Maintaining Reeb Graphs of Triangulated 2-Manifolds*†

Pankaj K. Agarwal[1], Kyle Fox[2], and Abhinandan Nath[3]

1 **Duke University, Durham, NC, USA**
pankaj@cs.duke.edu
2 **Duke University, Durham, NC, USA**
kylefox@cs.duke.edu
3 **Duke University, Durham, NC, USA**
abhinath@cs.duke.edu

Abstract

Let $\mathbb{M}$ be a triangulated, orientable 2-manifold of genus g without boundary, and let h be a height function over $\mathbb{M}$ that is linear within each triangle. We present a kinetic data structure (KDS) for maintaining the Reeb graph $\mathcal{R}$ of h as the heights of $\mathbb{M}$'s vertices vary continuously with time. Assuming the heights of two vertices of $\mathbb{M}$ become equal only $O(1)$ times, the KDS processes $O((\kappa + g)n \operatorname{polylog} n)$ events; n is the number of vertices in $\mathbb{M}$, and κ is the number of *external* events which change the combinatorial structure of $\mathcal{R}$. Each event is processed in $O(\log^2 n)$ time, and the total size of our KDS is $O(gn)$. The KDS can be extended to maintain an augmented Reeb graph as well.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases Reeb graphs, 2-manifolds, topological graph theory

Digital Object Identifier 10.4230/LIPIcs.FSTTCS.2017.8

1 Introduction

Let $h : \mathbb{X} \to \mathbb{R}$ be a function over a manifold $\mathbb{X}$. We refer to h as the height function for simplicity. Given a height value ℓ, the *ℓ-level set* of h is the set of all points on $\mathbb{X}$ whose heights equal ℓ. A *contour* is a connected component of a level set. The *Reeb graph* [15] of h encodes the topological changes to the contours of ℓ-level sets as one continuously varies the value ℓ. Reeb graphs are the generalization of contour trees, which are trees that encode the contours of functions defined on zero-genus manifolds, to manifolds of non-zero genus, and contain cycles if the genus of $\mathbb{X}$ is not zero.

The "height" functions may measure any manner of things including temperature, pressure, brightness of points on a shape, etc. Reeb graphs have a variety of applications in graphics, data visualization, and shape matching and retrieval; see [5,10,20] for some of them. However, many of the potential quantities used for height functions including those listed above may change over time. New heat measurements may be taken, or a user interacting with the lighting on a shape may wish to change the brightness of certain locations. In this work, we develop an efficient data structure, using the kinetic data structure framework [4], for

* Work on this paper is supported by NSF under grants CCF-15-13816, CCF-15-46392, and IIS-14-08846, by ARO grant W911NF-15-1-0408, and by grant 2012/229 from the U.S.-Israel Binational Science Foundation.

† See https://users.cs.duke.edu/~abhinath/dynamicReebGraphs.pdf for a full version of the paper.

37th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS 2017).
Editors: Satya Lokam and R. Ramanujam; Article No. 8; pp. 8:1–8:14

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

maintaining the Reeb graph as the height function varies continuously. Note that despite the height function changing continuously, the combinatorial structure of the Reeb graph changes only at certain discrete times, called *events*. Our goal is to characterize and detect these events, and to update the Reeb graph efficiently at these events.

Related work. Shinagawa and Kunii [16] describe an $O(n^2)$-time algorithm for constructing the Reeb graph of a piecewise linear height function over a 2-manifold with n triangles. Cole-McLaughlin *et al.* [6] improved the running time to $O(n \log n)$. Later, $O(n \log n)$-time algorithms were designed for the more general case of piecewise linear functions over simplicial complexes with n vertices, edges, and triangles [12, 13]. Pascucci *et al.* [14] give an online algorithm for computing Reeb graphs. Doraiswamy and Natarajan [8] describe an algorithm that splits the input domain containing s saddles into regions whose Reeb graphs are loop-free, computes these Reeb graphs, then combines them to get the final Reeb graph in $O(n \log n + ns)$-time.

Most work on time-varying height functions has focused on contour trees. Sohn and Bajaj [18] and Szymczak [19] compute a mapping between the contour trees of a changing height function at successive discrete time steps, but they ignore combinatorial changes in the contour tree between the time steps and compute each new tree using a static algorithm. Agarwal *et al.* [2] provide a detailed characterization of the changes in the contour tree of a time-varying function over a terrain, and they describe a kinetic data structure (KDS) [4] that can update the contour tree in $O(\log n)$ time per major data structure modifying event. See [11] for a survey on other kinetic data structures.

Edelsbrunner *et al.* [9] actually characterize the combinatorial changes in the *Reeb graph* of a time-varying function; however, their approach only considers smooth functions on a manifold that is compactified to what is effectively a 3-sphere (hence the Reeb graph has no loops). Moreover, they assume that the height function over all time values is given in advance, and the algorithm takes $O(n)$ time to update the graph at each event.

Our work. *et al.* [2] to Reeb graphs of more general domains than the terrain. However, their results rely heavily on the planarity of terrains and the simple connectivity of their contour trees. Therefore, we consider domains that nearly feature both of these properties while still capturing far more general settings. Namely, we focus on piecewise linear height functions over triangulated 2-manifolds of low genus. For simplicity, we will assume these manifolds are orientable and lack boundary.

We describe the first KDS to efficiently maintain the Reeb graphs of such functions. Fix a triangulated 2-manifold $\mathbb{M}$ of genus g and let $h : \mathbb{M} \to \mathbb{R}$ be a height function. We assume the height of each vertex is a constant degree polynomial function of time and that the roots of these polynomials can be computed in $O(1)$ time[1]. Let κ be the number of combinatorial changes that occur to the Reeb graph of h. Our KDS processes $O((\kappa + g)\lambda_{O(1)}(n) \log n)$ events in the worst case, where $\lambda_{O(1)}(.)$ is the maximum length of a Davenport-Schinzel sequence of order $O(1)$ and is almost linear [3]. Each event can be processed in $O(\log^2 n)$ time, and the total size of our KDS is $O(gn)$. Note that if we are allowed $O(n^2)$ space, it is not too difficult to maintain the Reeb graph in $O(\text{polylog}\, n)$ time per event, since we can explicitly store the $O(n)$ combinatorially different level sets, each of size $O(n)$, and easily maintain them dynamically.

[1] Our algorithm defines events as those time instances where the relative ordering of vertices according to their function values changes. We can also simulate arbitrarily changing the height of a vertex v from h_1 to h_2 through a sequence of vertex swaps involving v and the vertices with function values lying between h_1 and h_2.

To obtain our data structure, we extend and simplify the detailed characterization of the combinatorial changes in the contour tree described by Agarwal *et al.* [2] to the case of Reeb graphs over 2-manifolds. One advantage of our analysis is that we no longer distinguish between the inside and outside of contours; indeed, doing so is impossible unless we define an outer face with which to define containment. We also keep our list of distinct changes to the Reeb graph as generic as possible, helping us avoid a potentially lengthy case analysis for situations our data structure's algorithms can handle implicitly.

On the other hand, we need to process more events than the $O(\kappa + n)$ needed for contour trees [2]. Moreover, because Reeb graphs are more general than contour trees, and the changes in Reeb graphs at an event are more complex, the data structure itself is more involved. A major component of Agarwal *et al.*'s data structure is to locate a given vertex v's contour on the contour tree. To do so, they find a pair of nodes μ_1 and μ_2 on the contour tree such that there is a unique height-monotone path between them containing v, and then they perform a binary search along the path. Unfortunately, the presence of loops in a Reeb graph means that it may contain multiple such paths between a pair of nodes. To solve this problem, we explicitly maintain g level sets of the height function and a spanning tree of the Reeb graph. Using the level sets, we can find a pair μ_1 and μ_2 as before such that the unique height-monotone path *in the spanning tree* contains v. We take advantage of the topology of 2-manifolds to efficiently maintain these level sets. In particular, we recognize that contours essentially form cycles in the *dual graph* of a manifold, and as the level sets change these cycles should and can be modified easily.

The rest of the paper is organized as follows. We briefly discuss the necessary background in Section 2. We introduce our model of time-varying height functions, and give a characterization of the changes in the Reeb graph in Section 3. We describe a KDS, including a few useful operations for maintaining a dynamic Reeb graph in Section 4, and we describe how to efficiently update the KDS in Section 5.

2 Preliminaries

2-manifolds. A triangulated 2-manifold $\mathbb{M} = (V, E, F)$ without boundary consists of a set of triangles F bounded by a set of edges E incident to a set of vertices V. Let $n = |F|$. We require each edge to bound two distinct triangles and to contain two distinct incident vertices. Thus, every point on $\mathbb{M}$ has an open neighborhood homeomorphic to the plane $\mathbb{R}^2$. A 2-manifold is orientable if it does not contain a subset homeomorphic to the Möbius band. We consider only orientable 2-manifolds in this paper. Let $\chi = |V| - |E| + |F|$ denote the *Euler characteristic* of $\mathbb{M}$. The *genus* g of a 2-manifold is the maximum number of disjoint simple cycles through the manifold that can be removed without disconnecting it. By Euler's formula, we have $\chi = 2 - 2g$. For simplicity, we assume every vertex v has degree $O(1)$. Our data structure can be modified easily to handle arbitrary vertex degree; however the processing time of events will increase proportionally to the degree of vertices involved in the event. Let $\mathbb{M}^*$ denote the *dual graph* of $\mathbb{M}$. The graph $\mathbb{M}^*$ contains a vertex for every triangle of $\mathbb{M}$, and two of its vertices are adjacent if and only if their corresponding triangles share an edge in $\mathbb{M}$. Let $h : \mathbb{M} \to \mathbb{R}$ be a *height function*. We assume that the restriction of h to each triangle of $\mathbb{M}$ is linear and that the heights of all vertices are distinct.

Critical points, level sets, and contours. The *link* of v, denoted by $\mathrm{Lk}(v)$, is the cycle formed by those edges of $\mathbb{M}$ whose endpoints are neighbors of v. The lower/down (resp. higher/up) link of v, $\mathrm{Lk}^-(v)$ (resp. $\mathrm{Lk}^+(v)$), is the subgraph of $\mathrm{Lk}(v)$ induced by vertices

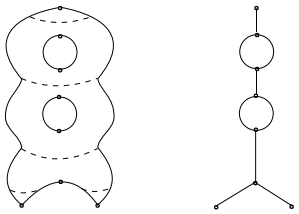

Figure 1 Reeb graph $\mathcal{R}$ on 2-manifold $\mathbb{M}$; the function h is given by height from the base.

u with $h(u) < h(v)$ (resp. $h(u) > h(v)$). The down (resp. up) edges of v are the edges between v and its lower (resp. upper) link. A *minimum* (resp. *maximum*) of $\mathbb{M}$ is a vertex v for which $\mathrm{Lk}^-(v)$ (resp. $\mathrm{Lk}^+(v)$) is empty. Such a vertex is also called an *extremal* vertex. A non-extremal vertex v is *regular* if $\mathrm{Lk}^-(v)$ (and also $\mathrm{Lk}^+(v)$) is connected, and *saddle* otherwise. A vertex that is not regular is called a *critical* vertex.

For $\ell \in \mathbb{R}$, the *ℓ-level set* of $\mathbb{M}$, denoted by $\mathbb{M}_\ell$, consists of points $x \in \mathbb{M}$ with $h(x) = \ell$. We refer to a level set $\mathbb{M}_\ell$ where $\ell = h(v)$ for some critical vertex v as a *critical level*. A *contour* of $\mathbb{M}$ is a connected component of a level set of $\mathbb{M}$. Each vertex $v \in V$ is contained in exactly one contour in $\mathbb{M}_{h(v)}$, which we call *the contour of v*. A contour not passing through a critical vertex is a simple polygonal cycle. A contour passing through an extremal vertex is a single point, and a contour passing through a saddle v consists of two or more simple cycles, one per component of $\mathrm{Lk}^-(v)$, with v being their only intersection point. A contour C not passing through a vertex can be represented by the cyclic sequence of edges of $\mathbb{M}$ that it passes through. Similarly, a contour C passing through a vertex can be represented by zero or more sequences of edges, depending on whether the vertex is critical, where each sequence represents a cycle of C.

Let $\varepsilon = \varepsilon(\mathbb{M})$ denote a positive value smaller than the height difference between any two vertices of $\mathbb{M}$. An *up-contour* (resp. *down-contour*) of a saddle vertex v is any contour of $\mathbb{M}_{h(v)+\varepsilon}$ (resp. $\mathbb{M}_{h(v)-\varepsilon}$) that intersects an edge incident on v. If v has two up-contours (resp. down-contours) and one down-contour (resp. up-contour) it is called a *positive (resp. negative) saddle vertex*. A saddle vertex v that is either negative or positive (but not both) is called a *simple* saddle (this also means that $\mathrm{Lk}^-(v)$ and $\mathrm{Lk}^+(v)$ each consist of two connected components). For simplicity, we assume that each saddle vertex v is *simple*. As before, our data structure can be modified easily to handle vertices that are not simple.

Reeb graphs. Consider raising ℓ from $-\infty$ to ∞. The contours continuously deform, but the level set's topology does not change while $\mathbb{M}_\ell$ varies between two consecutive critical levels. A new contour appears as a single point at a minimum vertex, and an existing contour contracts into a single point and disappears at a maximum vertex. An existing contour (the down-contour of v) splits into two new contours (the up-contours of v) at a positive saddle vertex v, and two contours (the down-contours of v) merge into one contour (the up-contour of v) at a negative saddle vertex v. The *Reeb graph* $\mathcal{R}$ of h is a graph on the critical vertices of $\mathbb{M}$ that encodes these topological changes of the level set. An arc (v, w) of $\mathcal{R}$ *represents* a contour that appears at v and disappears at w. If $\mathbb{M}$ has genus g, then $\mathcal{R}$ is a tree with exactly g additional arcs [6] ($\mathcal{R}$ may contain *parallel arcs* between two nodes; each such arc is considered distinct). We will refer to vertices and edges in $\mathbb{M}$ and *nodes* and *arcs* in $\mathcal{R}$.

More formally, two contours C_1 and C_2 at heights ℓ_1 and ℓ_2, respectively, are called *equivalent* if C_1 and C_2 belong to the same connected component of $\Gamma = \{x \in \mathbb{M} \mid \ell_1 \leq h(x) \leq \ell_2\}$ and that component of Γ does not contain any critical vertex. An equivalence

class of contours starts and ends at critical vertices. If a class starts at a critical vertex v and ends at w, then (v, w) is an arc in $\mathcal{R}$. We refer to v as a *down neighbor* of w and to w as an *up neighbor* of v. A down arc (resp. up arc) of v goes to another node on $\mathcal{R}$ of lesser (resp. greater) height. Equivalently, $\mathcal{R}$ is the quotient space in which each contour is represented by a point and connectivity is defined in terms of the quotient topology. Let $\rho : \mathbb{M} \to \mathcal{R}$ be the associated quotient map, which maps all points of a contour to a single point on $\mathcal{R}$. Fix a point p in $\mathbb{M}$. If p does not lie in the contour of a critical vertex, $\rho(p)$ lies in the relative interior of an arc in $\mathcal{R}$; if p is an extremal vertex, $\rho(p)$ is a leaf node of $\mathcal{R}$; and if p lies in the contour of a saddle vertex then $\rho(p)$ is a non-leaf node of $\mathcal{R}$. See Figure 1. We extend the height function h to $\mathcal{R}$ by defining $h(\rho(p)) = h(p)$ for all $p \in \mathbb{M}$.

Each node of $\mathcal{R}$ is labeled with the corresponding critical vertex of $\mathbb{M}$. A non-leaf node of $\mathcal{R}$ is called *positive* (resp. *negative*) if its corresponding saddle vertex is positive (resp. negative). The combinatorial description of $\mathcal{R}$ is the set of its nodes along with their labels, and the set of its arcs.

3 Time varying Reeb graph

We use the same model for the time varying function as the one in [2]. Specifically, we have a one-parameter family of height functions over $\mathbb{M}$, $h : \mathbb{M} \times \mathbb{R} \to \mathbb{R}$, where the extra dimension in the domain is time. We assume that at any given time at most two vertices have equal height value, and the heights of any two vertices become equal $O(1)$ times. Finally, if $h(u, t_0) = h(v, t_0)$ then the function $h(u, t) - h(v, t)$ changes sign at $t = t_0$. We now consider the different changes that may occur in $\mathcal{R}$ as time passes. We call each time t where the topology or node labels of $\mathcal{R}$ change an *external event*. We focus on external events in this section. In later sections, we define other types of events that only affect our data structure but not $\mathcal{R}$ itself; we refer to these events as *internal events*. If an event occurs at time t, we let t^- (resp. t^+) denote the time immediately before (resp. after) the event, i.e., $t^- = t - \delta$ (resp. $t^+ = t + \delta$) for some arbitrarily small $\delta > 0$.

We characterize external events by whether or not they change the set of nodes in $\mathcal{R}$. We refer to events that do change the nodes of $\mathcal{R}$ as *node* events. Recall that the nodes of $\mathcal{R}$ are the images of critical vertices of $\mathbb{M}$. Therefore, a node $\rho(u)$ enters or leaves $\mathcal{R}$ only when the uplinks or downlinks of u in $\mathbb{M}$ change their connectivity or when u becomes an extremum. These events occur only at a time t for which $h(u, t) = h(v, t)$ for some neighbor v of u. Immediately before and after the node event, no other vertex has the topology of its links changed, so only $\rho(u)$ and $\rho(v)$ can be added or removed from $\mathcal{R}$.

Changes to $\mathcal{R}$ that do not change the set of nodes only occur when two adjacent non-leaf nodes in $\mathcal{R}$ go to the same height value and the number of arcs remains the same, and are referred to as *interchange* events. We now explore both types of events in more detail.

Node events. Suppose a node $\rho(u)$ enters or leaves $\mathcal{R}$ at a time t, and let v be the neighbor of u for which $h(u, t) = h(v, t)$. Suppose u becomes a saddle vertex so that $\rho(u)$ is added to $\mathcal{R}$ with three incident arcs. One of two situations may occur. First, $\rho(u)$ subdivides an arc of $\mathcal{R}$, adding one additional arc to $\mathcal{R}$. In this case, $\rho(v)$ must enter $\mathcal{R}$ as well with only a single arc incidence. In other words, v is a new extremal vertex. We call this node event a *birth* event. Second, $\rho(u)$ replaces a node of $\mathcal{R}$ which must be $\rho(v)$. We call this type of node event a *shift* event (shift events occur because $\mathbb{M}$ is a piecewise-linear manifold; they do not occur for smooth surfaces). In opposition to the birth events, both $\rho(u)$ and $\rho(v)$ may leave $\mathcal{R}$. The removal of a saddle and extremal vertex is a *death* event. See Figure 2.

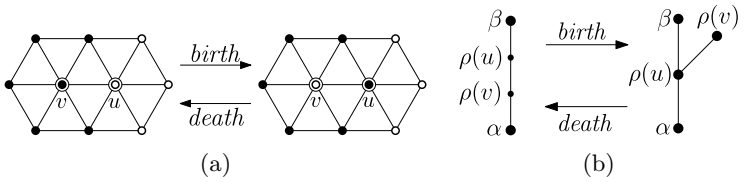

Figure 2 Birth/death event : (a) shows the changes in $\mathbb{M}$, and (b) shows the changes in $\mathcal{R}$. Solid vertices have lesser height than u and v; empty vertices have greater height. Figures are from [2].

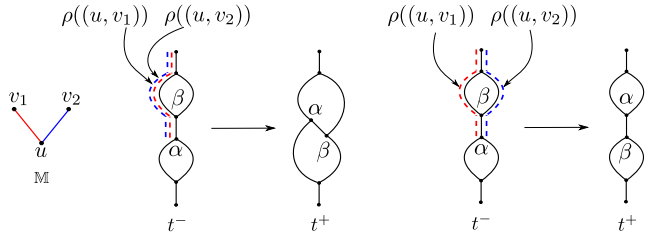

Figure 3 An interchange event where α is negative and β is positive at time t^-. We have $\alpha = \rho(u)$, and v_1 and v_2 are two vertices, one from each uplink of u. The figure shows the two possible ways in which (u, v_1) and (u, v_2) can map to $\mathcal{R}$, and how it determines the changes in $\mathcal{R}$.

Interchange events. The combinatorial changes that occur in $\mathcal{R}$ during node events are easy to deduce simply by looking at which vertices are involved in each event and examining the changes to their links' topology. However, interchange events are more subtle.

Suppose an interchange event occurs at time t where for an arc (α, β) we have $h(\alpha, t^-) < h(\beta, t^-)$ and $h(\beta, t^+) < h(\alpha, t^+)$. Only the neighborhood of α and β changes, because the relative ordering over vertices mapping to arcs outside the neighborhood do not change. We will focus on what the new up-arcs for α should be; other new up-arcs to neighbors of α and β must come from β. The case for the new down-arcs of β is symmetric. The following lemma describes the up-arcs based purely on which arcs of $\mathcal{R}$ incident to α and β intersect the projections of up-edges emanating from u. Note that in the presence of genus, there may be parallel arcs between a pair of nodes undergoing an interchange event. The two arcs are still present after the event, but their endpoints switch labels.

▶ **Lemma 1.** *Let (α, β) be an arc at time t^- such that $h(\alpha, t^-) < h(\beta, t^-)$ and $h(\beta, t^+) < h(\alpha, t^+)$. Let ξ be a node such that $h(\xi, t) > h(\alpha, t) = h(\beta, t)$. There exists one arc (α, ξ) at time t^+ for each instance of the following occurrences: either (i) there exists an arc (α, ξ) at time t^-, or (ii) there exists an arc (β, ξ) at time t^- such that there is an up-edge (u, v) of vertex u with $\alpha = \rho(u)$ and $\beta \neq \rho(v)$ such that $\{\beta\} \subsetneq \rho((u, v)) \cap (\beta, \xi)$.*

See Figure 3 for an example, where α is negative and β is positive at time t^-, and the two ways in which $\mathcal{R}$ can change. We remark that in condition *(ii)*, the requirement states that $\{\beta\}$ is a *proper* subset of $\rho((u, v)) \cap (\beta, \xi)$. See the full paper for the proof of Lemma 1.

According to Lemma 1, the changes to $\mathcal{R}$ during an interchange event between nodes $\alpha = \rho(u)$ and $\beta = \rho(u')$ can be completely determined after figuring out how the maps of edges incident to u and u' intersect arcs incident to α and β. In the next section, we present a kinetic data structure that aids in determining these intersections.

4 KDS for $\mathcal{R}$

We begin with a high level overview of our KDS for maintaining $\mathcal{R}$, including details on some auxiliary data structures we use. Our data structure, although similar in some aspects to the one of Agarwal *et al.* [2], is necessarily more complicated to handle the additional topology present in $\mathbb{M}$ and $\mathcal{R}$. Unlike in their KDS, ours explicitly maintains a collection of g level sets. These level sets are chosen so that there is at most one path between any pair of points in $\mathcal{R}$ that avoids crossing the level sets. We use this property in the following way: in order to compute changes to $\mathcal{R}$ during external events, we need to be able to locate $\rho(v)$ on $\mathcal{R}$ for any vertex v. This is accomplished by computing the first extremal vertices or the aforementioned level sets reached while greedily walking along up-edges (resp. down-edges) from v and then searching for the unique arc at v's height in the aforementioned path in $\mathcal{R}$ between these positions.

Our data structure maintains the Reeb graph $\mathcal{R}$, a spanning tree $\mathcal{T}$ of $\mathcal{R}$, and the set $\mathcal{L} = \mathcal{R} - \mathcal{T}$ of *loop arcs*. Set $\mathcal{L}$ contains exactly g arcs [6, Lemma A]. For each loop arc, we select a vertex $x \in \mathbb{M}$ such that $\rho(x)$ lies in the *closure* of the loop arc (i.e., the arc along with its two endpoints). We designate this set of vertices $\mathcal{V}$ as the *representative vertices* of $\mathcal{L}$. Since the maximum degree of $\mathcal{R}$ is at most 3, a single node may be incident to up to 2 loop arcs. That node's saddle vertex may therefore be representative for up to 2 loop arcs.

We store $\mathbb{M}$ in the form of a doubly connected edge list (DCEL) [7]. For each negative (positive) saddle vertex, we associate with each connected component of its down (up) link the arc of $\mathcal{R}$ holding down(up)-contours through the component. We maintain an *event queue* as a priority queue that, at any point in time, helps us to efficiently compute the next event to occur. Finally, we maintain two kinds of auxiliary data structures detailed below: (i) the *crossing level sets* which we define as the level sets at the same heights as representative vertices, and (ii) a forest of *descent and ascent trees*.

Crossing level sets. Let x be a representative vertex. We refer to the $h(x)$-level set $\mathbb{M}_{h(x)}$ as x's *crossing level set*. We explicitly maintain the edges intersected internally by x's crossing level set. For each regular contour avoiding x, we store a single *anchor edge* e and a path through $\mathbb{M}^*$ between e's incident triangles (which are vertices in $\mathbb{M}^*$) containing all triangles and edges other than e intersected by the contour (see Figure 4). For the contour passing through x, we store one or two paths (depending on whether x is a saddle) through $\mathbb{M}^*$ containing all the triangles and edges intersected internally by the one or two cycles in x's contour (see Figure 4). These paths are all stored in a single forest of edge and vertex-disjoint trees. Note that each edge and triangle of $\mathbb{M}$ is intersected by 0 to g crossing level sets. We store distinct copies of the edges and triangles in each crossing level set intersecting them internally. For each edge, we store a list of its copies ordered by height of their crossing level set. Let $\mathrm{Cr}_-(e)$ and $\mathrm{Cr}^-(e)$ denote the lowest and highest copies of e respectively. The total size of these crossing level sets is $O(gn)$.

We annotate crossing level set contours with the arcs/nodes they lie on in $\mathcal{R}$. If a crossing level set contour does not contain a vertex or passes through a regular vertex, then it is simply annotated with the arc in $\mathcal{R}$ containing its image under ρ. If the contour passes through a saddle vertex x, the annotations are slightly different. Suppose x is a negative (positive) saddle vertex. Let $C(x)$ be x's contour, and let C_1 and C_2 be x's two down(up)-contours, corresponding to the two down (up) arcs a_1 and a_2 of $\rho(x)$ respectively. Observe that each cycle of $C(x)$ shares intersected edges with exactly one of C_1 and C_2. For $i \in \{1, 2\}$, we annotate the contour cycle sharing edges with C_i with the arc a_i. Finally, we annotate each

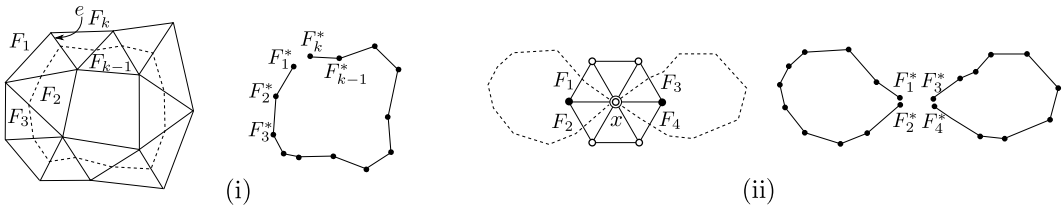

Figure 4 (i) (left) A crossing level set contour avoiding vertices of $\mathbb{M}$, along with its anchor edge e; (right) the contour is stored as a path in the dual graph between e's adjacent triangles. (ii) (left) A crossing level set contour passing through saddle vertex x; (right) the contour is stored as two paths. Recall that the dual of a triangle $F \in \mathbb{M}$ is a vertex $F^* \in \mathbb{M}^*$.

contour with the crossing level set it belongs to, and separately annotate each crossing level set with its representative vertex so it can be changed easily. Later, we describe how to use these annotations to take any edge e of $\mathbb{M}$, discover which crossing level sets intersect e, and learn which arc(s) of $\mathcal{R}$ contain the intersection and its neighborhood.

Finally, for each representative vertex x, we store two *dynamic and kinetic tournaments* $\psi^-(x)$ and $\psi^+(x)$ for vertices incident to edges crossed by x's crossing level set [1]. Tournament $\psi^-(x)$ stores such vertices u with $h(u) < h(x)$. At any point in time, $\psi^-(x)$ can be efficiently queried to provide the highest vertex lower than x. We call this vertex the *winner* of $\psi^-(x)$. The other tournament $\psi^+(x)$ stores the vertices u with $h(u) > h(x)$ and can be queried for its winner, i.e., the lowest vertex in $\psi^+(x)$ higher than x. Vertices can be added or removed from $\psi^-(x)$ and $\psi^+(x)$ as required by our data structure. Maintaining the tournaments introduces additional internal *tournament events*. We discuss the running times associated with modifying and maintaining the tournaments later.

Recall that a single vertex x may be the representative of up to two loop arcs. In this case, we store two copies of x's crossing level set so that they can be modified independently as our data structure requires.

Descent and ascent trees. We maintain a forest of rooted descent and ascent trees [2]. The descent trees are defined as follows. Each vertex of $\mathbb{M}$ appears exactly once in the set of descent trees. We also include a collection of special root vertices, called *level set roots*, that each have exactly one child and a label to an edge of $\mathbb{M}$. For each vertex $u \in \mathbb{M}$, if u is not a minimum and not a representative vertex, then we choose an arbitrary vertex $w \in \mathrm{Lk}^-(u)$. If edge (u, w) does not intersect a crossing level set internally, then w is made u's parent in the descent tree. Otherwise, we create a level set root c, set c to be u's parent, and record on c the edge (u, w) that would have been added to the descent trees if the level set were not crossed. Ascent trees are constructed in a similar manner by taking arbitrary vertices from uplinks and either adding edges to them or adding edges to level set roots. Note that the roots of our ascent and descent trees can be extremal vertices, representative vertices, or level set roots.

Final details. The spanning tree $\mathcal{T}$ of $\mathcal{R}$, the paths representing contours of the crossing level sets, and the ascent and descent trees are all stored as (rooted) *link-cut trees*. Link-cut trees support many basic tree operations in $O(\log n)$ time each [1, 17]. See the full paper

[2] Our definitions of descent and ascent trees differ from those of Agarwal *et al.* [2] due to the existence of crossing level sets.

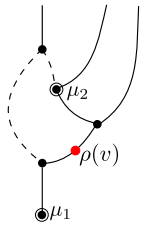

Figure 5 A height monotone path between μ_1 and μ_2 through $\mathcal{T}$ and containing $\rho(v)$. The dotted arcs are loop arcs.

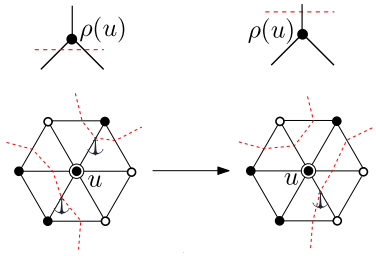

Figure 6 An LSCross(x, u) operation where u is a negativesaddle. The red dotted line represents x's crossing level set, and the anchor edges are shown.

for further details. It is possible to look up or store a constant amount of information on individual trees of a link-cut tree forest (such as a contour's anchor edge) in $O(\log n)$ time per operation.

Initialization. The Reeb graph $\mathcal{R}$, spanning tree $\mathcal{T}$, loop arcs $\mathcal{L}$, and the set $\mathcal{V}$ can be computed in $O(n \log n)$ time [6]. The ascent and descent trees can be initialized in $O(n \log n)$ time. The edges in the crossing level set of a vertex $x \in \mathcal{V}$ can be computed by traversing the edges of $\mathbb{M}$ crossing height $h(x)$ in $O(n)$ time, and can be stored as link-cut trees and tournaments $\psi^-(x)$ and $\psi^+(x)$ in $O(n \log n)$ time [1, 17]. There are at most g such level sets, so overall the time taken is $O(gn \log n)$.

4.1 Useful operations

We define the following operations used by our data structure. The first two are useful in determining the combinatorial changes to $\mathcal{R}$ during external events. The latter two aid us in maintaining crossing level sets as their heights change.

- FindArc(v) : Returns the arc of $\mathcal{R}$ containing $\rho(v)$ for a regular vertex v.
- LocateArc(α, β, v) : Suppose $(\alpha, \beta) \in \mathcal{R}$ is an arc and no other vertex has height between $h(\alpha)$ and $h(\beta)$. Let (u, v) be an edge of $\mathbb{M}$ with $\alpha = \rho(u)$, $\beta \neq \rho(v)$, and $h(\beta)$ lies between $h(\alpha)$ and $h(v)$. The operation LocateArc(α, β, v) returns the only arc $(\beta, \beta') \in \mathcal{R}$ incident to β, other than (α, β), such that $\{\beta\} \subsetneq (\beta, \beta') \bigcap \rho(u, v)$.
- LSCross(x, u): Let $x \in \mathcal{V}$ and u be any other vertex such that $h(x, t) = h(u, t)$. The operation LSCross(x, u) updates x's crossing level set(s) from intersecting down edges (up edges) of u at time t^- to intersecting up edges (down edges) of u at time t^+.
- Handoff(x, x'): Let $x \in \mathcal{V}$ and x' be any other non-extremal vertex such that no other vertex has height between $h(x)$ and $h(x')$. The operation Handoff(x, x') updates one crossing level set of x from containing x and intersecting the down edges (up edges) of x' to containing x' and intersecting the up edges (down edges) of x. Vertex x' is then made a representative owning the crossing level set.

We now sketch how to implement each operation. The details are given in the full paper.

FindArc(v) and LocateArc(α, β, v): For FindArc(v), we want to find a pair of nodes μ_1 and μ_2 bounding a height monotone path through $\mathcal{T}$ that contains $\rho(v)$. Afterward, we can binary search for the projection of v on that path. See Figure 5 for an example. The high level idea behind our strategy to find μ_1 is to search for the root of v's descent tree, and use it to find an extremal vertex reachable via a height monotone path through $\mathcal{T}$ or to find

an arc of $\mathcal{R}$ reachable in a similar way; the higher of the two nodes on the arc is reachable through $\mathcal{T}$. We compute μ_2 symmetrically. For LocateArc(α, β, v), we use a procedure similar to that of FindArc to quickly find a height monotone path in $\mathcal{T}$ from α that passes through β and $\rho(v)$.

▶ **Lemma 2.** *Both* FindArc(v) *and* LocateArc(α, β, v) *can be implemented in* $O(\log n)$ *time.*

LSCross(x, u) and Handoff(x, x'): For LSCross(x, u), our goal is to remove down edges of u from x's crossing level set one-by-one (assuming $h(x, t^-) < h(u, t^-)$ without loss of generality) using link-cut tree operations before adding the up edges of u to x's crossing level set. The details on which edges to link, unlink, or designate as anchors vary depending on whether u is an extremal vertex, a saddle, or a regular vertex. See Figure 6. Handoff(x,x') is also implemented in a similar fashion.

▶ **Lemma 3.** *Both* LSCross(x, u) *and* Handoff(x, x') *can be implemented in* $O(\log^2 n)$ *time.*

5 Handling events

Internal events and certificates. We define a few internal events that are needed to keep its auxiliary structures accurate. A *local event* occurs whenever two adjacent vertices in $\mathbb{M}$ have equal heights. A *crossing event* occurs whenever a representative vertex x and any vertex $u \neq x$ have equal height. A *winner event* occurs when the winner of a representative's tournament changes. Finally, a *tournament event* occurs when any of the representative vertices' tournaments need to be repaired as described by Agarwal *et al.* [1].

We maintain a collection of certificates that certify our data structure to be accurate. When a certificate fails, an event may occur. We maintain certificates of the following types:

- For each edge of $\mathbb{M}$, a certificate that fails when its two endpoints' heights become equal.
- For every arc of $\mathcal{R}$, a certificate that fails when its two endpoints' heights become equal.
- For every representative vertex $x \in \mathcal{V}$ and its two tournaments $\psi^-(x)$ and $\psi^+(x)$, a certificate that fails when the heights of x and the tournament winner become equal. One can easily verify that one of these two certificates for x will fail any time the height of any vertex $u \neq x$ becomes equal to $h(x)$.
- For every representative vertex $x \in \mathcal{V}$, a collection of certificates as described by Agarwal *et al.* [1] certifying the correctness of the tournaments $\psi^-(x)$ and $\psi^+(x)$.

Note that multiple certificate failures can occur at the same time, since the same two vertices can be involved in the failure of multiple certificates. For instance, during a death event, the certificates for both an arc of $\mathcal{R}$ and an edge of $\mathbb{M}$ will fail.

Repairing the KDS. It is straightforward from the definition of the different events to know what type of event is occurring with each certificate failure. We now discuss how our data structure handles events. Each event involves only a constant number of vertices (under our assumption that the vertex degree is constant).

To avoid opening the 'black box' we leave the handling of tournament events and the repairing of tournaments to the procedures described by Agarwal *et al.* [1]. Winner events are handled separately and in addition to the other type of events in $O(\log n)$ time by simply updating the certificates for representative vertices and their tournaments with new winners.

We now describe how to handle the remaining types of events, beginning with external events, and then focusing on non-external crossing events. We conclude with updating the descent and ascent trees, a process that must be done after every event. In general, we handle events by performing local changes that guarantee $\mathcal{T}$ is still a spanning tree of $\mathcal{R}$. We must be sure to appropriately hand off crossing level sets so their representative vertices still lie on loop arcs in $\mathcal{L}$. In addition, we must make sure that each edge still has its crossing level set copies ordered by height. To do so, we will generally perform HANDOFF operations so that only one vertex involved in each event represents crossing level sets. LSCROSS operations can then be performed safely (as we essentially move from time t^- to time t^+) before doing some number of HANDOFF operations to guarantee each vertex again represents the correct number of crossing level sets.

Shift event: Suppose u is critical and v is regular at t^-, and vice versa at t^+. We relabel node $\rho(u)$ in $\mathcal{R}$ by v. Node $\rho(v)$ is now incident to every loop arc that originally contained $\rho(u)$, so we call HANDOFF(u, v) for each crossing level set that belongs to u (recall, a vertex can be a representative up to two times). Finally, we call LSCROSS(v, u) if v is now a representative, because its (newly acquired) crossing level sets pass over u at time t.

Birth event: Suppose u and v are not critical at t^-, but they become critical at t^+. Let v and u be a maximum and positive saddle respectively at time t^+ (the other cases are similar). We need to figure out where the new arc $(\rho(u), \rho(v))$ will lie in $\mathcal{R}$ and possibly hand off a crossing level set from v, because $\rho(v)$ will soon be a leaf of $\mathcal{R}$. We call FINDARC(u) to figure out which arc (ξ, ζ) with $h(\xi) < h(\zeta)$ contains u and v. If $(\xi, \zeta) \in \mathcal{T}$, then we remove (ξ, ζ) from $\mathcal{T}$, and add arcs $(\xi, \rho(u))$, $(\rho(u), \zeta)$, and $(\rho(u), \rho(v))$ to $\mathcal{T}$ using link-cut tree operations. We are done, because $u, v \notin \mathcal{V}$ in this case.

Suppose $(\xi, \zeta) \in \mathcal{L}$. If $v \in \mathcal{V}$, we call HANDOFF(v, u). If u now has a crossing level set, we call LSCROSS(u, v). We replace (ξ, ζ) in $\mathcal{L}$ by $(\rho(u), \zeta)$, and add $(\xi, \rho(u)), (\rho(u), \rho(v))$ to $\mathcal{T}$. For the case of $u, v \notin \mathcal{V}$, we must guarantee the representative vertex w for (ξ, ζ) still lies on a loop arc. If $h(w) > h(u)$ (resp. $h(w) < h(v)$) at t^-, replace (ξ, ζ) in $\mathcal{L}$ by $(\rho(u), \zeta)$ (resp. $(\xi, \rho(u))$), and add $(\rho(u), \rho(v))$ and $(\rho(u), \xi)$ (resp. $(\zeta, \rho(u))$) to $\mathcal{T}$.

Death event: Suppose at t^-, $\alpha = \rho(u)$ was a positive saddle and $\beta = \rho(v)$ was a maximum, and they both become regular at t^+ (the other cases are similar). We have $(\alpha, \beta) \notin \mathcal{L}$, and in particular, v is not a representative vertex. We remove arc (α, β) from $\mathcal{T}$. Let $(\xi, \alpha), (\alpha, \zeta)$ be the other two arcs adjacent to α with $h(\xi) < h(\zeta)$. We cannot have both (ξ, α) and (α, ζ) as loop arcs at t^-. If neither arc is a loop arc or $(\xi, \alpha) \in \mathcal{L}$ then we relabel (ξ, α) to be (ξ, ζ) and remove (α, ζ) from $\mathcal{T}$. Otherwise, we relabel (α, ζ) to be (ξ, ζ) and remove (ξ, α) from $\mathcal{T}$. If either arc was a loop arc, then the new loop arc still contains the image of its representative. Finally, if $u \in \mathcal{V}$, we call LSCROSS(u, v).

Interchange event: Consider an interchange event between two nodes $\alpha = \rho(u)$ and $\beta = \rho(v)$ where $h(\alpha, t^-) < h(\beta, t^-)$. We must first figure out the new incident arcs of α and β. Afterward, we will pick which incident arcs will be designated as loop arcs and finally move crossing levels as appropriate to guarantee the new loop arcs have their representatives. We pick two up-neighbors w_1, w_2 of u such that $\rho(w_i) \neq \beta$ for $i \in \{1, 2\}$, one from each uplink of u. We then figure out the new up arcs of α using LOCATEARC(α, β, w_i) for $i \in \{1, 2\}$ (see Lemma 1). Similarly, we figure out the new down arcs of β, thus giving us the combinatorial changes in $\mathcal{R}$.

We define the *outer nodes* as the nodes incident to α and β at time t^- other than α and β themselves. Suppose α and β share a single arc in $\mathcal{R}$. Further suppose $(\alpha, \beta) \notin \mathcal{L}$ at t^-. Thus, $(\alpha, \beta) \in \mathcal{T}$. If we contract (α, β), $\mathcal{R}$ is identical at t^- and t^+, and all the arcs in $\mathcal{T} \setminus \{(\alpha, \beta)\}$ still constitute a spanning tree of the contracted $\mathcal{R}$ at t^+. Thus, expanding

(α, β) and including it along with all the other arcs in $\mathcal{T}$ not incident to (α, β) at time t^- does not create a cycle at time t^+. Any outer node that was adjacent to either α or β in $\mathcal{T}$ at time t^- still needs to be adjacent to α or β in $\mathcal{T}$ at time t^+. However, its neighbor may change from α to β or vice versa. We add the appropriate arcs connecting these outer nodes to (α, β) to get $\mathcal{T}$ at t^+. These changes can be done in $O(\log n)$ time using link-cut tree operations.

If instead $(\alpha, \beta) \in \mathcal{L}$ at t^-, then $(\alpha, \beta) \notin \mathcal{T}$. There exists a unique path from α to β in $\mathcal{T}$. Let ζ be an outer node incident to α lying on this path; ζ can be found in $O(\log n)$ time using a link-cut tree operation. Replacing the arc (α, ζ) by (α, β) in $\mathcal{T}$ still keeps it a spanning tree of $\mathcal{R}$, and we can now use the argument from the previous paragraph.

Now, if α and β share two arcs in $\mathcal{R}$ at time t^-, then they continue to share two arcs at time t^+ and the combinatorial structure of $\mathcal{R}$ does not change. We simply keep the same number of loop arcs between α and β.

We now discuss handling changes to level set contours. We call $\textsc{Handoff}(u, v)$ once for each arc that u represents. If any level set contours then belong to v, we call $\textsc{LSCross}(v, u)$. Finally, we call $\textsc{Handoff}(v, u)$ as many times as necessary so that both u and v have the same number of level sets as there are incident loop arcs not already represented by other vertices at time t^+.

***Other crossing event*:** We now describe how to handle non-external crossing events. Suppose we have representative vertex x and a vertex $u \neq x$ with $h(x, t) = h(u, t)$. We call $\textsc{Handoff}(u, x)$ for each loop arc represented by u followed by a single call to $\textsc{LSCross}(x, u)$ and then the same number of calls to $\textsc{Handoff}(x, u)$ as were done earlier for (u, x). In doing so, the crossing level sets of x and u retain the relative order by height and the lists of edge copies for the level sets retain their proper ordering.

***Repairing descent and ascent trees*:** During an event at time t, a constant number of edges gain or lose an intersecting crossing level set or have their endpoints change relative order by height. For each such edge $(u, v) \in \mathbb{M}$, we proactively update the descent and ascent trees of u and v in $O(\log n)$ time. We use a link-cut tree operation to remove both u and v's parents in the descent trees; if either parent was a level set root, then we delete it. Let $w \neq u$ be any down neighbor of v after the event is over (if w does not exist, v is a minimum at t^+ and hence made a root). If (v, w) does not intersect a crossing level set internally, we set w to be v's parent; else we add a new level set root c and make it v's parent by a link-cut tree operation, recording (v, w) on the arc (v, c). We perform a similar operation for u. The ascent tree is handled similarly.

We now state our main theorem.

▶ **Theorem 4.** *We can maintain the Reeb graph of a time-varying, piecewise-linear function, defined on a triangulated orientable 2-manifold without boundary of size n and genus g, in $O(\log^2 n)$ time per event and $O(gn)$ total space. The total number of events processed is $O((\kappa + g)\lambda_{O(1)}(n) \log n)$, where κ is the total number of combinatorial changes in the Reeb graph.*

Proof. There are $O(n)$ local events by our assumptions on the height function. There are g representative vertices $\mathcal{V}$ (counting with multiplicity). The set $\mathcal{V}$ changes only at a shift, birth, death, or interchange events, but these events change the combinatorial structure of the Reeb graph as well, and there are κ such events. Thus, at most $g + \kappa$ vertices can become representatives. Crossing events only occur when a representative vertex and some other vertex achieve the same height; by our assumptions on the height function, there are at most $O((\kappa + g)n)$ such events. Finally, we add and remove vertices from tournaments only

during crossing events. Therefore, there are $O((\kappa + g)\lambda_{O(1)}(n) \log n)$ tournament events [1]. Every event described above involves a constant number of link-cut tree operations, calls to LSCross and Handoff, or tournament operations. Therefore, every event, including tournament events, is handled in $O(\log^2 n)$ time [1]. ◀

Remarks. (i) Our approach can be extended to *augmented Reeb graphs*, although it does create additional types of interchange events. The augmented Reeb graph contains an additional degree two node at $\rho(v)$ for each regular v in $\mathbb{M}$. The total number of events remains the same as in the case above.
(ii) As mentioned, our approach can also handle vertices of high degree, however the running time to handle an event increases by a factor proportional to the degree of the vertices involved. Our approach can also handle non-simple saddles, without major modifications.
(iii) One of the main difficulties with extending our technique to higher dimensional manifolds is that the contours are not polygonal cycles but are higher dimensional surfaces. It is not clear how to represent them using link-cut trees, unlike the one-dimensional case.

References

1 Pankaj K. Agarwal, Haim Kaplan, and Micha Sharir. Kinetic and dynamic data structures for closest pair and all nearest neighbors. *ACM Trans. Algo.*, 5(1):4:1–4:37, 2008.

2 Pankaj K. Agarwal, Thomas Mølhave, Morten Revsbæk, Issam Safa, Yusu Wang, and Jungwoo Yang. Maintaining contour trees of dynamic terrains. In *Proc. 31st Int. Symp. Comp. Geom.*, pages 796–811, 2015.

3 Pankaj K. Agarwal and Micha Sharir. Davenport-schinzel sequences and their geometric applications. *Handbook of computational geometry*, pages 1–47, 2000.

4 Julien Basch, Leonidas J Guibas, and John Hershberger. Data structures for mobile data. *J. Algo.*, 31(1):1–28, 1999.

5 Dmitriy Bespalov, William C Regli, and Ali Shokoufandeh. Reeb graph based shape retrieval for cad. In *Proc. Int. Des. Engg. Tech. Conf. Comput. Inf. Engg.*, pages 229–238, 2003.

6 Kree Cole-McLaughlin, Herbert Edelsbrunner, John Harer, Vijay Natarajan, and Valerio Pascucci. Loops in Reeb graphs of 2-manifolds. In *19th Int. Symp. Comp. Geome.*, pages 344–350. ACM, 2003.

7 Mark de Berg, Marc van Kreveld, Mark Overmars, and Otfried Cheong. *Computational geometry*. Springer, 3rd edition, 2000.

8 Harish Doraiswamy and Vijay Natarajan. Computing reeb graphs as a union of contour trees. *IEEE Trans. Visualiz. Comput. Graph.*, 19(2):249–262, 2013.

9 Herbert Edelsbrunner, John Harer, Ajith Mascarenhas, and Valerio Pascucci. Time-varying Reeb graphs for continuous space-time data. In *20th Int. Symp. Comp. Geome.*, pages 366–372. ACM, 2004.

10 A. T. Fomenko and T. L. Kunii. *Topological Methods for Visualization*. Springer-Verlag, Tokyo, Japan, 1997.

11 Leonidas J. Guibas. Modeling motion. In *Handbook of Discrete and Computational Geometry, Second Edition.*, pages 1117–1134. 2004.

12 William Harvey, Yusu Wang, and Rephael Wenger. A randomized $O(m \log m)$ time algorithm for computing Reeb graphs of arbitrary simplicial complexes. In *Proc. 26th Symp. Comput. Geom.*, pages 267–276, 2010.

13 Salman Parsa. A deterministic $O(m \log m)$ time algorithm for the Reeb graph. *Disc. Comput. Geom.*, 49(4):864–878, 2013.

14 Valerio Pascucci, Giorgio Scorzelli, Peer-Timo Bremer, and Ajith Mascarenhas. Robust on-line computation of reeb graphs: simplicity and speed. *ACM Trans. Graph.*, 26(3):58, 2007.

15 Georges Reeb. Sur les points singuliers d'une forme de pfaff completement intégrable ou d'une fonction numérique. *CR Acad. Sci. Paris*, 222(847-849):2, 1946.

16 Yoshihisa Shinagawa and Tosiyasu L Kunii. Constructing a Reeb graph automatically from cross sections. *IEEE Comp. Graph. App.*, 11(6):44–51, 1991.

17 Daniel D. Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983.

18 B-S Sohn and Chandrajit Bajaj. Time-varying contour topology. *IEEE Trans. Visual. and Comp. Graph*, 12(1):14–25, 2006.

19 Andrzej Szymczak. Subdomain aware contour trees and contour evolution in time-dependent scalar fields. In *Inter. Conf. Shape Modeling Appli.*, pages 136–144. IEEE, 2005.

20 Tony Tung and Francis Schmitt. The augmented multiresolution reeb graph approach for content-based retrieval of 3d shapes. *Int. J. Shape Modeling*, 11(01):91–120, 2005.