Computing Reeb Graphs as a Union of Contour Trees

Harish Doraiswamy, *Student Member*, *IEEE*, and Vijay Natarajan, *Member*, *IEEE*

Abstract—The Reeb graph of a scalar function tracks the evolution of the topology of its level sets. This paper describes a fast algorithm to compute the Reeb graph of a piecewise-linear (PL) function defined over manifolds and non-manifolds. The key idea in the proposed approach is to maximally leverage the efficient contour tree algorithm to compute the Reeb graph. The algorithm proceeds by dividing the input into a set of subvolumes that have loop-free Reeb graphs using the join tree of the scalar function and computes the Reeb graph by combining the contour trees of all the subvolumes. Since the key ingredient of this method is a series of union-find operations, the algorithm is fast in practice. Experimental results demonstrate that it outperforms current generic algorithms by a factor of up to two orders of magnitude, and has a performance on par with algorithms that are catered to restricted classes of input. The algorithm also extends to handle large data that do not fit in memory.

Index Terms—Computational topology, scalar functions, Reeb graphs, level set topology, out-of-core algorithm

1 Introduction

The Reeb graph of a scalar function is obtained by mapping each connected component of its level sets to a point. Level set components that contain critical points of the function map to nodes of the graph. The abstract representation of the level set topology in the Reeb graph facilitates the development of methods for modeling objects and visualizing scientific data. Reeb graphs and their loop-free version, called contour trees, have a variety of applications including computer aided geometric design [1], [2], [3], [4], topology-based shape matching [5], topological simplification and cleaning [6], [7], [8], [9], surface segmentation and parameterization [10], [11], [12], and efficient computation of level sets [13]. The Reeb graph serves as an effective user interface for selecting meaningful level sets [14], [15], for designing transfer functions for volume rendering [16], [17], [18], [19] and for exploring high-dimensional data [20], [21].

Rapidly increasing data sizes and the interactivity requirement in the above mentioned applications necessitate the development of algorithms for fast computation of Reeb graphs that are capable of handling relatively large input sizes. Further, in several cases the domain is not simply connected, is non-manifold, and may be high dimensional. While an efficient and fast algorithm is available for computing contour trees in all dimensions, such an algorithm for computing Reeb graphs is still elusive. In this paper, we attempt to solve this problem by aggressively employing the contour tree algorithm to construct the Reeb graph. This approach results in an algorithm that is efficient both theoretically, in terms of the worst case running time, and practically, in terms of performance on real-world data. The algorithm is also amenable to handle large data that do not fit in memory.

- *H. Doraiswamy is with the Department of Computer Science and Automation, Indian Institute of Science, IISc PO, Bangalore 560012, India. E-mail: harishd@csa.iisc.ernet.in.*
- *V. Natarajan is with the Department of Computer Science and Automation and Supercomputer Education and Research Centre, Indian Institute of Science, IISc PO, Bangalore 560012, India. E-mail: vijayn@csa.iisc.ernet.in.*

Manuscript received 6 Sept. 2011; revised 14 Jan. 2012; accepted 9 Apr. 2012; published online 18 Apr. 2012.
Recommended for acceptance by S. Takahashi.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org, and reference IEEECS Log Number TVCG-2011-09-0218.
Digital Object Identifier no. 10.1109/TVCG.2012.115.

1.1 Related Work

Several algorithms have been proposed for computing the Reeb graph of a scalar function. We refer the reader to the following surveys [22], [23], [24], [25] for a detailed discussion of these approaches. In this section, we restrict the discussion to those algorithms that produce provably correct Reeb graphs. We categorize the algorithms based on how they partition the input domain to analyze the topology of the level sets. Table 1 summarizes the properties and worst case running time of the best known Reeb graph computation algorithms.

The Reeb graph of a scalar function defined on a simply connected domain is called a contour tree. Carr et al. [26] describe an elegant $O(v \log v + n\alpha(n))$ algorithm for computing contour trees that works in all dimensions. Here, v is the number of vertices and n is the number of triangles in the input. This algorithm uses a series of union-find [34] operations to track the connectivity of the super-level sets and sub-level sets, and constructs a join tree and split tree, respectively. These two trees are merged to generate the contour tree. Chiang et al. [35] propose an output sensitive (OS) approach that first finds all component critical points using local neighborhoods and connects these critical points using monotone paths to obtain the join and split trees. This algorithm has a running time of $O(t \log t + n)$, where t is the number of critical points of the input.

Early algorithms for computing Reeb graphs followed the direct approach of tracking its level sets with increasing/decreasing function values during a sweep of the input. Shinagawa and Kunii proposed the first algorithm for

TABLE 1
Comparison of Various Algorithms for Computing Reeb Graphs Based on the Type of Input They Can Handle, Worst Case Running Time, and Approach

Algorithm	Input Type	Time Complexity	Approach	Notes
Carr et al. 2003 [26]	simply connected d-dimensional simplicial complex	$O(v\log v+n\alpha(n))$	track components of sub-level sets and super-level sets using a series of union-find operations	Computes contour trees. It requires data in memory.
Cole-Mclaughlin et al. 2004 [27]	2-manifolds	$O(n\log n)$	sweep and explicitly maintain level sets	Algorithm does not extend to three and higher dimensions. It requires data in memory.
Doraiswamy et al. 2009 [28]	3-manifolds d-manifolds	$O(n\log n+n\log g(\log\log g)^3)$ $O(n\log n(\log\log n)^3)$	sweep and explicitly maintain level sets	Has best known theoretical bound on running time. It requires data in memory.
Pascucci et al. 2007 [29]	arbitrary simplicial complex	$O(n^2)$	explicitly maintain level sets	Has the best performance for 2D data and is capable of handling large data sizes.
Harvey et al. 2010 [30]	arbitrary simplicial complex	$O(n\log n)$ expected time	collapse triangles	Requires data in memory.
Patanè et al. 2009 [31]	2-manifold	$O(ns)$	split and compute	Time complexity degenerates to $O(n^2)$ in the worst case. It requires data in memory.
Doraiswamy et al. 2011 [32]	arbitrary simplicial complex	$O(n+l+t\log t)$	split and compute	Time complexity degenerates to $O(n^2)$ in the worst case. It requires data in memory.
Tierny et al. 2009 [33]	3-manifolds embedded in $\mathbb{R}^3$	$O(n\log n+hn)$	split and compute	1. Has the best performance for such input. 2. Time complexity degenerates to $O(n^2)$. 3. Requires data in memory.

v is the number of vertices in the input, n is the number of triangles, t is the number of critical points, s is the number of saddles, l is the total size of all critical level sets, g is the maximum genus over all level sets, and h is the number of loops in the Reeb graph.

constructing the Reeb graph of a scalar function defined on a triangulated 2-manifold [36] in $O(n^2)$ time. This algorithm explicitly tracks connected components of the level sets. Cole-McLaughlin et al. [27] store the level sets using balanced search trees and improved the running time to $O(n\log n)$. Doraiswamy and Natarajan [28] follow a similar approach to store the connected components of level sets using dynamic connectivity data structures resulting in an algorithm that computes the Reeb graph of a 3D scalar function in $O(n\log n+n\log g(\log\log g)^3)$ time. Here, g is the maximum genus over all level sets of the input function. They extend this approach to higher dimensional manifolds and design a $O(n\log n(\log\log n)^3)$ time algorithm. While this algorithm has the best known theoretical bound on the running time, the sophisticated data structures used in the algorithm do not lend themselves to efficient implementations in practice.

Pascucci et al. [29] propose an online algorithm that constructs the Reeb graph for streaming data. Their algorithm takes advantage of the input coherence to construct the Reeb graph efficiently. In a streaming model, where triangles are processed during a single pass through triangles in the input mesh, 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 for 2D scalar functions. However, the optimizations that result in fast incremental construction of Reeb graphs for 2D data do not provide a performance benefit in higher dimensions.

Harvey et al. [30] propose a randomized algorithm that computes the Reeb graph of an arbitrary simplicial complex. They repeatedly collapse all triangles constituting the level set component of randomly chosen vertices resulting in a reduced input whose Reeb graph is equal to that of the original scalar function. This algorithm has an expected running time of $O(n\log n)$.

Other recent algorithms follow an approach that explicitly split the input, compute the Reeb graph for each subdomain, and stitch the graphs together to obtain the Reeb graph of the input. Patanè et al. [31] focus on 2-manifolds and propose a contouring approach to compute the Reeb graph in $O(ns)$ time, where s is the number of saddles in the input. Tierny et al. [33] perform a surgery on a 3-manifold domain that cuts open all handles on the domain's boundary, thereby reducing the problem to the computation of contour trees. This approach leads to a very efficient algorithm that computes the Reeb graph in $O(n\log n+hn)$ time, where h is number of loops in the Reeb graph. This algorithm however works only on 3-manifolds that are embedded in $\mathbb{R}^3$. Doraiswamy and Natarajan [32] propose a two-step output-sensitive algorithm to compute the Reeb graph for both manifold and non-manifold input. They use an alternate definition of the Reeb graph that maps the nodes of the Reeb graph to level set components of critical points, and its arcs to the interval volume between these critical level sets. This results in a $O(n+l+t\log t)$ algorithm, where l is the size of all critical level sets and t is the number of critical points.

The methods used by the output sensitive [32] and loop surgery [33] algorithms can be considered as two extremes of the "split and compute" approach. The output sensitive approach splits the domain at all saddles resulting in loop-free subdomains whose contour trees consists of a single arc. These arcs are subsequently stitched together to form the Reeb graph of the input. On the other hand, the loop surgery method splits the domain only at potential loops resulting in a single loop-free domain. The contour tree of this surgically modified domain is computed and processed to close the loops. Even though the worst case running times of these algorithms degenerate to $O(n^2)$, they were shown to perform better than their sweep-based counterparts in practice. This can be attributed to the fact that both methods are able to achieve significant speed up by reducing the problem to that of computing contour trees. In both algorithms, explicitly storing the splits causes the memory required to increase linearly with the number of saddles or loops. This results in a large overhead in practice when several triangles span a large function range and thus are repeatedly stored at multiple splits. In this paper, we adopt an approach that simulates an optimal number of splits to maximally leverage the benefit of the contour tree algorithm. This approach results in an algorithm to compute the Reeb graph that outperforms existing algorithms and is applicable to manifold and non-manifold domains in any dimension.

1.2 Results

We present a fast and efficient algorithm that computes the Reeb graph of a piecewise-linear (PL) function in $O(v \log v + sn)$ time, where v is the number of vertices, n is the number of triangles and s is the number of saddles in the input. The algorithm first identifies loops in the Reeb graph using a combination of neighborhood-based critical point classification and level set topology-based classification of critical points. The algorithm then implicitly splits the domain into a set of subvolumes whose Reeb graphs are loop-free, and constructs the Reeb graph of the input by combining the Reeb graphs (contour trees) of these subvolumes. The algorithm has the following properties:

- Efficient in practice. Experimental results indicate that the algorithm is up to two orders of magnitude faster than existing algorithms.
- Easy to implement. Potential loops in the Reeb graph are identified using the join tree of the input. The main operation performed by the algorithm is a series of simple union-find operations.
- Works without any modification on d-manifolds ($d \geq 2$) and non-manifolds.
- Additional memory required is $O(n)$. The algorithm does not explicitly store the splits performed. We notice that in practice the algorithm uses only $O(s)$ additional memory. The number of saddles, s, is usually much smaller than n.
- Handles large data that do not fit in memory.

We perform extensive experiments to demonstrate the efficiency of our algorithm. More specifically, we show that our algorithm outperforms existing generic algorithms, and is comparable with algorithms that are specialized for restricted classes of the input.

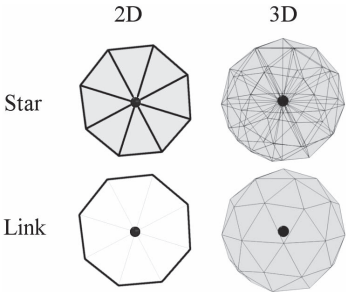

Fig. 1. Local neighborhood of a vertex in a 2D and 3D mesh. The star of the vertex represents its local neighborhood and consists of all simplicies incident on it. The link of a vertex in 2D is a triangulation of a circle, and in 3D, it is a triangulation of a sphere.

2 Background

In this section, we briefly introduce some of the necessary definitions and refer the reader to appropriate textbooks [37], [38], [39] for more detailed definitions and discussions of these concepts.

2.1 Simplicial Complex and PL Functions

A d-simplex σ is the convex hull of $d+1$ affinely independent points. A simplex τ is a *face* of σ if it is the convex hull of a subset of the $d+1$ points and is denoted as $\tau \leq \sigma$. A simplex σ is called the *coface* of τ if τ is a face of σ. A simplicial complex K is a finite collection of nonempty simplices that satisfies two properties.

1. $\sigma \in K$ and $\tau \leq \sigma$ implies $\tau \in K$,
2. $\sigma_1, \sigma_2 \in K$ implies that $\sigma_1 \cap \sigma_2$ is either empty or a face of both σ_1 and σ_2.

We assume that the input to our algorithm is a triangulated mesh represented by a simplicial complex K together with a piecewise-linear function $f\colon K \rightarrow \mathbb{R}$. The function is defined on the vertices of K and linearly interpolated within each simplex.

The *star* of a vertex u consists of the set of cofaces of u. All simplices in the star in which the function values are greater than at u constitute the *upper star*. All simplices in the star in which the function values are lower than at u constitute the *lower star*. The *link* of a vertex u is the set of all faces of simplicies of its star that are disjoint from u. It consists of all vertices adjacent to u 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*. Fig. 1 shows the star and link for vertices in 2D and 3D meshes.

2.2 Critical Points and Morse Functions

Let $\mathbb{M}$ denote a d-manifold with or without boundary. Given a smooth, real-valued function $f : \mathbb{M} \rightarrow \mathbb{R}$ defined on $\mathbb{M}$, the *critical points* of f are exactly where the gradient becomes zero. The function f is called a *Morse function* if it satisfies the following conditions [27]:

1. All critical points of f are non-degenerate and lie in the interior of $\mathbb{M}$.
2. All critical points of the restriction of f to the boundary of $\mathbb{M}$ are non-degenerate.

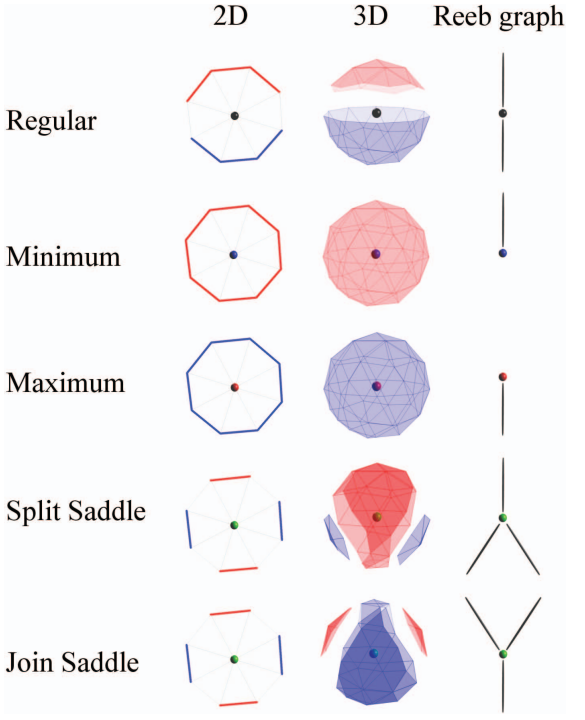

Fig. 2. Classifying a vertex as regular, minimum, maximum, or saddle using the topology of its local neighborhood. The lower link of a vertex is colored blue; its upper link is colored red. The structure of the Reeb graph is shown in the right column. The connectivity of the level sets further classifies the saddle as a split or join saddle.

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

Critical points of a Morse function can be classified based on the behavior of the function within a local neighborhood. Banchoff [40] and later Edelsbrunner et al. [41] extend these ideas to PL functions and describe a combinatorial characterization for its critical points, which are always located at vertices of the mesh. We describe this characterization in the following section after introducing level sets and their connectivity.

2.3 Level Set Topology

The preimage $f^{-1}(a)$ of a real value a of a PL function f 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. A *sub-level set* of a real value a is the preimage of the interval $(-\infty, a]$, while the *super-level set* of a is the preimage of the interval $[a, +\infty)$. We are interested in the evolution of level sets against increasing function value. Topological changes occur at critical points, whereas topology of the level set is preserved across regular points [42].

Critical points are characterized by the number of connected components of the lower and upper links, as shown in Fig. 2. The vertex is *regular* if it has exactly 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. Else, it is classified as a *saddle*.

2.4 Reeb Graphs

The *Reeb graph* of f is obtained by contracting each connected component of a level set to a point [43]. Formally, it is the quotient space under an equivalence relation that identifies all points within a connected component of a level set. 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. 3 shows the Reeb graph for the height function defined on a solid vase.

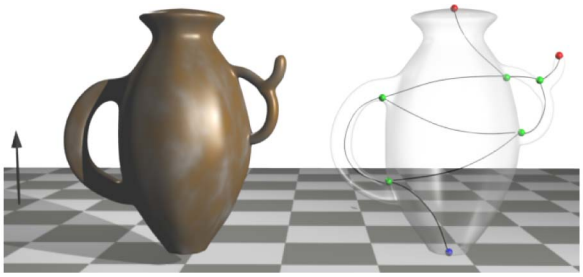

Fig. 3. The Reeb graph for the height function defined on a solid vase.

Consider a sweep of the input in decreasing order of function value. A vertex is a *join saddle* if two-level set components merge at that vertex during the sweep. It is a *split saddle* if a single-level set component splits into two components at that vertex. Fig. 2 illustrates the local structure of the Reeb graph at various types of nodes. Nodes corresponding to minima and maxima have degree one, while a node that corresponds to a simple join or split saddle has degree three. Other simple saddles have degree two, and do not modify the number of connected components of the level set. In the context of Reeb graphs, we are interested only in critical points that modify the number of level set components.

The conditions for a Morse function typically do not hold in practice for PL functions. Degeneracy in PL functions results in the presence of saddles that have degree greater than three in the Reeb graph. In order to simplify the description, we assume that all saddles are degree-2 or degree-3 nodes in the following sections. The extension of the algorithm to handle higher degree saddles is straightforward and is described in Section 3.4. Simulated perturbation of the function [44, Section 1.4] ensures that no two critical values are equal. The simulated perturbation imposes a total order on the vertices, which helps in consistently identifying the vertex with the higher function value between a pair of vertices. Regular vertices and degree-2 saddles are often inserted into the Reeb graph as degree-2 nodes to obtain the *augmented Reeb graph*.

2.5 Contour Trees

The Reeb graph of a simply connected domain has no loops and is called a *contour tree*. The key idea in our proposed approach to compute Reeb graphs is to optimize the use of the efficient contour tree algorithm. For completeness, we briefly describe the contour tree algorithm. For a more detailed description, we refer the reader to the paper by Carr et al. [26].

The contour tree algorithm makes two passes over the data to compute the *join tree* and *split tree* of the input. The join tree tracks the connectivity of super-level sets of the input scalar function and identifies all the maxima and join saddles of the contour tree. It is computed by first sorting the vertices of the input in decreasing order of

function value. Next, for each vertex u in this sorted list, the algorithm performs the following operations:

- If u is a maximum (its upper link is empty), create a new component containing u and set u as its *head*.
- If the upper link is not empty, find the components that contain the vertices in the upper link of u. Add an arc between u and the head of each of the components. Merge these components and set u as the head of the merged component. If the number of components is greater than one then u is a join saddle.

Similarly, the split tree tracks the connectivity of the sublevel sets and identifies the set of minima and split saddles. It is computed by traversing the vertices in increasing order of function values. The join and split trees are merged to obtain the contour tree.

3 Algorithm

This section describes our Reeb graph computation algorithm. For ease of explanation, we illustrate the algorithm using the solid vase model (3-manifold with boundary) shown in Fig. 3. We note that the algorithm works without any modifications for d-manifolds, $d \geq 2$, and non-manifolds. The input to our algorithm is a triangle mesh representing the input domain together with scalar values defined at vertices of the mesh. The algorithm computes the Reeb graph of this input in four stages.

1. Identify the loop saddles of the input.
2. Split the input at a function value infinitesimally above that of the loop saddles to obtain a set of interval volumes.
3. Compute the contour trees for each interval volume.
4. Construct the Reeb graph by computing the union of these contour trees.

The rest of this section is organized as follows: We first describe the characterization used for identifying loop saddles in Section 3.1, followed by a detailed description of the algorithm in Section 3.2. We analyze the time and space complexity of the algorithm in Section 3.3. Extension of the algorithm to handle saddles with degree greater than three is described in Section 3.4. Finally, in Section 3.5, we discuss the generality of our algorithm in handling d-manifold (d $\geq$ 2) and non-manifold input.

3.1 Loop Saddles

A *chord* in an undirected graph is an edge that connects two nodes of a cycle in the graph, but is not part of the cycle. A cycle is an *induced cycle* if the subgraph induced by the nodes of the cycle does not contain a chord [45]. *Loops* in a Reeb graph correspond to the set of all induced cycles in the graph.

Consider any loop L in the augmented Reeb graph of the given input. Define the set V_L as the set of all vertices of the input that belong to loop L. Fig. 4b shows the Reeb graph, corresponding to a loop in the vase model shown in Fig. 4a, augmented with degree-2 nodes. The set V_L for this loop is highlighted in cyan. Define $c_j = \inf(V_L)$ and $c_s = \sup(V_L)$ where the vertices are ordered based on the function values. If we sweep the input with decreasing function value, then the split saddle c_s *begins* the loop L, while the join saddle c_j *ends* it. We are interested in finding all such *loop saddles*—a set of saddles that begin or end a loop in the Reeb graph.

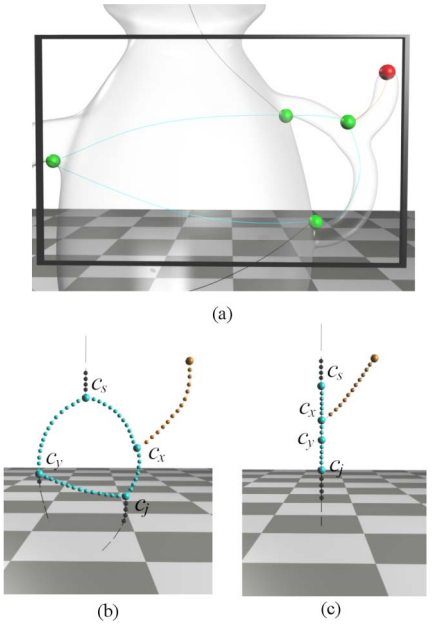

Fig. 4. Join saddles that end a loop in the Reeb graph form degree-2 nodes in the join tree. (a) A loop in the vase model is highlighted. (b) The Reeb graph within the highlighted region augmented with degree-2 nodes. (c) The augmented join tree corresponding to the loop. Note that c_j, the join saddle that ends the loop, forms a degree two node in the join tree.

The function values at these saddles are in turn used to obtain a set of loop-free interval volumes. This is accomplished by splitting the input at these saddles.

The set of all join or split saddles is a superset of the set of loop saddles. Using it as a conservative estimate while splitting the input domain may lead to an unnecessarily large number of interval volumes. In order to reduce this overhead, we utilize the contour tree algorithm to find a better estimate of the set of loop saddles. The following lemma provides us with the necessary condition to compute this set.

Lemma 1. *Let G_R be the Reeb graph of a scalar function f. Consider the join tree T_J of f. Any join saddle that ends a loop in G_R appears as a degree-2 node in T_J.*

Proof. Consider a split saddle c_s that begins a loop L in G_R. Let this loop end at the join saddle c_j. Let V_L be the set of vertices that belong to the loop L. Consider the algorithm that computes the join tree T_J of f and a vertex $u_j \in V_L$, with $f(u_j) < f(c_s)$ and $u_j \neq c_j$, that is processed by the algorithm. Let $u_k \in Lk^+(u_j)$ be a vertex in the upper link of u_j such that $u_k \in V_L$. Due to the way the join tree algorithm works, u_k will be in the same component as c_s. This is true for all vertices in V_L.

Now, consider the step when c_j is processed by the join tree algorithm. Its upper link $Lk^+(c_j)$ will consist of two components. Vertices in both components belong to L, and will therefore belong to the same component as c_s. Hence, the join tree algorithm will add only a single arc between c_j and the head of this component to T_J. For every node in the join tree T_J, the number of neighbors with function value less than the node is always one. Hence, the degree of c_j in the join tree T_J is two. □

Fig. 4c shows the join tree corresponding to the loop in Fig. 4a where c_j is a degree-3 node in the Reeb graph and a

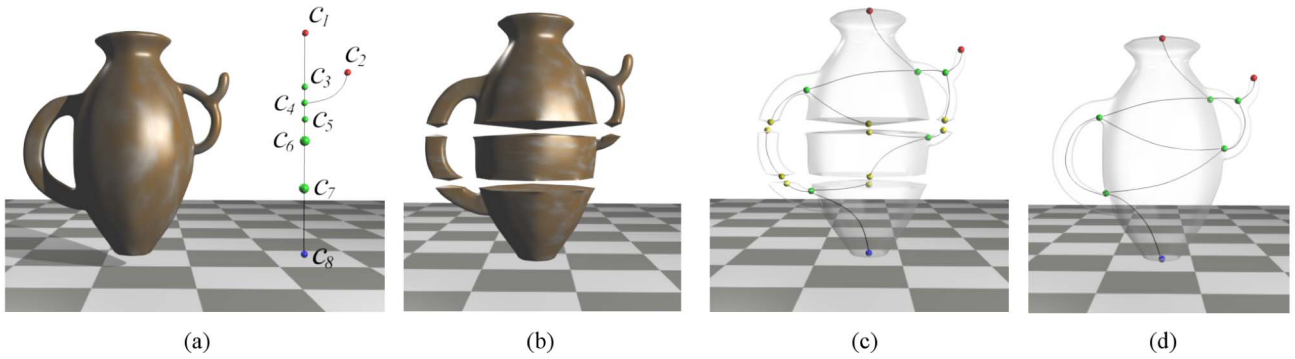

Fig. 5. The Reeb graph computation algorithm. (a) The join tree for the input is computed and the loop saddles are identified. Here, c_6 and c_7 are identified as loop saddles. (b) The input is split at a function value infinitesimally above that of the loop saddles to obtain a set of interval volumes. (c) The contour tree for each interval volume is computed. (d) The contour trees are merged to obtain the Reeb graph of the input.

degree-2 node in the join tree. A similar proof can be used to show that any split saddle that begins a loop appears as a degree-2 node in the split tree.

3.2 Computing the Reeb Graph

We now explain the first two steps of the Reeb graph computation algorithm in detail, while the third and fourth steps of the algorithm are straightforward.

3.2.1 Step 1: Identify Loop Saddles

The algorithm first identifies all potential loops of the Reeb graph. Each loop is represented by the join saddle that ends the loop. The algorithm computes the set of loop saddles as follows:

1. Compute all critical points of the input. This is accomplished by counting the number of components in the upper and lower links of every vertex via a breadth-first search in the graph formed by vertices and edges in the upper and lower links, respectively. Fig. 5a highlights the critical points of the vase input. The blue, green, and red nodes denote the set of minima, saddles, and maxima, respectively.
2. Compute a superset S of the set of join saddles. This set S is equal to the set of all critical points whose upper link consists of two components. For the vase input, $S = \{c_4, c_6, c_7\}$.
3. Compute the join tree T_J of f. The join tree of the vase input is shown on the right in Fig. 5a.
4. Identify the set of loop saddles. A superset of the set of loop saddles S_L is obtained using the characterization from Lemma 1. It is defined as $S_L = \{c_i \in S \mid deg(c_i)\ in\ T_J = 2\}$. The join saddles c_6 and c_7 correspond to loop saddles in Fig. 5a, since they form degree-2 nodes in the join tree.

In case of a 2-manifold input, the set of split saddles of the input are also identified as loop saddles. This is because the upper and lower links of all saddles in such input consists of two components. Similarly, degree-2 saddles present in 3D and higher dimensional input are included in the set of loop saddles. However, such false positives can be identified and eliminated when splitting the input.

3.2.2 Step 2: Split the Input

For each potential loop saddle c_j in S_L, the algorithm splits the input at a function value $f(c_j) + \epsilon$, for an appropriately small value of ϵ. The set of interval volumes thus obtained have the property that their Reeb graphs do not contain loops, and can therefore be computed using the contour tree algorithm of Carr et al. This operation is illustrated in Fig. 5b. It is accomplished during a sweep of the vertices in the input in increasing order of function value. While processing each vertex u in this sweep, the algorithm maintains the set of all triangles T that contains the level set at $f(u)$. The set T is initially empty. Each vertex u is processed as follows:

1. Remove the triangles in the lower star of u from T.
2. Add the triangles in the upper star of u to T.
3. If $u \in S_L$, perform a breadth-first traversal along adjacent triangles in T to obtain the set of connected components of the level set at $f(u) + \epsilon$.
4. Split the input along this level set, creating a new maximum-minimum pair for each component.

In step (3) above, if the traversal starting from one component of a loop saddle c_j's upper link reaches the other component, then it implies that the level set at $f(c_j) + \epsilon$ contains just one component, and c_j is therefore not a join saddle. When the algorithm encounters such a situation, it classifies c_j as a false positive, and does not split the input at that vertex.

The split performed in step (4) is realized by "cutting" each triangle in the maintained set T and connecting the edges of the level set to two new extrema, a maximum and a minimum, thereby creating a set of triangles that belong to either the lower star of the maximum or the upper star of the minimum. Fig. 6 shows the set of created triangles which are part of the lower star of a new maximum. Vertices colored

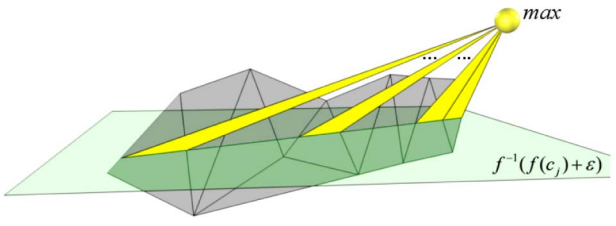

Fig. 6. The split operation performed on triangles (gray) that contain the level set (green) at a function value infinitesimally above that of a loop saddle. The yellow triangles denote the lower star for the new maximum that is inserted. Each triangle in this star contains an edge that lies within a gray triangle. The lower star of the new maximum is represented using the gray triangles.

yellow in Fig. 5c denote all the additional extrema created for the vase input.

Each new triangle corresponds to an edge in the level set, which can in turn be uniquely identified by a triangle in the input mesh. This property is utilized by our algorithm to represent each new triangle using an existing triangle. Also, the lower star of a new maximum (respectively, the upper star of a new minimum) is a connected component of a level set. This fact enables the algorithm to represent all triangles in the lower star of the maximum (respectively, upper star of the minimum) using a single triangle that contains this level set component. When required, the other triangles in the star are obtained by marching along adjacent triangles starting from the representative. If a triangle is already split then it implies that this triangle spans a function range that encompasses more than one loop, and is therefore not split a second time.

3.2.3 Steps 3 and 4

The algorithm computes the contour trees of the individual interval volumes (Fig. 5c) and constructs the Reeb graph of the input by merging the maximum-minimum pairs created in Step 2 (Fig. 5d).

3.3 Analysis

We now analyze the running time of the algorithm and the space required in the worst case scenario.

3.3.1 Time Complexity

Let v be the number of vertices, n be the number of triangles and s be the number of saddles in the input. Identifying the loop saddles requires the computation of critical points which in turn requires the computation of the number of connected components of the lower and upper links of each vertex. This takes $O(n)$ time if the mesh is represented using the triangle-edge data structure [46]. Computing the join tree requires sorting the input that takes $O(v \log v)$ time, and a set of $O(n)$ union-find operations, that takes $O(n\alpha(n))$ time. Here, α is the inverse Ackermann function.

In order to obtain the set of loop-free interval volumes, the algorithm performs at most $|S_L|$ cuts to the input. The size of the set S_L is bounded by the number of saddles s. Each cut requires the traversal of a level set, whose size is at most n. Hence, the time required to perform these cuts is bounded by $O(sn)$. The new extrema are then inserted into the sorted list of vertices at the appropriate positions.

Computing the set of contour trees requires computing the join and split trees of each interval volume. Since the set of critical points are already identified and sorted, the join and split trees can be computed in $(n + sn)$ time using monotone paths [35]. Reconstructing the Reeb graph from the set of contour trees is done by merging the various maximum-minimum pairs, which can be done in $O(n)$ time. Combining the above steps, we obtain an $O(v \log v + sn)$ bound on the running time of the algorithm.

3.3.2 Space Complexity

The triangle-edge data structure stores the triangles adjacent to each edge in the form of a triangle fan. This requires storing pointers to previous and next triangle for each edge of a triangle, a total of $6n$ pointers. The algorithm requires the star of each vertex in order to classify the critical points. For manifold input, this can be accomplished by traversing adjacent triangles using the triangle-edge data structure. But this method does not work for non-manifold input where the star of a vertex could consist of multiple components. Hence, all triangles in the star of each vertex need to be stored, a total of $3n$ triangles.

While splitting the domain into interval volumes, the algorithm stores a seed triangle for each component of the $|S_L|$ level sets. The number of such components is bounded by n. The auxiliary data structures required during the computation, such as the union-find data structure, vertex list, the edge lists representing the join and split trees, and the reconstructed Reeb graph together requires $O(n)$ space. Thus the over all space required for computing the Reeb graph is bounded by $O(n)$.

3.4 Handling Higher Degree Saddles

A PL function may contain saddles that appear as Reeb graph nodes with degree greater than three. The sum of the number of components in the upper and lower link of such a saddle is greater than three. Loop saddle identification, based on Lemma 1, extends to these higher degree saddles. Consider a higher degree saddle which merges m level set components and ends k loops. The degree of the corresponding node in the join tree becomes $m - k + 1$.

We extend our algorithm to handle higher degree saddles by modifying the loop saddle identification step as follows—given a join saddle c_j having m components in its upper link, if the degree of c_j in the join tree T_J is less than $m + 1$, then c_j is added to the set S_L. The rest of the algorithm remains unchanged.

3.5 d-Manifolds and Non-Manifolds

The connectivity of a level set is represented by its 1-skeleton. Therefore, tracking the connected components of the level set requires only the edges of the level set, which can be extracted from the triangles of the input mesh. So the algorithm works without any modifications for a d-manifold input represented by its triangles.

The algorithm expects the input to be a collection of triangles. In case of non-manifold input, edges that are not incident on any triangle in the input are replaced by a triangle, with the additional vertex of this new triangle having a function value equal to the average of the function values of the two end points of the edge [32]. This modification of the input does not affect the Reeb graph of the input because the newly introduced vertex is regular. Candidate critical points are again located by counting the number of connected components of the lower and upper link. Note that the size of the input remains unaffected asymptotically.

4 Implementation

In this section, we describe some of the design choices made during the implementation to handle generic input and to improve the performance of our algorithm. Our implementation accepts a function sampled at vertices of a simplicial mesh as input and computes the Reeb graph, which is stored as a set of arcs. In Section 4.1, we discuss the implementation issues related to handling input data in higher dimensions. We then describe the in-memory

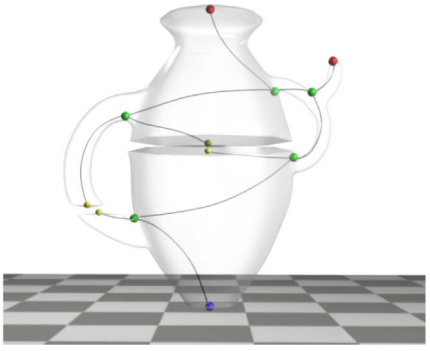

Fig. 7. Our in-memory implementation performs splits on one of the arcs corresponding to a loop saddle. The maximum-minimum pairs shown in yellow denotes the two cuts performed by this modified procedure for the vase input instead of the five cuts shown in Fig. 5c.

implementation of the algorithm in Section 4.2, followed by its extension to handle large data in Section 4.3.

4.1 Triangle-Edge Data Structure

The triangle-edge data structure stores the triangles of the input sorted around an edge. While this is feasible for two and 3D input, there is no natural order among triangles for higher dimensional data. Our algorithm uses the triangle-edge adjacencies only for breadth-first traversal performed during the split operation. Since this operation does not depend on the ordering of the triangles around an edge, we arbitrarily order the triangles around each edge.

4.2 In-Memory Implementation

We modify the Reeb graph computation algorithm to take advantage of the fact that the entire data set along with all the auxiliary data structures is available in memory. Instead of splitting the entire level set at a loop saddle, it suffices to split the level set component corresponding to one of the arcs of the loop. We compute the contour tree of the resulting connected loop-free subvolume, merge the newly inserted maximum-minimum pairs to obtain the loops and hence construct the Reeb graph. Fig. 7 illustrates this modified procedure on the vase input, where we perform cuts only on two components of the level set instead of the five components as required by the unmodified algorithm. The use of this modified procedure not only reduces the overhead of additional splits, but also enables us to perform further optimizations to our implementation.

The in-memory implementation first sorts the vertices of the input in increasing order of function value. This sorted set is stored as a list. Finding all critical points followed by computing the join tree of the input to identify loop saddles would require two passes over the data, while computing the join tree of the split domain would require an additional pass. Our implementation combines all three operations into a single pass, thus saving processing time. To achieve this, we modify the join tree/split tree computation algorithm as follows.

4.2.1 Computing the Join Tree

A key difference between the join tree algorithm described by Carr et al. and our implementation is that we keep track of the components of the super-level sets using triangles instead of vertices. While processing each vertex, it is first classified as regular or critical and processed accordingly.

- **Regular.** The component containing the vertex is obtained from the triangles incident on its upper link. All triangles in the lower star of the vertex are assigned to this component.
- **Maximum.** A new component is created and all triangles in the lower star are assigned to this component. The maximum is assigned as the head of this component.
- **Join saddle.** The triangles incident on the upper link of a join saddle c_j belong to m ($m \geq 2$) level set components. m arcs are inserted into the join tree, between c_j and the heads of the m components. The components are then merged, all triangles in c_j's lower star assigned to the resulting component, and c_j is set to be the head of this component.
- **Loop saddle.** The vertex is essentially a join saddle c_j, where all triangles incident on at least two components of the upper link belong to a single component of the super-level set. Let the number of upper link components sharing a super-level set component be k. This results in a join saddle that closes $k-1$ loops with respect to the super-level set component. The loop is split along $(k-1)$ components of the level set passing through the upper star of c_j, to create the required maximum-minimum pairs. The function value at each maximum is set to $f(c_j)+\varepsilon$, while that at each minimum is set to $f(c_j)+2\varepsilon$ where ε is an infinitesimally small positive value. These extrema are inserted into the sorted list after c_j. The new minima and maxima are then processed before processing c_j again. Note that when c_j is processed the second time, it will be processed as a normal join saddle. Each time a level set traversal starting from one component of the upper link reaches another component, the value of k is decreased by one. If k becomes one, then this loop saddle is classified as a false positive, and processed as a regular vertex.
- **Split saddle.** The triangles incident on the upper link of a split saddle c_s belong to a single component. An arc is inserted into the join tree between c_s and the head of that component. The triangles in the lower star of c_s are then added to this component and c_s is set as its head.
- **Minimum.** The triangles in the upper star of the minimum belong to a single component. An arc is inserted into the join tree between the minimum and the head of this component. The minimum is set to be the head of the component.

The split tree is computed in a similar manner. Note that since the domain is already split while computing the join tree, it is not necessary to check for loop saddles and perform any additional processing while computing the split tree. The contour tree is computed using the merge procedure as described by Carr et al., and finally the newly introduced maximum-minimum pairs are merged to obtain the Reeb graph.

The advantage of using triangles to track the super-level and sub-level set components is two-fold—1) the first, second, and third steps of the algorithm are executed during a single pass instead of three passes, and 2) additional

vertices need not be generated for each triangle that is cut, an operation that is necessary for computing the join and split trees using the algorithm described by Carr et al. This results in savings on the memory used by the algorithm.

4.3 Handling Large Input

We now discuss a direct extension of our algorithm to compute Reeb graphs for large data that do not fit in memory. The main idea is to split the input into interval volumes that fit in memory, and compute the Reeb graph of the input scalar function by combining the Reeb graphs of the individual interval volumes.

4.3.1 Dividing the Input

In our implementation of the out-of-core algorithm, the input is split into multiple interval volumes based on the input function. The required interval volumes are obtained by dividing the entire function range based on the number of vertices in the input. This requires sorting the vertices of the mesh on the associated scalar values and partitioning them into intervals. Our implementation currently assumes that the vertices fit in memory and performs an in-memory sort. For example, given a memory limit of 4 GB, an input having 500 million vertices can be easily stored in memory. This is because it is necessary to store only the function values of each vertex along with the pointer reference to its sorted position, a total of 8 bytes per vertex.

The triangles are assigned intervals depending on the lowest function value of its vertices. Note that a triangle can belong to multiple intervals, in which case it gets carried over to the appropriate intervals when processing those intervals. One persistent file is created for each interval and triangles belonging to that interval are written into the file. This process requires one pass over the input data.

4.3.2 Computing the Reeb Graph

Our implementation first computes the Reeb graph of each interval volume, which now fits in memory. The intervals are processed in increasing order of function value. Each interval creates a maximum-minimum pair for every arc of the Reeb graph that spans that interval volume's upper boundary. Each extrema pair have an associated set of triangles that belong to their star. These triangles span more than one interval volume, and are retained in memory until all interval volumes that overlap their span are processed. A component consisting of an extrema pair together with its associated triangles is called a *boundary component*.

The in-memory implementation is used to compute the Reeb graph of each interval volume. In practice, many boundary components may not be accessed while processing a particular interval. Such situations occur when triangles of the boundary component are not incident on any vertex within the interval volume. Retaining all such components in memory can potentially result in exceeding the available memory. To avoid such situations, these unused boundary components are stored in persistent temporary files until they are necessary. This requires an additional step of determining the interval volume that requires a particular boundary component and storing it appropriately.

Once the Reeb graph of all interval volumes are computed, the Reeb graph of the input is constructed by merging the additional maximum-minimum pairs that were created.

4.3.3 Handling Large Interval Volumes

A potential problem with the method described above to handle large data is that, storing the entire interval volume in memory is not feasible when it is large. Such a situation may occur, for example, when a single level set is large. We have noticed that in practice, such interval volumes typically consist of multiple components. Our implementation stores these components in persistent storage and loads them into memory only when required. While this strategy handles almost all memory issues, this method will still run into trouble if a single level set component cannot fit into memory. Also, if large level sets are common then the I/O overhead increases, potentially slowing down the Reeb graph computation. A possible solution, we are currently exploring in order to avoid such issues is to use the collapse operation proposed by Harvey et al. [30], but applied to a single level set. Since this operation essentially collapses triangles, it reduces the size of the level sets, thus allowing the algorithm to handle large level set components.

5 Experimental Results

The in-memory variant of our Reeb graph computation algorithm is implemented in C++, while we use Java to implement the out-of-core version. We evaluated the performance of our implementations on an Intel Xeon workstation with a 2.0 GHz processor and 16 GB main memory. First, we report the performance of our algorithm when working with data in memory and compare it with existing algorithms. We then discuss the performance of the out-of-core implementation on large data. For the out-of-core experiments, we restrict the memory availability of the program to 4 GB. For all comparisons with existing algorithms, we used the implementations provided by the respective authors. In cases where the implementation was not available, we use the timings from the corresponding paper. In the remaining discussion, we refer to our implementation as RECON.

5.1 In-Memory Experiments

5.1.1 2- and 3-Dimensional Data.

Table 2 shows the time taken by RECON to compute the Reeb graph for surface meshes. We compare our algorithm with the online algorithm (ONLINE) [29], which exhibits the best performance for 2D meshes among existing algorithms, the output sensitive algorithm [32], and the randomized algorithm (RAND) [30]. Our algorithm is an order of magnitude faster than the generic algorithms, and at least 50 percent faster than the online algorithm.

Table 3 compares RECON with loop surgery (LS) [33], output sensitive and the randomized algorithm for 3D input. We do not consider the online algorithm since it was previously shown to perform poorly for such input [32], [33]. For 3D input, RECON performs at least an order of magnitude faster than the generic algorithms, while its performs at least as fast as loop surgery, the fastest known algorithm for such input. The LS algorithm first computes the Euler characteristics of the 2D boundary of the input. If it detects that the boundary has no loops, then the contour tree is directly computed. The identification of loop saddles is performed only when a loop is detected in the input's boundary. Note that RECON performs at least twice as fast

TABLE 2
Reeb Graph Computation Time for Various 2D Input

2D Model	# Triangles	# Critical Points	# Potential Loop Saddles	# Loops	Time taken (sec) RECON	ONLINE	OS	RAND
Youthful	3.4M	27,627	5,588	506	2.4	6.0	47.2	54.0
Neptune	4.0M	1,752	563	3	2.6	8.7	42.0	37.3
Awakening	4.0M	17,031	2,360	1,643	2.4	6.7	41.4	51.8
Day	6.0M	104,898	12,546	2,161	4.3	10.3	91.3	67.5
Dawn	6.6M	91,640	8,592	757	4.5	11.5	73.2	71.8
Lucy	28.0M	9,521	2,462	15	34.2	60.1	*Mem*	*Mem*

For all models, the Reeb graph was computed for the height function defined by the y-axis. Mem denotes that the algorithm ran out of memory when trying to compute the Reeb graph. RECON is at least 50 percent faster than ONLINE and at least 10 times faster than OS and RAND.

TABLE 3
Reeb Graph Computation Time for Various 3D Input

3D Model	# Triangles	# Critical Points	# Potential Loop Saddles	# Loops	Time taken (sec) RECON	LS	OS	RAND
Fighter (3D)	143,881	6,787	1,717	0	0.2	0.1	123.8	6.5
Blunt fin	451,601	1,921	560	0	0.2	0.3	23.3	12.5
Bucky Ball	2,524,284	6,664	1,230	0	1.2	1.6	197.9	63.6
Plasma	2,646,016	4,719	935	0	1.5	1.9	396.3	132.7
SF Earthquake	4,198,057	20,655	529	0	2.4	2.8	598.1	166.9
Skull	336,296	26	15	2	0.1	0.3	3.4	2.2
Post	1,243,200	247	64	0	0.4	0.9	13.0	14.5
CubeHoles	2,355,234	2,402	1,300	1,200	3.2	*Seg*	97.7	23.0

The scalar function was provided with the data set. RECON is at least one order of magnitude faster and up to two orders of magnitude faster than OS and RAND. The time taken to compute the Reeb graph is comparable for RECON and LS. Seg indicates that the code exited with a segmentation

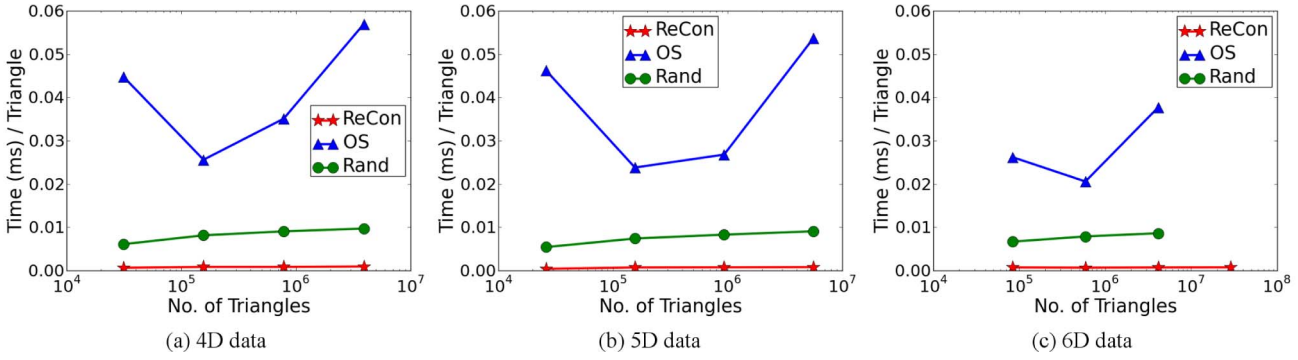

Fig. 8. Comparison of RECON, OS, and RAND algorithms for different sized Sierpinski simplexes in four, five, and six dimensions. The time required to process each triangle remains consistent for RECON with increase in size as well as dimension of the input. Further, the time required is lower than that for OS and RAND

as LS in the latter case, for example in the Skull, Post, and CubeHoles data sets. See Section 6.2 for a detailed comparison between RECON and LS algorithm.

Tables 2 and 3 also show the number of vertices classified as critical points along with the number of potential loop saddles, and the actual number of loops in the input. Note that for 2D input, the split saddles are also classified as potential loop saddles. In the case of 3D input, majority of the points classified as potential loop saddles correspond to regular points on the boundary. In both cases, these false positives are handled while performing the breadth-first traversal to split the input. We also observe that the additional processing does not significantly impact computation time. For example, even in the case of the *Day* model, which contains the largest number of false positives (greater than 10,000), RECON requires less than 10 milliseconds to discard these vertices.

5.1.2 Higher Dimensional Data

Fig. 8 compares the performance of RECON, OS, and RAND algorithms for 4D, 5D and 6D data. The input to these experiments were a set of Sierpinski simplexes in the corresponding dimension together with the height function. The number of loops in the Reeb graph also increases with increasing input size for these data sets. For the largest 6D model (s6d-7) with approximately 28.8 million triangles, OS algorithm threw a memory exception, while the RAND algorithm ran out of memory and started thrashing. These plots show that our RECON implementation not only performs much better than the other algorithms, but also exhibits a consistent performance with increasing dimensions.

In Table 4, we group models of approximately same size, but with different number of loops in the Reeb graph. Notice that RECON's execution times do not change significantly when the number of loops in the input changes. This is true even when the number of loops changes from 15 in Lucy to around 2 million in the case of s6d-7. Fig. 9 shows the Reeb graphs computed for the height function of a 2D, 3D, and non-manifold input, respectively.

5.1.3 Noisy Data

Our 2D and 3D experiments were performed on real-world data sets. Also, most of these data sets are noisy, as can be

TABLE 4
RECON Exhibits Consistent Performance for Similar Sized Models with Different Number of Loops

Model	# Triangles	# Loops	Time taken (sec)
s4d-7	0.8M	$1x10^5$	0.7
s5d-6	0.9M	$1x10^5$	0.7
s4d-8	3.9M	$5.8x10^5$	3.6
s5d-7	5.6M	$5.6x10^5$	4.4
s6d-6	4.1M	$2.9x10^5$	2.9
Lucy	28.0M	15	34.2
s6d-7	28.8M	$2x10^6$	21.2

The model sxd-y denotes an x-dimensional Sierpinski simplex subdivided y times.

TABLE 5
Reeb Graph Computation Times for Noisy Versions of Various Models (Denoted by *)

Model	# Critical Points	Time taken (sec)		
		RECON	ONLINE	LS
Dawn*	1,924,221	10.8	26.8	NA
Lucy*	7,520,211	71.2	585.9	NA
Plasma*	6,765	1.5	NA	2.1
SF Earthquake*	46,908	3.0	NA	3.1

seen by the number of critical points in them. In order to stress test the algorithm, we artificially introduced Gaussian noise to the input, and computed the Reeb graph for the resulting data set. Table 5 shows the results from this experiment. Note that in the Dawn model, which was already noisy to begin with, adding additional noise increased the number of critical points by a factor of 20 to around 2 million critical points. This accounts for about 60 percent of the input vertices. Similarly, for the Lucy model, the increase in the number of critical points is almost three orders of magnitude and consists of more than half the input vertices. We notice that RECON performs efficiently even in such extreme scenarios. The running time doubles for RECON while the performance of ONLINE and LS reduces significantly. These observations hold for the remaining data sets also. For a given input size, the time required to sort the input vertices and identify critical points remains constant, and contributes to about 50 percent of the total running time for the above data sets. The increase in number of critical points, caused due to the introduction of noise, mainly affects the steps corresponding to the identification of false positive loop saddles, and the merge procedure that constructs the contour tree from the join and split tree. The time taken by these two operations, which is less than 1 percent of the total running time, increases proportionally with the number of critical points. The increase in running time of the algorithm is primarily due to these two steps.

5.2 Experiments with Large Data

Table 6 shows experimental results of the out-of-core implementation. We compare it with the online algorithm for large data sets available from the Stanford data archive [47]. Note that even for the Atlas model which has approximately 500 million triangles, the total time taken to compute the Reeb graph is only around 40 minutes. The timings for the online algorithm are as reported in the paper [29]. The online algorithm requires finalization of the input to be performed only once irrespective of the input function used to compute the Reeb graph. Since our method constructs the interval volumes based on the input function, it has to perform this operation once for each function.

6 DISCUSSION

We now discuss a space-time tradeoff issue related to the in-memory implementation. We also present a detailed comparison between our algorithm and the loop surgery algorithm and the consequences of the various difference between the two approaches.

6.1 Storing Triangle Adjacencies

Our in-memory implementation stores the star of each vertex as well as the triangle adjacencies using the triangle-edge data structure. These triangle adjacencies can be obtained from the star, and thus it is not necessary to explicitly store the adjacencies. However, this results in increased effort for performing traversals, since the algorithm has to process more triangles to determine adjacencies, and hence the overall computation time increases. But an advantage of this approach is that the algorithm can process larger data sets in memory because of the $6n$ space that is saved by not storing triangle adjacencies. Table 7 compares the running time when triangle adjacencies are stored (RECON) and when they are not stored explicitly (RECON').

6.2 Comparison with Loop Surgery Algorithm

While the goal of our algorithm is similar to that followed by the loop surgery algorithm, namely to remove loops from the input in order to use the contour tree algorithm, it differs in

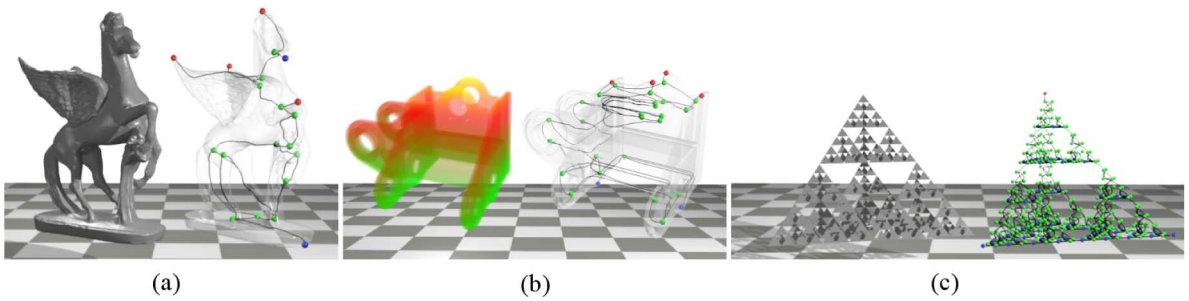

Fig. 9. Reeb graphs of height function defined on various models. (a) 2D Pegaso model. (b) Volume rendering of a 3D CAD model along with the simplified Reeb graph. (c) Reeb graph of a non-manifold mesh representing a 3D Sierpinski simplex.

TABLE 6
Performance of the Out-of-Core Implementation of RECON for Large 2D Data Sets

Model	# Triangles (millions)	Function	RECON			ONLINE		
			Creating interval volumes	Reeb graph computation	Total Time	Finalizing input	Reeb graph computation	Total Time
David	56M	x	37.0s	3.0m	3.6m	2.6m	2.1m	4.7m
		y	41.0s	3.1m	3.8m	2.6m	2.2m	4.8m
		z	27.0s	2.7m	3.2m	2.6m	14.0m	16.6m
St. Matthew	372M	x	5.5m	21.4m	26.9m	25.0m	15.0m	40.0m
		y	3.9m	22.8m	26.7m	25.0m	3.8h	4.2h
		z	2.8m	22.4m	25.2m	25.0m	16.0m	41.0m
Atlas	507M	x	7.2m	34.3m	41.5m	*	*	*
		y	5.8m	32.7m	38.5m	*	*	*
		z	7.3m	35.3m	42.6m	*	*	*

*RECON is faster than the online algorithm for all inputs, up to a factor of 8 for the St. Matthew data set (y-coordinate). * denotes that the running time for the data set is not available.*

the way this is accomplished. The main differences between the two approaches are listed below

- The Loop surgery algorithm analyzes the 2-manifold boundary of the 3-manifold input to identify loops, and is therefore restricted to 3-manifold input that is embedded in $\mathbb{R}^3$. Our algorithm, on the other hand identifies loops directly in the input. Our approach thus avoids the additional processing required to find the boundary. It is also generic because we work directly on the input.
- The LS algorithm follows a 3-step process to identify the loop saddles of the input—it first determines if the input has any tunnels by computing the Euler characteristic of its boundary, and then uses the join and split trees of the boundary to identify potential loop saddles. Further analysis is performed at these saddles to remove a subset of the false positives. Our approach identifies the potential loop saddles directly using the join tree of the input and removes false positives using a simple BFS operation, which also guarantees that all false positives are removed. Since, it uses the join tree directly to locate potential loop saddles, our algorithm works on both manifold and non-manifold input in any dimension.
- The LS algorithm splits the input domain to generate a single loop-free domain whose contour tree is used to recover the Reeb graph. Our approach creates multiple loop-free subvolumes, whose contour trees are stitched together to obtain the Reeb graph. The advantage of the latter approach is that the resulting algorithm scales to handle large data. Additionally, since the contour tree computation of each subvolume is independent of each other, the contour trees can potentially be computed in parallel.
- The LS algorithm explicitly stores the cuts that are performed to remove loops, while our algorithm stores only a representative triangle for each cut that is performed. Thus, not only is the memory usage of our algorithm reduced, but it is also bounded.
- The LS algorithm requires multiple passes over the input in order to compute the Reeb graph. These include passes over the input to obtain its boundary, identifying saddles in the boundary, computing the join and split trees of the boundary, and finally computing the join and split trees of the input after performing the surgery. Since our algorithm uses triangles to track connectivity of the sub-level and super-level sets, the operations corresponding to identifying critical points and loop saddles, performing the cut, and computing the join tree of the split domain is accomplished in a single pass over the input. Computing the split tree requires a second pass. Our algorithm is therefore efficient, especially for large data.

TABLE 7
Comparison of Running times Between RECON', Which Uses the Star of a Vertex to Find Triangle Adjacencies, and RECON

Model	# Triangles	Time taken (sec)		Memory saved
		RECON	RECON'	
Dawn	6.6M	4.5	5.0	338 MB
Lucy	28.0M	34.2	34.5	1.3 GB
Plasma	2.6M	1.5	2.5	91 MB
SF Earthquake	4.1M	2.4	2.8	135 MB

7 CONCLUSIONS

We have presented a practical algorithm to compute the Reeb graph of PL functions defined on manifolds and non-manifolds in any dimension. Since the algorithm essentially performs a series of union-find operations, it is easy to implement and efficient in practice. Experimental results demonstrated that the algorithm performs at least as good as other algorithms that are customized for a restricted subset of the input and outperforms existing generic algorithms by at least an order of magnitude. The algorithm's performance is consistent for various dimensions and with increase in number of loops of the Reeb graph. It can also efficiently handle large data sets that do not fit in memory.

Since the algorithm exhibits excellent practical performance, we believe that the bound on the worst case running time is loose. It will be interesting to either provide a tighter analysis of the algorithm or prove a lower bound that is different from that of the contour tree computation.

ACKNOWLEDGMENTS

Harish Doraiswamy was supported by Microsoft Corporation and Microsoft Research India under the Microsoft Research India PhD Fellowship Award. This work was supported by the Department of Science and Technology, India, under Grant SR/S3/EECE/048/2007.

REFERENCES

[1] F. Lazarus and A. Verroust, "Level Set Diagrams of Polyhedral Objects," *Proc. ACM Symp. Solid Modeling and Applications,* pp. 130-140, 1999.

[2] Y. Shinagawa, T.L. Kunii, and Y.L. Kergosien, "Surface Coding Based on Morse Theory," *IEEE Computer Graphics and Applications,* vol. 11, no. 5, pp. 66-78, Sept. 1991.

[3] S. Takahashi, Y. Shinagawa, and T.L. Kunii, "A Feature-Based Approach for Smooth Surfaces," *Proc. Symp. Solid Modeling and Applications,* pp. 97-110, 1997.

[4] H.S.Y. Shinagawa, T.L. Kunii, and M. Ibusuki, "Modeling Contact of Two Complex Objects: With an Application to Characterizing Dental Articulations," *Computers and Graphics,* vol. 19, no. 1, pp. 21-28, 1995.

[5] M. Hilaga, Y. Shinagawa, T. Kohmura, and T.L. Kunii, "Topology Matching for Fully Automatic Similarity Estimation of 3D Shapes," *Proc. ACM SIGGRAPH,* pp. 203-212, 2001.

[6] I. Guskov and Z. Wood, "Topological Noise Removal," *Proc. Graphics Interface,* pp. 19-26, 2001.

[7] Y.-J. Chiang and X. Lu, "Progressive Simplification of Tetrahedral Meshes Preserving all Isosurface Topologies," *Computer Graphics Forum,* vol. 22, no. 3, pp. 493-504, 2003.

[8] S. Takahashi, G.M. Nielson, Y. Takeshima, and I. Fujishiro, "Topological Volume Skeletonization Using Adaptive Tetrahedralization," *Proc. Geometric Modeling and Processing,* pp. 227-236, 2004.

[9] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder, "Removing Excess Topology from Isosurfaces," *ACM Trans. Graphics,* vol. 23, no. 2, pp. 190-208, 2004.

[10] F. Hétroy and D. Attali, "Topological Quadrangulations of Closed Triangulated Surfaces Using the Reeb Graph," *Graphical Models,* vol. 65, nos. 1-3, pp. 131-148, 2003.

[11] M. Mortara and G. Patané, "Affine-Invariant Skeleton of 3D Shapes," *Proc. Shape Modeling Int'l,* pp. 245, 2002.

[12] E. Zhang, K. Mischaikow, and G. Turk, "Feature-Based Surface Parameterization and Texture Mapping," *ACM Trans. Graphics,* vol. 24, no. 1, pp. 1-27, 2005.

[13] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D.R. Schikore, "Contour Trees and Small Seed Sets for Isosurface Traversal," *Proc. 13th Ann. Symp. Computational Geometry,* pp. 212-220, 1997.

[14] C.L. Bajaj, V. Pascucci, and D.R. Schikore, "The Contour Spectrum," *Proc. IEEE Conf. Visualization,* pp. 167-173, 1997.

[15] H. Carr, J. Snoeyink, and M. van de Panne, "Simplifying Flexible Isosurfaces Using Local Geometric Measures," *Proc. IEEE Conf. Visualization,* pp. 497-504, 2004.

[16] I. Fujishiro, Y. Takeshima, T. Azuma, and S. Takahashi, "Volume Data Mining Using 3D Field Topology Analysis," *IEEE Computer Graphics and Applications,* vol. 20, no. 5, pp. 46-51, Sept./Oct. 2000.

[17] S. Takahashi, Y. Takeshima, and I. Fujishiro, "Topological Volume Skeletonization and Its Application to Transfer Function Design," *Graphical Models,* vol. 66, no. 1, pp. 24-49, 2004.

[18] G.H. Weber, S.E. Dillard, H. Carr, V. Pascucci, and B. Hamann, "Topology-Controlled Volume Rendering," *IEEE Trans. Visualization and Computer Graphics,* vol. 13, no. 2, pp. 330-341, Mar./Apr. 2007.

[19] J. Zhou and M. Takatsuka, "Automatic Transfer Function Generation Using Contour Tree Controlled Residue Flow Model and Color Harmonics," *IEEE Trans. Visualization Computer Graphics,* vol. 15, no. 6, pp. 1481-1488, 2009.

[20] W. Harvey and Y. Wang, "Topological Landscape Ensembles for Visualization of Scalar-Valued Functions," *Computer Graphics Forum,* vol. 29, pp. 993-1002, 2010.

[21] P. Oesterling, C. Heine, H. Jänicke, G. Scheuermann, and G. Heyer, "Visualization of High Dimensional Point Clouds Using Their Density Distribution's Topology," *IEEE Trans. Visualization and Computer Graphics,* vol. 17, no. 11, pp. 1547-1559, Nov. 2011.

[22] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno, "Reeb Graphs for Shape Analysis and Applications," *Theoretical Computer Science,* vol. 392, pp. 5-22, Feb. 2008.

[23] S. Biasotti, L. De Floriani, B. Falcidieno, P. Frosini, D. Giorgi, C. Landi, L. Papaleo, and M. Spagnuolo, "Describing Shapes by Geometrical-Topological Properties of Real Functions," *ACM Computing Surveys,* vol. 40, pp. 12:1-12:87, Oct. 2008.

[24] S. Biasotti, L. Floriani, B. Falcidieno, and L. Papaleo, "Morphological Representations of Scalar Fields," *Shape Analysis and Structuring,* Mathematics and Visualization, L. Floriani, M. Spagnuolo, G. Farin, H.-C. Hege, D. Hoffman, C.R. Johnson, and K. Polthier, eds., pp. 185-213, Springer, 2008.

[25] S. Biasotti, D. Attali, J.-D. Boissonnat, H. Edelsbrunner, G. Elber, M. Mortara, G.S. Baja, M. Spagnuolo, M. Tanase, and R. Veltkamp, "Skeletal Structures," *Shape Analysis and Structuring,* Mathematics and Visualization, L. Floriani, M. Spagnuolo, G. Farin, H.-C. Hege, D. Hoffman, C.R. Johnson, and K. Polthier, eds., pp. 145-183, Springer, 2008.

[26] H. Carr, J. Snoeyink, and U. Axen, "Computing Contour Trees in all Dimensions," *Computational Geometry: Theory and Application,* vol. 24, no. 2, pp. 75-94, 2003.

[27] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Loops in Reeb Graphs of 2-Manifolds," *Discrete and Computational Geometry,* vol. 32, no. 2, pp. 231-244, 2004.

[28] H. Doraiswamy and V. Natarajan, "Efficient Algorithms for Computing Reeb Graphs," *Computational Geometry: Theory and Applications,* vol. 42, nos. 6/7, pp. 606-616, 2009.

[29] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas, "Robust On-Line Computation of Reeb Graphs: Simplicity and Speed," *ACM Trans. Graphics,* vol. 26, no. 3, article 58, 2007.

[30] W. Harvey, Y. Wang, and R. Wenger, "A Randomized O(m log m) Time Algorithm for Computing Reeb Graphs of Arbitrary Simplicial Complexes," *Proc. Symp. Computational geometry,* pp. 267-276, 2010.

[31] G. Patanè, M. Spagnuolo, and B. Falcidieno, "A Minimal Contouring Approach to the Computation of the Reeb Graph," *IEEE Trans. Visualization and Computer Graphics,* vol. 15, no. 4, pp. 583-595, July/Aug. 2009.

[32] H. Doraiswamy and V. Natarajan, "Output-Sensitive Construction of Reeb Graphs," *IEEE Trans. Visualization and Computer Graphics,* vol. 18, no. 1, pp. 146-159, Jan. 2012.

[33] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci, "Loop Surgery for Volumetric Meshes: Reeb Graphs Reduced to Contour Trees," *IEEE Trans. Visualization and Computer Graphics,* vol. 15, no. 6, pp. 1177-1184, Nov./Dec. 2009.

[34] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press, 2001.

[35] Y.-J. Chiang, T. Lenz, X. Lu, and G. Rote, "Simple and Optimal Output-Sensitive Construction of Contour Trees Using Monotone Paths," *Computational Geometry: Theory and Application,* vol. 30, no. 2, pp. 165-195, 2005.

[36] Y. Shinagawa and T.L. Kunii, "Constructing a Reeb Graph Automatically from Cross Sections," *IEEE Computer Graphics and Application,* vol. 11, no. 6, pp. 44-51, Nov. 1991.

[37] H. Edelsbrunner and J. Harer, *Computational Topology: An Introduction.* Am. Math. Soc., 2009.

[38] A. Hatcher, *Algebraic Topology.* Cambridge Univ. Press, 2002.

[39] J. Milnor, *Morse Theory.* Princeton Univ. Press, 1963.

[40] T.F. Banchoff, "Critical Points and Curvature for Embedded Polyhedral Surfaces," *Am. Math. Monthly,* vol. 77, pp. 475-485, 1970.

[41] H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci, "Morse-Smale Complexes for Piecewise Linear 3-Manifolds," *Proc. 19th Ann. Symp. Computational Geometry,* pp. 361-370, 2003.

[42] Y. Matsumoto, *An Introduction to Morse Theory,* Am. Math. Soc., 2002.

[43] G. Reeb, "Sur Les Points Singuliers D'une Forme de Pfaff Complètement Intégrable ou D'une Fonction Numérique," *Comptes Rendus de L'Académie ses Séances,* vol. 222, pp. 847-849, 1946.

[44] H. Edelsbrunner, *Geometry and Topology for Mesh Generation.* Cambridge Univ. Press, 2001.

[45] R. Diestel, *Graph Theory,* Graduate Texts in Mathematics, third ed., vol. 173, Springer-Verlag, 2005.

[46] E.P. Mücke, "Shapes and Implementations in Three-Dimensional Geometry," PhD dissertation, Dept. of Computer Science, Univ. of Illinois, Urbana-Champaign, Illinois, 1993.

[47] The Digital Michelangelo Project. http://graphics.stanford.edu/projects/mich/, 2012.

Harish Doraiswamy received the BE degree from Visveswaraiah Technological University, and the ME degree from Indian Institute of Science, both in computer science and engineering. Currently, he is working toward the PhD degree at the Department of Computer Science and Automation, Indian Institute of Science, Bangalore. His research interests include scientific visualization and computational topology. He is a student member of the IEEE.

Vijay Natarajan received the BE degree in computer science, the MSc degree in mathematics from Birla Institute of Technology and Science, Pilani, India, and the PhD degree in computer science from Duke University in 2004. He is an assistant professor in the Department of Computer Science and Automation and the Supercomputer Education and Research Centre at the Indian Institute of Science, Bangalore. His research interests include scientific visualization, computational geometry, computational topology, and meshing. He is a member of the IEEE.