Algorithms for the Reeb Graph and Related Concepts

by

Salman Parsa

Department of Computer Science
Duke University

Date: ____________________________

Approved:

____
Herbert Edelsbrunner, Co-Supervisor

____
Pankaj Agarwal, Co-Supervisor

____
Sayan Mukherjee

____
Paul Bendich

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2014

Abstract

Algorithms for the Reeb Graph and Related Concepts

by

Salman Parsa

Department of Computer Science
Duke University

Date: ______________________

Approved:

____
Herbert Edelsbrunner, Co-Supervisor

____
Pankaj Agarwal, Co-Supervisor

____
Sayan Mukherjee

____
Paul Bendich

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2014

Copyright © 2014 by Salman Parsa
All rights reserved except the rights granted by the
Creative Commons Attribution-Noncommercial Licence

Abstract

This thesis is concerned with a structure called the Reeb graph. There are three main problems considered. The first is devising an efficient algorithm for constructing the Reeb graph of a simplicial complex with respect to a generic simplex-wise linear real-valued function. We present an algorithm that builds the Reeb graph in $O(m \log m)$ worst-case deterministic time, where m is the size of the complex. This was the first deterministic algorithm with this time bound. Without loss of generality, the complex is assumed to be 2-dimensional. The algorithm works by sweeping the function values and maintaining a spanning forest for the preimage, or the level-set, of the value. Using the observation that the operations that change the level-sets are off-line, we was able to achieve the above bound.

The second topic is the dynamic Reeb graph problem. As the function changes its values, the Reeb graph also changes. The problem is to update the Reeb graph after a change in function values. We reduce the problem into a graph theoretic problem which we call retroactive graph connectivity. The approach allows us to solve the problem in special cases efficiently, for example, for 2-manifolds and the offline setting. However, for the general case, we only state results based on the running time of the retroactive graph connectivity algorithm.

The third topic is an argument regarding the complexity of computing Betti numbers. This problem is also related to the Reeb graph by means of the vertical Homology classes. The problem considered here is whether the realization of

a simplicial complex in $\mathbb{R}^4$ can result in an algorithm for computing its Homology groups faster than the usual matrix reduction methods. Using the observation that the vertical Betti numbers can always be computed more efficiently using the Reeb graph, we show that the answer to this question is in general negative. For instance, given a square matrix over the field with two elements, we construct a simplicial complex in linear time, realized in $\mathbb{R}^4$ and a function on it, such that its Horizontal homology group, over the same field, is isomorphic with the null-space of the matrix. It follows that the Betti number computation for complexes realized in $\mathbb{R}^4$ is as hard as computing the rank for a sparse matrix.

Contents

Abstract **iv**

List of Tables **ix**

List of Figures **x**

1 Introduction **1**

1.1 Overview of the Thesis 3

1.2 Formal Statements of Results 4

2 Background Material **7**

2.1 Simplex and Simplicial Complex 7

2.2 Reeb Graph 9

2.3 Simplicial Homology 11

2.4 Computing Simplicial Homology 13

2.5 Singular Homology 14

2.6 Horizontal and Vertical Homology Classes 16

3 Computing the Reeb Graph **17**

3.1 Related work 17

3.2 Algorithm 19

3.2.1 The Reduction 20

3.2.2 Graph Connectivity 23

3.2.3 Correctness 26

3.3 Implementation 26
3.3.1 Simple Reeb-Regular Vertices 27
3.3.2 Performance Comparison 28
3.4 More on Graph Connectivity 30
3.4.1 Reduction to the Reeb Graph 31

4 Dynamic Updates of the Reeb Graph and Retroactive Graph Connectivity 33

4.1 Introduction 33
4.2 A Retroactive Graph Connectivity Problem 39
4.2.1 Off-The-Shelf Algorithms for the Retroactive Graph Connectivity 41
4.3 The Framework of an Algorithm for Dynamic Reeb Graphs 42
4.4 Transforming an Interchange Event to Swaps 43
4.4.1 Notation and terminology 43
4.4.2 Reducing an interchange to `swap` operations 45
4.4.3 Updating the Reeb graph 48
4.5 Discussion 50

5 Loops in the Reeb Graph and Computing Homology of (Embedded) Simplicial Complexes 53

5.1 Reducing Rank Computation to Betti Numbers of Simplicial Complexes 55
5.2 Extensions 63
5.2.1 More general complexes. 63
5.2.2 Finite fields 64
5.2.3 Matrix sparsification 65
5.3 Integer Coefficients and Invariant Factors 67

6 Conclusion 72

Bibliography **74**

Biography **79**

List of Tables

3.1 Comparison of running times. 30

List of Figures

2.1 a drawing of the Reeb graph 10

3.1 processing of a Reeb-critical vertex 22

3.2 A non-simple Reeb-regular vertex 28

3.3 making a complex 32

4.1 Interchange of two vertices 36

4.2 schematic view of the notation 40

4.3 level-set graphs 45

4.4 Removing two nodes creates dangling arcs. 49

4.5 Large number of loops are created/removed by a single interchange. . . 51

5.1 the first reduction 58

5.2 the second construction 61

5.3 prime field example 65

5.4 the Mobuis band 68

Acknowledgements

I would like to thank all my teachers, from the very first, who were my family, to later ones, all the way to the time of this writing. Some teachers I never met.

I want to thank my advisor, Herbert, for his patience with me, and, for everything I learnt from him, in the way of work and beyond. My thanks goes also to Pankaj, for accepting to be my committee chair and his help during my PhD. I thank other committee members too for their time and help.

I would like moreover, to thank all the great people, staff and faculty and administration, at IST Austia, who hosted me during a large part of my studies. IST never stopped providing a great environment for research.

Never did my heart stop receiving knowledge,

A few mysteries remained unresolved.

I contemplated for seventy years, day and night,

I realized that, nothing I have realized.

-Omar Khayyam (1048-1131)

1

Introduction

The central object of study in this thesis is a 1-dimensional space constructed from a simplicial complex and a generic map defined on it. It is called the *Reeb graph* of the simplicial complex with respect to the function. A simplicial complex is a finite structure and it is commonly used in computers for representing mathematical spaces such as manifolds, more general cell complexes, and other topological spaces. The input data for an algorithm that computes a property of a space is most often a simplicial complex or can be easily turned into a simplicial complex. In computational topology, the goal is to develop theory and algorithms for topological properties of a space. In general however, a simplicial complex representation might be too big in size, or computing a topological property might not need the whole complex. Therefore, simpler representations are required in many applications. Reeb graphs are one such construction.

Roughly speaking, the Reeb graph tracks the connected components of levelsets of a function on topological space. A contour tree is a special Reeb graph, one that is a tree. Imagine the surface of the earth. The altitude map assigns to each point of the surface a real value. Fix an altitude value and consider all points on the surface

that have this fixed altitude. This is a *levelset* of the altitude map. If the surface is smooth this levelset generically consists of closed curves which are called *contours*, since the surface has no boundary. For example, points on the flanks of a mountain having the same altitude. Observe that the peak of the mountain is a point in its levelset (a constant curve) disjoint from others. If one varies the fixed altitude value, the connected components of the levelset, or contours, move on the surface. They might merge with others, split into more components, or just disappear or appear. The mountain peak disappears when the altitude value increases.

Informally, the Reeb graph is the space of the contours. In the case of the earth surface, the Reeb graph has no loops and is called the *contour tree*. It can be shown that the loops of the Reeb graph correspond to non-trivial loops of the domain.

Formally, let K be a simplicial complex and $f : |K| \to \mathbb{R}$ a continuous real valued piece-wise linear map on its underlying space. Call two points of $|K|$ *equivalent* if i) they have the same function value and, ii) they are connected in the levelset of that value. It is easily checked that this relation is indeed an equivalence and one can take the quotient space of $|K|$ with respect to this relation. Since the complex is assumed finite, the resulting space is a 1-dimensional topological space called the *Reeb graph* of f and denoted $R(K, f)$. More details will be given in Chapter 2.

Reeb graphs were first defined by Reeb (1946). They were applied to visualization and surface reconstruction by Shinagawa et al. (1991). Besides being theoretically interesting, Reeb graphs have found many more applications in recent years. The areas in which they are applied include shape matching and retrieval, Hilaga et al. (2001); Tung and Schmitt (2005); Bespalov et al. (2003), shape segmentation and simplification, Shi et al. (2008); Wood et al. (2004), animation, Kanongchaiyos and Shinagawa (2000); Aujay et al. (2007), high-dimensional data visualization and analysis, Natali et al. (2011); Fujishiro et al. (2000) and robotics, Rekleitis et al. (2004). We refer to Biasotti et al. (2008a,b) for a survey of the Reeb graph and its applications. More

recent applications are in Buchin et al. (2013); Chazal and Sun (2014).

1.1 Overview of the Thesis

In this section we give an informal overview of the problems and results of this thesis. We consider two algorithmic problems regarding the Reeb graph. In addition, we obtain a general complexity result for computing homology groups which we obtain as a consequence of the algorithms. The first algorithm constructs the Reeb graph of a simplex-wise linear generic function defined on the simplicial complex. The high-level idea in efficient construction of the Reeb graph is to sweep the values of the function and to maintain the connected components of the level sets. This approach can be found in Doraiswamy and Natarajan (2009) and elsewhere. They use dynamic tree data structures for maintaining the levelsets and, as a consequence, incur computational costs that make their algorithm not efficient. In this work, we use a weighting technique to construct the Reeb graph more efficiently and in an almost worst-case optimal running time. This algorithm is described in Chapter 3. This is the first deterministic algorithm for constructing the Reeb graph which has runtime less than quadratic in the size of the complex, in the worst case. See also Chapter 3 for the history of algorithms for the Reeb graph.

The second algorithmic problem asks for the dynamic updates of the Reeb graph, given dynamic updates of the function values. The fundamental operation is the interchange of the values of two vertices with neighboring values. This is an *interchange event*. In general, the interchange event can force a global reorganization of the Reeb graph. However, the size of the combinatorial change is a function of the size of stars of the vertices involved. We give a reduction of this problem to a simple graph problem, we call retroactive graph connectivity. The dynamic Reeb graph problem was not considered before in full generality. Our approach to this problem gives almost optimal algorithms in certain cases, however, not in general. We state the running

times of the update algorithm for the Reeb graph based on running times of the graph problem. The retroactive graph connectivity problem is of interest in itself. For more on the history and related work on this problem see the introduction to Chapter 4.

In the last chapter, we give a lower bounds argument for the complexity of computing homology groups for simplicial complexes. For the most part we will be considering homology with coefficients from the field with two elements $\mathbb{F}_2$. At first glance, this topic might appear unrelated. The connection arises when we distinguish the vertical first homology, which is the homology of the Reeb graph, from the horizontal first homology, which is the orthogonal complement of the vertical homology in the first homology of the simplicial complex (with field coefficients). As a result of efficient Reeb graph computation, the vertical first homology group is easy to compute. Therefore the horizontal first homology group must be the difficult part.

Informally, the result implies that computing the homology of simplicial complexes cannot be done by any other method faster than matrix computations. By matrix computation we mean computing the cokernel of the map defined by the matrix. This result holds for matrices and homology over prime fields or integers. In the case of the field coefficients, the result says that computing the first Betti number of complexes is equivalent to rank computation for sparse matrices. The results of Chapter 5 where published in Edelsbrunner and Parsa (2014).

1.2 Formal Statements of Results

Construction of the Reeb graph, the static algorithm. Let K be a simplicial complex of size m. Let $f : |K| \to \mathbb{R}$ be a simplex-wise linear generic map. We show that the Reeb graph $R(K, f)$ can be constructed in $O(m \log m)$ worst-case deterministic time, given K and f.

Dynamic Reeb graphs. Let K be as above. A generic real-valued simplex-wise linear map on $|K|$ is uniquely determined by the vertex values. The Reeb graph is determined by the ordering of these values in $\mathbb{R}$. Let an *interchange event* be the change of f by a swap of two consecutive values in this ordering, thus changing the ordering by a transposition. Let l be size of stars of the vertices involved in the interchange. We present an example with $\Omega(l)$ combinatorial changes in the Reeb graph after a single interchange. This example shows that in general the change in the Reeb graph can be $\Omega(m)$ and hence it is necessary to state the running times as a function of l. We reduce this problem to a graph problem which we call retroactive graph connectivity. Using a data structure for retroactive graph connectivity we then update the Reeb graph. This reduction gives efficient algorithms on certain domains, for example 2-manifolds. In the general setting, we do not know if the retroactive graph connectivity can be solved efficiently.

Reducing matrix computations to computing homology of embedded simplicial complexes. One of the consequences of the efficient Reeb graph algorithm is that the vertical first homology group of a generic simplex-wise linear function on a complex can always be computed efficiently. We show that computing the rest of the homology, namely computing the first horizontal homology group for simplicial complexes is equivalent to certain well-known matrix problems. This equivalence can be proved already for 2-dimensional simplicial complexes (geometrically) realized in $\mathbb{R}^4$. We can state the result as follows. Let M be an n-by-n integer matrix. In linear time it is possible to construct a 2-complex $K = K(M)$ and a generic map $f : |K| \to \mathbb{R}$ such that the horizontal homology of K with respect to f is isomorphic with $\mathbb{Z}^n/ker(M)$, where $ker(M)$ is kernel of the homomorphism $\mathbb{Z}^n \to \mathbb{Z}^n$ defined by M. The size of K is proportional to the bit-complexity of M, and K embeds in linear time in $\mathbb{R}^4$, i.e., K is realized in $\mathbb{R}^4$ after a constant number of subdivisions. The above is also true

when $\mathbb{Z}$ is replaced with a prime field.

For instance, it follows that computing the the rank of an n-by-n matrix over $\mathbb{F}_2$, the field with two elements, reduces to computing the first Betti number of a 2-dimensional simplicial complex. On the other direction, the Betti numbers of a simplicial complex can be computed by rank computations for a finite number of sparse matrices. It follows that computing the Betti numbers of simplicial complexes is equivalent to computing the rank of sparse matrices.

The above results are easy to show for general cell complexes. The point here is that simplicial complexes essentially are as inefficient as cell complexes in the worst-case and even if they are 2-complexes embedded in $\mathbb{R}^4$. However, dealing with a simplicial complex representation is like dealing with a sparse matrix. See Chapter 5 for more details.

2

Background Material

In this chapter, we present the required definitions and basic concepts required throughout the main body of the dissertation. We start by defining simplicial complexes and related concepts in Section 2.1. Then we define the Reeb graph and present its basic properties in Section 2.2. Next, we define homology (in its simplest form) and give a quick overview of its computation for simplicial complexes in Sections 2.3 and 2.4. In the last section, we also define singular homology since it will be needed in Chapter 5. We refer to textbooks on algebraic topology for more detailed treatments of the concepts below, for instance, Munkres (1984) and Hatcher (2001).

2.1 Simplex and Simplicial Complex

A *d-simplex* is the convex hull of $d+1$ affinely independent points $V = \{v_0, \ldots, v_d\}$ in some Euclidean space, e.g. $\mathbb{R}^d$, where $d \geqslant 0$. The set V 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 $\sigma \leqslant \delta$ or by $\sigma < \delta$ if the subset is proper. 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 $\sigma \leqslant \delta \Rightarrow \sigma \in K$, 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. Note that the simplicial complex thus defined is a collection of simplices glued at shared faces. Denote by K_i the set of i-simplices of K and by n_i the number of elements of K_i. The *size* of K is the total number of simplices, namely, $m = \sum_{i \geqslant 0} n_i$. A *subcomplex* of the simplicial complex K is a set of simplices $L \subset K$ such that L is a simplicial complex. Let $\sigma \in K$ be a simplex. The *star* of σ is the set of simplices in K having σ as a face.

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 example, K is a subset of a standard $(n_0 - 1)$-simplex in $(n_0 - 1)$-dimensional Euclidean space and can have a topology induced by this realization. One can also define $|K|$ to be the topological space obtained from the disjoint union of the simplices in K and when one identifies corresponding points in the common faces of simplices. 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. This set is also a complex.

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 $K_0 = \{v_1, v_2, \ldots, v_{n_0}\}$ and $\sum_i b_i = 1$ and $b_i \geqslant 0$, with strict inequality when v_i is a vertex of σ. The numbers b_i are called the *barycentric coordinates* of $x \in |K|$.

A usual graph can be turned into a simplicial 1-complex if one takes a 1-simplex for an arc. However, in general a graph is not a 1-dimensional simplicial complex as defined above. It may have more than one edge connecting same two vertices, or

other violations of the definition of the complex. In general a graph for us is a set of edges over vertices rather than a geometric structure. To express the difference with a complex, we call the vertices of a graph *nodes* and its edges *arcs*.

2.2 Reeb Graph

Let $f : K_0 \to \mathbb{R}$ be a function. We say that f is *generic* if it is injective. 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 *simplex-wise linear* (or sometimes piece-wise 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 difficult to see that the restriction to the 20-skeleton of the preimage is a simplicial complex, namely, $f^{-1}(r) \cap K^2 = \{\sigma \cap f^{-1}(r) : \sigma \in K^2\}$. 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* $R(K, f)$ of the function $f : |K| \to \mathbb{R}$ is the topological graph obtained by contracting every connected component of $f^{-1}(r)$ to a point, for every $r \in \mathbb{R}$. 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 that meet at points where corresponding connected components merge.

It is easily seen that when r changes continuously without passing any value in

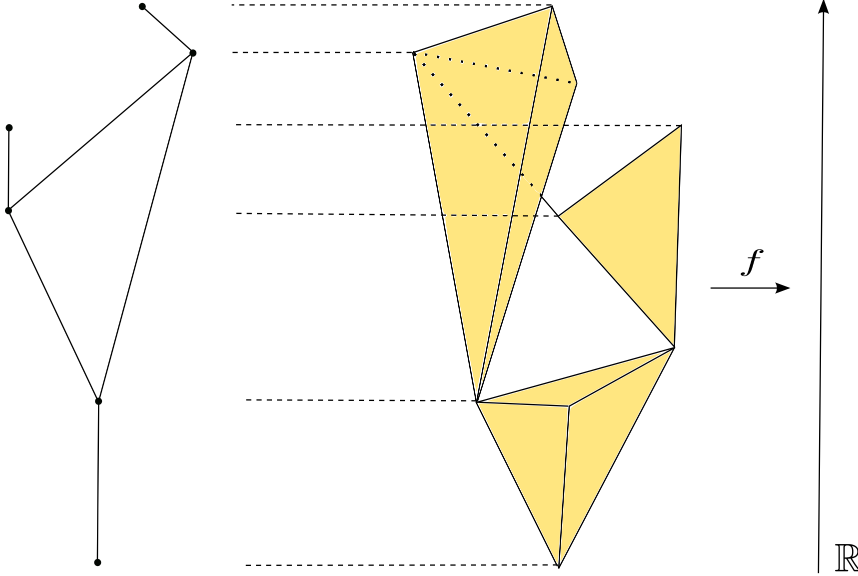

FIGURE 2.1: A drawing of the Reeb graph for the height function on a simplicial complex realized in in 3-dimensional space. The dotted horizontal lines show correspondence of vertices of the complex with nodes of the Reeb graph. Note that the Reeb graph is not augmented.

$f(K_0)$, the connected components of the preimages $f^{-1}(r)$ remain unchanged. 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 such that K is realized in $\mathbb{R}^k$. 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. This is why we look at the Reeb graph as an abstract graph. We also note that, the Reeb graph is in fact a multigraph.

Sweeping the Reeb graph in increasing function value, we note that 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*.

An important result that is also easy to prove is the following.

Lemma 1. *The Reeb graph of the complex K and $f : K \to \mathbb{R}$ depends only on the restriction of f to the 2-skeleton of K.*

Proof. let $\sigma \in K$ be a d-simplex with $d \geqslant 3$. Assume $|\sigma| \subset \mathbb{R}^d$. The preimage of r in $|\sigma|$ is the intersection of a $(d-1)$-plane P with $|\sigma|$. Let σ^2 be the 2-skeleton of σ. Every point in $P \cap |\sigma|$ is connected to a point of $P \cap |\sigma^2|$ and the latter space is connected. Hence, $P \cap |\sigma^2|$ defines the same point as $P \cap |\sigma|$ in the preimage. Therefore, to know the component of the contribution of σ to the preimage it is enough to know the component of the contribution of the 2-skeleton σ^2. Hence, we can remove all the simplices of dimension $\geqslant 3$ without changing the preimage components.

Let R and R' be the original Reeb graph and the Reeb graph defined for the 2-skeleton. Define $h : R \to R'$ to be the function that assigns to a connected component of preimage of K the one in the 2-skeleton defined above. Using the fact that K^2 and K differ by a union of disjoint open sets, namely interior of simplices, it is straightforward to verify that h is a homeomorphism. □

2.3 Simplicial Homology

In the following, we define what is called the homology of a simplicial complex with coefficients in the field $\mathbb{F}_2$ of two elements. We mostly need only this definition. For a general definition with arbitrary coefficients refer to any basic text book on algebraic topology such as Munkres (1984) or Hatcher (2001) .

Let K be a simplicial complex. Assuming a numbering, we write $\Delta_{i,j}$ for the j-th

i-simplex, and we let

$$S_i = S_i(K) = \{\Delta_{i,j} \mid j = 1, 2, \ldots, n_i\}$$

be the set of i-simplices in K. We will drop the subscript, i, when the dimension is clear. Any subset of S_i is called an *i-chain* of K. It can be written as an n_i-vector of 0s and 1s: $c = (c(1), c(2), \ldots, c(n_i))$, where $c(j) = 1$ iff the simplex $\Delta_{i,j}$ belongs to c, for $1 \leqslant j \leqslant n_i$. Two i-chains, c and d, can be added by vector addition modulo 2:

$$(c + d)(j) = c(j) + d(j) \pmod 2.$$

This means that the sum is the chain that consists of all i-simplices in the symmetric difference of the two chains: $c + d = (c \cup d) - (c \cap d)$. With this addition, the set of i-chains forms a group, denoted as $C_i = C_i(K)$. More specifically, C_i is an n_i-dimensional vector space over $\mathbb{F}_2$ with basis S_i.

The *boundary* of an i-simplex, denoted by $\partial_i(\Delta_j)$, is the collection of its faces of dimension $i-1$. It is a chain in C_{i-1}. Since S_i is a basis for C_i, this definition can be extended to a unique homomorphism between vector spaces, $\partial_i : C_i \to C_{i-1}$ defined by

$$\partial_i(c) = \sum_{\Delta_j \in c} \partial_i(\Delta_j),$$

called the *i-th boundary homomorphism.* By definition, the zeroth boundary homomorphism, ∂_0, is the zero map. A chain with empty boundary is called a *cycle.* Hence, the i-cycles are the chains in the kernel of ∂_i, and we write $Z_i = Z_i(K) \subseteq C_i$ for this kernel. For example, if K is a triangulation of a 2-manifold (without boundary), then the chain c that includes all triangles in K is a 2-cycle. Indeed, every edge of K belongs to the boundary of exactly two triangles, which implies that the sum of the boundaries of all triangles is empty. A chain that is the boundary of another chain of one higher dimension is called a *boundary.* Hence, the i-boundaries are the

chains in the image $\partial_{i+1}(C_{i+1})$, and we write $B_i = B_i(K) \subseteq C_i$ for this image. Note that the Z_i and B_i are also vector spaces over $\mathbb{F}_2$, and that $B_i \subseteq Z_i$ because the boundary of a boundary is necessarily empty. We can therefore form the quotient, $H_i = H_i(K) = Z_i/B_i$, called the *i-th homology group* of K (with coefficients in $\mathbb{F}_2$). This quotient is again a vector space over $\mathbb{F}_2$, and its dimension is called the *i-th Betti number* of K, denoted as $\beta_i = \beta_i(K)$. Since H_i is a quotient, its elements are classes of *homologous* cycles. If such a class contains a cycle c, then we denote the class by $[c]$, and we call c a *representative* of the class. Note that the sum of any two representative cycles of the same class is a boundary.

2.4 Computing Simplicial Homology

We already have a basis for the vector space C_i, namely S_i or, equivalently, the vectors of length n_i with only a single 1 each. We need to find bases for Z_i and B_i. Let D_i be the matrix of the i-th boundary homomorphism, ∂_i, with respect to the bases S_i and S_{i-1}. The rows of D_i are indexed by $(i-1)$-simplices and the columns by i-simplices. The j-th column of D_i corresponds to $\Delta_{i,j}$ and is the vector $\partial_i(\Delta_{i,j})$. It follows that we can write the boundary of a chain $c \in C_i$ in matrix notation as $\partial_i(c) = D_i c$. To find a basis for Z_i, we only need to find a basis for the null-space of D_i. We thus reduce the matrix D_i to diagonal form using the usual row- and column-operations. In this form, an initial segment of the diagonal consists of 1s and all other matrix entries are zero. Let R_i and R_{i-1} be the new bases with respect to which the matrix D_i is diagonal. The vectors in R_i that are associated with zero columns form a basis of Z_i, and the vectors in R_{i-1} that are associated with non-zero rows form a basis of B_{i-1}. After reducing all boundary matrices, we have a basis for each Z_i and each B_i. If our interest is in the Betti numbers, we could count basis

vectors and this way get the ranks of the D_i and the dimensions of the vector spaces:

$$\begin{aligned}\mathsf{dim}\, Z_i &= n_i - \mathsf{rank}\, D_i,\\ \mathsf{dim}\, B_i &= \mathsf{rank}\, D_{i+1},\\ \mathsf{dim}\, H_i &= \mathsf{dim}\, Z_i - \mathsf{dim}\, B_i.\end{aligned}$$

However, to get bases for the homology groups, we need to do more. One possibility is to find a basis of B_i that is a subset of a basis for Z_i. Then a basis for homology is given by classes of the basis elements of Z_i not in B_i. This can be done by writing B_i in terms of the given basis for Z_i and computing a basis for the complementary space of B_i, that is, the space of $z \in Z_i$, $z^t B = 0$, where B is a matrix whose columns are the basis for B_i. This is a basis for the nullity of B^t. These operations can be done in matrix multiplication time.

To summarize, assuming a constant dimension of K, bases for all homology groups can be computed in a constant number of matrix reductions, and all Betti numbers can be computed with a constant number of rank computations for sparse matrices.

2.5 Singular Homology

Recall that a cycle in simplicial homology is a collection of simplices in the complex. We also need cycles that possibly cross over simplices and therefore introduce the formalism of singular homology. A *singular i-simplex* is a (continuous) map $\sigma_i : \Delta_i \to |K|$. We write $\bar{S}_i$ for the set of all such maps. A *singular i-chain*, $\bar{c}$, is a finite subset of $\bar{S}_i$. Equivalently, the chain is a function, $\bar{c} : \bar{S}_i \to \{0, 1\}$ with finite support. Thinking of such a function as an infinite vector, and using $\mathbb{F}_2$, we again define addition by setting

$$(\bar{c} + \bar{d})(\sigma_j) = \bar{c}(\sigma_j) + \bar{d}(\sigma_j) \pmod 2.$$

The *i-th singular chain group*, $\bar{C}_i$, is the set of singular i-chains together with addition, which is again a vector space over $\mathbb{F}_2$.

The *boundary* of a singular i-simplex mapping Δ_i to $|K|$ is the sum of the restrictions of the map to the $(i-1)$-dimensional faces of Δ_i. Extending this definition to chains, we get the *boundary homomorphism*, $\bar{\partial}_i : \bar{C}_i \to \bar{C}_{i-1}$. With this, we define the *i-th singular cycle group*, $\bar{Z}_i$, as the kernel of $\bar{\partial}_i$, and the *i-th singular boundary group*, $\bar{B}_i$, as the image of $\bar{\partial}_{i+1}$. As before, we have $\bar{\partial}_{i-1} \circ \bar{\partial}_i = 0$, which implies that all singular boundaries are singular cycles. Finally, we define the *i-th singular homology group* by taking the quotient, $\bar{H}_i = \bar{Z}_i/\bar{B}_i$.

The simplices of K can also be thought of as members of $\bar{S}_i$. This inclusion induces homomorphisms $C_i \to \bar{C}_i$, which in turn define homomorphisms between the homology groups, $H_i \to \bar{H}_i$. A well-known result in algebraic topology asserts that this homomorphism is an isomorphism; see e.g. (Hatcher, 2001, Theorem 2.27). For a simplicial complex, the simplicial and singular homology groups are isomorphic and a basis for simplicial homology is also a basis for singular homology. For a more detailed description of these concepts see any introductory text in algebraic topology, such as Hatcher (2001). The reason for introducing singular in addition to simplicial homology is that it simplifies our definitions and proofs. From this point on, we write S for $\bar{S}$, c for $\bar{c}$, etc.

Remark. Besides facilitating the comparison of spaces with each other, homology groups in low dimensions also have intuitive meanings. For example, the rank of the zeroth homology group is the number of connected components. For a graph, the rank of the first homology group is the number of independent loops, which for a connected graph is the number of edges minus the number of vertices plus one. Note that for a tree, this number is zero.

2.6 Horizontal and Vertical Homology Classes

Let K be a simplicial complex and $f : |K| \to \mathbb{R}$ be a generic simplex-wise linear function on K. The function f divides the each homology group into two summands, or in this case a vector space into orthogonal vector spaces. Formally, we call a homology class $h \in H_i$ *horizontal* if it has a representative cycle whose image under f is a finite set of values in $\mathbb{R}$. The horizontal classes form a subgroup of homology, called the *i-th horizontal homology group* of K with respect to f, denoted as $H_i^{\text{hor}} = H_i^{\text{hor}}(K, f)$. The *i-th vertical homology group* is $H_i^{\text{vcl}} = H_i / H_i^{\text{hor}}$. The ranks of the horizontal and vertical homology groups are called the *horizontal* and *vertical Betti numbers*, denoted as β_i^{hor} and β_i^{vcl}. Note that the i-th Betti number of K satisfies $\beta_i = \beta_i^{\text{hor}} + \beta_i^{\text{vcl}}$. This distinction has been introduced in Cohen-Steiner et al. (2009) and studied in Dey and Wang (2013).

The distinction between first horizontal and first vertical homology classes is significant because there are fast algorithms for computing the latter but no such algorithms for computing the former. More specifically, the first vertical homology group of the complex and the function is isomorphic to the first homology group of the Reeb graph: $H_1^{\text{vcl}}(K, f) \cong H_1(R(K, f))$; see (Dey and Wang, 2013, Theorem 3.2).

We show in Chapter 3 that the Reeb graph of a generic function on a simplicial complex of size m can be computed in $O(m \log m)$ time in the worst case and in Chapter 5 that computing the first horizontal Betti number is equivalent to computing the rank of sparse matrix of size m. It seems unlikely that the rank of such a matrix can be computed in $O(m \log m)$ time. If this is indeed true, then our results prove that vertical and horizontal first homology groups have different computational complexities: the latter one is harder than the former.

3

Computing the Reeb Graph

In this chapter we present an algorithm for constructing the Reeb graph $R(K, f)$ for a given simplicial complex K and generic simplex-wise linear map $f : K \to \mathbb{R}$. Before discussing the algorithm we present the history and related work in this area.

3.1 Related work

Carr et al. (2003) gave an efficient algorithm for computing the contour tree for a function on a simplicial complex domain in time $O(n_0 \log n_0 + n_1 \alpha(n_1))$, where n_0 is the number of vertices and n_1 is the number of edges of the input complex. The first algorithm to compute the Reeb graph was given by Shinagawa and Kunii (1991), it works for the triangulation of a 2-manifold and runs in time $\Theta({n_0}^2)$. 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. (2003) improved the running time to $O(n_0 \log n_0)$. They used circular lists to maintain the preimage.

The Reeb graph of a d-dimensional simplicial complex for $d \geqslant 2$, depends only on the 2-skeleton, whose size we denote by m as usual. One can maintain the preimage

components as graphs, reducing the computation of the Reeb graph to $O(m)$ dynamic graph connectivity operations 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 (2009) were the first to use this reduction to compute the Reeb graph, see Section 3.2 for details.

Holm et al. (2001) gave a deterministic algorithm for dynamic graph connectivity with $O(\log^2 m)$ amortized time per operation. As used by Doraiswamy and Natarajan (2009), this connectivity algorithm resulted in the best deterministic algorithm for the Reeb graph for a general simplicial complex before this work. Moreover, Thorup (2000) 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 (2009) 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. (2009) 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 by Pascucci et al. (2007) with $\Theta(n_0 m)$ running time. Harvey et al. (2010) 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 by Edelsbrunner et al. (2008b). Dey and Wang (2013) 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 Edelsbrunner et al. (2008a).

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 (2006) proved an $\Omega(\log m)$ lower bound for dynamic graph connectivity updates in the cell-probe model. Eppstein (1994) uses a linear time minimum spanning tree algorithm to solve the offline minimum spanning forest 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 work. As is shown in this chapter, any algorithm for offline graph connectivity can be used to construct the Reeb graph. Therefore, Eppstein's offline algorithm implies an algorithm for the Reeb graph in $O(m \log m)$ worst-case time. However, we give a different and simpler algorithm for offline graph connectivity. Our method maintains a minimum spanning forest with respect to carefully chosen weights whose maintenance is easy, while Eppstein's algorithm can be used to maintain any minimum spanning forest.

3.2 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, for every $r \in \mathbb{R}$.

The input to our algorithm is described as a list of vertices, edges and triangles. Edges and triangles are specified as pairs and triplets of vertices. Also, for every vertex we need to know edges and triangles incident on it, which we compute for all

vertices in time linear in number of edges and triangles.

3.2.1 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 $f(v_1) < f(v_2) < \cdots < f(v_{n_0})$. 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 write the vertex with smaller function value first.

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:

In Algorithm 1, we use four subroutines. The *LowerComps*(v_i) subroutine consid-

Algorithm 1 Sweep Algorithm

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

ers 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 Figure 3.1. 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 = DynTrees.find(e)
  if c is not marked then
    Lc.add(c)
    mark c as listed
  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

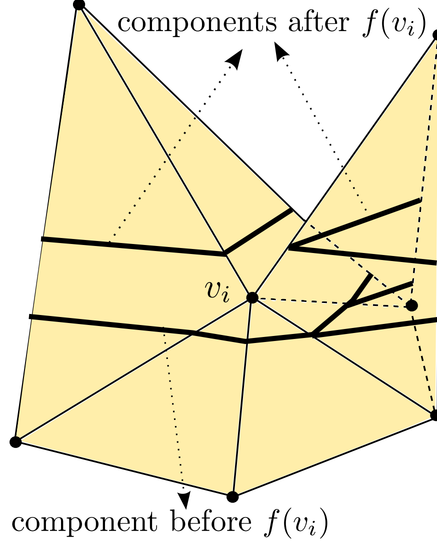

FIGURE 3.1: processing of a Reeb-critical (split) vertex in a 2-d simplicial complex

into the components in Uc. The vertex v_i is Reeb-regular, if and only if Lc and Rc both have 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, ν_{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 ν_{v_i}. A pseudo-code for this subroutine is given in Algorithm 4.

Algorithm 3 UpdatePreimage(v)

```
for all triangles t = {v1, v2, v3} incident on v while f(v1) < f(v2) < f(v3) do
  if v = v3 then DynTrees.delete((v1 v)(v2 v)) end
  if v = v2 then
    DynTrees.delete((v1 v3)(v1 v))
    DynTrees.insert((v v3)(v1 v3))
  end
  if v = v1 then DynTrees.insert((v v2)(v v3)) end
end
```

Algorithm 4 UpdateReebGraph(Lc,Uc)

---

create a new node ν in Reeb graph
assign the node to all $c \in Uc$
create an arc between ν and ν_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 $O(n_1 U(n_1))$ 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 $O(n_2 U(n_1))$. Summing all of these together, the algorithm runs in time $O(mU(m))$ where m is the size of the 2-skeleton.

3.2.2 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_1x_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 of Sleator and Tarjan (1983, 1985), top trees as in Alstrup et al. (2005); Tarjan and Werneck (2005) and RC-trees of Acar et al. (2004). See Tarjan and Werneck (2010) 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. RC-trees achieve the same bound in expected running time.

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

- *find*(x):

```
return root(x)
```

- *insert*(x_1x_2):

```
ω = weight of arc x1x2
if root(x1) = root(x2) then
  evert(x1)
  x′ = minWeight(x2)
  ω′ = x′.weight
  if ω′ < ω then
    cut(x′, parent(x′))
    link(x1,x2, ω)
  end
else
  link(x1,x2,ω)
end
```

- *delete*(x_1,x_2):

```
if (x1,x2) is a tree edge then
  cut(x1,x2)
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.

3.2.3 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.*

3.3 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. Therefore our implementation uses this heuristic. The implementation is $O(n_1)$ per tree operation in the worst-case time. However, as will be seen in Section 3.3.2, it obtains running times superior to earlier implementations. In Section 3.3.2 we compare our algorithm with that of Harvey et al. (2010). The latter algorithm was compared to other algorithms in Harvey et al. (2010).

3.3.1 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 $f : |K| \to \mathbb{R}$ 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, as explained below. 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.3.2. 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,

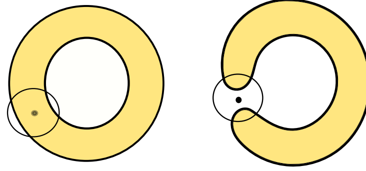

FIGURE 3.2: A non-simple Reeb-regular vertex

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 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.

3.3.2 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 Harvey et al. (2010) which we call RandReeb. This algorithm takes $O(m \log m)$ time in expectation, and has actual running times superior to earlier ones, see Harvey et al. (2010). The exception is the surgery method of Tierny et al. (2009). However, this approach cannot handle arbitrary 3-manifolds or simplicial complexes. The running times are shown in the table below. The input data sets are almost the same as those of Harvey et al. (2010) and were kindly provided to us by the authors. We ran the experiments on 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 Harvey et al. (2010); Tierny et al. (2009) 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. Thess 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 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.

Table 3.1: Comparison of running times. 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.

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

3.4 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 $C = c_1 c_2 \ldots c_m$ 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 $\{1, \ldots, n\}$. 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 Eppstein (1994) 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 algorithm that runs in time $O(\log n + d(n))$ in worst-case (amortized) for all of the operations.

3.4.1 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

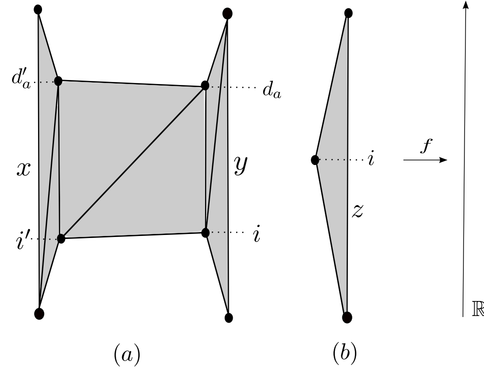

FIGURE 3.3: making a complex

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 Figure 3.3 (a), where the vertices are assigned the heights as in the figure. In Figure 3.3, i' and ${d_a}'$ are slightly perturbed values. If c_i is a query we add a regular vertex as in Figure 3.3 (b) 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.

4

Dynamic Updates of the Reeb Graph and Retroactive Graph Connectivity

In this chapter, we study the problem of dynamically updating the Reeb graph of a simplex-wise linear map on a simplicial complex. This is done by abstracting the problem into a graph problem, which we call retroactive graph connectivity. We start with an overview, continue with the precise problem definition, and, finally we present our reduction of the dynamic Reeb graph to the retroactive graph connectivity problem.

4.1 Introduction

Our goal is to reduce the dynamic updates of the Reeb graph, to operations of a particular graph data structure. Thus, in our approach, an algorithm for dynamic Reeb graphs requires a data structure solving this graph problem, hence there is a connection to graph algorithms. We abstract the main ingredient of the geometric problem into the graph problem, called retroactive graph connectivity. In addition to helping us present an algorithm for the dynamic Reeb graph, the retroactive graph

connectivity defines a variation of the dynamic graph connectivity problem that is of independent interest.

Dynamic vs retroactive data structures. We now explain in more detail what we mean by a dynamic and a retroactive data structure. Dynamic data structures usually support two types of operations: *updates* and *queries*. We show the difference between these two types of operations in an example. A dynamic spanning forest algorithm maintains a spanning forest of a graph. The update operations change the graph whereas query operations do not change the graph; they just ask for information. For example, inserting an edge is an update operation. Asking if two vertices are connected in the forest is a query operation. One can define a dynamic data structure problem by defining a set of operations that it has to support, that is, its *interface*. For instance, we can think of graphs over a fixed node set. We have only one update operation, $\texttt{insert}(a)$ for the arc a, and, one query operation, $\texttt{con}(v, w)$, which asks if nodes v and w are connected. This particular setting is essentially the union-find problem.

Any dynamic data structure changes its state over time depending on the operations requested from it. Assume that starting from time zero, a sequence of operations has been executed, where each operation advances time by one. For instance, a sequence of edge insertions into and edge deletions from a graph. A dynamic data structure has to answer queries about the properties of the graph such as connectivity, etc at the current time. A *retroactive* data structure has to be able to answer historic queries too, that is, queries asking about an earlier state of the graph. Moreover, it has to support retroactive updates, namely, updates that change what happened in the past, that is, updates that change the sequence of operations. This last property is what separates retroactive and persistent data structures. Thus, a persistent data structure only answers queries about the past and the sequence of

updates is not changed, see Demaine et al. (2007) for more information on retroactive data structures and Driscoll et al. (1989) for persistent data structures.

Dynamic Reeb graphs. Let as usual K be a simplicial complex and f a real-valued simplex-wise linear generic map on it. If f changes the Reeb graph changes. The dynamic Reeb graph problem asks for updating the Reeb graph after a change in the function. When f is simplex-wise linear, which we assume throughout, the change in function values can be broken into primitive changes. The Reeb graph only depends on the ordering of the vertices by their f-values. Then a primitive change in function values corresponds to the exchange of the place of two consecutive vertices in this ordering. We give algorithms for updating the Reeb graph after such an *interchange* update. Note that this primitive operation can have many effects on the Reeb graph and incorporates the two kinds of operations defined by Edelsbrunner et al. (2008b), namely interchange and birth-death events. In that paper, authors divide operations into groups depending on the change in the Reeb graph of a Morse function. Whereas we update the Reeb graph to reflect the interchange of values in any generic simplex-wise linear function. Figure 4.1 shows two Reeb graphs of two functions whose values differ only on two vertices. The change in the Reeb graph can also be seen. Our purpose here is to perform such updates on the Reeb graph.

Summary of results. Recall that n_i is the number of i-simplices of the input complex and $m = n_0 + n_1 + n_2$ is the size of the complex K, which we assume to be 2-dimensional. We give an example showing that the time for processing an interchange event in general is $\Omega(l)$ where l is the size of the stars of involved vertices. It follows from this example that in general the worst-case running time for the update cannot be $o(m)$, which leads us naturally to express the running time as a function of l. Next, we reduce the dynamic Reeb graph problem to a particular retroactive graph

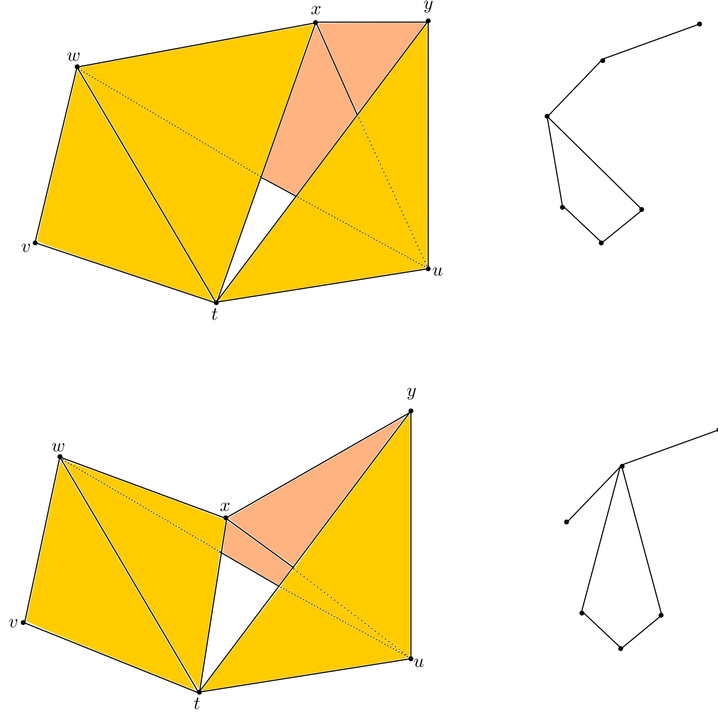

FIGURE 4.1: Interchange of two vertices: the simplicial complex with vertical height function on the left, and the Reeb graph on the right.

connectivity problem. We use existing algorithms and data structures to solve this problem, obtaining a time bound of $O(lt(n_1))$ for the update of the Reeb graph, where $t(n)$ is the worst-case time for any operation of a dynamic graph connectivity algorithm over n nodes. The space complexity is $O(n_0 s(n_1))$ where $s(n)$ is the space requirements of the graph connectivity algorithm.

This approach gives more efficient algorithms, whenever the dynamic connectivity problem can be solved more efficiently on the preimage graphs of the function. As important cases we mention the following.

1. When the domain is a 2-manifold. The preimages are circles or lines and dynamic graph connectivity on them can be solved efficiently. A special case is dynamic contour trees over a terrain. The running time in this case is $O(l \log m)$ per interchange.

2. In the offline setting, such as in Edelsbrunner et al. (2008b). Here, the dynamic graph connectivity on preimage graphs becomes offline connectivity, and, hence the running time is $O(l \log m)$. This is because, offline graph connectivity can be solved in $O(\log n)$ per operation, by methods of Parsa (2013) or Eppstein (1994).

3. When the preimage is embedded graph in a plane. This is the case when, for example, the complex itself is embedded in 3-space, and, the function is one of the coordinates. Since dynamic graph connectivity can be solved in this case in $O(\log n)$ per operation as in Eppstein et al. (1990), the update of the Reeb graph can be performed in $O(l \log n)$ time in the worst case. This case might be interesting in some applications.

4. If one is interested in probabilistic algorithms, then there are algorithms for the dynamic graph connectivity in polylogarithmic runtime by Kapron et al. (2013), which give corresponding probabilistic algorithms for the dynamic Reeb graph.

Related work. As discussed in previous chapters, the dynamic graph connectivity is an important problem and is well studied. Here we give an account of the algorithms solving this problem, see Section 4.2 for a formal definition. The best deterministic algorithm takes time $O(\sqrt{n})$ for update operations and $O(\log n)$ for queries Frederickson (1985); Eppstein et al. (1997), where n is the number of vertices or nodes of the graph. Henzinger and King (1999) first found data structures with polylogarithmic randomized runtime for all operations. The best amortized running times are achieved by the algorithm of Holm et al. (2001), with improvements in Wulff-Nilsen (2013); Thorup (2000). In this algorithm, updates can be performed in $O(\log^2 n / \log \log n)$ time amortized over n operations and queries can

be answered in $O(\log n/\log\log n)$. Recently, a probabilistic algorithmic with polylogarithmic running time has been announced in Kapron et al. (2013). We refer to Wulff-Nilsen (2013) for a fuller account of algorithms for this problem. Improving upper and lower bounds of the dynamic graph connectivity is a major challenge in graph algorithms.

In the offline case, where the sequence of operations is known in advance, Eppstein (1994) presented an algorithm for minimum spanning forests which works in $O(\log n)$ time per operation including weight change. The simple algorithm presented in Chapter 3 for constructing the Reeb graph can be used for maintaining *a* spanning forest for the offline connectivity problem in $O(\log n)$ per update. However, we do not know if such an approach can be used for maintaining a minimum spanning forest in the offline setting.

The retroactive data structures were first studied by Demaine et al. (2007). They gave general procedures for making data structures retroactive. The running times of algorithms obtained in this way depend on the length of the sequence. In special problems like union-find they obtain more efficient retroactive data structures, i.e., not depending on the length of the history of operations.

Perhaps the nearest work to the problem of this chapter is Edelsbrunner et al. (2008b), where authors consider changes of the Reeb graph when the function changes continuously in a time-interval and the values of the function are known in advance. This is different from our setting in that the change in function values are not dynamic. Moreover, they consider a very restricted case of complexes embedded in $\mathbb{R}^3$ since they need to compute the Jacobi curve of the function. The method employed there, namely, computing the Jacobi curve is not suitable for a totally dynamic setting. With methods of this chapter, we generalize the results in that paper as mentioned above.

4.2 A Retroactive Graph Connectivity Problem

In this section, we define our retroactive graph connectivity problem.

If G is a graph, we denote the node-set of G by $N(G)$ and the arc-set by $A(G)$. As usual, a *spanning forest* of a graph is a subgraph that consists of trees and includes all the nodes of the graph. With a fixed spanning forest, an arc is said to be a *tree arc* if it belongs to a tree. Let a be a tree arc. Removing a from its tree splits the tree into two connected components. An arc b of G is called a *replacement arc* of a if it connects these same two components together. Hence, a can be replaced with b to get a new spanning tree. Note that the relation "b is a replacement arc of a" depends on the tree. If arcs of the graph have weights, then a minimum spanning forest is a spanning forest whose total weight is a minimum.

Graph connectivity. The *dynamic graph connectivity* problem is defined as follows. A fixed set N of n nodes is given. A dynamic graph connectivity data structure has to support the following three operations for a graph with vertex set N.

1. `insert`(a): insert the arc a into the graph
2. `delete`(a): delete the arc a from the graph
3. `con`(v, w): return 1 if the node v is connected to the node w in the current graph, and return 0 otherwise[1].

We call the first two operations *update* operations and the last the *query* operation. Assume $c = c_1 c_2 \ldots c_m$ is a sequence of m update operations, that is, each c_i is either an insertion of an arc or a deletion of an arc. In applications of dynamic connectivity as a subroutine in a static algorithm, it might very well happen that

[1] This operation can also be written as `con`(e) for an arc $e \in V \times V$.

the sequence of all the operations is known in advance or can be computed easily. Then, one is in the *offline* setting where the sequence c is given.

We always assume that in the sequence c, c_1 is an operation on an empty graph. The set $[m] = \{1, \ldots, m\}$ is called *time* so that c_t is the operation at time t. The graph after the operation c_t is denoted by G_t and called the graph at time t, see Figure 4.2. Thus, the graph after c_1 is the graph at time 1 which has only one edge. Each operation advances the time by one.

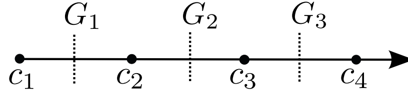

FIGURE 4.2: schematic view of the notation, the arrow shows increasing time.

Given the sequence c, we define a *retroactive graph connectivity data structure* to be a data structure supporting the following two operations.

1. $\texttt{con}(v_1, v_2, t)$: return 1 if the vertex v_1 is connected to vertex v_2 at time t, i.e. in G_t. Otherwise, return 0.

2. $\texttt{swap}(t)$: in the sequence c, exchange the position of c_t and c_{t+1}.

Note that since we are not extending the sequence c, regardless, we still cal this problem retroactive graph connectivity. The $\texttt{swap}(t)$ operation exchanges the place of two neighboring operations c_t and c_{t+1}, thus changing the sequence c into c'. We denote the new graphs and updated forests after the operation by a prime ($'$), for instance, G'_t is the new graph at time t, i.e. after the swap $c' = c_1 \ldots c_{t-1}c_{t+1}$. For simplicity we assume that in the sequence c each edge is inserted at most once. This conditions is satisfied in our application of the dynamic Reeb graphs.

4.2.1 Off-The-Shelf Algorithms for the Retroactive Graph Connectivity

In this section we review some algorithms that can be used to solve our retroactive connectivity problem.

The basic graph structure. In this approach, we store a graph data structure with all possible edges. To each edge, we associate the time interval the edge exists in the graph. Thus, the query $\texttt{con}(v, w, t)$ asks if vertices v and w are connected in the subgraph whose edges have t in their intervals. A tree traversal can be used to answer the query. A swap corresponds to changing an interval of an edge by moving one of the endpoints by one unit. This approach uses a minimal space, however, in general the time to answer a single query is linear in the number of edges.

Dynamic connectivity structures for snapshots, the trivial data structure. Let $S \subset [m]$ be a fixed set and call S the set of *snapshot*s. An data structure for retroactive graph connectivity can maintain a spanning forest for every snapshot. In other words, for each $s \in S$, the data structure has a spanning forest F_s for the graph which results from the operations $c_1, \ldots, c_s$.

The next algorithm uses a dynamic connectivity data structures for each snapshots, this is a general technique for making data structures retroactive, see Demaine et al. (2007). That is, using a version of the dynamic data structure at intervals. We can use $p \geqslant 1$ dynamic connectivity structures to maintain p of the graphs G_t at regular distances. We call this data structure the *trivial data structure* for the retroactive graph connectivity problem.

If the swap operation is requested for c_t and c_{t+1} and t is a snapshot time, then the swap can be done by a constant number of dynamic graph connectivity operations. In general, one has to move the snapshots at most $O(m/p)$ steps and then perform the swap and move the snapshots backwards. If $t(n)$ denotes the time complexity

of connectivity data structures then swap can be executed in $O((m/p)t(n))$ time in the worst-case. A query can also be answered in the same time by moving a nearest snapshot to the query position.

In an extreme case, we make each time a snapshot and therefore we will have the same time bounds for the retroactive swap as for dynamic connectivity. This will be $O(\sqrt{n})$ in deterministic worst-case time. Here n is the number of nodes, and $\sqrt{n}$ is the best worst-case running time known for the dynamic connectivity problem for all of the operations, see above in related work. Probabilistic algorithms also give their respective bounds. However, the polylogarithmic amortized data structures do not give the same amortized time per retroactive update. This is because update operations are performed on different structures. The space complexity of this approach is $O(ps(n))$ where $s(n)$ is the space complexity of a dynamic graph data structure used.

4.3 The Framework of an Algorithm for Dynamic Reeb Graphs

After introducing the retroactive graph connectivity problem and some off-the-shelf algorithms for solving it, in this section, we give a high-level description of an algorithm for dynamic Reeb graphs using a trivial data structure. The details of this reduction will be clarified in the rest of the chapter.

The input consists of a simplicial complex K and a function f on the vertex-set of K to real numbers. Our task is then to devise an algorithm to update the Reeb graph when an interchange event happens.

In the beginning, the Reeb graph of f will be built using the algorithm of Parsa (2013). During the execution of this algorithm, a sequence of `insert-delete` operations are performed on the preimage graph, let c denote this sequence of dynamic graph connectivity updates. We construct the data structure $\mathcal{D}$, which is a trivial data structure for retroactive graph connectivity, defined for the sequence c, and the

levelsets as snapshot graphs. We assume moreover that there are pointers from the components of the snapshots graphs, which $\mathcal{D}$ maintains, to the arcs of the Reeb graph. Here the preprocessing step finishes. Recall that there exist n_0 different levelset graphs, defined by values between those of the consecutive vertices.

Suppose f changes by an interchange event into f'. If we run the static algorithm again, it will generate a new sequence of operations c'. We will prove shortly in Proposition 1, that c and c' are related by at most l^2 `swap`s. These swaps are around the snapshot time that corresponds to the level-set graph that changes. We then perform these `swap`s on the data structure $\mathcal{D}$. "Performing swaps" means updating the dynamic graph connectivity structures of snapshots to correspond to the new sequence after swaps. Therefore, after performing the swaps on c, and transforming the sequence into c', we have the trivial data structure $\mathcal{D}$ of the level-set graphs for the updated function. It is then not difficult to update the Reeb graph by querying the dynamic connectivity structures in $\mathcal{D}$ and using pointers to the Reeb graph, this is done in Section 4.4.3.

4.4 Transforming an Interchange Event to Swaps

In this section, we show that an interchange event results in a collection of swaps on a certain sequence of graph connectivity operations. This sequence is the one that the sweep algorithm for construction of an static Reeb graph produces. Consequently, one can use a data structure for retroactive graph connectivity as defined above to fix the Reeb graph. The rest of this chapter explains the details of this procedure.

4.4.1 Notation and terminology

The Reeb graph depends only on the ordering of the vertices by their function values. If S denotes a fixed ordering of the vertices then we write $v <_S w$ to say that v precedes w in the ordering S. Often the ordering is understood and we remove the

subscript.

Let $r \in \mathbb{R}$ and consider the preimage or level-set $f^{-1}(r)$. Since K is 2-dimensional, each such preimage is a graph. We are interested in preimages of values r that are not values of vertices. Hence, the vertex set of a preimage graph can be thought to be a subset of the edges of K and the set of edges is a subset of triangles of K. There is no combinatorial change in the graph as long as r stays between the values of two consecutive vertices in the ordering S. Therefore, there can be at most n_0 different level-set graphs for any fixed ordering. For a vertex v, we denote the level set graph for the value $f(v) - \epsilon$ by $G(S, v)$, where ϵ is arbitrarily small but positive. For convenience, we assume that the set of vertices for each such graph is all the edge-set of K. This does not cause a problem since edges of K which do not intersect the preimage will be isolated vertices of the preimage graph and hence do not change the connectivity of other vertices. We call these *augmented preimage graph*s; See Fig. 4.3.

We have three different structures: the 2-dimensional simplicial complex, its Reeb graph, and, the levelset graphs. In the rest of this chapter, we use the following convention. We always call a 1-simplex of the complex an *edge*. A graph has *node*s connected by *arc*s. Each node of the Reeb graph is associated to a unique vertex of the complex, and vice versa. We denote the node corresponding to the vertex v by $\nu(v)$. Moreover, arcs of the Reeb graph will be distinguished from arcs of the preimage graphs by the fact that they are incident on nodes of the Reeb graph.

Interchange updates. The update procedure consists in exchanging the place of two consecutive vertices in S, that is, applying a transposition to the ordering S. We denote the interchange of v and w by $I = I(v, w)$ and assume $v <_S w$. If S' is the new ordering after the interchange, we write $S' = IS$. If $v <_S w$ we say v is *below* w and w is *above* v, similarly for the level-set graphs. With respect to a fixed ordering

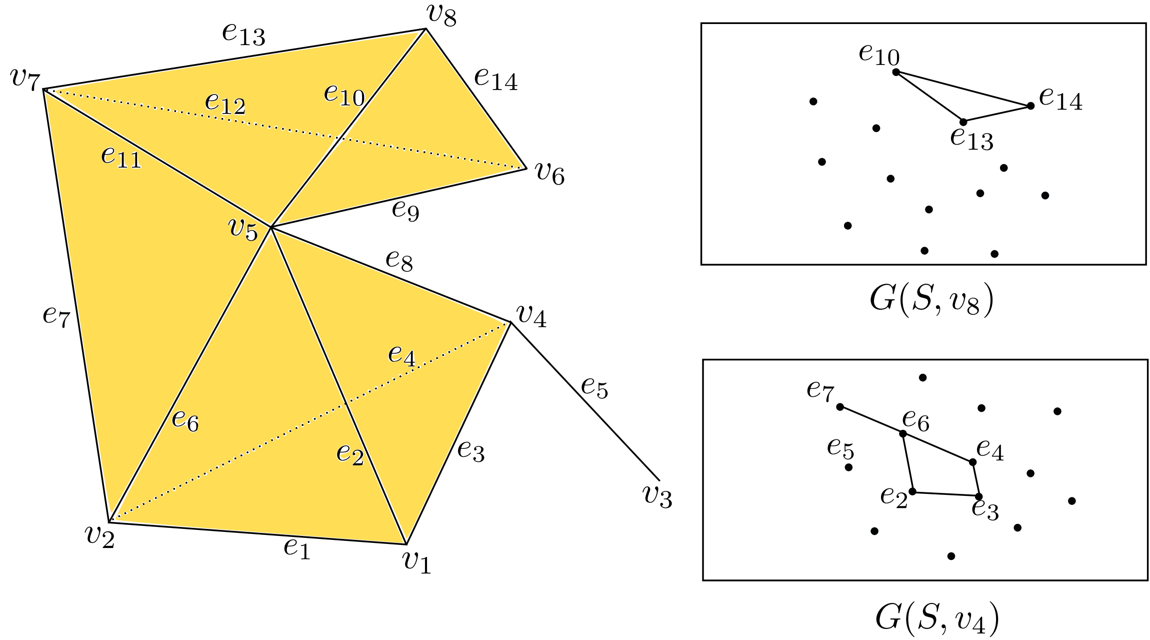

FIGURE 4.3: Left: a 2-dimensional simplicial complex consisting of two hollow tetrahedra, a triangle and an edge. Right: two level-set graphs.

S, an edge *starts at* its lower vertex and *ends at* its upper vertex. Similarly, an arc of the Reeb graph *starts at* its lower node and *ends at* its upper node. We also sometimes say that a component starts at or ends at a vertex v, by which we mean the corresponding arc of the Reeb graph starts or ends at the corresponding node $\nu(v)$.

4.4.2 Reducing an interchange to `swap` operations

In Chapter 3, we showed how constructing the Reeb graph of f and K boils down to maintaining connected components of a graph under a sequence of graph connectivity operations. Let this sequence be $c(f)$ and we know that it is defined by the ordering S, hence we can write $c(S)$. The aim of this section is to show that an interchange event changes the sequence $c(S)$ into $c(IS)$, and to compute the difference between these sequences. We summarize in the following proposition. Recall that `swap` exchanges

the place of two consecutive operations in c.

Proposition 1. *After an interchange event I, the sequence $c(IS)$ differs from $c(S)$ by swaps of a constant number of batches of operations, each of size $O(l)$, where l is the size of stars of involved vertices.*

This subsection is the proof of the above proposition. By swap of batches of operations we mean performing the swap on consecutive batches instead of single operations.

Level-sets before and after an interchange. Consider computing the Reeb graph for an ordering S. This construction can be done using the sweep algorithm that sweeps $\mathbb{R}$ from $-\infty$ to $+\infty$. At each time during the sweep, a spanning forest of the current preimage is maintained. Upon crossing the value of a vertex v, the preimage graph is changed by removing arcs that correspond to triangles ending at v and inserting arcs which correspond to triangles starting at v. For those triangles with v as the middle vertex, one arc is removed and one is inserted. See Chapter 3 above for the details of the static construction algorithm.

After these comments, we are ready to consider the effect of an interchange of two consecutive vertices in the level-set graphs. Let $I = I(v, w)$ with $v < w$ be the interchange event and let S be the current ordering and IS the updated ordering.

Lemma 2. *For each vertex $x \neq v, w$, $G(IS, x) = G(S, x)$ and $G(IS, w) = G(S, v)$. Moreover, $G(IS, v)$ is obtained from $G(S, w)$ by removing arcs starting at v and inserting arcs ending at v with respect to S, and, removing arcs ending at w and inserting those starting at w with respect to IS.*

Proof. It is easy to see that interchanging v and w does not change the level-set graphs below v or above w. To find the change in the level-set between v and w, we move the level-set of S down and then bring it up in the new ordering. In more

details, consider the level-set graph $G(S, w)$. We transform $G(S, w)$ into $G(IS, v)$ in two steps. First, we remove from $G(S, w)$ arcs that start at v and insert arcs that end at v with respect to S. These operations bring the level-set one level down. We will obtain $G(S, v) = G(IS, w)$. In the second step, we move up the level-set with respect to the new ordering, so we remove arcs ending at w and insert arcs starting at w with respect to IS. This gives us the graph $G(IS, v)$. □

As above, let $c(S)$ be the sequence of `insert` and `delete` operations performed during the construction of the initial Reeb graph for S. Consider the same algorithm for constructing the static Reeb graph applied to the new ordering of vertices (i.e. updated function). Let $x <_S v$ be the vertex before v in S. The sequence of insertions and deletions does not change before the start of processing of v; let this sequence be c_x. Let d_v be sequence of deletions of arcs ending at v in S, i_v the sequence of insertions of arcs starting at v in S, and similarly for d_w and i_w. Then $c = c_x d_v i_v | d_w i_w \ldots$. The bar "|" shows the place of the snapshot graph we are interested in. Observe that the ordering of operations between two level-set graphs is not important. If we denote the corresponding sequences with respect to the new ordering IS by d'_v, i'_v, d'_w, i'_w, then $c' = c_x d'_w i'_w | d'_v i'_v \ldots$. Now the difference between d_v and d'_v corresponds to arcs that end at v in S' but not in S, so they start at w in S' (or end at w in S) and therefore correspond to the triangles in the star of the edge vw. Similarly, always the primed sequences and original ones, with the same subscript, differ by arcs corresponding to triangles in the star of the edge vw. The arcs of these triangles will always be in the graph $G(S, v)$ or $G(IS, w)$ and they need not be moved around. Let i and d be insertions and deletions of these arcs respectively. Remove the operations of i and d from the sequences $d_\star$ and $i_\star$. Then $c = c_x d_v i_v i | d d_w i_w \ldots$ and $c' = c_x d_w i_w i | d d_v i_v$. Therefore, the sequence c is transformed into c' by moving four batches of insertion and deletion operations. This finishes the proof of Proposition 1. □

4.4.3 Updating the Reeb graph

After we have updated the level-set connectivity structure $\mathcal{D}$, we have to update the current Reeb graph. For this purpose we have to store pointers from each level set component to the corresponding arc of the Reeb graph. With the help of these pointers, the following procedure updates the Reeb graph.

Recall that $\nu(v)$ denotes the node of the Reeb that corresponds to the vertex v. At the very beginning, we disconnect all the arcs incident on the node $\nu(v)$ or the node $\nu(w)$ such that no arc is incident on these two nodes and we have some arcs which are dangling from below or from above, we remove the arcs connecting $\nu(v)$ and $\nu(w)$, if any, see Figure 4.4. Next, we query all the edges ending at w with respect to IS in $G(IS, w) = G(S, v)$. This results in a set of components which is in bijection with arcs ending at $\nu(w)$ in the new Reeb graph. We use the pointers to the Reeb graph structure to find those arcs which now should end at $\nu(w)$. We make $\nu(w)$ the new upper endpoint of these arcs, whose upper endpoint was dangling. In the next step, we query all the edges starting from w in $G(IS, v)$ to find their components. Then, we will have a set of components associated with arcs which will start at $\nu(w)$ in the new Reeb graph. Let this set of components be C_d. For each component in C_d we create a dangling (from above) arc whose lower endpoint is $\nu(w)$.

In the next step, we query edges ending at v (with respect to IS) in $G(IS, v)$. This gives us a set of components which will end at $\nu(v)$, call this set C_u. These correspond to arcs with dangling upper endpoint, either those recently generated or those that were disconnected. It is easy to find out which of these components are in C_d, hence are arcs from $\nu(w)$ to $\nu(v)$. This subset of C_d must end at $\nu(v)$. From Lemma 2 it follows that all of the other arcs not in C_d and with a dangling upper endpoint must end at $\nu(v)$. Therefore, the arcs with dangling upper endpoint at this

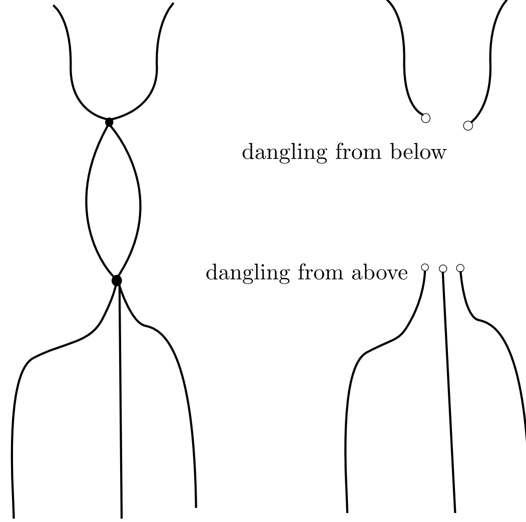

FIGURE 4.4: Removing two nodes creates dangling arcs.

point are in bijection with $C_u - C_d$. We update the graph structure by making $\nu(v)$ the upper endpoint of these arcs. The arcs with dangling lower endpoint must start at $\nu(v)$ or $\nu(w)$. Symmetrically to the above argument, we find those which should start at $\nu(v)$ and what remains should start at $\nu(w)$ and hence is in bijection with still dangling arcs from C_d, which is $C_d - C_u$. This finishes the update procedure for the Reeb graph. Observe that, disregarding the time spent for updating the connectivity data structure, the above procedure performs $O(\ell)$ queries to various level-set graphs and $O(\ell)$ changes to the Reeb graph data structure.

Updating the pointers. Note that since only the level-set $G(S, w)$ is affected by the swap, the pointers from components of level-sets to arcs of the Reeb graph remain valid other than for this level-set. Moreover, there are at most $O(\ell)$ number of components of $G(S, w)$ which are affected by the above procedure, namely those starting or ending at w or v. The components that start at $\nu(w)$ and end at $\nu(v)$ are easily assigned their (new) arcs, these correspond to $C_d \cap C_u$. What remains to be done is assigning to each component in $C_u - C_d$ a pointer to the corresponding

arc and the symmetric operation. Let $c \in C_u - C_d$ and e an edge ending at v whose query resulted in component c. e does not start at w and hence exists in $G(IS, w)$. We query e in $G(IS, w)$ which gives us a component whose arc is the arc we seek. Therefore, updating the pointers from components of the preimage graphs to arcs of the Reeb graph takes also $O(\ell)$ queries to the level-set connectivity structure. And hence can be done by the time bounded by the update of the data structure $\mathcal{D}$. Therefore, we have updated the Reeb graph in time which is dominated by the update of $\mathcal{D}$.

From this discussion the following.

Theorem 2. *Let $u(n)$ denote the running time of a data structure for retroactive graph connectivity, for all the operations. Then there exists an algorithm for updating the Reeb graph after an interchange event in $O(l^2 u(n))$ time, where l is the size of star of the vertices involved in the interchange.*

Moreover, if one uses the trivial algorithm, then a constant number of swaps of batches of operation of $O(l)$ each can be performed by $O(l)$ operation on the dynamic connectivity structures at snapshots. Hence we obtain

Theorem 3. *Let $t(n)$ denote the running time of the trivial data structure for retroactive graph connectivity, for all the operations. Then there exists an algorithm for updating the Reeb graph after an interchange event in $O(lt(n))$ time, where l is the size of star of the vertices involved in the interchange.*

4.5 Discussion

Dependency on the size of stars. We give an example in which an interchange introduces $\Omega(l)$ combinatorial changes in the Reeb graph. This proves that $\Theta(l)$ time is necessary in this case, where l is the size of the star of the involved vertices. Consider the complex in Figure 4.5, a. This is an empty box but with two triangles removed.

To improve visibility, the front facing square is not shown. The ordering of the vertices is as usual implied by their vertical position in the figure. In b the Reeb graph of the complex is drawn. Now if the two vertices v_9 and v_{10} exchange their place in the ordering, all the level-sets become connected and the Reeb graph will be a line. Now take l copies of the complex and identify the vertices that correspond to v_9 together and similarly identify all those that correspond to v_{10} to a vertex. Define the function values on each copy similar to the one drawn in the picture. Then the Reeb graph consists of l copies of b attached together at nodes corresponding to v_9 and v_{10}. In c the Reeb graph for $l = 3$ is drawn. Exchanging the values of v_9 and v_{10} removes a loop from each copy, the result then has l fewer loops. The drawing in d shows the Reeb graph after the interchange for $l = 3$.

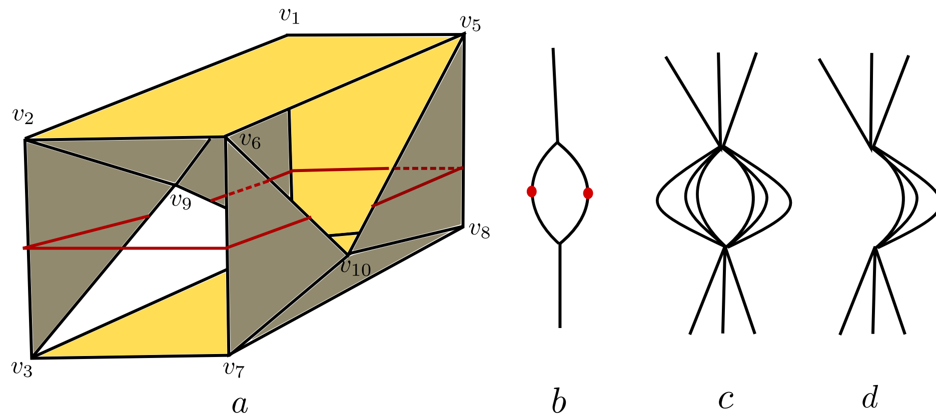

FIGURE 4.5: Large number of loops are created/removed by a single interchange.

The offline (time-varying) interchange events. Edelsbrunner et al. (2008b) have considered the dynamic Reeb graph problem but in a more restricted setting where a time-varying function is given on a triangulation of the 3-space. Thus the evolution of the function is known beforehand and the setting is not totally dynamic. As a result the same techniques of offline connectivity can be used to maintain each level-

set. Namely, one has to compute for each snapshot all the operations that will be performed on that snapshot graph from beginning to the end. Once the sequence of operations for a snapshot graph is computed, in $O(\log m)$ time per operation, spanning forests for levelsets can be maintained. Thus, the running time would be $O(lE \log m)$ in total, where E is the number of interchange events produced by the time-varying function and l is an upper bound on size of the stars of vertices in the complex K. If we assume l is constant, which is reasonable in the setting of Edelsbrunner et al. (2008b), then the running time would be $O(E \log m)$ for processing E events. However, in the beginning we have to build n_0 trees (in the dynamic tree data structure), one for each preimage graph. We denote the cost of this by k, which depends on the sum of the number of vertices of all the preimages and hence $k = \tilde{O}(n_0 n_1)$. The run-time would be $O(k + E \log m)$ in total to compute all the changes of the Reeb graph. The running times of Edelsbrunner et al. (2008b) is $O(m + En)$ where n is the size of a level-set. When $m = \Omega(k)$, or when E is large, the running time would be $O(m + E \log m)$ which answers the following question asked in Edelsbrunner et al. (2008b) in the affirmative, under the above conditions. They asked if each interchange of critical vertices can be processed in logarithmic time. For the complex in Figure 4.5, the running times will depend on l as shown in the above example and hence $O(m + E \log m)$ is not achievable.

5

Loops in the Reeb Graph and Computing Homology of (Embedded) Simplicial Complexes

In this chapter we present an application of Reeb graphs and their algorithms to the problem of the complexity of computing Betti numbers for an arbitrary simplicial complex. The motivation for this work is the fact that in some cases, Betti numbers (or a basis for homology groups) can be computed more efficiently than reducing the boundary matrices. One of these cases is when an embedding of a complex is given in a 3-dimensional Euclidean space; see Delfinado and Edelsbrunner (1995). In this case, a 2-dimensional homology generator corresponds to a connected component of the complementary space. Therefore, the second and the zeroth Betti numbers can be computed by just finding connected components of a complex, which takes linear time. The first Betti number can be computed using the Euler relation. The question arises: does embedding in the Euclidean 4-space provide us with any method of computing the Betti numbers more efficiently? We answer this question in the negative way. We show that computing the Betti numbers of embedded complexes in $\mathbb{R}^4$ is "harder" than computing the rank of a matrix. And as was shown in the Back-

ground chapter, Betti numbers can be computed by computing ranks of matrices. Therefore, we have reductions of problems both ways. For example, computing the Betti numbers over $\mathbb{F}_2$ of a complex embedded in $\mathbb{R}^4$ (and also given its complement) is equivalent to computing the rank of a sparse m-by-m matrix of $O(m)$ elements over $\mathbb{F}_2$. More detailed statements can be found in the main body of the chapter.

Computational complexity. We use the term *computational complexity* to refer to the measure of complexity of computing. We reduce problems to each other using worst-case linear-time reductions on a usual RAM model. However, our results are also valid when the underlying model of computation allows these reductions to be done in linear complexity. This is true if we consider for example randomized algorithms with average-case running time as complexity, etc. At times, we use *run-time* to mean computational complexity in this sense. Since we are reducing problems to each other, as long as the reduction is linear in the measure of complexity one considers, then the reductions will be relevant in that measure of complexity.

Complexity of rank computation. It is known that the rank of an arbitrary matrix can be computed in matrix multiplication time; see Bunch and Hopcroft (1974). The best asymptotic run-time for multiplying two matrices is a major open problem in algebraic complexity theory. Let ω be a number such that a worst-case optimal algorithm that multiplies two n-by-n matrices runs in $O(n^{\omega+\varepsilon})$ time, for each $\varepsilon > 0$. The number ω is called the *exponent* of matrix multiplication. The currently best upper bound is $\omega < 2.3727$; see Coppersmith and Winograd (1990); Williams (2012). However, it is not known whether the sparsity of matrices can help in computing the rank. While there exists a theoretical algorithm for multiplying two n-by-n matrices each with $O(n)$ non-zero entries in $O(n^{2+\varepsilon})$ time, for every $\varepsilon > 0$; see Yuster and Zwick (2005), it is not known whether this helps in rank computation or Gaussian

elimination for sparse matrices.

It is worth mentioning that there is a randomized algorithm that computes the rank of a matrix in a time that is roughly proportional to n^2. Specifically, Wiedemann's Monte Carlo algorithm computes the rank of an n-by-n matrix with m non-zero entries in $O(n^{2+\varepsilon} + nm)$ time; see Wiedemann (1986). Moreover, there exists a Las Vegas algorithm whose expected run-time for matrices with $O(n)$ non-zero entries is $O(n^{2.28})$; see Eberly et al. (2007) but also Yuster (2008).

5.1 Reducing Rank Computation to Betti Numbers of Simplicial Complexes

In this section, we state and prove our main theorems. They consist of reductions from computing the rank of a matrix to computing the Betti numbers of a complex. For simplicity, we consider only square matrices, while the generalization to rectangular matrices is straightforward. Any n-by-n matrix, M, determines a linear map from $\mathbb{R}^n$ to $\mathbb{R}^n$. The kernel of this map is the *null-space*, and the dimension of the kernel is the *nullity* of the matrix, denoted $\mathsf{null}\, M$. Since the rank of the matrix is the dimension of the image of this map, we have $\mathsf{null}\, M = n - \mathsf{rank}\, M$. The nullity is therefore the maximum number of independent solutions to the equations $Mx = 0$.

Statements.

Assume we have a representation of the input matrix that gives amortized constant time access to its non-zero entries and moreover with no zero row or column.

Theorem 4. *Let M be an n-by-n 0-1 matrix with m non-zero entries. In time $O(m)$, it is possible to build a 2-dimensional simplicial complex, $K = K(M)$, of size $O(m)$ and a piecewise linear function $f : |K| \to \mathbb{R}$, such that $\beta_2(K) = \beta_1^{\text{hor}}(K, f) = \mathsf{null}\, M$. Moreover, the complex K embeds in $\mathbb{R}^4$ by a linear time procedure.*

For the definition of horizontal and vertical homology groups and Betti numbers see the Backgrounds chapter. Theorem 4 implies

$$b(m) \quad = \quad \Omega(r(n,m)), \tag{5.1}$$

where $b(m)$ is the complexity of computing the Betti numbers of a 2-dimensional simplicial complex of size m, and $r(n,m)$ is the computational complexity of computing the rank of an n-by-n 0-1 matrix with m non-zero entries. The constructed complex, $K(M)$, embeds in $\mathbb{R}^4$.

Recall that for a complex of size m, the Betti numbers can be computed by a constant number of rank computations for matrices which are at most m-by-m and have $O(m)$ non-zero entries. Therefore, $b(m) = O(r(m,m))$. The theorem shows $b(m) = \Omega(r(m,m))$ which gives $b(m) = \Theta(r(m,m))$.

Corollary 1. *Computing Betti numbers of a* 2*-dimensional complex embedded in* $\mathbb{R}^4$ *is equivalent with a linear time reduction to computing the rank of sparse m-by-m matrix of $O(m)$ non-zeroes over* $\mathbb{F}_2$*.*

Our second construction produces a complex of $O(n)$ vertices with the cost of making the complex denser and which is not guaranteed to be embeddable.

Again, we assume access in constant amortized time to the non-zero entries of the input matrix.

Theorem 5. *Let M be an n-by-n 0-1 matrix with m non-zero entries. In time $O(m)$, it is possible to build a* 2*-dimensional simplicial complex, $L = L(M)$, with $O(n)$ vertices and a piecewise linear function $g : |L| \to \mathbb{R}$, such that $\beta_2(L) = \beta_1^{\mathrm{hor}}(L) = \mathsf{null}\, M$.*

Theorem 5 proves the second main result,

$$B(n) \quad = \quad \Omega(r(n)), \tag{5.2}$$

where $B(n)$ is the computational complexity of computing the Betti numbers of a 2-dimensional simplicial complex with n vertices, and $r(n)$ is the computational complexity of computing the rank of an n-by-n 0-1 matrix. The complex, $L(M)$, does not necessarily embed in $\mathbb{R}^4$.

The difference between the two theorems is that in the second theorem we consider the complexity of computing Betti numbers not as a function of the size of the complex, rather, as a function of number of vertices. It turns out that this complexity is related to the complexity of computing rank for matrices, as in Theorem 5 above, when the latter complexity is described as number of rows (or columns) of a square matrix.

The first reduction

We use the matrix M in Theorem 4 to construct a simplicial complex, K, and a piecewise linear function $f : |K| \to \mathbb{R}$, such that the rank of the first horizontal homology group is isomorphic to the null-space of M and therefore its rank equals the nullity of M. We interpret M as the matrix of a system of linear equations. The null-space is the space of solutions to the equations $\sum_{\ell=1}^{n} M(k,\ell)x_\ell = 0$, for $1 \leqslant k \leqslant n$.

Construction. We start by constructing a cycle made out of a constant number of edges for each column. We refer to the cycle corresponding to column ℓ by x_ℓ. Placing these cycles disjointly in a 2-dimensional plane Π in $\mathbb{R}^4$, we set the function values of their vertices to 0. For each row, we add a sphere with as many holes as there are non-zero entries, gluing the boundaries of the holes to the cycles corresponding to the non-zero entries with a simple map of degree 1; see Figure 5.1. Letting p be the number of holes, we call this surface a *p-cap*, since it is obtained by removing p disks from a sphere. It generalizes a cap, which is a sphere with a single disk removed. It is easy to construct a triangulation of the p-cap that consists of $O(p)$

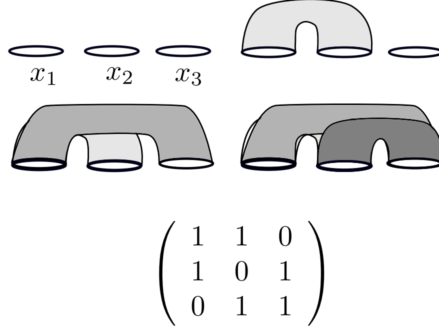

FIGURE 5.1: Starting with three cycles, the simplicial complex that corresponds to the 3-by-3 matrix is constructed by adding one cap at a time. Note that all the caps are disjoint in 4-space.

vertices, edges, and triangles and embeds in a 3-dimensional plane containing Π. The function values of the vertices in the triangulation that are glued to the initial cycles are 0, and those of all other vertices are chosen to be strictly larger than 0 and smaller than 1. As suggested in Figure 5.1, we can think of the function values as the heights of the vertices above Π. Choosing a pencil of 3-dimensional planes, all passing through Π, we get n caps that are pairwise disjoint except for possibly shared cycles in Π. It follows that the k-th cap is a 2-chain that introduces the relation $\sum_{\ell=1}^{n} M(k,\ell)[x_\ell] = 0$ on the classes of cycles, for $1 \leqslant k \leqslant n$. After adding the n caps, we obtain a simplicial complex, which we call $K = K(M)$.

Analysis. At the beginning, after adding the cycles and before adding any caps, every x_ℓ represents a 1-dimensional horizontal homology class. The class represented by x_ℓ remains horizontal throughout the construction, but it can of course become zero. Indeed, the effect of the cap constructed is to render the corresponding sum of classes to be the zero class. We show that the cap does not affect the first horizontal homology group in any other way.

Lemma 3. *For every $1 \leqslant k \leqslant n$, the addition of the k-th cap does not create any new horizontal (1-dimensional) homology class, and it kills at most one class, namely*

$\sum_{\ell=1}^{n} M(k,\ell)[x_\ell]$.

PROOF. Let p be the number of non-zero entries in the k-th row of M, and recall that the corresponding p-cap is a sphere with p holes. To show that the addition of the p-cap does not add any new horizontal classes, we construct the p-cap from the p circles that bound its holes as follows. Connect the p circles with $p-1$ arcs whose interior points have function values strictly larger than 0. Because each arc covers an interval of function values that has a non-empty interior, these arcs do not change the horizontal homology. We can form a closed curve that traverses each circle once and each arc twice, once in each direction. The p-cap can now be completed by adding a disk whose boundary is glued to the closed curve. Finally, we note that the boundary cycle of the added disk is homologous to $\sum_{\ell=1}^{n} M(k,\ell)x_\ell$. This is the only relation implied by the disk, which finishes the proof.□

Lemma 3 implies that the first horizontal homology group of K is generated by $[x_1],[x_2],\ldots,[x_n]$ subject to $Mx = 0$, where $x = ([x_1],[x_2],\ldots,[x_n])$. The number of independent generators is therefore $n - \mathsf{rank}\, M = \mathsf{null}\, M$. The cap added for row k creates a new 2-cycle iff its boundary can be written as a linear combination of preceding caps. It follows that the second Betti number is equal to the number of rows minus the rank of M, which is again the nullity of M.

Complexity. To finish the proof of Theorem 4, we recall that M provides access in constant amortized time to its non-zero entries. It follows that the above construction can be done in $O(m)$ time. This is because we only need to access non-zeros of each row.

We argue as follows that the complex can be embedded in $\mathbb{R}^4$ by a linear time procedure. The set of circles can easily be embedded in the plane Π. We show how to attach a cap in a 3-plane. Different 3-planes are transformed together by a simple transformation. Then consider a 3-plane passing through the 2-plane Π. We embed

a 2-sphere away from Π. We want to turn the 2-sphere into a m_i-cap an attach the boundary to the circles. This can be done easily by taking m_i disjoint points from the 2-sphere and connect them to the centers of the sphere by disjoint curves. By thickening these curves we obtain tubes connected to the 2-sphere. We then expand the other end of the tube to be glued to the circle c_i. Therefore, each cap can be embedded in linear time with respect to its size. This completes the proof of Theorem 4.

As explained above, the complex can be embedded in $\mathbb{R}^4$ in linear time and its complement space can also be constructed during the embedding, also in linear time. Here, we also note that the first vertical Betti number can be computed as follows. We observe that the Reeb graph of $K(M)$ is homotopy equivalent to a bipartite graph whose nodes are the rows and the columns of M, with an arc from a row to a column iff they intersect in a non-zero entry of M. We have $2n$ nodes and m arcs, and we can compute ℓ, the number of connected components in $O(m)$ time. The number of independent loops in the Reeb graph is $m - 2n + \ell$, which is also the first vertical Betti number of K.

Remark. The assumption of access in constant amortized time to the non-zero entries of the matrix is not essential. Without it, we can construct the complex K in $O(n^2 + m)$ time, which implies a slightly weaker claim that suffices for our purposes.

Remark. If we are not interested in the difficulty of computing the horizontal and vertical Betti numbers of a function defined on a complex and rather only Betti numbers it is possible to simplify the above proof by not constructing the function over the complex at each step. When attaching the caps to the x_i, whether the k-th cap creates a new 2-cycle or not depends only on whether the previous caps have killed that cycle or not. Therefore, it does not depend on the new 1-classes that adding these caps might create. These classes are vertical classes in the above proof.

The second reduction.

Similar to Theorem 4, we prove Theorem 5 by interpreting the matrix M as a system of linear equations from which we construct a simplicial complex. The main difference is that we now allow ourselves only $O(n)$ vertices, which limits the possibilities. We still manage to construct a simplicial complex, $L = L(M)$, and a simplex-wise linear function $g : |L| \to \mathbb{R}$ such that the first horizontal Betti number of L with respect to g is the nullity of M. However, L will not necessarily embed in $\mathbb{R}^4$.

Construction. We start by creating n square cycles, denoted as y_ℓ, one for each column of M. We assign the same function value, g_ℓ to all four vertices of y_ℓ, making sure that different square cycles receive different function values. For each row k of M, we introduce the relation $\sum_{\ell=1}^{n} M(k,\ell) y_\ell = 0$ by adding some edges and triangles to the complex. We cannot afford adding a cap, as in the proof of Theorem 4, because this would require an additional number of vertices proportional to the number of non-zero entries in row k. Instead, we connect the square cycles by pairs of triangles, as illustrated in Figure 5.2. We connect all the squares corresponding to non-zero

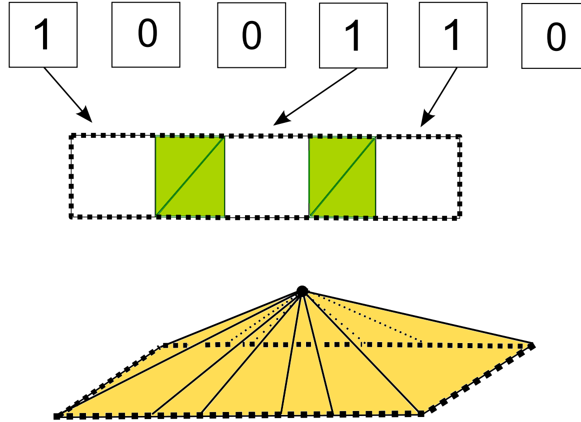

FIGURE 5.2: Three of the six square cycles are connected by two pairs of triangles, and a cone is erected over the cycle that surrounds the three square cycles and the connecting triangles.

entries together in a row. In particular, if non-zero entries in row k belong to columns

$\ell_1 < \ell_2 \ldots < \ell_q$, then we connect the right edge of y_{ℓ_j} with the left edge of $y_{\ell_{j+1}}$, for $j = 1, 2, \ldots, q-1$. We add the connecting pair of triangles unless the two square cycles are already so connected.

Finally, we consider the cycle, c_k, that goes around the sequence of connected squares (the ones corresponding to non-zero entries in row k), and we add a cone over c_k. Assuming row k has p non-zero entries, c_k has $4p$ vertices and $4p$ edges. The cone over c_k thus consists of $4p$ edges and $4p$ triangles, and it adds only one new vertex to the complex. We choose the function value of this vertex different from the function values of all previous vertices.

Analysis. We argue that L has the desired homology groups. Indeed, adding a pair of connecting triangles does not alter the first homology group. To see this, we add the three edges and two triangles in sequence. The first edge does not affect the first horizontal homology group for the simple reason that it connects vertices with different function values, say $g_\ell < g_{\ell'}$. The function values of the points on the edge thus cover the interval $[g_\ell, g_{\ell'}]$, which has non-empty interior. We now add the other two edges and the two triangles using two anti-collapses, which preserve the homotopy type of the complex, and therefore also its homology groups, and also the splitting into horizontal and vertical.

Adding the cone over c_k is like adding a single disk to the complex. Since c_k does not intersect itself, this disk does not affect the homology other than by introducing the relation $[c_k] = 0$. We argue that c_k is homologous to $d_k = \sum_{\ell=1}^{n} M(k, \ell) y_\ell$. In other words, $c_k + d_k$ is a boundary. But this is clear because $c_k + d_k$ is the boundary of the sum of triangles connecting square cycles of contiguous non-zero entries in row k.

In summary, the first horizontal homology group of the final complex L is generated by the $[y_\ell]$, for $\ell = 1, 2, \ldots, n$, subject to the relation $My = 0$, where

$y = ([y_1], [y_2], \ldots, [y_n])$. The first horizontal Betti number is therefore $\beta_1^{\text{hor}}(L) = n - \mathsf{rank}\, M = \mathsf{null}\, M$. In the case in which c_k is null-homologous before the cone is added, the addition of the cone creates a new 2-cycle. Hence, we also have $\beta_2(L) = \mathsf{null}\, M$, as required.

Complexity. The number of vertices in the complex $L = L(M)$ is $5n$, namely 4 for each column and 1 for each row. The number of edges is at most $4n + 7m$, and the number of triangles is less than $6m$. All these simplices can be constructed in $O(m)$ time, assuming again a representation that permits access to the non-zero entries of M in constant amortized time. This completes the proof of Theorem 5.

The remark given after the proof of Theorem 4 also applies here.

5.2 Extensions

In this section, we consider three extensions of our results: from simplicial to more general complexes, from $\mathbb{F}_2$ to more general finite fields and integers, and from sparse matrices to matrices that have at most three non-zero entries per column.

5.2.1 More general complexes.

Given a matrix, M, with integer entries, there is a standard construction of a 2-dimensional CW complex whose boundary matrix is M; see for example (Hatcher, 2001, Corollary 1.28). To construct this complex, we start with a wedge of n oriented circles pinned together at a common point, which we denote as ω. We order the circles and write z_ℓ for the ℓ-th circle in this ordering. Each circle corresponds to a column of the matrix. Consider any loop that starts at ω and ends at ω. We can write this loop as $\sum_\ell a_\ell z_\ell$, in which a_ℓ is the number of times the path traverses z_ℓ (using the sign distinguishing traversals with or against the orientation of the circle). To complete the construction, it suffices to attach a disk by gluing its boundary to the

loop corresponding to the row, for each row. It is not difficult to see that the first homology group of the final complex is generated by the circles z_ℓ subject to the relations given by the equations $Mz = 0$, in which $z = (z_1, z_2, \ldots, z_n)$ is a column vector. In particular, the $\mathbb{F}_p$-Betti numbers of the complex give the nullity and hence the rank of the matrix M over $\mathbb{F}_p$, for p prime.

The above construction is related to computing the homology of a simplicial complex by simplification that merges simplices into larger and more complicated cells. The number of cells is reduced but at the cost of making the boundary matrix denser, albeit smaller in size. Eventually, we compute the rank of a smaller but denser matrix. This approach to homology computation is justified as long as the complexity of computing the rank of a sparse matrix is not known to be less that of computing the rank of a dense matrix.

5.2.2 Finite fields

Our two theorems and the implied complexity bounds for computing horizontal Betti numbers extend from $\mathbb{F}_2$ to more general finite fields. Assume we are given a matrix, M, with elements in $\mathbb{F}_q$; that is: integer numbers modulo q, with q a prime number. Let $m_k = \sum_\ell M(k, \ell)$ be the sum of entries in row k, where we take the sum in $\mathbb{Z}$ and not modulo q. We construct the simplicial complex, $K = K(M)$, as before but with an m_k-cap added for the k-th row such that the number of legs that are attached to the cycle x_ℓ is $M(k, \ell)$. The same proof then shows that after attaching the m_ℓ-caps, we will have $\sum_\ell M(k, \ell)[x_\ell] = 0$ in homology with $\mathbb{F}_q$ coefficients. It follows that the first horizontal $\mathbb{F}_q$-Betti number of K is the nullity of M, and similarly for the second Betti number of K. That is, after adding an m_k-cap, the second Betti number will increase if and only if the boundary of the cap is already null-homologous with coefficients $\mathbb{F}_q$. In other words, we have a statement for $\mathbb{F}_q$ like that given in Theorem 4 for $\mathbb{F}_2$. It follows that the complexity of computing the Betti numbers over $\mathbb{F}_q$ is at

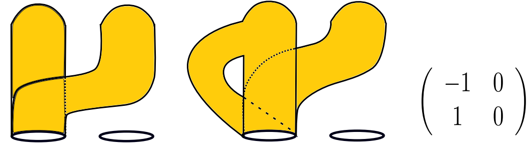

FIGURE 5.3: Left: the complex constructed for the matrix on the right assuming the entries are in $\mathbb{F}_2$. Middle: the complex constructed for the same matrix but assuming the entries are in $\mathbb{F}_3$.

least that of computing the rank of an n-by-n matrix with entries in $\mathbb{F}_q$ whose sum of entries is m. In the next section we show how to construct a complex with size exactly the bit-complexity of the given matrix.

We illustrate the construction in Figure 5.3, which shows two complexes for the same matrix, the complex on the left for the 3-element field, and the complex in the middle for the 2-element field. For $\mathbb{F}_2$, we have $-1 = 1$ and a complex that consists of a 2-sphere and a circle. Its first horizontal and its second Betti numbers equal the nullity of the matrix, as claimed. For $\mathbb{F}_3$, we have $-1 = 2$ and a complex that consists of a Klein bottle with an additional disk attached and an isolated cycle. As before, the first horizontal and the second Betti numbers

5.2.3 Matrix sparsification

As in Yuster (2008) one can define sparsification of an n-by-n matrix A with $O(m)$ non-zero entries to be constructing a sparse matrix B of size m-by-m such that one can compute the rank or determinant or other quantities of A from those of B. If the reduction and computation of rank for A from rank of B can be done efficiently say in $\tilde{O}(m)$ time, then the general matrix problem has been reduced to a sparse problem. In Yuster (2008) the sparse matrix has the additional structure that each row and column contains at most three non-zero entries. Here we show that the above constructions can be used for such a sparsification of a 0-1 matrix.

Observe that we can view our construction of a two dimensional simplicial com-

plex for a matrix M as a process that from M generates two matrices, that is, boundary matrices of K, such that rank of M can be computed from the rank of these matrices. Each column of these boundary matrices has exactly two or exactly three 1s corresponding to the boundary of an edge or a triangle. Consider the second boundary matrix D_2. The rows of D_2 correspond to edges of the complex. Most of the edges have exactly two incident triangles other than possibly edges of the cycles x_ℓ. If an edge belongs to a cycle x_ℓ, the number of incident triangles equals the number of rows in M with a 1 in position ℓ.

If we apply the construction again to D_1 and D_2 we will obtain four matrices, D_{11}, D_{12}, D_{21} and D_{22}. Consider $K(D_1)$. In D_1 each column has exactly two 1s, it follows that $K(D_1)$ is a 2-manifold without boundary of size $O(m)$. This surface can be built to be orientable. Hence its Betti numbers can be determined in $O(m)$ time using its Euler characteristic number. This corresponds to the fact that the rank of D_1 can be computed easily since each column has only two non-zeros.

On the other hand, $K(D_2)$ is a complex which may not be a manifold. Therefore, the difficulty is in finding the Betti numbers of this complex. This we can do by computing the rank of the boundary matrices. Rank of D_{21} can again be computed efficiently. So we turn to D_{22} whose columns and rows have at most three non-zero entries. After computing the rank of D_{22} and an additional $O(m)$ work we will obtain the rank of the original matrix M. Therefore, we have a sparsification of 0-1 matrices which is a special case of Theorem 1.1 in Yuster (2008):

Theorem 6. Yuster (2008) *Let A be a square* 0-1 *matrix of order n with m non-zero entries. Another* 0-1 *matrix B of order $O(m)$ can be constructed in $O(m)$ time so that rank of A can be computed from rank of B with $O(m)$ computation. Moreover, each row and column of B contains at most three non-zero entries.*

Remark. We note that in the complex $K(D_2)$ exactly three triangles are incident

on an edge which belongs to the cycles x_ℓ and exactly two triangles are incident on other edges. Informally, the complex $K(D_2)$ deviates from being a manifold by a little but still computing its Betti numbers is equivalent to computing the rank of any sparse matrix of the same size.

5.3 Integer Coefficients and Invariant Factors

In this section, we extend the intuitive construction for matrices over $\mathbb{F}_2$ and prime fields to the general case of matrices over integers. Thus assume M is a given integer matrix. M has a diagonal form called Smith normal form. That is, there exist invertible integer matrices P and Q such that

$$D = QMP^{-1}$$

with D a diagonal matrix. Let $d_1, \ldots, d_r$ be the diagonal entries of the reduced matrix D and assume $d_i \geqslant d_{i+1}$. We consider the cokernel of the map defined by M or D

$$\operatorname{coker}(M) = \mathbb{Z}^n/\operatorname{im}(M) \cong \mathbb{Z}^n/\operatorname{im}(D) \cong \mathbb{Z}^n/\bigoplus_i d_i\mathbb{Z} \cong \bigoplus_i \mathbb{Z}/d_i\mathbb{Z}.$$

Our goal in this section is to build a simplicial complex, embeddable in $\mathbb{R}^4$ and of the same complexity as M, whose first horizontal homology group is isomorphic with the above cokernel. It then follows that computing the diagonal elements of D reduces to computing the invariant factors d_i of the first horizontal homology group. Note that in the case of field coefficients, all of the d_i are one and the cokernel is determined up to isomorphism by its dimension, which since the matrix is square equals nullity of M.

The construction can be done totally analogous to that of $\mathbb{F}_2$ and $\mathbb{F}_q$, however, since we want the complex to be of the size which is bit-complexity of M we make use of a two-dimensional complex which we call a Mobius telescope.

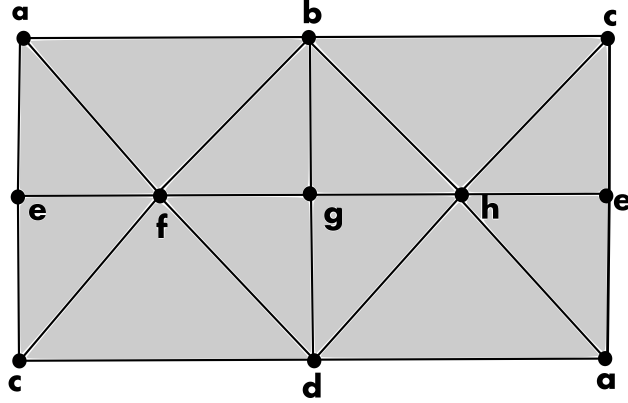

FIGURE 5.4: the Mubius band with equivalent middle and outer circles.

Mobius telescope. A *Mobius telescope* of height 1 is depicted in Figure 5.4. In this figure c_0 is the 1-cycle consisting of the edges in the middle of the band and c_1 is the boundary 1-cycle. It is clear that $[c_1] = 2[c_0]$ in homology. Note that the triangulation is such that c_0 and c_1 are combinatorially the same complex. Now consider a second Mobius band with corresponding cycles c'_0 and c'_1. We can identify c'_0 with c_1 without introducing any more simplices. In the resulting complex we will have $[c'_1] = 2[c'_0] = 2[c_1] = 4[c_0]$. The resulting complex is called a Mobius tower of height 2. A *Mobius telescope* of height k is built by repeating this process $k-1$ times. The complex of height k has evidently size linear in k and has a 1-cycle $[c_k] = 2^k[c_0]$, which is the boundary of the topmost Mobius band. It is also easy to see that the homology of the Mobius telescope is free cyclic with c_0 as a generator.

Construction. The construction of the complex proceeds as in the 0-1 case with the following changes. First we select an orientation for each circle x_i. Let N be the largest entry in absolute value of the matrix M. For each cycle x_i we create a Mobius telescope of height $\lfloor \log |N| \rfloor$ and identify the base of the telescope with the circle x_i with a degree 1 map. Since the telescope is homotopy equivalent to the base circle, this does not change the homology classes.

For the row i, we take again a sphere with m_i holes denoted by C, where m_i is the number of non-zero entries of row i. Let $\mu = M(i,j)$ be a non-zero entry of row i and let $\pm\mu_p\mu_{p-1}\ldots\mu_0$ be the binary representation of μ so that $p = \lfloor \log|\mu| \rfloor$. Assume that m_μ of the μ_i are non-zero.

We take a m_μ-cap C_μ and glue one of its boundary circles to the boundary circle of C corresponding to μ, c_μ, with a map of degree opposite to sign of μ. For each k such that $\mu_k = 1$, we connect one boundary circle of C_μ with the corresponding cycle c_k of the telescope by a degree 1 map.

The surface C_μ introduces the relation

$$\mp c_\mu + \mu_p 2^p[c_0] + \mu_{p-1}2^{p-1}[c_0] + \ldots + \mu_1 2[c_0] + \mu_0[c_0] = 0.$$

It follows that $[c_\mu] = \mu[c_0] = \mu[x_j]$. The proof that this is the only relation introduced is similar to the 0-1 case, namely, one only has to use one disk with boundary $\mu_p 2^p[c_0] + \mu_{p-1}2^{p-1}[c_0] + \ldots + \mu_1 2[c_0] + \mu_0[c_0] \mp c_\mu$. One dimensional homology classes created are all vertical if we assign different function values to vertices, similar to the 0-1 case.

After executing the above for each entry μ of the row i, we have introduced the relation $\sum_j M(i,j)[x_j] = 0$. Moreover, the size of the complex is linear in the bit complexity of the matrix M.

From the construction it follows that the first horizontal homology group of K is generated by $[x_j]$ subject to the relations $Mx = 0$, where $x = ([x_1], \ldots, [x_n]) \in \mathbb{Z}^n$ is the set of generators of the first homology in the initial complex before attaching cells. Therefore the horizontal homology group is isomorphic to $\mathbb{Z}^n$ generated by $[x_i]$'s with relations $Mx = 0$ which is isomorphic to the cokernel above. In other words, we have killed the image of transpose of M in $\mathbb{Z}^n$.

Embedding in $\mathbb{R}^4$. We again embed the n circles x_i in a 2-plane. For each x_i we embed a Mobius telescope as follows. For each level of the telescope we choose a 3-plane disjoint from the 3-planes of lower levels. We embed a usual Mobius band in each of these 3-planes. Next we attach a cylinder to any two consecutive Mobius bands which glues together the cycles which are identified in the definition of the Mobius telescope. To see that this is indeed possible observe that the boundary of a Mobius band is ambient isotopic inside a 3-plane to the middle circle. This is equivalent to existence of the above cylinder in the strip $[0,1] \times P$ where $I = [0,1]$, $\{0\} \times P$ and $\{1\} \times P$ are consecutive disjoint 3-planes. It follows that the Mobius strip can be embedded for each x_i in time proportional to its size.

For each circle x_i, we can moreover embed the Mobius telescope such that the circles c_k of the telescope all lie in the 2-plane of x_i. This is as follows. Let c_k be embedded on the plane. We have to embed Mobius bands between c_k and c_{k+1} such that c_{k+1} is the boundary circle and c_k is the middle circle of the band. This is possible by attaching a cylinder between them. Moreover, we make sure that the cylinders are all in the same half-space created by a fixed 3-plane P which passes throw the 2-plane of circles. Note that in general the two Mobius bands incident on a cycle c_k are disjoint since the intersection of a 3-plane parallel to P with the bands are at most two unlinked circles. This process can be done in time linear in the size of the telescope. In addition, these maps are standard and for each matrix and each x_i the embedded telescope is a copy of others.

By the above process we can embed each Mobius telescope such that all the cylinders are in the same half-space defined by P. For each x_i, we also create a circle denoted $-x_i$ and embedded in the 2-plane of x_i's. This we also connect to x_i by a mapping cylinder of a map of degree -1. This mapping cylinder can also be embedded in the same half space as the other cylinders.

Now we embed the rest of the complex as follows. Consider C_μ of the entry

$\mu = M(i, j)$. This m_μ-cap attaches to the complex by attaching one boundary circle to c_μ in C and others to the c_k of the telescope of x_j. We can embed the circle c_μ also in the plane of the other circles. Other boundary circles will be glued to the corresponding circles among c_k of the telescope of x_j. All of these circles are in the same plane. We choose a new 3-plane P' for this cap. P together with P' divide $\mathbb{R}^4$ into two half-spaces. We embed the cap c_μ in intersection of P' and the half-space which is empty from the cylinders. This is easily possible since all the mapping degrees are 1. Note that if we had to glue the cap to c_μ by a map of degree -1, we will embed a standard mapping cylinder of degree -1 attached to c_μ with the other end $-c_\mu$ embedded in the plane of circles. After each C_μ is embedded for the i'th row of the matrix, we will embed C similarly in a new 3-plane.

Observe that since the embedded 2-complex is the same for each matrix just before adding the disks C_μ, and these disks can be embedded easily in linear time in their size, the whole embedding of the matrix takes time proportional to the total size of the disks C_μ which up to a constant is the same as bit-complexity of the matrix M.

Thus we have proved the following theorem. Note that we assume the matrix is presented in a data structure that allows access to non-zero entries, otherwise, we assume that the bit complexity of M is at least n^2, that is, zeros are counted as one bit.

Theorem 7. *Given an integer matrix of bit-complexity m, in $O(m)$ time it is possible to build a simplicial complex K of size $O(m)$ such that*

$$coker(M) \cong H_1^h(K),$$

where $H_1^h(K)$ is the horizontal homology of K with respect to a suitable function. Moreover, in the same time it can be embedded in $\mathbb{R}^4$. In other words, K can be realized in $\mathbb{R}^4$ after a constant number of subdivisions.

6

Conclusion

We considered the Reeb graph, its algorithms and concepts related to it, such as vertical first homology groups, etc.. We conclude by pointing out some possible directions that this research can be continued.

We mentioned that the running time for the static algorithm, that is, $O(m \log m)$, is *almost* optimal. It is not difficult to see that the Reeb graph algorithm can sort the vertices, and hence it can have a $O(m \log m)$ lower bound, where m is size of the complex. Nevertheless, there are several obstacles to calling our algorithm for the Reeb graph optimal. The issue is that the running time can be expressed as a function of vertices and perhaps there are algorithms that perform better. The reduction from sorting to the Reeb graph results in a complex with number of vertices proportional to the number of edges of the graph. Hence vertices dominate the size. It will be interesting to know if it is possible to have faster algorithms for very dense complexes. Intuitively, for dense complexes it is not necessary to consider all of the triangles. Another direction is to consider special complexes, such as Rips- or Cech-complexes, or embedded complexes and see if the Reeb graph can be constructed more efficiently.

One can also generalize the Reeb graph by changing the range of the function.

One such generalization, namely, when the range is a high-dimensional Euclidean space is called a Reeb space, see Edelsbrunner et al. (2008a). It is interesting to have efficient algorithms for constructing Reeb spaces. However, it seems that the complexity of the image of f in this case plays a role. By complexity of image of f we mean size of a triangulation of the image in the Euclidean space. It is not difficult to build an example in which the Reeb space itself is a disk, whereas, the map from this disk to the image of f is of large complexity. If one is interested in maintaining the map from the Reeb space to the range of f, then building the Reeb space at least needs time proportional to the image of f.

For the dynamic setting, the main question is if there exists algorithms for the dynamic Reeb graph better than the trivial data structure. Moreover, it would be interesting if one can obtain an algorithm which is output sensitive for the interchange update. Another interesting research direction is to consider changes in the complex, rather than the function.

Regarding the last part of the thesis, we have observed a reduction from matrix computations to the Betti number computations. We have shown that embeddability of the complex in $\mathbb{R}^4$ gives no new information as long as the computational complexity of Betti numbers is concerned. This was achieved specifically first by use of vertical classes. The vertical classes result from defining a function on the complex. The author believes that this approach can also help in attacking other problems regarding complexes.

Bibliography

Acar, U. A., Blelloch, G. E., Harper, R., Vittes, J. L., and Woo, S. L. M. (2004), "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, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics.

Alstrup, S., Holm, J., Lichtenberg, K. D., and Thorup, M. (2005), "Maintaining information in fully dynamic trees with top trees," *ACM Transactions on Algorithms*, 1, 243–264.

Aujay, G., Hétroy, F., Lazarus, F., and Depraz, C. (2007), "Harmonic skeleton for realistic character animation," in *Proceedings of the 2007 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '07, pp. 151–160, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association.

Bespalov, D., Regli, W. C., and Shokoufandeh, A. (2003), "Reeb graph based shape retrieval for CAD," *ASME Conference Proceedings*, 2003, 229–238.

Biasotti, S., De Floriani, L., Falcidieno, B., Frosini, P., Giorgi, D., Landi, C., Papaleo, L., and Spagnuolo, M. (2008a), "Describing shapes by geometrical-topological properties of real functions," *ACM Computing Surveys*, 40, 12:1–12:87.

Biasotti, S., Giorgi, D., Spagnuolo, M., and Falcidieno, B. (2008b), "Reeb graphs for shape analysis and applications," *Theoretical Computer Science*, 392, 5 – 22, Computational Algebraic Geometry and Applications.

Buchin, K., Buchin, M., van Kreveld, M., Speckmann, B., and Staals, F. (2013), "Trajectory grouping structure," in *Algorithms and Data Structures*, eds. F. Dehne, R. Solis-Oba, and J.-R. Sack, vol. 8037 of *Lecture Notes in Computer Science*, pp. 219–230, Springer Berlin Heidelberg.

Bunch, J. R. and Hopcroft, J. E. (1974), "Triangular factorization and inversion by fast matrix multiplication," *Mathematics of Computation*, 28, pp. 231–236.

Carr, H., Snoeyink, J., and Axen, U. (2003), "Computing contour trees in all dimensions," *Computational Geometry*, 24, 75 – 94.

Chazal, F. and Sun, J. (2014), "Gromov-Hausdorff approximation of filament structure using Reeb-type graph," in *Proceedings of the Thirtieth Annual Symposium on Computational Geometry*, SOCG'14, p. 491:500, New York, NY, USA, ACM.

Cohen-Steiner, D., Edelsbrunner, H., and Harer, J. (2009), "Extending persistence using Poincare and Lefschetz duality," *Foundations of Computational Mathematics*, 9, 79–103.

Cole-McLaughlin, K., Edelsbrunner, H., Harer, J., Natarajan, V., and Pascucci, V. (2003), "Loops in Reeb graphs of 2-manifolds," in *Proceedings of the Nineteenth Annual Symposium on Computational Geometry*, SCG '03, pp. 344–350, New York, NY, USA, ACM.

Coppersmith, D. and Winograd, S. (1990), "Matrix multiplication via arithmetic progressions," *Journal of Symbolic Computation*, 9, 251 – 280, Computational Algebraic Complexity Editorial.

Delfinado, C. J. A. and Edelsbrunner, H. (1995), "An incremental algorithm for Betti numbers of simplicial complexes on the 3-sphere," *Computer Aided Geometric Design*, 12, 771 – 784, Grid Generation, Finite Elements, and Geometric Design.

Demaine, E. D., Iacono, J., and Langerman, S. (2007), "Retroactive data structures," *ACM Transactions on Algorithms*, 3.

Dey, T. and Wang, Y. (2013), "Reeb graphs: approximation and persistence," *Discrete and Computational Geometry*, 49, 46–73.

Doraiswamy, H. and Natarajan, V. (2009), "Efficient algorithms for computing Reeb graphs," *Computational Geometry*, 42, 606 – 616.

Driscoll, J. R., Sarnak, N., Sleator, D. D., and Tarjan, R. E. (1989), "Making data structures persistent," *Journal of Computer and System Sciences*, 38, 86 – 124.

Eberly, W., Giesbrecht, M., Giorgi, P., Storjohann, A., and Villard, G. (2007), "Faster inversion and other black box matrix computations using efficient block projections," in *Proceedings of the 2007 international symposium on Symbolic and algebraic computation*, ISSAC '07, pp. 143–150, New York, NY, USA, ACM.

Edelsbrunner, H. and Parsa, S. (2014), "On the computational complexity of Betti Numbers: reductions from matrix rank," in *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pp. 152–160.

Edelsbrunner, H., Harer, J., and Patel, A. K. (2008a), "Reeb spaces of piecewise linear mappings," in *Proceedings of the Twenty-fourth Annual Symposium on Computational Geometry*, SCG '08, pp. 242–250, New York, NY, USA, ACM.

Edelsbrunner, H., Harer, J., Mascarenhas, A., Pascucci, V., and Snoeyink, J. (2008b), "Time-varying Reeb graphs for continuous space-time data," *Computational Geometry*, 41, 149 – 166.

Eppstein, D. (1994), "Offline algorithms for dynamic minimum spanning tree problems," *Journal of Algorithms*, 17, 237–250.

Eppstein, D., Italiano, G. F., Tamassia, R., Tarjan, R. E., Westbrook, J., and Yung, M. (1990), "Maintenance of a Minimum Spanning Forest in a Dynamic Planar Graph," in *Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '90, pp. 1–11, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics.

Eppstein, D., Galil, Z., Italiano, G. F., and Nissenzweig, A. (1997), "Sparsification a technique for speeding up dynamic graph algorithms," *Journal of the ACM*, 44, 669–696.

Frederickson, G. (1985), "Data structures for on-line updating of minimum spanning trees, with applications," *SIAM Journal on Computing*, 14, 781–798.

Fujishiro, I., Takeshima, Y., Azuma, T., and Takahashi, S. (2000), "Volume data mining using 3D field topology analysis," *IEEE Computer Graphics and Applications*, 20, 46–51.

Harvey, W., Wang, Y., and Wenger, R. (2010), "A randomized $O(m \log m)$ time algorithm for computing Reeb graphs of arbitrary simplicial complexes," in *Proceedings of the thwenty-sixth Annual Symposium on Computational Geometry*, SoCG '10, pp. 267–276, New York, NY, USA, ACM.

Hatcher, A. (2001), *Algebraic Topology*, Cambridge University Press.

Henzinger, M. R. and King, V. (1999), "Randomized fully dynamic graph algorithms with polylogarithmic time per operation," *Journal of the ACM*, 46, 502–516.

Hilaga, M., Shinagawa, Y., Kohmura, T., and Kunii, T. L. (2001), "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, New York, NY, USA, ACM.

Holm, J., de Lichtenberg, K., and Thorup, M. (2001), "Poly-logarithmic deterministic fully-dynamic algorithms for connectivity, minimum spanning tree, 2-edge, and biconnectivity," *Journal of the ACM*, 48, 723–760.

Kanongchaiyos, P. and Shinagawa, Y. (2000), "Articulated Reeb graphs for interactive skeleton animation," *Modeling Modeling Multimedia Information and System*, pp. 451–467.

Kapron, B. M., King, V., and Mountjoy, B. (2013), "Dynamic graph connectivity in polylogarithmic worst case time." in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA '13, pp. 1131–1142.

Munkres, J. (1984), *Elements of Algebraic Topology*, Advanced book classics, Perseus Books Group.

Natali, M., Biasotti, S., Patane, G., and Falcidieno, B. (2011), "Graph-based representations of point clouds," *Graphical Models*, 73, 151 – 164.

Parsa, S. (2013), "A deterministic $O(m \log m)$ time algorithm for the Reeb graph," *Discrete and Computational Geometry*, 49, 864–878.

Pascucci, V., Scorzelli, G., Bremer, P.-T., and Mascarenhas, A. (2007), "Robust online computation of Reeb graphs: simplicity and speed," *ACM Transactions on Graphics*, 26.

Patrascu, M. and Demaine, E. D. (2006), "Logarithmic lower bounds in the cell-probe model," *SIAM Journal on Computing*, 35, 932–963.

Reeb, G. (1946), "Sur les points singuliers d'une forme de Pfaff complètement intégrable ou d'une fonction numérique," *Comptes Rendus Acad. Sciences*, 222, 847–849.

Rekleitis, I., Lee-Shue, V., and Choset, H. (2004), "Limited communication, multi-robot team based coverage," *Proceedings of IEEE International Conference on Robotics and Automation 2004*, 4, 3462–3468.

Shi, Y., Lai, R., Krishna, S., Sicotte, N., Dinov, I., and Toga, A. W. (2008), "Anisotropic Laplace-Beltrami eigenmaps: bridging Reeb graphs and skeletons," *Computer Vision and Pattern Recognition Workshop*, pp. 1–7.

Shinagawa, Y. and Kunii, T. L. (1991), "Constructing a Reeb graph automatically from cross sections," *IEEE Computer Graphics and Applications*, 11, 44–51.

Shinagawa, Y., Kunii, T. L., and Kergosien, Y. L. (1991), "Surface coding based on Morse theory," *IEEE Computer Graphics and Applications*, 11, 66–78.

Sleator, D. D. and Tarjan, R. E. (1983), "A data structure for dynamic trees," *Journal of Computer and System Sciences*, 26, 362 – 391.

Sleator, D. D. and Tarjan, R. E. (1985), "Self-adjusting binary search trees," *Journal of the ACM*, 32, 652–686.

Tarjan, R. E. and Werneck, R. F. (2005), "Self-adjusting top trees," in *Proceedings of the Sixteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '05, pp. 813–822, Philadelphia, PA, USA, Society for Industrial and Applied Mathematics.

Tarjan, R. E. and Werneck, R. F. (2010), "Dynamic trees in practice," *Journal of Experimental Algorithmics*, 14, 5:4.5–5:4.23.

Thorup, M. (2000), "Near-optimal fully-dynamic graph connectivity," in *Proceedings of the Thirty-second Annual ACM Symposium on Theory of Computing*, STOC '00, pp. 343–350, New York, NY, USA, ACM.

Tierny, J., Gyulassy, A., Simon, E., and Pascucci, V. (2009), "Loop surgery for volumetric meshes: Reeb graphs reduced to contour trees," *IEEE Transactions on Visualization and Computer Graphics*, 15, 1177–1184.

Tung, T. and Schmitt, F. (2005), "The augmented multiresolution Reeb graph approach for content-based retrieval of 3D shapes," *International Journal of Shape Modeling*, 11, 91–120.

Wiedemann, D. (1986), "Solving sparse linear equations over finite fields," *IEEE Transactions on Information Theory*, 32, 54–62.

Williams, V. V. (2012), "Multiplying matrices faster than Coppersmith-Winograd," in *Proceedings of the 44th ACM Symposium on Theory of Computing*, STOC '12, pp. 887–898, New York, NY, USA, ACM.

Wood, Z., Hoppe, H., Desbrun, M., and Schröder, P. (2004), "Removing excess topology from isosurfaces," *ACM Transactions on Graphics*, 23, 190–208.

Wulff-Nilsen, C. (2013), "Faster deterministic fully-dynamic graph connectivity," in *Proceedings of the ACM-SIAM Symposium on Discrete Algorithms*, SODA '13.

Yuster, R. (2008), "Matrix sparsification for rank and determinant computations via nested dissection," in *Proceedings of IEEE 49th Annual Symposium on Foundations of Computer Science, 2008*, FOCS '08, pp. 137–145.

Yuster, R. and Zwick, U. (2005), "Fast sparse matrix multiplication," *ACM Transactions on Algorithms*, 1, 2–13.

Biography

Salman Parsa was born on 8th of March, 1985, in Khomeinishahr, which is a town near the historic city of Esfahan, Iran. He finished his high-school and pre-university studies in his home town. He then took part in the nation-wide university exam and was accepted into the Computer Science field at Sharif University of Technology, Tehran. He earned his B.Sc. degree from Sharif. Then he started an M.Sc. in Computer Science from the same department and finished in 2010.After that he moved to the U.S. as a Ph.D. student at Duke. Later, he traveled to Austria to work alongside his advisor, Herbert Edelsbrunner, at the newly founded Institute for Science and Technology, Austria.

At the time of his graduation from Ph.D studies, Parsa is interested broadly in the relations between Computer Science and algorithmic techniques and ideas to more theoretical and abstract methods and concepts from algebraic topology, geometry and other well-established mathematical fields. He believes that consideration of algorithmic problems is part of the mathematical theory, and on the other hand, methods developed in the Theoretical Computer Science over the recent years can have a say in advancing mathematical theories and solving open problems.