A Randomized $O(m \log m)$ Time Algorithm for Computing Reeb Graphs of Arbitrary Simplicial Complexes

William Harvey*, Yusu Wang* and Rephael Wenger*

Abstract

Given a continuous scalar field $f : X \to \mathbb{R}$ where X is a topological space, a *level set* of f is a set $\{x \in X : f(x) = \alpha\}$ for some value $\alpha \in \mathbb{R}$. The level sets of f can be subdivided into connected components. As α changes continuously, the connected components in the level sets appear, disappear, split and merge. The Reeb graph of f encodes these changes in connected components of level sets. It provides a simple yet meaningful abstraction of the input domain. As such, it has been used in a range of applications in fields such as graphics and scientific visualization.

In this paper, we present the first sub-quadratic algorithm to compute the Reeb graph for a function on an arbitrary simplicial complex K. Our algorithm is randomized with an expected running time $O(m \log n)$, where m is the size of the 2-skeleton of K (i.e, total number of vertices, edges and triangles), and n is the number of vertices. This presents a significant improvement over the previous $\Theta(mn)$ time complexity for arbitrary complex, matches (although in expectation only) the best known result for the special case of 2-manifolds, and is faster than current algorithms for any other special cases (e.g, 3-manifolds). Our algorithm is also very simple to implement. Preliminary experimental results show that it performs well in practice.

*Computer Science and Engineering Department, The Ohio State University, Columbus, OH 43210. Emails: harveywi,yusu,wenger@cse.ohio-state.edu

1 Introduction

Let $f : X \to \mathbb{R}$ be a continuous function defined on a topological space X. For each scalar value $\alpha \in \mathbb{R}$, the set $\{x \in X : f(x) = \alpha\}$ is called a *level set* of f. A level set of f can be partitioned into connected components. The *Reeb graph* of f is obtained by continuously collapsing every component of each level set into a single point. As α increases, connected components appear, disappear, split and merge, and the Reeb graph of f tracks such changes.

The mathematical concept of the Reeb graph was first defined by George Reeb [19] for smooth manifolds. It was introduced in graphics applications by Shinagawa et al. [22]. Since then, it has been used in a range of shape analysis applications, including shape understanding [2, 3, 22], segmentation and matching [14, 20, 25, 27], shape repairing and animation [15, 29], surface quadrangulations and parametrization [13, 18, 30]. See [4] for a very nice survey paper on the Reeb graph and its applications. The loop-free version of the Reeb graph, commonly known as the contour tree, is also a popular tool for analyzing scalar fields [5], and has a long history in fields such as geographic information systems and scientific visualization. In all these applications, it is important to be able to construct the Reeb graph in an efficient manner.

We consider the problem of computing Reeb graphs of $f : X \to \mathbb{R}$, where f is a piecewise linear function defined on a simplicial complex X. A straightforward approach to computing the Reeb graph is to track the connected components of the level sets and update the Reeb graph as the components change. While this approach is conceptually simple, it can be tricky to implement. Moreover, on a 2-complex with n vertices and m triangles, this approach can take $\Theta(mn)$ time.

Our contribution. As discussed in Section 2, a number of papers present efficient algorithms for computing Reeb graphs in special cases, such as when X is a two or three manifold. This paper describes an efficient randomized algorithm for arbitrary simplicial complexes. The algorithm runs in $O(m \log n)$ expected time, where m is the number of 0, 1, and 2-simplices in the complex and n is the number of vertices.

Specifically, for a point $q \in X$, let L_q be the level set $\{x \in X : f(x) = f(q)\}$ and let C_q be the connected component of L_q which contains q. Our algorithm picks a random vertex q in the complex and "collapses" all the triangles intersecting C_q onto q and its adjacent edges. The algorithm then picks another random vertex and repeats this collapse procedure. New triangles will be created and collapsed during the process, and the expected total number of these intermediate triangles is $O(m \log n)$. As a result, the expected running time of our algorithm is also $O(m \log n)$. Our contributions are as follows:

- We give the first sub-quadratic time algorithm for an arbitrary simplicial complex. Our expected running time matches the running time of the best current algorithm for 2-manifolds (although only in expectation) and is faster than any other current algorithm.
- Our algorithm and its implementation are very simple. The implementation uses a repetition of just one "collapse" operation, and involves only array and linked-list data structures.
- We show preliminary experimental results to demonstrate the practical potential of our algorithm.

2 Related work

In [6], Carr et al. presented a simple and efficient algorithm to compute the contour tree of an arbitrary simplicial complex domain X in $O(n \log n + m\alpha(m))$ time, where n is the number of vertices in the input simplicial complex and m is the size of its 2-skeleton (i.e, the total number of vertices, edges and triangles). The algorithm improved ideas from several previous contour-tree algorithms [24, 28].

Shinagawa and Kunii presented the first provably correct algorithm to compute Reeb Graphs for a triangulation of a 2-manifold [21]. Their algorithm runs in $\Theta(n^2)$ time. It sweeps vertices of the mesh in

References	Underlying Domain	Time Complexity
[7]	2-manifolds	$O(n \log n) = O(m \log m)$
[10]	3-manifolds	$O(m \log m + m \log g (\log \log g)^3)$
	3-manifolds	$O(mg \log^2 m)$ (implemented algorithm)
	d-manifolds	$O(m \log m (\log \log m)^3)$
[9]	3-manifolds	$O(m \log m + L) = \Theta(mn)$
	arbitrary simplicial complex	$O(mn)$
[26]	3-manifolds embedded in $\mathbb{R}^3$	$O(m \log m + hm) = \Theta(mn)$
[17]	arbitrary simplicial complex	$O(mn)$

Table 1: Current best algorithms to compute Reeb graph for a simplicial complex X. m is the size of the 2-skeleton of X (i.e, total number of vertices, edges and triangles), and n is the number of vertices.

increasing order of function values, and maintains the level-sets explicitly during the sweeping. This sweeping idea can be used to compute the Reeb graph for arbitrary simplicial complex in $\Theta(nm)$ time. Since then, there have been several more efficient algorithms, mainly focusing on 2 or 3-manifolds. We summarize the results in Table 2 and briefly review them below.

For the case of 2-manifold, Cole-McLaughlin et al. [7] improved the running time from $O(n^2)$ to $O(n \log n)$ based on their insights in the structure of the level sets and Reeb graph for 2-manifolds. Doraiswamy and Natarajan [10] further extended the sweeping idea to handle 3-manifolds, and presented an $O(m \log m + m \log g (\log \log g)^3)$ time algorithm, where g is the maximum genus of any iso-surface. Their algorithm uses dynamic spanning tree / co-trees to maintain connectivity of level sets during the sweeping. It can be extended to d-manifolds (by using a fully-dynamic graph connectivity algorithm) with a running time $O(m \log m (\log \log m)^3)$. They also provided a simpler but slower implementation that runs in $O(mg \log^2 m)$ time to compute the Reeb graph for functions on 3-manifolds. An interesting and simple semi-output sensitive algorithm is proposed in [9] to compute the Reeb graph of 3-manifolds in time $O(m \log m + L)$, where L is the total complexity of all level-sets passing through critical points and can be $\Theta(nm)$ in worst case. The same algorithm can handle arbitrary simplicial complexes, but the bound of $O(m \log m + L)$ will then not hold. Very recently in [26], Tierny et al. proposed an algorithm that computes the Reeb graph for a 3-manifold with boundary embedded in $\mathbb{R}^3$ in time $O(m \log m + hm)$, where h is number of independent loops in the Reeb graph. Their approach leverages the efficient contour tree algorithm in [5] by using a novel surgery idea to first cut the input simplicial complex so that its Reeb graph is loop-free. The worst case time complexity can be $\Theta(nm)$, but it runs much faster in practice as h is usually small. Finally, a streaming algorithm was presented in [17] to compute the Reeb graph for an arbitrary simplicial complex in $\Theta(nm)$ time. Reeb graphs for time-varying functions defined on 3-dimensional domains was studied in [12] exploiting Jacobi sets. In the field of graphics, there are also several practical implementations to approximate the Reeb graphs for triangular meshes (see references in [4]).

3 Problem Definition and Preliminaries

Reeb graphs. Let X be a topological space, and $f : X \to \mathbb{R}$ a continuous function defined on f. The level-set of f with respect to a value $\alpha \in \mathbb{R}$ is denoted by $f^{-1}(\alpha) = \{x \in X \mid f(x) = \alpha\}$. Two points $x, y \in X$ are *equivalent*, denoted by $x \sim y$, if and only if x and y belong to the same connected component of some $f^{-1}(\alpha)$; that is, $\alpha = f(x) = f(y)$ and x is connected to y in $f^{-1}(\alpha)$. Now consider the quotient space $X_\sim$, which is the set of equivalent classes equipped with the quotient topology induced by this equivalent relation; $X_\sim$ is also called the *Reeb graph* of X with respect to f, denoted by $\mathcal{R}_X(f)$ from now on, where X may be omitted when its choice is clear.

Note that $\mathcal{R}_X(f)$ can also be thought of the image of the continuous surjection map $\Phi_f : X \to X_\sim$ where $\Phi_f(x) = \Phi_f(y)$ *if and only if* x and y come from the same connected component of a level set of f. In this sense, $\mathcal{R}_X(f)$ is obtained by continuously identifying each connected component C. The map Φ_f induces a scalar function $\tilde{f} : \mathcal{R}_X(f) \to \mathbb{R}$ on $\mathcal{R}_X(f)$ where $\tilde{f}(p) = f(x)$ if $p = \Phi_f(x)$. Since $f(x) = f(y)$ whenever $\Phi_f(x) = \Phi_f(y)$, the function $\tilde{f}$ is well-defined. Since f is continuous, so is $\tilde{f}$.

Reeb graph nodes and arcs. As the name suggests, the Reeb graph $\mathcal{R}_X(f)$ can be represented as a graph. See Figure 1 (b) for an example, where the height of a point $q \in \mathcal{R}_X(f)$ is simply $\tilde{f}(q)$. Now for every point $q \in \mathcal{R}_X(f)$, there is a sufficiently small neighborhood $\mathcal{R}' \subseteq \mathcal{R}_X(f)$ containing q such that $\mathcal{R}'$ is connected, has no loops, and $\tilde{f}(p) \neq \tilde{f}(q)$ for any point $p \in \mathcal{R}' - \{q\}$. Let γ be a connected component of $\mathcal{R}' - \{q\}$. Since $\tilde{f}(p) \neq \tilde{f}(q)$ for any point $p \in \gamma$ and $\tilde{f}$ is continuous, either $\tilde{f}(p) > \tilde{f}(q)$ for all $p \in \gamma$ or $\tilde{f}(p) < \tilde{f}(q)$ for all $p \in \gamma$. The *up degree* of a point $q \in \mathcal{R}_X(f)$ is the number of connected components γ in $\mathcal{R}' - \{q\}$ where $\tilde{f}(p) > \tilde{f}(q)$ for $p \in \gamma$. The *down degree* of $q \in \mathcal{R}_X(f)$ is the number of connected components γ in $\mathcal{R}' - \{q\}$ where $\tilde{f}(p) < \tilde{f}(q)$ for $p \in \gamma$. A *node* of $\mathcal{R}_X(f)$ is a point $q \in \mathcal{R}_X(f)$ whose up degree is not one or whose down degree is not one. Deleting the nodes of $\mathcal{R}_X(f)$ from $\mathcal{R}_X(f)$ leaves a set of curves which are the *arcs* of the Reeb graph.

The Reeb graph $\mathcal{R}_X(f)$ encodes partial information about the 0-th and 1-st homology group information of the topological space X. Specifically, it can be shown [11] that $\beta_0(\mathcal{R}(f)) = \beta_0(X)$ and $\beta_1(\mathcal{R}(f)) \leq \beta_1(X)$. If X is an orientable 2-manifold, then one can in fact obtain the full 1-st homology group information (including its rank and generators) about X from $\mathcal{R}(f)$ [7].

Piecewise-linear (PL) settings. We consider a simplicial complex K and a piecewise-linear (PL) function $f : K \to \mathbb{R}$ defined on K. Specifically, f is defined at vertices of K, and linearly interpolated in the interior of every simplex of dimension higher than 0. Since $\mathcal{R}(f)$ depends only on the connectivity of each level set, for a generic function f (where no two vertices have the same function value), the Reeb graph of f depends only on the 2-skeleton of K. Hence from now on, we assume that f is generic and $K = (V, E, T)$ is a simplicial 2-complex with vertices V, edges E and triangles T.

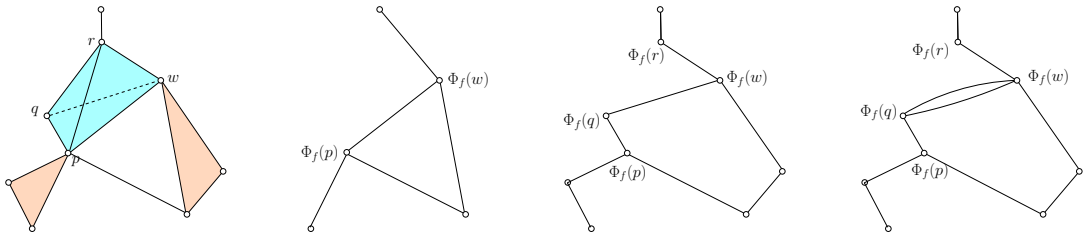


Figure 1: A simplicial complex K in (a) and the Reeb graph of the height function is in (b). Shaded triangles are 2-simplices in K, and the two front faces of the tetrahedron $pqrw$ as well as one back face $\triangle qrw$ are present in K. The corresponding augmented Reeb graph is shown in (c). If the back face $\triangle qrw$ is missing, then the corresponding augmented Reeb graph is shown in (d). One can obtain (b) by removing monotonic degree-2 nodes from (c).

Augmented Reeb graph. An *augmented Reeb graph* of $f : X \to \mathbb{R}$ is a Reeb graph of f with some additional degree-2 vertices inserted in the original Reeb graph arcs. In particular, if f is a piecewise-linear function on a simplicial complex K, then the *augmented Reeb graph generated by* K is a Reeb graph of f whose nodes are $\{\Phi_f(v) : v \in V(K)\}$. Note that all nodes of the original Reeb graph are in $\{\Phi_f(v) : v \in V(K)\}$. See Figure 1 (c) and (d) for examples. One can obtain the original Reeb graph from

the augmented one by removing *monotonic* degree-2 nodes, which are nodes with up-degree 1 and down-degree 1. (The augmented Reeb graph generated by K is similar to the augmented contour tree in [5].)

Some notations. Let f be a function on triangle $t = \triangle pqr$ such that $f(p)$, $f(q)$ and $f(r)$ are all distinct. The *middle vertex* of t, denoted by $\mathrm{mid}(t)$, is the vertex q where $f(p) < f(q) < f(r)$. The edge of t opposite to $\mathrm{mid}(t)$, pr in this example, spans the range of function values for all points in t, and is called *the spanning edge of* t.

Our initial input K is a simplicial complex. However, as our algorithm proceeds, we will obtain intermediate complexes that may no longer be simplicial. Specifically, there may be multiple edges between the same pair of vertices. Similarly, two triangles may have the same set of vertices but not the same set of edges. Hence we usually denote a triangle t by its boundary edges $t(e_1, e_2, e_3)$. However, for simplicity, we sometimes still denote a triangle by its vertices $\triangle pqr$ when its choice of edges is clear.

4 Algorithm RANDREEB

Input to Algorithm RANDREEB is a simplicial 2-complex $K = (V, E, T)$ and a piecewise linear function $f : K \to \mathbb{R}$. We assume that $f(v)$ are distinct for all $v \in V(K)$. We can enforce this condition by symbolically perturbing the scalar value at all vertices. The algorithm compute the augmented Reeb graph of f in expected time $O(m \log n)$ where $n = |V|$ and $m = |V| + |E| + |T|$ is the size of K. In what follows, after briefly describing the data structure our algorithm uses, we introduce main components of our algorithm. From now on, in all figures, we use the height function as the input piecewise-linear function f.

Data structure. The set of vertices is stored in an array. The set of edges and triangles are each stored in a doubly-linked list. Each edge e maintains a list of (pointers to) triangles incident on e, denoted by $IncT(e)$. Each triangle maintains three pointers to its three boundary edges, e_1, e_2 and e_3, and three pointers to its position in these incident lists, $IncT(e_1)$, $IncT(e_2)$ and $IncT(e_3)$. Similarly, a vertex v maintains a list of triangles having v as middle-vertex, and each triangle stores a pointer to its position in the corresponding list. Adding or removing a triangle takes $O(1)$ time. The total size of the data structure is $O(|V|+|E|+|T|)$.

4.1 The Collapse Operation

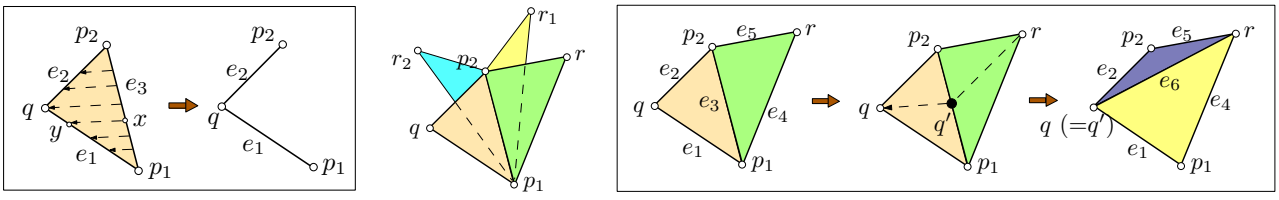


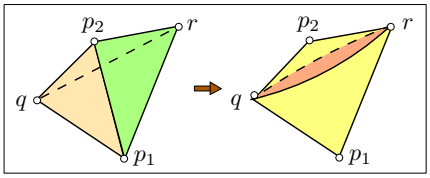
Figure 2: (a) $\mathcal{C}ollapse(t)$ with $t = (e_1, e_2, e_3)$. All points from t on the same contour (e.g, segment xy) collapse onto its corresponding point on edges e_1 or e_2 (e.g, point y). (b) Edge e_3 has three incident triangles other than t. (c) Each incident triangle of e_3, say $\triangle rp_1p_2$, will be split into two ($\triangle rp_1q'$, $\triangle rp_2q'$ with $f(q) = f(q')$).

A fundamental operation of our algorithm is the $\mathcal{C}ollapse(t)$ procedure defined for a triangle t. Let e_1 and e_2 be the edges of t incident on $q = \mathrm{mid}(t)$, and let e_3 be the edge of t opposite q. Since e_3 is the spanning edge of t, no two points in $e_1 \cup e_2$ share the same function value. Procedure $\mathcal{C}ollapse(t)$ collapses t by mapping $\{x \in t : f(x) = f(y)\}$ to the point $y \in e_1 \cup e_2$. See Figure 2 (a) for an illustration.

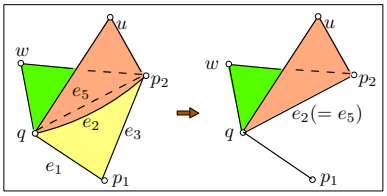
```
Procedure COLLAPSE(t)
(1)     remove triangle t
(2)     let q = mid(t), and e3 = (p1, p2) the spanning edge of t
(3)     let e1 = (q, p1) and e2 = (q, p2) be the edges of t incident on q
(4)     merge all edges identified with e3
(5)     For (every σ incident to e3 and σ ≠ t)
(6)         set r be the vertex of σ opposite to e3
(7)         let e4 = (r, p1) and e5 = (r, p2) be the edges of σ incident on r
(8)         If (r == q)          // then t and σ have the same vertices
(9)             If (e1 ≠ e4)     Then   identifyEdges(e1, e4);
(10)            If (e2 ≠ e5)     Then   identifyEdges(e2, e5);
(11)        Else
(12)            create new edge e6 with endpoints q, r
(13)            create triangle (e1, e4, e6) and triangle (e2, e5, e6)
(14)        Endif
(15)        remove triangle σ;
(16)    Endfor
(17)    remove edge e3;
```

Figure 3: Pseudo-code for the collapse operation.

Collapsing t splits all the other triangles incident on the spanning edge e_3. Let $\sigma = (e_3, e_4, e_5)$ be a triangle incident on e_3, where e_4 shares a vertex with edge $e_1 \in t$ and e_5 shares a vertex with edge $e_2 \in t$. Triangle σ has a vertex r opposite e_3. We add the edge e_6 with endpoints q, r and replace σ with triangles (e_1, e_4, e_6) and (e_2, e_5, e_6). See Figure 2 (c). We refer to σ as a *splitting triangle*. Every incident triangle to edge e_3 other than t itself is a splitting triangle. We remark that one should consider simplicial complexes in this paper as abstract simplicial complexes. After the collapse operation, it is possible that the geometric realization of the new complex in the embedding Euclidean space may have self-intersections.



Procedure $\mathcal{C}ollapse$ described above may create double or even multiple edges between the same two points. For example, see the figure on the right where the complex contains four faces of a tetrahedron. Collapsing $\triangle qp_1p_2$ splits $\triangle p_1p_2r$ and creates a double edge between q and r. (Also, recall Figure 1 (b) where such a pair of double edge creates a loop in the final Reeb graph.) Hence the resulting complex may not be a simplicial complex any more, but rather, is a cell complex with the property that each face is a simplex.



Because the complexes created by Procedure $\mathcal{C}ollapse$ are not necessarily simplicial, a triangle σ incident to the spanning edge e_3 of t may have the same vertices as the collapsing triangle t (i.e, $r = q$ in Figure 2 (c)). In this case, edges e_1 and e_4 may be the same edges or they may be "double" edges, sharing the same endpoints but nevertheless distinct. Similarly, edges e_2 and e_5 may be the same edges or they may be distinct double edges. For example, in the figure on the right, there are two edges e_2 and e_5 between q and p_2 and two triangles with the same set of vertices q, p_1, p_2. One triangle has edges e_1, e_3 and e_2 while the other one has edges e_1, e_3 and e_5.

In the case where σ and t share the same vertices, collapsing t and splitting σ would create two degenerate triangles, spanned by edges e_1 and e_4, and by edge e_2 and e_5, respectively. Instead of creating such

degenerate triangles, if edges e_1 and e_4 are distinct double edges, then we simply identify these two edges. Similarly, if edges e_2 and e_5 are distinct, then we identify e_2 and e_5. In each case the algorithm calls a procedure to merge the two edges and their data structures into a single edge. In the previous example on the right, triangle $t_1(e_2, qu, up_2)$ is incident to edge e_2 while triangle $t_2(e_5, qw, wp_2)$ is incident to edge e_5. Collapsing triangle $t(e_1, e_2, e_3)$ causes the deletion of triangle $\sigma(e_1, e_5, e_3)$ and the identification of edge e_5 with e_2. After the identification of e_2 and e_5, both t_1 and t_2 are incident to edge e_2 and edge e_5 is removed. The pseudo-code of Procedure $\mathcal{C}ollapse$ is described in Figure 3.

Finally, let K_1 and K_2 denote the complex before and after Procedure $\mathcal{C}ollapse(t)$. Note that K_1 and K_2 share the same set of vertices V, and each cell in K_2 is still a simplex. Let e_1 and e_2 be the edges of t incident on $\mathrm{mid}(t)$. Procedure $\mathcal{C}ollapse(t)$ induces a continuous surjection $\phi_t : |K_1| \to |K_2|$ where $\phi_t(y) = y$ if $y \notin t$, and $\phi_t(y) = \phi_t(x)$ if $y \in t$ and $x \in e_1 \cup e_2$ and $f(x) = f(y)$. Since each cell in K_2 is still a simplex, it can be shown that the function $f_t : |K_2| \to \mathbb{R}$ defined as $f_t := f \circ \phi_t$ is exactly the piecewise-linear function on K_2 induced by the function values $f(v)$ at $v \in V$. We thus abuse the notation slightly and still refer to f_t as f.

Lemma 4.1 *Let K_1 and K_2 denote the complex before and after a collapse operation. The augmented Reeb graphs remains the same; that is, $\mathcal{R}_{K_1}(f) = \mathcal{R}_{K_2}(f)$.*

Proof: Let $\Phi_2 : |K_2| \to \mathcal{R}_{K_2}(f)$ be the surjection mapping K_2 to its Reeb graph. Set $\Phi = \Phi_2 \circ \phi_t$. Function Φ maps each connected component of a level set to a distinct point. Thus $\mathcal{R}_{K_2}(f)$ equals $\mathcal{R}_{K_1}(f)$. ■

Vertex-collapse with respect to q. A *vertex-collapse w.r.t a vertex q*, denoted by $\mathcal{S}eqCollapse(q)$, refers to a sequence of collapse operations for all triangles with q being the middle vertex. These triangles may include new triangles created from earlier collapse operations involving q. In fact, it is necessary that one of the two new triangles created by every splitting triangle during $\mathcal{C}ollapse(t)$ will have q as the middle vertex, and thus will be later collapsed while executing $\mathcal{S}eqCollapse(q)$. For example, in Figure 2 (c), the triangle $\triangle qp_1r$ may be later also collapsed in $\mathcal{S}eqCollapse(q)$. At the end of $\mathcal{S}eqCollapse(q)$ operation, there is no triangle left with q being the middle-vertex. In other words, $\mathcal{S}eqCollapse(q)$ provides a systematic way to collapse the entire contour passing through q onto q. See Figure 4 for an example.

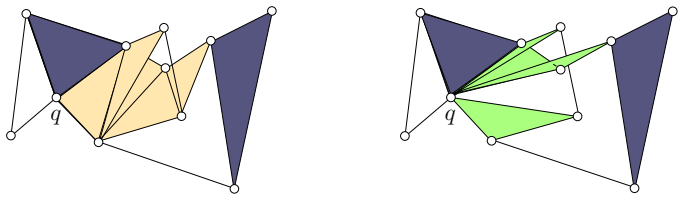


Figure 4: (a) Before and (b) after vertex-collapse operation $\mathcal{S}eqCollapse(q)$. Shaded triangles are 2-simplices. Two dark-shaded triangles are not affected by this operation.

Claim 4.2 *After performing $\mathcal{S}eqCollapse(q)$, no subsequent vertex-collapse operations can ever create a triangle with q being the middle vertex.*

Proof: Each splitting triangle σ will create two new triangles σ_1 and σ_2. The key observation is that the two extreme vertices of σ (i.e, the vertex with largest and smallest function values) remain extreme in the new triangles. Hence after Procedure $\mathcal{S}eqCollapse(q)$ (at which point q is not the middle-vertex of any triangle), no subsequent collapse operation can change q from being an extreme vertex of some triangle to being the middle vertex of another triangle. The claim then follows. ■

4.2 Identifying Edges

The last remaining operations to describe are the *identifyEdges*(e, e') in Steps (9) and (10) of Procedure $\mathcal{C}ollapse$ and the merging of these edges in Step (4) in Figure 3. The simplest way to implement *identifyEdges* is to traverse the lists of triangles, $IncT(e)$ and $IncT(e')$, merging the two lists into one and updating the incident triangles. However, this merging and updating associated data-structures of incident triangles takes time proportional to the length of one of the two lists. If the simplicial complex is a 2-manifold, then each list $IncT(e)$ never has more than two elements (see Appendix B). However, for a general simplicial complexe, these lists may grow arbitrarily long. To ensure that Steps (9) and (10) do not take too much time, we use a "lazy" identification procedure.

Specifically, instead of merging e and e' and their associated data structures, *identifyEdges*(e, e') simply records the identity relation between e and e'. Each edge has a list of edges with which it has a identity-relation. Calling *identifyEdges*(e, e') adds e' to the list of e and adds e' to the list of e.

The actual merging of edges only occurs in Step (4) of Procedure $\mathcal{C}ollapse$. All the edges identified *transitively* with the spanning edge e_3 and their associated incident triangle lists are merged with e_3. Note e_3 is removed at the end of the call to $\mathcal{C}ollapse(t)$. All the edges identified with e_3 and all the triangles incident on those edges are also removed in this call. Thus, each edge is only involved in one merge operation.

4.3 The Randomized Algorithm

Algorithm RANDREEB(K, f) randomly permutes vertices of K and performs $\mathcal{S}eqCollapse(v)$ for each vertex v in the permuted order. Because of lazy edge identification, some edges may be identified but not yet merged. As a final step, RANDREEB merges these edges using a linear scan.

Termination. Algorithm RANDREEB calls Procedure $\mathcal{S}eqCollapse(v)$ for each vertex v of K. Procedure $\mathcal{S}eqCollapse(v)$ calls $\mathcal{C}ollapse(t)$ for each triangle t whose middle vertex is v. Procedure $\mathcal{C}ollapse(t)$ can create new triangles whose middle vertex is t. However, the number of such triangles is bounded by the number of triangles intersected by the level set $f^{-}(v)$. Therefore, $\mathcal{S}eqCollapse(v)$ creates only a finite number of new triangles and Algorithm RANDREEB terminates.

Correctness. Let $(v_1, v_2, \ldots, v_n)$ be the vertices of K listed in the order their permuted order. At the i'th iteration of the algorithm, any triangle with middle vertex v_i will be collapsed. By Claim 4.2, no new triangle can be created with middle vertex v_i after the i'th iteration. Hence no triangle with middle vertex v_i can remain at the end of the algorithm, for any $i \in [1, n]$. Thus, Algorithm RANDREEB constructs a graph $G = (V, E_r)$, possibly with multiple edges having the same endpoints.

Finally, note that the algorithm is simply a sequence of $\mathcal{C}ollapse$ procedures. By Lemma 4.1, Procedure $\mathcal{C}ollapse$ does not change the Reeb graph. Thus, the final outcome, which is itself a graph, is exactly the augmented Reeb graph $\mathcal{R}_K(f)$.

4.4 Time Complexity

We now analyze the time complexity of the above randomized algorithm RANDREEB(K, f). The worst case time complexity is $\Omega(|V||T|)$ even when K is a 2-manifold without boundary. A lower-bound example is shown in Appendix A. We now claim that for an arbitrary simplicial 2-complex K, the expected running time of RANDREEB is $O(m \log n)$. Below we first argue that each $\mathcal{C}ollapse$ procedure takes time proportional to the number of triangles it collapses and splits. We then bound the (expected) total number of triangles that will ever be collapsed or split.

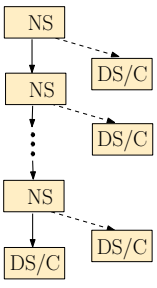
One collapse operation. Given a triangle $t = (e_1, e_2, e_3)$ with spanning edge e_3, $\mathcal{C}ollapse(t)$ destroys all $k = |IncT(e_3)|$ triangles incident to edge e_3, and creates at most $2(k-1)$ new triangles. Destroying the triangle t itself takes $O(1)$ time. Consider a splitting triangle σ incident to e_3. If σ and t do not share the same vertices, then splitting σ creates two new triangles, which can be done in $O(1)$ time. If they share the same vertices, then edges e_1 and e_4 (or e_2 and e_5) may need to be identified. Using lazy edge identification, the identity relation between e_1 and e_4 is recorded in $O(1)$ time. Thus Steps (5)–(17) run in time $O(k)$.

Step (4) merges all the edges identified with e_3, as well as their lists of incident triangles. Since each incident triangle merged becomes an entry in $IncT(e_3)$, merging all incident triangles takes $O(|IncT(e_3)|) = O(k)$ time. The cost of finding and merging all edges identified with e_3 in a transitive way is proportional to the total number of identify-relations recorded in all these edges. We charge the cost of processing each entry to the cost of creating this identity relation in Steps (9) and (10). Since each identity relation, once processed, will be destroyed, each Step (9) or (10) is charged at most once. Thus bounding the total time of Steps (5)–(17) over all calls to Procedure $\mathcal{C}ollapse$ bounds also the total time for Step (4).

Overall analysis. The above analysis says that the total time for all executions of Step (4) is bounded by the total time for all executions of Steps (5)–(17). Hence the total number of triangles ever destroyed provides an upper bound for Algorithm RANDREEB. We now focus on bounding total number of triangles ever destroyed.

Note that a triangle σ can be destroyed either by performing $\mathcal{C}ollapse$ procedure on this triangle itself, or by splitting it while collapsing some other triangle. We call σ a collapsing triangle in the former case, and a splitting triangle as usual in the latter case.

If σ is a collapsing triangle, then it will not create any new triangle. Now suppose σ is split while performing Procedure $\mathcal{C}ollapse(t)$. If σ shares the same vertices with t, then we call it a *degenerate splitting triangle*. In this case, it will not create any further triangle either. Otherwise, it is a *non-degenerate splitting triangle*, and creates two new triangles σ_1 and σ_2. Exactly one of the new triangles, say σ_2, has the same middle-vertex as $q = \mathrm{mid}(t)$ (e.g, $\sigma_2 = \triangle qp_1r$ in Figure 2 (c)). Hence σ_2 will be destroyed within the same $\mathcal{S}eqCollapse(q)$ operation as $\mathcal{C}ollapse(t)$, either being collapsed or split. In fact, if σ_2 is destroyed by splitting, then σ_2 is destroyed when the algorithm collapses another triangle t' that shares the same vertices as σ_2. In this case σ_2 is a degenerate splitting triangle. In summary, whether σ_2 is a collapsing or a splitting triangle, it will be destroyed within Procedure $\mathcal{S}eqCollapse(q)$, and will not create any new triangle. The other new triangle σ_1 will remain untouched throughout the $\mathcal{S}eqCollapse(q)$ operation.



We now charge the existence of the short-lived triangle σ_2 to σ. On the other hand, we say that σ_1 is the *child* of the splitting triangle σ, and σ is the *parent* of σ_1. If σ_1 turns out to be also a collapsing triangle, we charge it to σ as well. In summary, each splitting triangle has at most one child, and can be charged at most twice. These relations are illustrated in the right figure, where 'NS', 'DS' and 'C' stand for non-degenerate splitting, degenerate splitting, and collapsing triangles, respectively. Solid segments indicate parent-child relations, and dotted segments indicate charging relations. Note that the parent-child relations induce a collection of directed paths (the solid path in the right figure is one example). Each such path starts with a triangle $t \in T$ from the original complex K, consists of a sequence of non-degenerate splitting triangles descended from t, and ends with either a collapsing triangle or a degenerate splitting triangle. We call such a path the *descendant-path with head* t. There are exactly $|T|$ number of descendant-paths, each headed by an original triangle from K. If an original triangle is a collapsing triangle, then its descendant-path has length 1 (containing only itself). Every non-degenerate splitting triangle belongs to exactly one descendant-path. The total size of these paths upper-bounds the total number of splitting and collapsing triangles ever existed. We now bound the expected length of each descendant-path.

Expected length of a descendant-path. Let the *index* of a vertex $v \in V$, denoted by $\mathcal{I}(v)$, be its position in the sorted list of vertices based on their function values; that is, $f(v_i) > f(v_j)$ if and only if $\mathcal{I}(v_i) > \mathcal{I}(v_j)$. Given a triangle $t = \triangle abc$, the *range* of t, denoted by $rg(t)$, is the interval spanned by maximum and minimum indices of vertices of t; that is, $rg(t) = [\min\{\mathcal{I}(a), \mathcal{I}(b), \mathcal{I}(c)\}, \max\{\mathcal{I}(a), \mathcal{I}(b), \mathcal{I}(c)\}]$. Now consider a non-degenerate splitting triangle σ with range $rg(\sigma) = [I_1, I_2]$. Let $V_\sigma \subseteq V$ be the set of vertices v of K such that (1) $f(v) \in rg(\sigma)$, and (2) σ intersects the level set connected component containing v. Obviously, as we perform $\mathcal{S}eqCollapse(v)$ for some $v \in V_\sigma$, triangle σ will either be collapsed (thus terminating the descending-path it lies in), or be split. In the latter case, the range of its child σ_1 is either $[I_1, \mathcal{I}(v)]$ or $[\mathcal{I}(v), I_2]$. In either case, $\mathcal{R}(\sigma_1)$ is strictly contained in the range $rg(\sigma)$ of σ. Furthermore, if we choose v randomly from V_σ, then with constant probability, the size of V_{σ_1} is at most half of that of V_σ. Thus intuitively, it can be further split $O(\log |V_\sigma|) = O(\log |V|)$ expected number of times.

This can be made more precise by a standard probabilistic analysis using indicator functions for the length of the range that may appear in a dependent-path with a fixed head. Alternatively, this can be seen by thinking of this splitting process as building a randomized binary search tree, where splitting σ against a vertex $q \in V_\sigma$ corresponds to insert a random key $\mathcal{I}(q)$. At each node of the tree, we record the range of keys in its subtree. The sequence of ranges of triangles on the descendant-path with head σ corresponds to ranges of nodes along some tree path in a randomly built binary tree over keys $I_1, I_1 + 1, \ldots, I_2$. Since the expected height of such a tree is $O(\log(I_2 - I_1)) = O(\log |V|)$ [8], the dependent path has expected length of $O(\log |V|)$. Since there are $|T|$ number of descendant-paths, putting everything together, we conclude with our main result:

Theorem 4.3 *Given an arbitrary simplicial complex K with 2-skeleton (V, E, T) and a piecewise-linear function $f : K \to \mathbb{R}$, Algorithm* RANDREEB*(K, f) computes the augmented Reeb graph $\mathcal{R}(f)$ in expected time $O(m \log n)$ where $m = |V| + |T| + |E|$ and $n = |V|$.*

4.5 Variations

2-manifolds. When K is the triangulation of a 2-manifold, there is no need to use lazy edge identification. Thus Algorithm RANDREEB can be simplified while retaining its $O(|V| \log |V|)$ expected running time. The details can be found in Appendix B.

Using potential critical points. As we sweep K in increasing function values, the 0-th homology of the level sets does not change till some "critical" moments (corresponding to nodes in the Reeb graph). For example, if $|K|$ is a manifold and f is a Morse function, then these critical moments correspond to a certain subset of critical points of f. Instead of applying the $\mathcal{S}eqCollapse(v)$ to all the vertices of K, we need only apply it to the "critical" vertices of K.

We identify a subset of vertices $V_c \subseteq V$ which we call *potential critical points*, randomly permute them, and then apply Procedure $\mathcal{S}eqCollapse$ to these vertices in their permuted order. The resulting complex is not a graph, but it turns out that the Reeb graph can be easily extracted from this complex in linear time. The time complexity of running RANDREEB using only the critical vertices is $O(m \log |V_c|) = O(m \log n)$. In the worst case, this running time is the same as the running time when all vertices are processed, but the revised algorithm is faster when $|V_c|$ is substantially smaller than m. The details are in Appendix C.

5 Experiments

We implemented Algorithm RANDREEB() using the version where we only collapse w.r.t potential critical points as discussed in Section 4.5. In our experiments, this usually improves the running time of the version that processes all points by 10%–50%. Experiments are conducted on a standard desktop with a 64-bit 3.16 GHz Intel Core 2 Duo CPU and 8 GB of RAM. Most data are courtesy of the AIM@SHAPE database [1].

Performance comparison. We compare the performance of our algorithm with that of two state-of-the-art algorithms: the output-sensitive algorithm proposed in [9], denoted by OS, and the loop-surgery algorithm proposed in [26], denoted by LS. Algorithm OS can handle arbitrary simplicial complexes. We use the original implementation of OS kindly provided by the authors of [9]. Algorithm LS processes only tetrahedral meshes modeling 3-manifolds with boundary embedded in $\mathbb{R}^3$. We use published running times reported in [26], which were obtained on a computer of similar configuration as ours. Time for our algorithm is averaged over 5 runs.

Table 5 compares the running times of the three algorithms on a set of tetrahedral meshes of volumetric data (i.e, 3-manifolds with boundary embedded in $\mathbb{R}^3$) used in previous papers [9, 26], and on a set of non-manifold simplicial complexes. See Appendix D.1 for a description of the non-manifold data.

Dataset Statistics					Running Time (sec)		
Classification	Dataset	#Triangles	$\lvert V_c\rvert$	# Loops	Our	OS [9]	LS [26]
Tetrahedron	Fighter	143881	3618	0	5.36	629.40	0.35
	Plasma	2646016	2852	0	114.76	3772.42	2.20
	Earthquake	4198057	11896	0	150.28	5866.71	4.07
	Buckyball	2524284	4378	0	56.75	1602.01	2.51
	Post	1243200	132	1	13.91	18.05	0.69
Non-manifold	Hand	1676884	209	0	6.60	14.10	*
	Camel	144971	176	24	0.26	0.86	*
	Simulation	104127	11804	214	1.31	193.966	*

Table 2: Entries marked with * are unavailable. V_c is the set of potential critical points used by our algorithm.

Our algorithm outperforms algorithm OS in all cases. Algorithm LS is specifically designed for 3-manifolds with boundary embedded in $\mathbb{R}^3$ and gives superior performance on all tetrahedral meshes.

Both our algorithm and algorithm LS identify and use a few potential critical points. Based on a nice observation about 3-manifolds embedded in $\mathbb{R}^3$, Algorithm LS processes only critical points which form a loop in the Reeb graph of the boundary of the 3-manifold. Thus its running time depends on the number of loops in this Reeb graph. While [26] reported timing results for some tetrahedral meshes whose boundary Reeb graphs contained 30-80 such loops, these data sets are not in the public domain. The tetrahedral mesh data sets listed in Table 5 all have 0 or 1 loops. Since the observation used in [26] does not hold for arbitrary simplicial complexes, Algorithm RANDREEB needs to consider a much larger set of potentially critical points, not just the ones contributing to loops in the Reeb graph of domain boundary. The number of such points are on the order of several thousands in the data sets we tested. (See column 4 in Table 5).

Effect of random collapsing. To measure the effect of collapsing in random order, we removed randomization from RANDREEB and called $\mathcal{S}eqCollapse$ on potential critical points in order of increasing function values. We counted the average number of times an input triangle was split by the original randomized version of RANDREEB ($AvgRand$) and by the sequential, non-random version ($AvgSeq$). The *Fighter* data has 3618 potential critical points, RANDREEB splits each triangle 21.16 times on average, while the sequential version splits each triangle 1138 times on average. Note that 21 is on the order of $\log_2(1138)$ which is what we expect by using a random order of collapsing. (Appendix D.2 contains statistics for other data.)

Let C_q denote the connected component of $f^{-1}(q)$ that contains vertex q. $AvgSeq$ is a lower bound on the number of times that a triangle intersects C_q for all potential critical points q. Thus each triangle in the *Fighter* data set intersects an average of (at least) 1138 such connected components. This may in some sense explain the performance difference between our algorithm and algorithm OS from [9], as it may process a triangle every time it is intersected by such a connected component.

References

[1] Aim@shape shape repository, 2006. `http://shapes.aimatshape.net/`.

[2] M. Attene, S. Biasotti, and M. Spagnuolo. Shape understanding by contour driven retiling. *The Visual Computer*, 19(2-3):127–138, 2003.

[3] S. Biasotti, B. Falcidieno, and M. Spagnuolo. Extended Reeb graphs for surface understanding and description. In *Proc. 9th Internat. Conf. Discrete Geom. for Computer Imagery*, pages 185–197, 2000.

[4] S. Biasotti, D. Giorgi, M. Spagnuolo, and B. Falcidieno. Reeb graphs for shape analysis and applications. *Theor. Comput. Sci.*, 392(1-3):5–22, 2008.

[5] H. Carr. *Topological manipulation of isosurfaces*. PhD thesis, The University of British Columbia (Canada), 2004. Adviser-Panne, Michiel Van.

[6] H. Carr, J. Snoeyink, and U. Axen. Computing contour trees in all dimensions. *Comput. Geom*, 24(2):75–94, 2003.

[7] K. Cole-McLaughlin, H. Edelsbrunner, J. Harer, V. Natarajan, and V. Pascucci. Loops in Reeb graphs of 2-manifolds. *Discrete Comput. Geom.*, 32(2):231–244, 2004.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, MA, second edition, 2001.

[9] H. Doraiswamy and V. Natarajan. Efficient output-sensitive construction of Reeb graphs. In *Proc. 19th Internat. Sym. Alg. and Comput.*, pages 556–567, 2008.

[10] H. Doraiswamy and V. Natarajan. Efficient algorithms for computing Reeb graphs. *Computational Geometry: Theory and Applications*, 42:606–616, 2009.

[11] H. Edelsbrunner and J. Harer. *Computational Topology. An Introduction*. Amer. Math. Soc., Providence, Rhode Island, 2009.

[12] H. Edelsbrunner, J. Harer, A. Mascarenhas, V. Pascucci, and J. Snoeyink. Time-varying Reeb graphs for continuous space-time data. *Comput. Geom.*, 41(3):149–166, 2008.

[13] F. Hétroy and D. Attali. Topological quadrangulations of closed triangulated surfaces using the Reeb graph. *Graph. Models*, 65(1-3):131–148, 2003.

[14] M. Hilaga, Y. Shinagawa, T. Kohmura, and T. L. Kunii. Topology matching for fully automatic similarity estimation of 3d shapes. In *Proc. SIGGRAPH '01*, pages 203–212, 2001.

[15] P. Kanongchaiyos and Y. Shinagawa. Articulated Reeb graphs for interactive skeleton animation. In S. Hashimoto, editor, *Multimedia Modeling: Modeling Multimedia Information and System*, pages 451–467. World Scientific, 2000.

[16] I.-H. Park and C. Li. Novel dynamic ligand-induced-fit simulation via enhanced conformational samplings and ensemble dockings: a survivin example, 2009. Submitted to The Journal of Physical Chemistry, B.

[17] V. Pascucci, G. Scorzelli, P.-T. Bremer, and A. Mascarenhas. Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26(3):58, 2007.

[18] G. Patanè, M. Spagnuolo, and B. Falcidieno. Para-graph: Graph-based parameterization of triangle meshes with arbitrary genus. *Comput. Graph. Forum*, 23(4):783–797, 2004.

[19] G. Reeb. Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique. *Comptes Rendus Hebdomadaires des Séances de l'Académie des Sciences*, 222:847–849, 1946.

[20] Y. Shi, R. Lai, S. Krishna, N. Sicotte, I. Dinov, and A. W. Toga. Anisotropic Laplace-Beltrami eigenmaps: Bridging Reeb graphs and skeletons. *Computer Vision and Pattern Recognition Workshop*, 0:1–7, 2008.

[21] Y. Shinagawa and T. L. Kunii. Constructing a Reeb graph automatically from cross sections. *IEEE Comput. Graph. Appl.*, 11(6):44–51, 1991.

[22] Y. Shinagawa, T. L. Kunii, and Y. L. Kergosien. Surface coding based on morse theory. *IEEE Comput. Graph. Appl.*, 11(5):66–78, 1991.

[23] J. Sun, M. Ovsjanikov, and L. J. Guibas. A concise and provably informative multi-scale signature based on heat diffusion. *Comput. Graph. Forum*, 28(5):1383–1392, 2009.

[24] S. P. Tarasov and M. N. Vyalyi. Construction of contour trees in 3D in $o(nlogn)$ steps. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 68–75, 1998.

[25] J. Tierny. *Reeb graph based 3D shape modeling and applications*. PhD thesis, Universite des Sciences et Technologies de Lille, 2008.

[26] J. Tierny, A. Gyulassy, E. Simon, and V. Pascucci. Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Trans. Vis. Comput. Graph.*, 15(6):1177–1184, 2009.

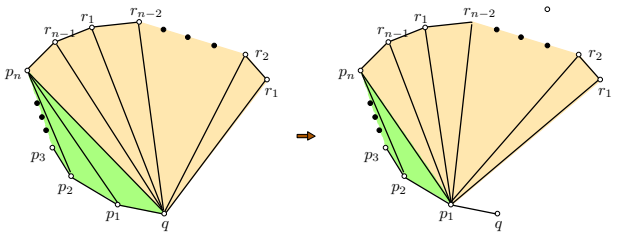
[27] T. Tung and F. Schmitt. The augmented multiresolution Reeb graph approach for content-based retrieval of 3d shapes. *Internat. J. Shape Modeling*, 11(1):91–120, 2005.

[28] M. van Kreveld, R. van Oostrum, C. Bajaj, V. Pascucci, and D. Schikore. Contour trees and small seed sets for isosurface traversal. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 212–220, 1997.

[29] Z. Wood, H. Hoppe, M. Desbrun, and P. Schröder. Removing excess topology from isosurfaces. *ACM Trans. Graph.*, 23(2):190–208, 2004.

[30] E. Zhang, K. Mischaikow, and G. Turk. Feature-based surface parameterization and texture mapping. *ACM Trans. Graph.*, 24(1):1–27, 2005.

A Lower-bound Example for Algorithm RANDREEB



The worst case time complexity for our algorithm RANDREEB(K, f) is $\Omega(|V||T|)$ even when K is a 2-manifold without boundary. A lower-bound example for a manifold with boundary is shown in the right figure. In this example, we process the vertices $\{p_1, p_2, \ldots, p_n\}$ in order. Each $\mathcal{S}eqCollapse(p_i)$ will destroy all $\Theta(n)$ triangles of the form $\triangle p_{i-1} r_j r_{j+1}$ and create $\Theta(n)$ triangles of the form $\triangle p_i r_j r_{j+1}$. Simple modifications of the figure on the right give a worst case $\Omega(|V||T|)$ bound for a 2-manifold without boundary.

B 2-Manifolds

When K is the triangulation of a 2-manifold, it turns out that there is no need to use lazy edge identification. Thus Algorithm RANDREEB can be simplified while retaining its $O(|V| \log |V|)$ expected running time. The simplification is based on the following. With this property, there is no longer any need to use lazy edge identification, since performing merging directly in Steps (9) and (10) takes constant time.

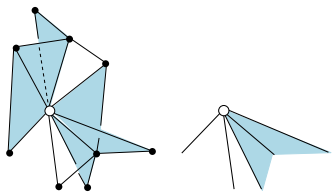
Claim B.1 *If K is the triangulation of a 2-manifold (with or without boundary,) then $|IncT(e)| \leq 2$ for any edge e in any intermediate complex produced by* RANDREEB*(K, f).*

Proof: The claim holds at the beginning for K, since its underlying space is a 2-manifold. Let $IncT_{K'}(e)$ denote the incident list of an edge e in an intermediate complex K'. Let K_1 and K_2 be the complex before and after a $\mathcal{C}ollapse(t)$ operation with $t = \triangle(qp_1p_2)$ and $q = \text{mid}(t)$. Recall Figure 2. After the collapse operation, edge qp_1 will lose one incident triangle (which is t), but gain $k-1$ incident triangles where $k = |IncT_{K_1}(p_1p_2)|$ Hence $|IncT_{K_2}(qp_1)| = |IncT_{K_1}(qp_1)| - 1 + (k-1) \leq 2$ as both $|IncT_{K_1}(qp_1)|$ and k are bounded by 2. The same argument holds for edge qp_2. For each splitting triangle $\sigma = \triangle p_1p_2r$ involved in $\mathcal{C}ollapse(t)$, if $r \neq q$, then the boundary edges p_1r and p_2r will lose one triangle (i.e, σ) but gain one triangle $\triangle qp_1r$ (resp. $\triangle qp_2r$). Hence the size of its incident list remains the same. In the case that $r = q$ and a pair of double edges e_1 and e_2 are identified, we have that $|IncT_{K_2}(e_1)| = |IncT_{K_1}(e_1)| - 1 + |IncT_{K_1}(e_2)| - 1 \leq 2$. The claim thus holds at any time during the algorithm. ■

C Using Potential Critical Points

Algorithm RANDREEB is conceptually simple and easy to implement. However, it ignores any property that input data may have. Specifically, observe that as we sweep K in increasing function values, the H_0 homology of the level sets does not change till some "critical" moments (corresponding to nodes in the Reeb graph). For example, if the $|K|$ is a 2-manifold (resp. d-manifold) and f is a Morse function, then these critical moments correspond to the set of critical points (resp. a certain subset of critical points) of f. Thus it seems wasteful that we perform Procedure $\mathcal{S}eqCollapse$ for all vertices in K, even though many of them do not carry key information.

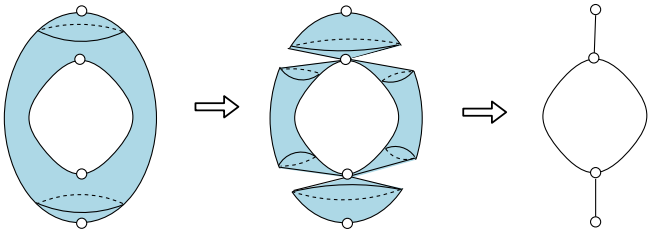
We thus present a revised algorithm RANDREEB(K, f) to potentially reduce the number of $\mathcal{C}ollapse$ operations. First, we say that a simplex α_1 is *lower* (resp. *higher*) than another simplex α_2, if for any two points x and y in the interior of α_1 and α_2, respectively, we have $f(x) < f(y)$ (resp. $f(x) > f(y)$). Given a vertex $v \in K$, its *lower star* (resp. *upper star*) is the union of the interior of simplices incident to v which are lower (resp. higher) than v. See the right figure for an example: The left illustration shows the closure of all simplices incident to the empty dot, and the right one shows its lower star, which contains three components. We say that a vertex $v \in K$ is a *potential critical point* if its lower star or upper star has more than one component. It is easy to verify that the pre-image of any node in the Reeb graph $\mathcal{R}_K(f)$ necessarily contains a potential critical point. The inverse is not true – the image of a potential critical point v may be a monotonic degree-2 point in the interior of an arc of $\mathcal{R}_K(f)$.



We now modify Algorithm RANDREEB() as follows. First, it computes the set of potential critical vertices $V_c \subseteq V$. In the second step, we randomly permute vertices in V_c and perform Procedure $\mathcal{S}eqCollapse$ for only these vertices. Let K' denote the resulting complex after these $\mathcal{S}eqCollapse$ operations. By Lemma 4.1, K' shares the same Reeb graph as K; that is, $\mathcal{R}_{K'}(f) = \mathcal{R}_K(f)$. Furthermore, the following result suggests that K' almost has the structure of its Reeb graph, even though it is not a graph.

Lemma C.1 *Let $\Phi_f : |K'| \to \mathcal{R}_{K'}(f)$ be the continuous surjection from K' to its Reeb graph. For every point $v \in V$, there is a one-to-one correspondence between components in the upper-star of v in K' and the up-branches of $\Phi_f(v)$ in $\mathcal{R}(f)$. Similarly, there is a one-to-one correspondence between components in the lower-star of v in K' and the lower-branches of $\Phi_f(v)$ in $\mathcal{R}(f)$.*

See the right figure for an example where the underlying domain of input simplicial complex is a torus. There are four potential critical points in V_c in this case (empty dots). Consider the lower saddle point. Note that its lower star in the input manifold has two components. After performing $\mathcal{S}eqCollapse$ for all points in V_c in K' (middle figure), its lower star has only one component, just like in the final Reeb graph (right figure).



Lemma C.1 states that the local neighborhood of every vertex in K' has the same connected component information as its image in the Reeb graph $\mathcal{R}(f)$. Hence in the third step of the algorithm, we use K' to connect vertices in V_c to obtain the Reeb graph of f[1]. In particular, for every $v \in V_c$ and for each component of its upper star, we start an upward path by connecting v to an arbitrary vertex v' of K in the closure of that component. We then connect v' to an arbitrary vertex v'' connected to v' by an edge and with a higher function value. We continue this tracing until we reach another potential critical vertex from V_c. Note that any intermediate vertex we reach along this upward path can have only one component in its upper star (as otherwise, this vertex should be in V_c and the tracing is terminated). It follows from Lemma C.1 that the resulting graph consisting of the upward tracing paths is the Reeb graph of f.

It is easy to see that the expected time complexity of the revised algorithm RANDREEBis $O(m \log |V_c|) = O(m \log n)$. In worst case this is the same as as the running time when processing all vertices, but is faster when V_c is substantially smaller than n.

[1]Note that this is not the augmented Reeb graph generated by K any more, as not all vertices will appear as a node in the final Reeb graph. It is a Reeb graph of f augmented by some vertices from K.

D More on Experiments

D.1 Data Sets

The tetrahedral meshes are downloaded from AIM@SHAPE database [1], and were used in previous papers [9, 26]. For non-manifold data set, the *Simulation* data is obtained by constructing the Rips complex from a set of high-dimensional points, where each point corresponds to a protein conformation produced by a molecular simulation process [16]. The function value at a point is the energy of the corresponding protein conformation. The *Camel* data is the Rips complex constructed from an incomplete point clouds of a camel model. The function value is the heat-signature defined in [23]. The *Hand* data is obtained from AIM@SHAPE database.

D.2 Effect on Random Collapse

Dataset	# Triangles	$\|V_c\|$	$AvgSeq$	$AvgRand$
Fighter	143881	3618	1138.40	21.16
Plasma	2646016	2852	*	15.57
Earthquake	4198057	11896	*	13.95
Buckyball	2524284	4378	104.3	11.1
Blunt	451601	827	103.6	12.46
Post	1243200	132	9.97	5.62
Hand	1676884	209	2.34	2.3
Camel	144971	176	3.44	3.16
Simulation	104127	11804	40.92	7.64

Table 3: The average number of times an input triangle is split, using sequential vertex-collapsing ($AvgSeq$) and random vertex-collapsing ($AvgRand$). Only input triangles that are destroyed are counted when computing this average. V_c is the set of potential critical points that our algorithm processes. Entries marked with (*) as the algorithm runs out of memory when using sequential vertex-collapsing operations.