

A Deterministic $O(m \log m)$ Time Algorithm for the Reeb Graph

Salman Parsa

Received: 13 July 2012 / Revised: 6 February 2013 / Accepted: 18 April 2013 /
Published online: 25 May 2013
© Springer Science+Business Media New York 2013

Abstract We present a deterministic algorithm to compute the Reeb graph of a PL real-valued function on a simplicial complex in $O(m \log m)$ time, where m is the size of the 2-skeleton. The problem can be solved using dynamic graph connectivity. We obtain the running time by using offline graph connectivity which assumes that the deletion time of every arc inserted is known at the time of insertion. The algorithm is implemented and experimental results are given. In addition, we reduce the offline graph connectivity problem to computing the Reeb graph.

Keywords Algorithms · Reeb graph · PL topology · Graph connectivity

1 Introduction

Let f be a continuous function from a topological space to the real line. Roughly speaking, the Reeb graph of f is achieved by considering connected components of preimages (or level-sets) $f^{-1}(r)$ as points, for $r \in \mathbb{R}$. When the domain contains no nontrivial loop, such as \mathbb{R}^d , Reeb graph is the same as the contour tree. For example, for the altitude function on a terrain, contours (preimage components) are drawn on the plane, and their evolution traces out the contour tree. On the other hand, if the domain is a torus, contours should be drawn on a torus and it is no longer true that their evolution traces a tree; it traces a graph called the Reeb graph.

The Reeb graph of f provides a compact description of the domain, seen through f . Reeb graphs have found applications in a wide range of areas such as shape matching and retrieval [4, 16, 31], shape segmentation and simplification [23, 32], animation [3, 18], high dimensional data visualization and analysis [14, 19] and robotics [22].

S. Parsa (✉)
Duke University, Durham, NC, USA
e-mail: salparsa@cs.duke.edu

We refer to [5,6] for a survey of the Reeb graph and some applications. In this paper we consider the efficient computation of the Reeb graph of a piecewise linear function on an arbitrary simplicial complex. This setting is general enough to approximate functions and spaces usually considered in practice.

Our contributions are the following:

- We give the first algorithm that runs in worst-case time $O(m \log m)$, where m is the size of the 2-skeleton. This is certainly optimal, if the number of edges and triangles of the complex is in the same order as the number of vertices.
- We implemented a simple version of the algorithm including a heuristic to speed up the computation. This implementation gives running times superior to those of existing algorithms. Given the simplicity and possible choices for the implementation, it might turn into the algorithm of choice for various applications.
- We use an offline variant of the dynamic graph connectivity and show that this offline problem is equivalent to Reeb graph construction.

2 Related Work

Carr et al. [7] gave an efficient algorithm for computing the contour tree for a function on a simplicial complex domain in time $O(n \log n + m\alpha(m))$, where n is the number of vertices and m is the number of edges of the input complex. The first algorithm to compute the Reeb graph was given by Shinagawa and Kunii [24] that works for the triangulation of a 2-manifold and runs in time $\Theta(n^2)$. This algorithm sweeps the vertices in increasing order of function values and maintains the preimage. For the case of 2-manifolds, Cole-McLaughlin et al. [8] improved the running time to $O(n \log n)$. They used circular lists to maintain the preimage.

The Reeb graph of a d -dimensional simplicial complex for $d \geq 2$, depends only on the 2-skeleton, whose size we denote by m . One can maintain the preimage components as graphs, reducing to dynamic graph connectivity on a graph of size m . Then, for an arbitrary simplicial complex, the sweep algorithm asymptotically runs in time m times the bound for an operation in the dynamic graph connectivity data structure. The number of nodes of the graph is $O(m)$. Doraiswamy and Natarajan [10] were the first to use this reduction to compute the Reeb graph, see Sect. 4 for details.

Holm et al. [17] gave a deterministic algorithm for dynamic graph connectivity with $O(\log^2 m)$ amortized time per operation. As used in [10], this connectivity algorithm resulted in the best deterministic algorithm for the Reeb graph for a general simplicial complex before this paper. Moreover, Thorup [29] presents an algorithm with $O(\log m (\log \log m)^3)$ expected amortized running time per operation for the dynamic graph connectivity. For computing the Reeb graph on a 3-manifold, Doraiswamy and Natarajan [10] give an algorithm that runs in expected time $O(m \log m + m \log g (\log \log g)^3)$, where g is the maximum genus over all preimages. This algorithm maintains a tree/co-tree partition of the graph, and uses Thorup's randomized graph connectivity.

Tierny et al. [30] present an algorithm that works on 3-manifolds-with-boundary embedded in \mathbb{R}^3 . Their algorithm runs in time $O(m \log m + hm)$, where h is the number of independent loops in the Reeb graph. This algorithm is not general but is

very efficient. A streaming algorithm for computing the Reeb graph of an arbitrary simplicial complex is presented in [20] with $\Theta(nm)$ running time. Harvey et al. [15] presented a randomized algorithm with the expected running time $O(m \log m)$. The algorithm works by collapsing triangles adjacent to a vertex. In the end, the complex collapses to a representation of the Reeb graph. The evolution of the Reeb graph, as the function varies over time, is studied in [11]. In [9], the authors study approximation of the Reeb graph and its persistence. Higher dimensional analogs of Reeb graphs are called Reeb spaces. They are more difficult to compute, however. These spaces are studied in [12].

We also mention that there has been extensive research on the dynamic graph connectivity and related problems, both from the upper bound and from the lower bound point of view. In addition to the above, Patrascu and Demaine [21] proved an $\Omega(\log m)$ lower bound in the cell-probe model. Eppstein [13] uses a linear time minimum spanning tree algorithm to solve the offline minimum spanning tree problem in $O(\log m)$ time per operation. This is the only reference to offline graph connectivity, and we came to know about it after publishing this paper.

3 Background

3.1 Simplices and Simplicial Complexes

A d -simplex is the convex hull of $d + 1$ affinely independent points $V = \{v_0, \dots, v_d\}$ in some Euclidean space, e.g. \mathbb{R}^d , where $d \geq 0$. The set $V = V(\sigma)$ is called the *vertex set* of the simplex. Let σ be a d_1 -simplex and δ a d_2 -simplex. If $V(\sigma) \subset V(\delta)$, we say σ is a d_1 -face of δ and denote it by $\sigma \leq \delta$. We also call a 0-simplex, a 1-simplex and a 2-simplex a *vertex*, an *edge* and a *triangle*, respectively. If K is a finite set of simplices, all in the same Euclidean space, then K is a *simplicial complex* provided (i) $\delta \in K$ and $\sigma < \delta \Rightarrow \sigma \in K$, (ii) $\sigma_1, \sigma_2 \in K \Rightarrow \sigma_1 \cap \sigma_2 < \sigma_1, \sigma_2$ if $\sigma_1 \cap \sigma_2$ is not empty. We define K_0, K_1 and K_2 to be the set of vertices, edges, and triangles, respectively, of the simplicial complex K . Moreover, set $n_0 = \#K_0, n_1 = \#K_1$ and $n_2 = \#K_2$, where $\#$ denotes the number of elements of the set. By $|K|$ we mean the underlying space of K , i.e. $|K| = \bigcup_{\sigma \in K} \sigma$ with the topology inherited from the ambient Euclidean space. For convenience, if no ambiguity is caused, we also write K instead of $|K|$. The *dimension* of a simplicial complex is the highest dimension of its simplices. The k -skeleton of the simplicial complex, denoted K^k , is the set of its simplices of dimension at most k . We denote by m the size of the 2-skeleton, i.e. $m = n_0 + n_1 + n_2$.

If $x \in |K|$, then there is a unique simplex with smallest dimension that contains x , say σ' . By definition of a simplex, x can be written as a convex combination of the vertices of σ' . Setting the coefficients of other vertices in K_0 to zero, we can write x as a convex combination of points in K_0 in a unique way: $x = \sum_{i=1}^{n_0} b_i v_i$ where $K_0 = \{v_1, v_2, \dots, v_{n_0}\}$ and $\sum_i b_i = 1$ and $b_i \geq 0$ for all i . The numbers b_i are called the *barycentric coordinates* of $x \in |K|$.

To express the difference with a complex, we call the vertices of a graph *nodes* and its edges *arcs*. Graphs in this paper will be abstract over a fixed, labeled set of nodes.

3.2 Reeb Graph

Let $f : K_0 \rightarrow \mathbb{R}$ be a function. We say that f is *generic* if it is injective. In this paper, we always assume the function f on the set of vertices to be generic. One can extend f to all of $|K|$ by setting $f(x) = \sum_i b_i f(v_i)$, where b_i are the barycentric coordinates of x . Then, the extended function f , is called a *piecewise linear* function from $|K|$ to \mathbb{R} . Now, fix $r \in \mathbb{R}$ and consider the preimage of r : $f^{-1}(r) = \{x \in |K|, f(x) = r\}$. If $\sigma \in K$ is a d -simplex, then $f^{-1}(r) \cap \sigma$ is the intersection of a $(d - 1)$ -plane with σ . It is not hard to see that $f^{-1}(r) \cap K^2$ is also a simplicial complex, namely, $f^{-1}(r) \cap K^2 = \{\sigma \cap f^{-1}(r) : \sigma \in K^2\}$. Therefore, every vertex of $f^{-1}(r) \cap K^2$, $r \notin f(K_0)$, corresponds to exactly one edge of K and every edge of it comes from a unique triangle of K .

We are interested in the connected components of $f^{-1}(r)$, and their behavior as r varies. The *Reeb graph* of a function $f : |K| \rightarrow \mathbb{R}$ is the topological graph obtained by contracting every connected component of $f^{-1}(r)$ to a point, for every $r \in \mathbb{R}$. So it is a quotient space of $|K|$ with the quotient topology. Formally this means that, two points in $|K|$ are equivalent if they belong to the same connected component of $f^{-1}(r)$ for some $r \in \mathbb{R}$, and the Reeb graph is the set of equivalence classes of this relation, with quotient topology. Intuitively, the points of the Reeb graph are connected together as the preimage components were connected together. Thus, $f^{-1}(r)$ reduces to a finite set of points and as r varies, these points trace out arcs of a graph which meet at points where corresponding connected components meet (Fig. 1).

It is easily seen that when r changes continuously, without passing any $f(v)$ value, the connected components of the preimages $f^{-1}(r)$ remain unchanged. Given a simplicial complex and a generic piecewise linear function, we are concerned with finding an efficient algorithm that computes its Reeb graph. Note that if the complex K is d -dimensional, one can easily embed the Reeb graph in $|K| \subset \mathbb{R}^k$, where k is as in the definition of the complex. However, there is no “natural” embedding of the Reeb graph on the plane (page) for example. This is true, even if the Reeb graph is planar.

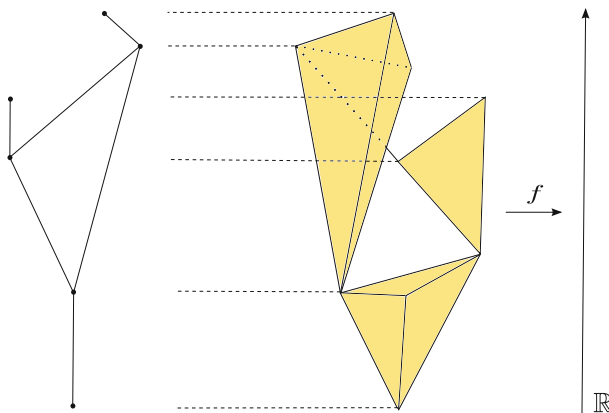


Fig. 1 A drawing of the Reeb graph for the height function on a simplicial complex embedded in space

In the following we look at the Reeb graph as an abstract graph. We also note that, the Reeb graph is in fact a multigraph.

If we sweep the Reeb graph in increasing function value, a node is a point where a component (arc) is created, merged with others, split or destroyed. Since these events can happen only at preimage of some $f(v)$ and the preimage only includes one vertex v , vertices can be used to identify Reeb graph nodes. Consider the component of $f^{-1}(f(v))$ containing v . If the contraction of this component in the Reeb graph is not a node, we call v *Reeb-regular*. In other words, Reeb-regular vertices correspond exactly to those v such that when the preimage changes from $f^{-1}(f(v) - \epsilon)$ to $f^{-1}(f(v) + \epsilon)$ no arcs of the Reeb graph are created, get destroyed, merge or split. If a vertex is not Reeb-regular, we call it *Reeb-critical*. Therefore, a Reeb-critical vertex identifies a node of the Reeb graph. Moreover, a *Reeb-critical value* is the function value of a Reeb-critical vertex. Other values are *Reeb-regular*.

4 Algorithm

We describe the algorithm in two parts. First we show how to reduce the problem to that of maintaining connected components of a graph through insertion and deletion of arcs, then we explain how the latter problem can be solved in our setting within optimal time bounds. The Reeb graph of a d -dimensional simplicial complex depends only on its 2-skeleton so we assume that the input is the 2-skeleton of the original complex, then $f^{-1}(r)$ is a 1-dimensional simplicial complex.

The input to our algorithm is described as a list of vertices, edges and triangles. Edges and triangles are defined by indexing their vertices. Also, for every vertex we need to know edges and triangles incident on it, which we compute in time linear in number of edges and triangles.

4.1 The Reduction

The outline of the algorithm is as follows. To obtain the Reeb graph of f , we need to know its nodes and its arcs. Since the components of the preimage (arcs of the Reeb graph) are created and/or destroyed only at Reeb-critical values, we need to find the Reeb-critical vertices of K . Sort the vertices so that $f(v_1) < f(v_2) < \dots < f(v_{n_0})$. We find the Reeb-critical vertices by sweeping f from $-\infty$ to $+\infty$ and for each value $f(v_i)$, we look how the preimage components change in number, when we pass that value. If the vertex is recognized as Reeb-critical, we add it to the Reeb graph as a node and connect it to other nodes appropriately.

Before going into more detail, we introduce some terms. We say an edge of K *starts* at its vertex of lower function value and *ends* at the one with higher function value. Similarly, a triangle starts at its vertex of lowest function value and ends at its vertex of highest function value. The vertex with the middle function value of a triangle we call its *middle* vertex. We denote by v_1v_2 the edge connecting vertices v_1 and v_2 . In this notation, we always first write the vertex with smaller function value.

The preimage $f^{-1}(r)$ can be abstracted into a graph G_r , which we call the *preimage graph* at value r . Nodes of G_r are edges of K intersecting the preimage and arcs of

G_r are triangles of K contributing to the preimage, so arcs indeed connect nodes to each other. G_r changes if and only if we pass a function value $f(v_i)$. The graph G_r is $f^{-1}(r)$, when viewed as an abstract graph. In the following, we use a data structure to maintain the connected components of the preimage graph which we call *DynTrees*. Its description is the topic of the next section. The pseudo-code for the sweep algorithm is given below:

Algorithm 1 Sweep Algorithm

```

set DynTrees to be an empty graph
for  $i = 1$  to  $n_0$  do
   $Lc = \text{LowerComps}(v_i)$ 
  UpdatePreimage()
   $Uc = \text{UpperComps}(v_i)$ 
  if  $\neg(\#Lc = \#Uc = 1)$  then
    UpdateReebGraph( $Lc, Uc$ )
  end
end
end
  
```

In Algorithm 1, we use four subroutines. The *LowerComps*(v_i) subroutine considers edges ending at the vertex v_i , one by one, and finds their corresponding components in the current preimage graph (which is $G_{f(v_i)-\epsilon}$). At the end, *LowerComps*(v_i) provides us with a set of nodes of the preimage graph, each representing a component. A pseudo-code for this procedure is given in Algorithm 2.

The *UpdatePreimage*() subroutine updates the preimage graph from that of immediately before $f(v_i)$ to that of immediately after $f(v_i)$. Triangles and edges ending at v_i are removed from the graph and edges and triangles starting at v_i are inserted into the preimage graph. Moreover, for every triangle of K that has v_i as a middle vertex, we delete the arc of the preimage graph that will no longer be in the graph and insert the new arc; see Fig. 2. A pseudo-code for this subroutine is given in Algorithm 3. This code assumes that edges not intersecting the preimage are also in the preimage graph as isolated nodes so there is no need to add and remove isolated nodes. The subroutine *UpperComps*(v_i) is symmetric to *LowerComps*(v_i).

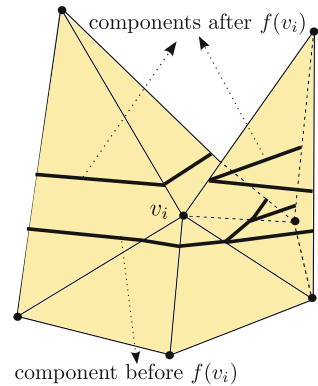
Algorithm 2 LowerComps(v)

```

Lc = empty list
for all edges  $e$  ending at  $v$  do
   $c = \text{DynTrees.find}(e)$ 
  if  $c$  is not marked then
    Lc.add( $c$ )
    mark  $c$  as listed
  end
end
end
  
```

The *UpdateReebGraph*(Lc, Uc) subroutine, updates the Reeb graph. Note that all components in Lc will be merged into one component at v_i and this one, will split into the components in Uc . The vertex v_i is Reeb-regular, if and only if Lc and Rc both have

Fig. 2 Processing of a Reeb-critical (split) vertex in a 2-D simplicial complex



exactly one element, therefore it is easy to decide if the vertex corresponds to a node of the Reeb graph. For such a vertex, we create a new node in the Reeb graph, v_{v_i} , and associate it to components in Uc . Intuitively, those components were first generated at v_i . We make an arc between corresponding Reeb graph nodes of components in Lc to the node v_{v_i} . A pseudo-code for this subroutine is given in Algorithm 4.

Algorithm 3 UpdatePreimage(v)

```

for all triangles  $t = \{v_1, v_2, v_3\}$  incident on  $v$  while  $f(v_1) < f(v_2) < f(v_3)$  do
  if  $v = v_3$  then DynTrees.delete( $(v_1v)(v_2v)$ ) end
  if  $v = v_2$  then
    DynTrees.delete( $(v_1v_3)(v_1v)$ )
    DynTrees.insert( $(v_1v_3)(v_1v_3)$ )
  end
  if  $v = v_1$  then DynTrees.insert( $(v_1v_2)(v_1v_3)$ ) end
end

```

Algorithm 4 UpdateReebGraph(Lc, Uc)

```

create a new node  $v$  in Reeb graph
assign the node to all  $c \in Uc$ 
create an arc between  $v$  and  $v_c$ , for all  $c \in Lc$ 

```

The total time spent in $UpdateReebGraph(Lc, Uc)$ is linear in the size of the Reeb graph. The *DynTrees* data structure supports three types of operations: finding component of a node, inserting an arc, and deleting an arc from the preimage graph. Assuming n is the number of nodes in this graph, we write $U(n)$ for the time needed for any of these operations. Every edge of K is considered once in $LowerComps(v_i)$ and once in $UpperComps(v_i)$. For each, there is one *find* operation for finding the component of the edge in preimage graph, therefore the total running time of the two subroutines is $O(n_1U(n_1))$ in the worst case. Moreover, every triangle gives rise to two arc

insertions and two arc deletions in the preimage graph, so the total running time of $UpdateGraph()$ is $O(n_2U(n_1))$. Summing all of these together, the algorithm runs in time $O(mU(m))$ where m is the size of the 2-skeleton.

4.2 Graph Connectivity

We complete the description of the algorithm by explaining how to implement the three operations *find*, *delete* and *insert* on the preimage graph, required by the sweep algorithm. The *DynTrees* data structure keeps track of the connected components of the graph, when arcs are inserted and deleted over a fixed node set. This is called the *dynamic graph connectivity* problem.

A dynamic graph connectivity algorithm usually works by maintaining a rooted spanning forest of the graph. The root is used to identify the component, so a *find* query will be just finding the root of the tree containing the node. This is the approach we will also take, but we exploit the fact that the operations requested by the sweep algorithm can be predicted to choose our spanning trees. In order to do so, we assign weights to arcs of the preimage graph. The *weight* of an arc is the time that the arc is going to be deleted. In other words, if the arc $(v_1 v_2)(v_3 v_4)$ corresponds to a triangle (so the v_i are not distinct), then the weight of the arc is the smallest function value of endpoints, i.e. $\min\{f(v_2), f(v_4)\}$. The weight of an arc is computed in constant time when the arc is inserted, and assigned to the arc.

The main idea is now to maintain the maximum spanning forest of the preimage graph. It has the important property that, the arc that is going to be deleted is not in this forest, unless it is absolutely necessary. To maintain the forest, we use a dynamic tree data structure. These data structures can keep a forest of node-disjoint trees and support various operations on those trees. Arcs can have weights and information about the weights on a tree or a path can be obtained. The operations that we require are as follows:

- *parent*(x): return the parent of node x , or **null** if x is the root.
- *root*(x): return the root of the tree containing node x .
- *link*(x_1, x_2, w): link distinct trees containing the two nodes x_1 and x_2 by adding the arc $x_1 x_2$. Assign the weight w to this arc.
- *cut*(x_1, x_2): remove the arc between x_1 and x_2 , splitting the tree in two.
- *minWeight*(x): return a node with minimum weight arc to its parent on the path from x to the root of its tree, or **null** if x is the root.
- *evert*(x): make the node x the root of its tree.

All of the above operations are supported by existing dynamic tree data structures that allow path operations, for example, ST-trees or Link–Cut trees [25, 26], top trees [2, 27] and RC-trees [1]. See [28] for an experimental comparison of these data structures. Different implementations of ST-trees and/or top trees support all of the above operations in worst-case or amortized time $O(\log n)$, where n is the number of nodes in the forest [28]. RC-trees achieve the same bound in expected running time.

Using the above, we implement our three tree operations as follows:

- *find*(x):
 - return $\text{root}(x)$
- *insert*(x_1x_2):
 - $\omega = \text{weight of arc } x_1x_2$
 - if $\text{root}(x_1) = \text{root}(x_2)$ then
 - $\text{evert}(x_1)$
 - $x' = \text{minWeight}(x_2)$
 - $\omega' = x'.\text{weight}$
 - if $\omega' < \omega$ then
 - $\text{cut}(x', \text{parent}(x'))$
 - $\text{link}(x_1, x_2, \omega)$
 - end
 - else
 - $\text{link}(x_1, x_2, \omega)$
 - end
- *delete*(x_1, x_2):
 - if (x_1, x_2) is a tree edge then
 - $\text{cut}(x_1, x_2)$
 - end

The Reeb graph node associated with a root transfers as the root node changes in *evert*. Reeb graph nodes are assigned to the roots of new trees generated, after each *cut* or *link* operation. The overall cost of keeping track of Reeb graph nodes is then constant number of *find* calls per every edge. So, we can maintain the Reeb graph data in $O(n_1U(n_1))$ time. Considering the above, we have $U(n) = O(\log n)$ using this semi-dynamic graph connectivity algorithm.

As explained later on, here we say semi-dynamic instead of offline, since for the approach to work, we do not need the entire sequence of updates, we only need to be able to compute the deletion time in time of insertion of an arc.

4.3 Correctness

At the beginning of the algorithm, the preimage graph has no arcs. We should show that the above operations result in a maximum spanning forest, if they are applied to one such forest. The *insert*(a) operation, breaks the cycle that is formed by adding the arc a between two nodes of a tree. It does that by removing the arc of minimum weight, say b , on this cycle, if that weight is less than the weight of a . Therefore the overall weight of the tree increases. If a tree with higher weight existed, then it should have a as an arc. By removing a and inserting a missing edge of the cycle (with weight at least that of b), we get a new tree with higher weight for the original graph before insertion, which is a contradiction.

In $delete(a)$, note that every other arc in the whole graph has weight at least as large as weight of a , since the weights are deletion times. Every arc that exists, either is deleted during the current call to $UpdatePreimage()$ or has a higher weight. If all of the weights are higher than that of a , then, the deletion indeed splits the maximum spanning tree in two. Otherwise, any arc reconnecting the resulting trees is also deleted before update process finishes and before any $find$ queries, therefore there is no harm in not connecting back the split trees. We have the following theorem:

Theorem 1 *Reeb graph of a piecewise linear function on a simplicial complex can be constructed in $O(m \log m)$ time in the worst case, where m is the size of the 2-skeleton.*

5 Implementation

We use “lazy insertion” to make the implementation faster. Roughly speaking, our goal is not to insert arcs that die (i.e. get deleted) before any Reeb-critical value is met.

5.1 Simple Reeb-Regular Vertices

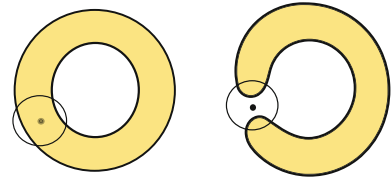
We recall some concepts before going into details. The *star* of a simplex $\sigma \in K$ is the collection of simplices of K that have σ as a face. This might not be a simplicial complex, however, if we add the missing faces of the simplices in the star, we obtain a subcomplex of K which is called *closed star* of σ . Here we are only concerned with stars of vertices. With $f : |K| \rightarrow \mathbb{R}$ as above, the *lower star* of a vertex $v \in K$ contains all the simplices in the star for which $f(v)$ is the maximum value among its vertices. Again, the *closed lower star* is obtained by adding missing simplicies. *Upper star* and *closed upper star* are defined symmetrically.

If the upper star and the lower star of a vertex both have just one component, then the vertex is Reeb-regular. We use this fact, to quickly decide if a vertex is Reeb-regular of this kind, which we call *simple* Reeb-regular. Note that, if this is not the case, the vertex still can be Reeb-regular. However, non-simple Reeb-regular vertices tend to be smaller in number compared to simple Reeb-regular vertices in practice.

Non-simple regular vertices happen for instance when the first Betti number of the preimage changes but the links do not have any non-trivial loop. For an example, consider the case depicted in Fig. 3. The figure on the right is the preimage before processing v and the figure on the left is the preimage after processing v . The small circles are intended to show the lower and upper links of v . We can think of these figures as being two dimensional, that is, a deformed disk and an annulus. Then, the links would be parts of the preimage inside the small circles. It is clear that the lower link has two components and the upper link has one component and the vertex is Reeb-regular. The space here could be a 3-manifold with boundary.

In our implementation, we first check if the vertex is simple Reeb-regular, if so, we do not insert the corresponding arcs into the preimage, rather, we merely keep them for insertion later in an *insertion list*. For such a vertex, the arcs that should be removed, will be removed from the insertion list and the current preimage spanning trees. This

Fig. 3 A non-simple Reeb-regular vertex



causes some of the spanning trees to be currently not valid and incomplete since we are just removing arcs and not inserting any arc into the preimage forest.

If a vertex is found to be not simple Reeb-regular, we build the preimage trees completely from the arcs survived in the insertion list and continue the algorithm as usual. This means, we insert the arcs in the insertion list one by one into the current preimage graph as if they are being first encountered.

We remark that building the preimage from insertion list involves also keeping track of the corresponding Reeb graph nodes of the spanning tree roots. This needs special care, since all the arcs incident to a root might have been removed, or the entire tree might have been removed before reaching a vertex which is not simple Reeb-regular.

5.2 Performance comparison

We did a preliminary implementation of the algorithm using the simple “linear” tree data structure, that is, the data structure is simply a set of nodes, every node contains a pointer to its parent and the weight of the arc to its parent if it is not a root. Each operation is done trivially by following the parent pointers. In the worst case, tree operations will need $O(n)$ time over a graph of n nodes, however, as the running times below demonstrate, it is promising.

We compare our implementation with that of [15] which we call RANDREEB. This algorithm takes $O(m \log m)$ time in expectation, and has actual running times superior to earlier ones, see [15]. The exception is the surgery method of [30]. However, this approach cannot handle arbitrary 3-manifolds or simplicial complexes. The running times are shown in Table 1. The input data sets are almost the same as those of Harvey et al. [15] and were kindly provided to us by the authors. We ran the experiments on

Table 1 Comparison of running times

Data set	m	IV	RANDREEB	Ours
Camel	110,785	0.03	0.32	0.99
Simulation	190,165	0.6	1.64	1.39
Fighter	245,300	0.49	6.70	1.95
Blunt	762,683	0.05	13.29	5.12
Post	2,086,950	0.003	17.32	13.45
Buckyball	4,322,620	0.05	69.11	36.63
Plasma	4,530,561	0.02	135.79	42.35
Earthquake	7,085,157	0.05	177.68	71.41

The running times are in seconds. The size of the 2-skeleton, m , is the total number of vertices, edges, and triangles of the input complex. IV is the fraction of Reeb-critical and non-simple Reeb-regular vertices of all the vertices

a 64-bit computer with Dual-Core 3.00 GHz CPU and 8 GB's of memory running a Linux operating system.

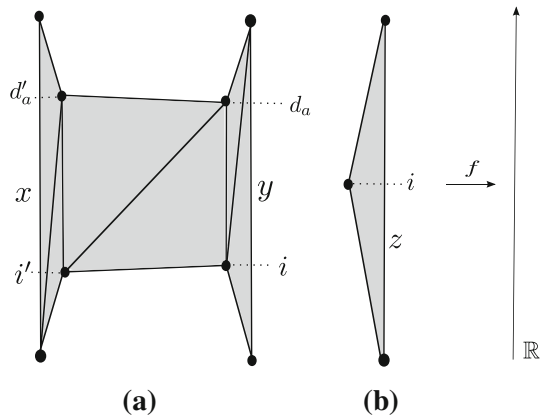
With the exception of the *Camel* and *Simulation*, all data sets are manifolds. For further information on the data sets we refer to [15,30] and the aim@shape database. As can be seen from the table, the algorithm shows an almost linear performance. We note that this linearity is mostly the result of lazy insertion trick. The IV (important vertex) column shows the fraction of Reeb-critical vertices and non-simple Reeb regular vertices of all the vertices. These vertices cause the non-linearity of the algorithm, which is barely noticeable, since the fraction is small. As long as this fraction remains low, the algorithm works specially well, even with trivial implementation of the tree data structure. With a full fledged dynamic tree data structure, we expect the running times to improve substantially, at least for large data sets, as building the preimage at the value of a IV involves a large number of tree operations. The source code is available upon request.

6 More on Graph Connectivity

We distinguish between two variants of the graph connectivity problem. The first is what we called the semi-dynamic graph connectivity. *Semi-dynamic* graph connectivity problem is answering the queries of the dynamic graph connectivity problem when we can compute the deletion times of every edge inserted at the time of its insertion. The other variant, which we call the *offline* dynamic graph connectivity, refers to answering the queries of a known sequence of dynamic graph connectivity operations. In the following, we first show that the offline connectivity is a special case of semi-dynamic connectivity when the deletion times can be computed in constant time. Later, we will give a linear (in the number of operations) time reduction from offline graph connectivity to the Reeb graph computation.

Let $C = c_1c_2 \dots c_m$ be a sequence of three types of dynamic graph connectivity operations over a fixed node set of size n , starting with an empty graph. The parameters of operations are fixed and we think of them as indexing the nodes of the graph, thus integers in $\{1, \dots, n\}$. The problem of answering the queries in a given sequence like C is what we call the offline graph connectivity problem. C consists of some arc insertions and deletions mixed with queries. We say i is the time when operation c_i happens. We can determine the deletion time of any arc in constant amortized time as follows. We make indices for arcs inserted and deleted using the two integers indexing its endpoint nodes, and then sort them using a linear time algorithm for sorting integers. Then we can find operations that index the same arc. This takes time linear in the number of edges inserted. Having found the deletion times in constant time, we can solve this problem in $O(\log n)$ time per operation, as above. Here we again note that this bound matches that of [13] which has given an algorithm for the harder problem of answering queries for an offline dynamic minimum spanning tree problem.

In the more general setting of semi-dynamic graph connectivity, i.e. when we can compute the deletion time of every arc at the time of its insertion, say with worst-case (amortized) time $d(n)$, we can apply the technique above and get an algo-

Fig. 4 Making a complex

rithm that runs in time $O(\log n + d(n))$ in worst-case (amortized) for all of the operations.

6.1 Reduction to the Reeb graph

Here we show a linear time reduction from the offline graph connectivity problem, as defined above, to the Reeb graph construction. This implies that the two problems are equivalent in terms of computational complexity. Given a sequence C of m graph connectivity operations, we construct a simplicial complex whose Reeb graph contains the answers to the queries and these can be read off in constant time. For this, we use the *augmented* Reeb graph. It is the same as the Reeb graph except every vertex of the input complex has a corresponding node in it. Reeb-regular vertices are degree-two nodes with one incoming and one outgoing arc. It is clear that with some modifications, our algorithm can compute the augmented Reeb graph within the same time bound. Moreover, every Reeb-regular vertex can contain an identifier of the preimage component (Reeb graph arc) that it belongs to.

We build the complex and the function at the same time. For simplicity, we assume that at the end of the sequence C , all arcs are deleted. For every node of the graph that will have an incident arc we create a sufficiently long vertical edge, say of length $m + 1$. So, at the beginning we have a collection of edges all of the same height. We consider each c_i in turn. If c_i is an insertion of an arc $a = xy$ with deletion time d_a , then we connect two edges x and y as in Fig. 4a, where the vertices are assigned the heights as in the figure. In Fig. 4, i' and d'_a are slightly perturbed values. If c_i is a query we add a regular vertex as in Fig. 4b with height i . If c_i is a deletion, we do nothing. The function on this complex is the height function. It is easy to verify that when the augmented Reeb graph of this complex is computed, the answer to the query is the component identifier kept with the node corresponding to the regular vertex.

Acknowledgments The author thanks Herbert Edelsbrunner for support and valuable suggestions on the paper. This work was partially done when the author was visiting IST Austria, Klosterneuburg, Austria.

References

1. Acar, U.A., Blueloch, G.E., Harper, R., Vittes, J.L., Woo, S.L.M.: Dynamizing static algorithms, with applications to dynamic trees and history independence. In: Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete algorithms, SODA '04, pp. 531–540. Society for Industrial and Applied Mathematics, Philadelphia (2004)
2. Alstrup, S., Holm, J., Lichtenberg, K.D., Thorup, M.: Maintaining information in fully dynamic trees with top trees. *ACM Trans. Algorithms* **1**, 243–264 (2005)
3. Aujay, G., Hétry, F., Lazarus, F., Depraz, C.: Harmonic skeleton for realistic character animation. In: Proceedings of the 2007 ACM SIGGRAPH/Eurographics Symposium on Computer Animation, SCA '07, pp. 151–160. Eurographics Association, Aire-la-Ville (2007)
4. Bepalov, D., Regli, W.C., Shokoufandeh, A.: Reeb graph based shape retrieval for CAD. *ASME Conf. Proc.* **2003**(36991), 229–238 (2003)
5. Biasotti, S., De Floriani, L., Falcidieno, B., Frosini, P., Giorgi, D., Landi, C., Papaleo, L., Spagnuolo, M.: Describing shapes by geometrical–topological properties of real functions. *ACM Comput. Surv.* **40**(4), 12:1–12:87 (2008)
6. Biasotti, S., Giorgi, D., Spagnuolo, M., Falcidieno, B.: Reeb graphs for shape analysis and applications. *Theor. Comput. Sci.* **392**(1–3), 5–22 (2008)
7. Carr, H., Snoeyink, J., Axen, U.: Computing contour trees in all dimensions. *Comput. Geom.* **24**(2), 75–94 (2003)
8. Cole-McLaughlin, K., Edelsbrunner, H., Harer, J., Natarajan, V., Pascucci, V.: Loops in Reeb graphs of 2-manifolds. In: Proceedings of the Nineteenth Annual Symposium on Computational Geometry, SCG '03, pp. 344–350. ACM, New York (2003)
9. Dey, T.K., Wang, Y.: Reeb graphs: approximation and persistence. In: Proceedings of the 27th Annual ACM Symposium on Computational Geometry, SoCG '11, pp. 226–235. ACM, New York (2011)
10. Doraiswamy, H., Natarajan, V.: Efficient algorithms for computing Reeb graphs. *Comput. Geom.* **42**(6–7), 606–616 (2009)
11. Edelsbrunner, H., Harer, J., Mascarenhas, A., Pascucci, V., Snoeyink, J.: Time-varying Reeb graphs for continuous space-time data. *Comput. Geom.* **41**(3), 149–166 (2008)
12. Edelsbrunner, H., Harer, J., Patel, A.K.: Reeb spaces of piecewise linear mappings. In: Proceedings of the Twenty-fourth Annual Symposium on Computational Geometry, SCG '08, pp. 242–250. ACM, New York (2008)
13. Eppstein, D.: Offline algorithms for dynamic minimum spanning tree problems. *J. Algorithms* **17**(2), 237–250 (1994)
14. Fujishiro, I., Takeshima, Y., Azuma, T., Takahashi, S.: Volume data mining using 3D field topology analysis. *IEEE Comput. Graph. Appl.* **20**, 46–51 (2000)
15. Harvey, W., Wang, Y., Wenger, R.: A randomized $O(m \log m)$ time algorithm for computing Reeb graphs of arbitrary simplicial complexes. In: Proceedings of the 2010 Annual Symposium on Computational Geometry, SoCG '10, pp. 267–276. ACM, New York (2010)
16. Hilaga, M., Shinagawa, Y., Kohmura, T., Kunii, T.L.: Topology matching for fully automatic similarity estimation of 3D shapes. In: Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '01, pp. 203–212. ACM, New York (2001)
17. Holm, J., de Lichtenberg, K., Thorup, M.: Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity. *J. ACM* **48**, 723–760 (2001)
18. Kanongchaiyos, P., Shinagawa, Y.: Articulated Reeb Graphs for Interactive Skeleton Animation. Modeling: Modeling Multimedia Information and System, Nagano (2000)
19. Natali, M., Biasotti, S., Patane, G., Falcidieno, B.: Graph-based representations of point clouds. *Graph. Models* **73**(5), 151–164 (2011)
20. Pascucci, V., Scorzelli, G., Bremer, P.-T., Mascarenhas, A.: Robust on-line computation of Reeb graphs: simplicity and speed. *ACM Trans. Graph.* **26**, 58 (2007)
21. Patrascu, M., Demaine, E.D.: Logarithmic lower bounds in the cell-probe model. *SIAM J. Comput.* **35**, 932–963 (2006)
22. Rekleitis, I., Lee-Shue, V., Choset, H.: Limited communication, multi-robot team based coverage. In: Proceedings of the 2004 IEEE International Conference on Robotics and Automation ICRA 04 2004, vol. 4, pp. 3462–3468 (2004)

23. Shi, Y., Lai, R., Krishna, S., Sicotte, N., Dinov, I., Toga, A.W.: Anisotropic Laplace–Beltrami eigenmaps: bridging Reeb graphs and skeletons. In: *Computer Vision and Pattern Recognition Workshop*, pp. 1–7. IEEE Computer Society, Anchorage (2008)
24. Shinagawa, Y., Kunii, T.L.: Constructing a Reeb graph automatically from cross sections. *IEEE Comput. Graph. Appl.* **11**, 44–51 (1991)
25. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. *J. Comput. Syst. Sci.* **26**(3), 362–391 (1983)
26. Sleator, D.D., Tarjan, R.E.: Self-adjusting binary search trees. *J. ACM* **32**, 652–686 (1985)
27. Tarjan, R.E., Werneck, R.F.: Self-adjusting top trees. In: *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA '05*, pp. 813–822. Society for Industrial and Applied Mathematics, Philadelphia (2005)
28. Tarjan, R.E., Werneck, R.F.: Dynamic trees in practice. *J. Exp. Algorithmics* **14**, 5:4.5–5:4.23 (2010)
29. Thorup, M.: Near-optimal fully-dynamic graph connectivity. In: *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing, STOC '00*, pp. 343–350. ACM, New York (2000)
30. Tierny, J., Gyulassy, A., Simon, E., Pascucci, V.: Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees. *IEEE Trans. Vis. Comput. Graph.* **15**, 1177–1184 (2009)
31. Tung, T., Schmitt, F.: The augmented multiresolution Reeb graph approach for content-based retrieval of 3D shapes. *Int. J. Shape Model.* **11**(1), 91–120 (2005)
32. Wood, Z., Hoppe, H., Desbrun, M., Schröder, P.: Removing excess topology from isosurfaces. *ACM Trans. Graph.* **23**, 190–208 (2004)