser or equal to $f_{i+1}$. If $T'_{ij}$ does not belong to $L_{i+1}$, we continue the search until we reach a node with cost greater or equal to $f_{i+2}$, in order to determine if an arc in the Reeb graph connects $c_i$ with $c_{i+2}$. We repeat the search until it is successful.

If a search initiated in the $i$th iteration from a node in the $j$th component of the upper star reaches a node with a tag $[i, j']$ ($j \neq j'$), then $c_i$ is a genus modifying saddle and therefore the Reeb graph remains unchanged. The critical point $c_i$ is again a genus modifying saddle if the search reaches a node, whose corresponding triangle lies in a critical isosurface component that was previously visited from a different upper star component of $c_i$. Note that it is impossible for the search initiated in the $i$th iteration to reach a vertex tagged $[i', j]$ $(i \neq i')$ for any $j$, because this will imply that two components of a level set merged into one at a regular vertex.

The Reeb graph is stored as an adjacency list whose nodes correspond to critical points of the function. An arc from $c_i$ to $c_p$ is added if the search finds a triangle in $L_p$. Once all critical points are processed, the adjacency list will represent the Reeb graph.

## 3.3 Analysis

**Correctness**. Let $c_p$ $(f_i < f_p)$ be a critical point such that there is an arc from $c_i$ to $c_p$ in the Reeb graph. So, if we track a component of the level set at function value infinitesimally above $f_i$ and keep increasing the function value until it reaches $f_p$, then the topology of that component remains unchanged until the function value reaches $f_p$. Consider a triangle $T_{ij}$ that contains the level set component when the tracking begins. As we increase the function value past the cost of the node $t_{ij}$, the level set component passes through an adjacent triangle with cost greater than that of $t_{ij}$. This is equivalent to the search in the *ls*-graph as performed by the algorithm. The algorithm continues the above procedure until we reach a node $t'_{ij}$ with cost greater than or equal to $f_p$. Since the cost of the preceding node is less than $f_p$, an isosurface component at a function value infinitesimally below $f_p$ will pass through the triangle $T'_{ij}$. Since the Reeb graph contains an arc between critical points $c_i$ and $c_p$, the triangle $T'_{ij}$ will belong to the set $L_p$ and our algorithm will identify the arc $(c_i, c_p)$ of the Reeb graph.

**Running time**. Let $n$ be the number of triangles in the input and $t$ be the number of critical points of the input PL function. Triangles adjacent to a given triangle can be found in $O(1)$ time using the triangle-edge data structure. The *ls*-graph is implicitly stored in this data structure. Critical points are located by computing the number of connected components of the lower and upper links, which can be done in $O(n)$ time using the triangle-edge data structure. Sorting the critical points takes $O(t \log t)$ time.

The sets $L_i$ for each critical point $c_i$ can be found by marching through the triangles that contain $f^{-1}(f_i - \varepsilon)$. This can be accomplished in $O(l)$ time, where $l = \sum_i |L_i|$, is the number of triangles in all the sets $L_i$. Though it is possible in theory that $l = O(n^2)$, the size of the critical level sets is usually $O(n)$ in practice.

Each node $t_i$ in the *ls*-graph has at most 6 neighbors (since each triangle can be in at most two tetrahedra). Hence, the number of edges in the *ls*-graph is $O(n)$. During the search procedure, each node is tagged exactly once and visited at most 6 times (from each of its tagged neighbors). Thus the traversal of the graph is accomplished in $O(n)$ time. Each update of the adjacency list representation takes constant time. Total number of such updates is equal to the number of arcs in the Reeb graph. A conservative bound for the number of edges in the Reeb graph is given by the number of triangles in the input. Hence, maintaining the Reeb graph takes $O(n)$ time. Combining the above steps, we obtain an $O(n + l + t \log t)$ running time for our algorithm.

### 3.4 $d$-manifolds and non-manifolds

The level set of a regular value for a Morse function defined on a $d$-manifold is a $(d-1)$-manifold. The connectivity of a level set is represented by its 1-skeleton. Hence, similar to 3-manifolds, tracking the connected components of the level set requires only a 1-skeleton representation, which can be extracted from the 2-skeleton of the input mesh. So, the algorithm works directly on the 2-skeleton representation of $d$-manifolds. In the case of non-manifolds, the algorithm will again work on the 2-skeleton representation. We relax the definition of critical points to include all vertices that modify the topology of the level set. Candidate critical points are again located by counting the number of connected components of the lower and upper link.

## 4 Visualization of Reeb graphs

### 4.1 Simplification of Reeb graphs

Simplification is necessary for effective visualization of large and feature rich data because it aids in noise removal and creation of feature-preserving multiresolution representations. A topological feature in the input is represented by a pair of critical points, typically an arc in the Reeb graph. Unimportant features in the data can be removed by repeated cancellation of low persistence critical point pairs [10], which also leads to a multiresolution representation of the input scalar field. Features can also be ordered and removed based on geometric measures like hypervolume [5]. Existing algorithms for contour tree simplification remove critical point pairs that create / destroy a level set component. We simplify the Reeb graph using a notion of extended persistence [1] that pairs genus modifying critical points in addition to pairing component creators with destroyers.

Our approach to Reeb graph simplification is similar to the one used to simplify contour trees [5]. In addition to the *leaf pruning* and *node reduction* simplification operations, we perform an additional *loop pruning* operation on the Reeb graph. Leaf pruning removes a leaf and the incident arc from the Reeb graph. A leaf connecting to a degree-2 saddle is not pruned. Node reduction removes a degree-2 node by merging the two adjacent edges. The loop pruning operation removes a loop defined by adjacent nodes connected by two parallel arcs, from the Reeb graph. This operation is equivalent to removing one of the parallel arcs and performing node reduction on the pair of adjacent nodes.

We simplify the Reeb graph using repeated application of the three mentioned operations: (1) Perform node reduction where possible, (2) Choose the least important leaf / loop and prune it. Leaves and loops that can be pruned are stored in a priority queue ordered based on the persistence of the corresponding critical point pair. If a pruning operation results in a reducible node, then node reduction is performed immediately. All new leaves and loops created by the above operations are in turn added to the priority queue. Note that we use the simplification process as an aid for visualizing Reeb graphs and not to modify the input function. Realizing the function representing the simplified Reeb graph may require changing the topology of the input.

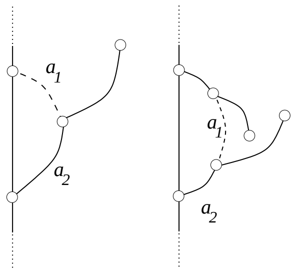
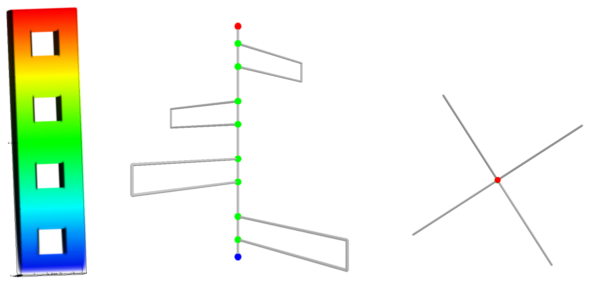


**Fig. 6.** The spanning contour tree of a Reeb graph is structurally similar to a contour tree. Removal of $a_1$ results in a spanning contour tree. Removing $a_2$ results in a tree with an invalid degree-2 node.



(a) model (b) Reeb graph: side view (c) Reeb graph: top view

**Fig. 7.** Radial layout of the Reeb graph of the height function on a 4-torus.

## 4.2 Reeb graph layout

We build upon the orrery layout proposed for contour trees [16] to obtain a layout for Reeb graphs. The extension to Reeb graphs is non-trivial because of the presence of loops. We overcome this difficulty by designing a four step layout scheme:

- First, extract a *spanning contour tree* of the Reeb graph.
- Second, compute a branch decomposition of this spanning tree.
- Third, use a radial layout scheme to embed the spanning tree in 3D.
- Finally, add the non-tree arcs to the layout.

The spanning contour tree is a spanning tree of the Reeb graph that satisfies the structural properties of a contour tree. All degree-2 nodes in this spanning tree have exactly one neighbor node with higher function value and one neighbor node with lower function value. Not all spanning trees satisfy this property. For example, in Figure 6, removing arc $a_1$ results in a spanning contour tree. Removal of $a_2$ also results in a spanning tree, but one that does not correspond to a valid contour tree.

A branch decomposition is an alternate representation of a contour tree that explicitly stores the topological features and their hierarchical relationship [16]. A branch is a path between two leaves of the contour tree or a path that connects a leaf to an interior node of another branch.

All branches of the spanning contour tree are drawn as L-shaped polylines and the $z$-coordinate corresponds to function value. The $(x, y)$ coordinates are computed for each branch using a radial layout scheme. The root branch is located at the origin and others are placed on concentric circles centered at the origin. All branches that connect to an interior node of the root branch are equally spaced around the origin at a fixed distance from it. Branches that connect to an interior node of a first-level branch are placed in the second concentric circle within a wedge centered at a level-one branch. The angle subtended is proportional to the number of descendant branches. In order to

**Table 1.** Reeb graph computation time for various 2D and 3D input. For all 2D models, solid 8-torus, and fighter, the Reeb graph was computed for the height function. In all other cases, the function is available with the data set.

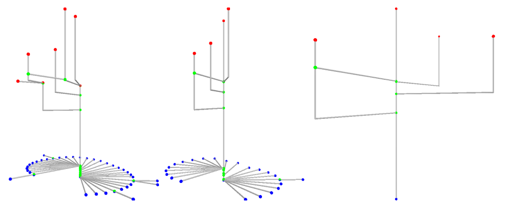
| | | | | Time taken (sec) | |
|---|---|---|---|---|---|
| Dimension | Model | #Triangles | #Critical points | Our algorithm | Online algorithm |
| 2D | bunny | 40000 | 217 | 0.7 | 0.1 |
| | Laurent Hand | 99999 | 92 | 1.5 | 0.43 |
| | Neptune | 998840 | 1757 | 24.3 | 3.7 |
| 3D | engine | 27252 | 160 | 2 | 0.7 |
| | solid 8-torus | 34832 | 18 | 0.56 | 0.14 |
| | fighter | 143881 | 8 | 2.2 | 28 |
| | PMDC | 237291 | 902 | 8 | 17 |
| | blunt | 451601 | 827 | 213 | 406 |
| | nucleon | 652964 | 2203 | 254 | 1638 |
| | post | 1243200 | 132 | 70 | 1671 |

avoid intersections when the non-tree arcs are added, we include a dummy branch for each loop arc before calculating the angular wedge subtended at each branch. Figure 7 shows the layout for the Reeb graph of a 4-torus.

## 5 Experimental Results

The Reeb graph construction algorithm was implemented in Java and tested on a Pentium 4, 2.4 GHz machine with 1 GB main memory. Our implementation accepts a function sampled at vertices of a simplicial mesh as input, computes the Reeb graph, and stores it as an edge list. Table 1 shows the time taken by our implementation to compute the Reeb graph for various models. We compare the performance of our algorithm with the online algorithm described in [17]. While the online algorithm performs well for 2D data, our algorithm performs substantially better for 3D data. We expect that the algorithm will also be efficient in practice for higher dimensional input. The running time depends on the number of critical points, clearly indicating the output sensitivity of our algorithm. For large datasets that do not fit in memory, our implementation can be extended similar to the contour tree algorithm described in [6]. Also, no code optimizations have been applied. An implementation in C / C++ will exhibit significant improvement in the performance.

Figure 8 shows the Reeb graph for the height function defined on the Laurent hand model. The near-horizontal branches in the Reeb graph indicate that the function is noisy near the wrist. Our implementation allows the user to interactively simplify and visualize the Reeb graph by specifying a persistence threshold. Simplification using a low persistence threshold removes these unimportant features. The remaining branches correspond to the palm and fingers and the loop corresponds to the thumb and forefinger. Figure 9 shows the Reeb graph computed for two biological data sets: dnaB and GroEL. In both data sets, the volumetric domain represents the molecule and the scalar field is the height function. Loops in the Reeb graph indicate possible tunnels in


(a) Reeb graphs before and after various iterations of simplification.

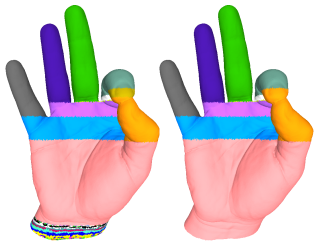
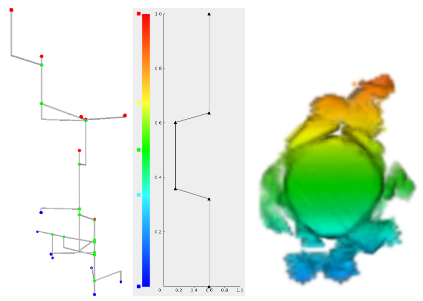

(b) partition induced by the full resolution and simplified Reeb graphs.

**Fig. 8.** Reeb graph computed for height function on Laurent hand and induced partition on the surface.


(a) dnaB

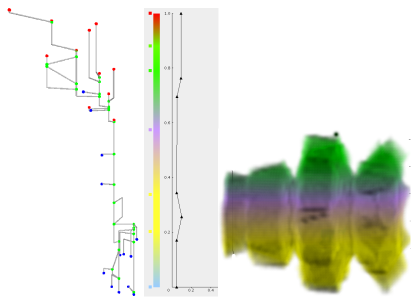

(b) GroEL

**Fig. 9.** Reeb graph computed for height function on volume representation of two molecules: dnaB and GroEL. The transfer function shown in the middle determines the color map.

the molecule. The Reeb graph of dnaB contains two loops and that of GroEL has five loops.

## 6 Conclusions and Future Work

We have described a simple output-sensitive near-optimal algorithm that constructs the Reeb graph of a PL function. Compared to prior known algorithms that run in $O(n^2)$ time, our algorithm has a worst case running time of $O(n+l+t\log t)$, where $n$ is the number of triangles in the mesh representing the domain, $t$ is the number of critical points of the function and $l$ is size of all critical level sets. The algorithm works without any modification for functions defined on manifolds in any dimension, and for non-manifold domains. We have also described a method to simplify the Reeb graph based on an extended notion of persistence and provided a feature-directed layout of the Reeb graph that serves as a useful interface for exploring and understanding three-dimensional scalar fields. We have shown through our experimental results that our algorithm performs efficiently in practice. The iterations of the algorithm being independent of each other, provide an inherent scope for parallelization.

## References

1. Agarwal, P. K., Edelsbrunner, H., Harer, J., and Wang, Y. Extreme elevation on a 2-manifold. *Disc. Comput. Geom. 36*, 4 (2006), 553–572.
2. Bajaj, C. L., Pascucci, V., and Schikore, D. R. The contour spectrum. In *Proc. IEEE Conf. Visualization* (1997), pp. 167–173.
3. Banchoff, T. F. Critical points and curvature for embedded polyhedral surfaces. *Am. Math. Monthly 77* (1970), 475–485.
4. Carr, H., Snoeyink, J., and Axen, U. Computing contour trees in all dimensions. *Comput. Geom. Theory Appl. 24*, 2 (2003), 75–94.
5. Carr, H., Snoeyink, J., and van de Panne, M. Simplifying flexible isosurfaces using local geometric measures. In *Proc. IEEE Conf. Visualization* (2004), pp. 497–504.
6. Chiang, Y.-J., Lenz, T., Lu, X., and Rote, G. Simple and optimal output-sensitive construction of contour trees using monotone paths. *Comput. Geom. Theory Appl. 30*, 2 (2005), 165–195.
7. Cole-McLaughlin, K., Edelsbrunner, H., Harer, J., Natarajan, V., and Pascucci, V. Loops in Reeb graphs of 2-manifolds. *Disc. Comput. Geom. 32*, 2 (2004), 231–244.
8. Edelsbrunner, H. *Geometry and Topology for Mesh Generation*. Cambridge Univ. Press, England, 2001.
9. Edelsbrunner, H., Harer, J., Natarajan, V., and Pascucci, V. Morse-Smale complexes for piecewise linear 3-manifolds. In *Proc. Symp. Comput. Geom.* (2003), pp. 361–370.
10. Edelsbrunner, H., Letscher, D., and Zomorodian., A. Topological persistence and simplification. *Disc. Comput. Geom. 28*, 4 (2002), 511–533.
11. Guskov, I., and Wood, Z. Topological noise removal. In *Proc. Graphics Interface* (2001), pp. 19–26.
12. Hétroy, F., and Attali, D. Topological quadrangulations of closed triangulated surfaces using the Reeb graph. *Graph. Models 65*, 1-3 (2003), 131–148.
13. Hilaga, M., Shinagawa, Y., Kohmura, T., and Kunii, T. L. Topology matching for fully automatic similarity estimation of 3d shapes. In *Proc. SIGGRAPH* (2001), pp. 203–212.
14. Matsumoto, Y. *An Introduction to Morse Theory*. Amer. Math. Soc., 2002. Translated from Japanese by K. Hudson and M. Saito.
15. Mücke, E. P. *Shapes and Implementations in Three-Dimensional Geometry*. PhD thesis, Dept. Computer Science, University of Illinois, Urbana-Champaign, Illinois, 1993.
16. Pascucci, V., Cole-McLaughlin, K., and Scorzelli, G. Multi-resolution computation and presentation of contour trees. Tech. rep., Lawrence Livermore Natl. Lab., 2005.
17. Pascucci, V., Scorzelli, G., Bremer, P.-T., and Mascarenhas, A. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Trans. Graph. 26*, 3 (2007), 58.
18. Reeb, G. Sur les points singuliers d'une forme de pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus de L'Académie ses Séances, Paris 222* (1946), 847–849.
19. Shinagawa, Y., and Kunii, T. L. Constructing a reeb graph automatically from cross sections. *IEEE Comput. Graph. Appl. 11*, 6 (1991), 44–51.
20. Shinagawa, Y., Kunii, T. L., and Kergosien, Y. L. Surface coding based on Morse theory. *IEEE Comput. Graph. Appl. 11*, 5 (1991), 66–78.
21. Tung, T., and Schmitt, F. Augmented reeb graphs for content-based retrieval of 3d mesh models. In *SMI '04: Proc Shape Modeling Intl.* (2004), pp. 157–166.
22. van Kreveld, M., van Oostrum, R., Bajaj, C., Pascucci, V., and Schikore, D. R. Contour trees and small seed sets for isosurface traversal. In *Proc. Symp. Comput. Geom.* (1997), pp. 212–220.

23. WEBER, G. H., DILLARD, S. E., CARR, H., PASCUCCI, V., AND HAMANN, B. Topology-controlled volume rendering. *IEEE Trans. Vis. Comput. Graph. 13*, 2 (2007), 330–341.
24. WOOD, Z., HOPPE, H., DESBRUN, M., AND SCHRÖDER, P. Removing excess topology from isosurfaces. *ACM Trans. Graph. 23*, 2 (2004), 190–208.
25. Y. SHINAGAWA, T. L. KUNII, H. S., AND IBUSUKI, M. Modeling contact of two complex objects: with an application to characterizing dental articulations. *Computers and Graphics 19*, 1 (1995), 21–28.
26. ZHANG, E., MISCHAIKOW, K., AND TURK, G. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph. 24*, 1 (2005), 1–27.