Barcodes of Towers and a Streaming Algorithm for Persistent Homology*

Michael Kerber[1] and Hannah Schreiber[2]

1 **Graz University of Technology, Graz, Austria**
kerber@tugraz.at
2 **Graz University of Technology, Graz, Austria**
hschreiber@tugraz.at

Abstract

A tower is a sequence of simplicial complexes connected by simplicial maps. We show how to compute a filtration, a sequence of nested simplicial complexes, with the same persistent barcode as the tower. Our approach is based on the coning strategy by Dey et al. (SoCG 2014). We show that a variant of this approach yields a filtration that is asymptotically only marginally larger than the tower and can be efficiently computed by a streaming algorithm, both in theory and in practice. Furthermore, we show that our approach can be combined with a streaming algorithm to compute the barcode of the tower via matrix reduction. The space complexity of the algorithm does not depend on the length of the tower, but the maximal size of any subcomplex within the tower. Experimental evaluations show that our approach can efficiently handle towers with billions of complexes.

1998 ACM Subject Classification G.4 Algorithm Design and Analysis

Keywords and phrases Persistent Homology, Topological Data Analysis, Matrix reduction, Streaming algorithms, Simplicial Approximation

Digital Object Identifier 10.4230/LIPIcs.SoCG.2017.57

1 Introduction

Motivation and problem statement. Persistent homology [16, 6, 15] is a paradigm to analyze how topological properties of general data sets evolve across multiple scales. Thanks to the success of the theory in finding applications (see, e.g., [25, 18] for recent enumerations), there is a growing demand for efficient computations of the involved topological invariants.

In this paper, we consider a sequence of simplicial complexes $(\mathbb{K}_i)_{i=0,\dots,m}$ and simplicial maps $\phi_i : \mathbb{K}_i \to \mathbb{K}_{i+1}$ connecting them, calling this data a *(simplicial) tower* of length m. Applying the homology functor with an arbitrary field, we obtain a *persistence module*, a sequence of vector spaces connected by linear maps. Such a module decomposes into a *barcode*, a collection of intervals, each representing a homological feature in the tower that spans over the specified range of scales.

Our computational problem is to compute the barcode of a given tower efficiently. The most prominent case of a tower is when all maps f_i are inclusion maps. In this case one obtains a *filtration*, a sequence of nested simplicial complexes. A considerable amount of work went into the study of fast algorithms for the filtration case, which culminated in fast software libraries for this task. The more general case of towers recently received growing

* The authors are supported by the Austrian Science Fund (FWF) grant number P 29984-N35.

interest in the context of sparsification technique for the Vietoris-Rips and Čech complexes; see the related work section below for a detailed discussion.

Results. As our first result, we show that any tower can be *efficiently* converted into a *small* filtration with the same barcode. Dey et al. [13] give an explicit construction, called "coning", for the generalized case of *zigzag towers*. Using a simple variant of their strategy, we obtain a filtration whose size is only marginally larger than the length of the tower. Furthermore, we experimentally show that the size is even smaller on realistic instances.

To describe our improved coning strategy, we discuss the case that a simplicial map in the tower contracts two vertices u and v. The coning strategy by Dey et al. proposes to join u with the closed star of v, making all incident simplices of v incident to u without changing the homotopy type. The vertex u is then taken as the representative of the contracted pair. We refer to the number of simplices that the join operation adds to the complex as the *cost* of the contraction. Quite obviously, the method is symmetric in u and v, and we have two choices to pick the representative, leading to potentially quite different costs. We employ the self-evident strategy to pick the representative that leads to smaller costs. This idea leads to an asymptotically improved size bound on the filtration. We prove this by an abstraction to path decompositions on weighted forest. Altogether, the worst-case size of the filtration is $O(\Delta \cdot n \cdot \log(n_0))$, where Δ is the maximal dimension of any complex in the tower, and n/n_0 is the number of simplices/vertices added to the tower.

We also provide a conversion algorithm whose time complexity is roughly proportional to the total number of simplices in the resulting filtration. One immediate benefit is a generic solution to compute barcodes of towers: just convert the tower to a filtration and apply one of the efficient implementations for barcodes of filtrations. Indeed, we experimentally show that on not-too-large towers, our approach is competitive with, and sometimes outperforms Simpers, an alternative approach that computes the barcode of towers with *annotations*, a variant of the persistent cohomology algorithm.

Our second contribution is a space-efficient version of the just mentioned algorithmic pipeline that is applicable to very large towers. To motivate the result, let the *width* of a tower denote the maximal size of any simplicial complex among the $\mathbb{K}_i$. Consider a tower with a very large length (say, larger than the number of bytes in main memory) whose width remains relatively small. In this case, our conversion algorithm yields a filtration that is very large as well. Most implementations for barcode computation read the entire filtration on initialization and algorithms based on matrix reduction are required to keep previously reduced columns. This leads to a high memory consumption for the barcode computation.

We show that with minor modifications, the standard persistent algorithm can be turned into a streaming algorithm with smaller space complexity in the case of towers. The idea is that upon contractions, simplices become *inactive* and cannot get additional cofaces. Our approach makes use of this observation by modifying the boundary matrix such that columns associated to inactive simplices can be removed. Combined with our conversion algorithm, we can compute the barcode of a tower of width ω keeping only up to $O(\omega)$ columns of the boundary matrix in memory. This yields a space complexity of $O(\omega^2)$ and a time complexity of $O((\Delta \cdot n \cdot \log(n_0)) \cdot \omega^2)$ in the worst case. We implemented a practically improved variant that makes use of additional heuristics to speed up the barcode computation in practice and resembles the *chunk algorithm* presented in [1]. We tested our implementation on various challenging data sets. The source code of the implementation is available at `https://bitbucket.org/schreiberh/sophia/`.

Related work. Already the first works on persistent homology pointed out the existence of efficient algorithm to compute the barcode invariant (or equivalently, the persistent diagram) for filtrations [16, 27]. As a variant of Gaussian elimination, the worst-case complexity is cubic. Remarkable theoretical follow-up results are a persistence algorithm in matrix multiplication time [23], an output-sensitive algorithm to compute only high-persistent features with linear space complexity [9], and a conditional lower bound on the complexity relating the problem to rank computations of sparse matrices [17].

On realistic instances, the standard algorithm has shown a quasi-linear behavior in practice despite its pessimistic worst-case complexity. Nevertheless, many improvements of the standard algorithm have been presented in the last years which improve the runtime by several orders of magnitude. One line of research exploits the special structure of the boundary matrix to speed up the reduction process [8]. This idea has led to efficient parallel algorithms for persistence in shared [1] and distributed memory [2]. Moreover, of same importance as the reduction strategy is an appropriate choice of data structures in the reduction process as demonstrated by the PHAT library [3]. A parallel development was the development of dual algorithms using persistent cohomology, based on the observation that the resulting barcode is identical [12]. The *annotation algorithm* [13, 4] is an optimized variant of this idea realized in the GUDHI library [21]. It is commonly considered as an advantage of annotations that only a cohomology basis must be kept during the reduction process, making it more space efficient than reduction-based approaches. We refer to the comparative study [24] for further approaches and software for persistence on filtrations.

Moreover, generalizations of the persistence paradigm are an active field of study. *Zigzag persistence* is a variant of persistence where the maps in the filtration are allowed to map in either direction (that is, either $\phi_i : \mathbb{K}_i \hookrightarrow \mathbb{K}_{i+1}$ or $\phi_i : \mathbb{K}_i \hookleftarrow \mathbb{K}_{i+1}$) – see [25] for a comprehensive introduction. The initial algorithms to compute this barcode [7] has been improved recently [22]. Our case of towers of complexes and simplicial maps can be modeled as a zigzag filtration and therefore sits in-between the standard and the zigzag filtration case.

Dey et al. [13] described the first efficient algorithm to compute the barcode of towers. Instead of the aforementioned coning approach explained in their paper, their implementation handles contractions with an empirically smaller number of insertions, based on the link condition. Recently, the authors have released the SIMPERS library that implements their annotation algorithm from the paper.

The case of towers has received recent attention in the context of approximate Vietoris-Rips and Čech filtrations. The motivation for approximation is that the (exact) topological analysis of a set of n points in d-dimensions requires a filtration of size $O(n^{d+1})$ which is prohibitive for most interesting input sizes. The first such type of result by Sheehy [26] resulted in a approximate filtration; however, it has been observed that the concept of towers somewhat simplifies the approximation schemes conceptually. See [13, 5, 20, 10] for examples. Very recently, the SimBa library [14] brings these theoretical approximation techniques for Vietoris-Rips complexes into practice.

Outline. We introduce the necessary basic concepts in Section 2. We describe our conversion algorithm from general towers to barcodes in Section 3. The streaming algorithm for persistence is discussed in Section 4. Several proofs and constructions are only sketched due to space constraints; see the arxiv version [19] for full arguments.

2 Background

Simplicial Complexes. Given a finite *vertex set* V, a *simplex* is a non-empty subset of V; more precisely, a *k-simplex* is a subset consisting of $k+1$ vertices, and k is called the *dimension* of the simplex. For a k-simplex σ, a simplex τ is a *face* of σ if $\tau \subseteq \sigma$. If τ is of dimension ℓ, we call it a ℓ-face. If $\ell < k$, we call τ a *proper face* of σ, and if $\ell = k-1$, we call it a *facet*. For a simplex σ and a vertex $v \notin \sigma$, we define the *join* $v * \sigma$ as the simplex $\{v\} \cup \sigma$.

An *(abstract) simplicial complex* $\mathbb{K}$ over V is a set of simplices that is closed under taking faces. We call V the *vertex set* of $\mathbb{K}$ and write $\mathcal{V}(\mathbb{K}) := V$. The *dimension* of $\mathbb{K}$ is the maximal dimension of its simplices. For $\sigma, \tau \in \mathbb{K}$, we call σ a *coface* of τ in $\mathbb{K}$ if τ is a face of σ. In this case, σ is a *cofacet* of τ if their dimensions differ by exactly one. A simplicial complex $\mathbb{L}$ is a *subcomplex* of $\mathbb{K}$ if $\mathbb{L} \subseteq \mathbb{K}$. Given $\mathcal{W} \subseteq \mathcal{V}$, the *induced subcomplex* by $\mathcal{W}$ is the set of all simplices σ in $\mathbb{K}$ with $\sigma \subseteq \mathcal{W}$. For a subcomplex $\mathbb{L} \subseteq \mathbb{K}$ and a vertex $v \in \mathcal{V}(\mathbb{K}) \setminus \mathcal{V}(\mathbb{L})$, we define the join $v * \mathbb{L} := \{v * \sigma \mid \sigma \in \mathbb{L}\}$. For a vertex $v \in \mathbb{K}$, the *star* of v in $\mathbb{K}$, denoted by $\mathrm{St}(v, \mathbb{K})$, is the set of all cofaces of v in $\mathbb{K}$. In general, the star is not a subcomplex, but we can make it a subcomplex by adding all faces of star simplices, which is denoted by the *closed star* $\overline{\mathrm{St}}(v, \mathbb{K})$. Equivalently, the closed star is the smallest subcomplex of $\mathbb{K}$ containing the star of v. The *link* of v, $\mathrm{Lk}(v, \mathbb{K})$, is defined as $\overline{\mathrm{St}}(v, \mathbb{K}) \setminus \mathrm{St}(v, \mathbb{K})$. It can be checked that the link is a subcomplex of $\mathbb{K}$. When the complex is clear from context, we will sometimes omit the $\mathbb{K}$ in the notation of stars and links.

Simplicial maps. A map $\mathbb{K} \xrightarrow{\phi} \mathbb{L}$ between simplicial complexes is called *simplicial* if with $\sigma = \{v_0, \ldots, v_k\} \in \mathbb{K}$, $\phi(\sigma)$ is equal to $\{\phi(v_0), \ldots, \phi(v_k)\}$ and $\phi(\sigma)$ is a simplex in $\mathbb{L}$. By definition, a simplicial map maps vertices to vertices and is completely determined by its action on the vertices. Moreover, the composition of simplicial maps is again simplicial.

A simple example of a simplicial map is the inclusion map $\mathbb{L} \xhookrightarrow{\phi} \mathbb{K}$ where $\mathbb{L}$ is a subcomplex of $\mathbb{K}$. If $\mathbb{K} = \mathbb{L} \cup \{\sigma\}$ with $\sigma \notin \mathbb{L}$, we call ϕ an *elementary inclusion*. The simplest example of a non-inclusion simplicial map is $\mathbb{K} \xrightarrow{\phi} \mathbb{L}$ such that there exist two vertices $u, v \in \mathbb{K}$ with $\mathcal{V}(\mathbb{L}) = \mathcal{V}(\mathbb{K}) \setminus \{v\}$, $\phi(u) = \phi(v) = u$, and ϕ is the identity on all remaining vertices of $\mathbb{K}$. We call ϕ an *elementary contraction* and write $(u, v) \rightsquigarrow u$ as a shortcut. These notions were introduced by Dey, Fan and Wang in [13] and they also showed that any simplicial map $\mathbb{K} \xrightarrow{\phi} \mathbb{L}$ can be written as the composition of elementary contractions[1] and inclusions.

A *tower* of length m is a collection of simplicial complexes $\mathbb{K}_0, \ldots, \mathbb{K}_m$ and simplicial maps $\phi_i : \mathbb{K}_i \to \mathbb{K}_{i+1}$ for $i = 0, \ldots, m-1$. From this initial data, we obtain simplicial maps $\phi_{i,j} : \mathbb{K}_i \to \mathbb{K}_j$ for $i \leq j$ by composition, where $\phi_{i,i}$ is simply the identity map on $\mathbb{K}_i$. A tower is called a *filtration* if all ϕ_i are inclusion maps. The *dimension* of a tower is the maximal dimension among the $\mathbb{K}_i$, and the *width* of a tower is the maximal number of simplices in a $\mathbb{K}_i$. For filtrations, dimension and width are determined by the dimension and size of the last complex $\mathbb{K}_m$, but this is not necessarily true for general towers.

Homology and Collapses. For a fixed base field $\mathbb{F}$, let $H_p(\mathbb{K}) := H_p(\mathbb{K}, \mathbb{F})$ the *p-dimensional homology group* of $\mathbb{K}$. It is well-known that $H_p(\mathbb{K})$ is a $\mathbb{F}$-vector space. Moreover, a simplicial map $\mathbb{K} \xrightarrow{\phi} \mathbb{L}$ induces a linear map $H_p(\mathbb{K}) \xrightarrow{\phi^*} H_p(\mathbb{L})$. In categorical terms, the equivalent

[1] They talk about "collapses" instead of "contractions", but this notion clashes with the standard notion of simplicial collapses of free faces that we use later. Therefore, we decided to use "contraction", even though the edge between the contracted vertices might not be present in the complex.

statement is that homology is a functor from the category of simplicial complexes and simplicial maps to the category of vector spaces and linear maps.

We will make use of the following homology-preserving operation: a *free face* in $\mathbb{K}$, is a simplex with exactly one proper coface in $\mathbb{K}$. An *elementary collapse* in $\mathbb{K}$ is the operation of removing a free face and its unique coface from $\mathbb{K}$, yielding a subcomplex of $\mathbb{K}$. We say that $\mathbb{K}$ *collapses to* $\mathbb{L}$, if there is a sequence of elementary collapses transforming $\mathbb{K}$ into $\mathbb{L}$. It is then well-known that the inclusion map $\mathbb{L} \overset{\phi}{\hookrightarrow} \mathbb{K}$ induces an isomorphism ϕ_* between $H_p(\mathbb{L})$ and $H_p(\mathbb{K})$.

Barcodes. A *persistence module* is a sequence vector spaces $\mathbb{V}_0, \ldots, \mathbb{V}_m$ and linear maps $f_{i,j} : \mathbb{V}_i \to \mathbb{V}_j$ for $i < j$ such that $f_{i,i} = \mathrm{id}_{\mathbb{V}_i}$ and $f_{i,k} = f_{j,k} \circ f_{i,j}$ for $i \leq k \leq j$. Persistence modules admit a decomposition into indecomposable summands in the following sense. Writing $I_{b,d}$ with $b \leq d$ for the persistence module

$$\underbrace{0 \xrightarrow{0} \ldots \xrightarrow{0} 0}_{b-1 \text{ times}} \xrightarrow{0} \underbrace{\mathbb{F} \xrightarrow{\mathrm{id}} \ldots \xrightarrow{\mathrm{id}} \mathbb{F}}_{d-b+1 \text{ times}} \xrightarrow{0} \underbrace{0 \xrightarrow{0} \ldots \xrightarrow{0} 0}_{m-d \text{ times}},$$

we can write every persistence module as the direct sum $I_{b_1,d_1} \oplus \ldots \oplus I_{b_s,d_s}$, where the direct sum of persistence modules is defined component-wise for vector spaces and linear maps in the obvious way. Moreover, this decomposition is uniquely defined up to isomorphisms and re-ordering, thus the pairs $(b_1, d_1), \ldots, (b_s, d_s)$ are an invariant of the persistence module, called its *barcode*. When the persistence module was generated by a tower, we also talk about the barcode of the tower.

Matrix reduction. In this paragraph, we assume that $(\mathbb{K}_i)_{i=0,\ldots,m}$ is a filtration such that $\mathbb{K}_0 = \emptyset$ and $\mathbb{K}_{i+1}$ has exactly one more simplex than $\mathbb{K}_i$. We label the simplices of $\mathbb{K}_m$ accordingly as $\sigma_1, \ldots, \sigma_m$, with $\mathbb{K}_i \setminus \mathbb{K}_{i-1} = \{\sigma_i\}$. The filtration can be encoded as a *boundary matrix* ∂ of dimension $m \times m$, where the (ij)-entry is 1 if σ_i is a facet of σ_j, and 0 otherwise. In other words, the j-th column of ∂ encodes the facets of σ_j, and the i-th row of ∂ encodes the cofacets of σ_i. Moreover, ∂ is upper-triangular because every $\mathbb{K}_i$ is a simplicial complex. We will sometimes identify rows and columns in ∂ with the corresponding simplex in $\mathbb{K}_m$. Adding the k-simplex σ_i to $\mathbb{K}_{i-1}$ either introduces one new homology class (of dimension k), or turns a non-trivial homology class (of dimension $k-1$) trivial. We call σ_i and the i-th column of ∂ *positive* or *negative*, respectively (with respect to the given filtration).

For the computation of the barcode, we assume for simplicity homology over the base field $\mathbb{Z}_2$, and interpret the coefficients of ∂ accordingly. In an arbitrary matrix A, a *left-to-right column addition* is an operation of the form $A_k \leftarrow A_k + A_\ell$ with $\ell < k$, where A_k and A_ℓ are columns of the matrix. The *pivot* of a non-zero column is the largest non-zero index of the corresponding column. A non-zero entry is called a pivot if its row is the pivot of the column. A matrix R is called a *reduction* of A if R is obtained by a sequence of left-to-right column additions from A and no two columns in R have the same pivot. It is well-known that, although ∂ does not have a unique reduction, the pivots of all its reductions are the same. Moreover, the pivots $(b_1, d_1), \ldots, (b_s, d_s)$ of R are precisely the barcode of the filtration. A direct consequence is that a simplex σ_i is positive if and only if the i-th column in R is zero.

The standard persistence algorithm processes the columns from left to right. In the j-th iteration, as long as the j-th column is not empty and has a pivot that appears in a previous column, it performs a left-to-right column addition. In this work, we use a simple improvement of this algorithm that is called *compression*: before reducing the j-th column, it first scans through the non-zero entries of the column. If a row index i corresponds to

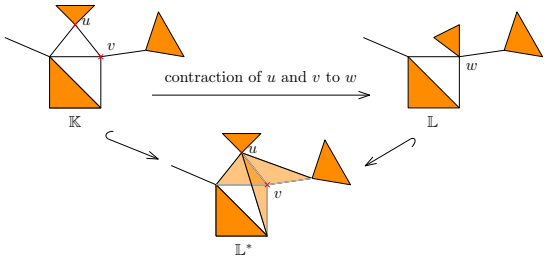

Figure 1 Construction example of $\mathbb{L}^*$, were u and v in $\mathbb{K}$ are contracted to w in $\mathbb{L}$.

a negative simplex (i.e., if the i-th column is not zero at this point in the algorithm), the row index can be deleted without changing the pivots of the matrix. After this initial scan, the column is reduced in the same way as in the standard algorithm. See [1, §. 3] for a discussion (we remark that this optimization was also used in [27]).

3 From towers to filtrations

We phrase now our first result which says that any tower can be converted into a filtration of only marginally larger size with a space-efficient streaming algorithm:

▶ **Theorem 1** (Conversion Theorem). *Let* $\mathcal{T}: \mathbb{K}_0 \xrightarrow{\phi_0} \mathbb{K}_1 \xrightarrow{\phi_1} \dots \xrightarrow{\phi_{m-1}} \mathbb{K}_m$ *be a tower where, w.l.o.g.,* $\mathbb{K}_0 = \emptyset$ *and each* ϕ_i *is either an elementary inclusion or an elementary contraction. Let* Δ *denote the dimension and* ω *the width of the tower, and let* $n \leq m$ *denote the total number of elementary inclusions, and* n_0 *the number of vertex inclusions. Then, there exists a filtration* $\mathcal{F}: \hat{\mathbb{K}}_0 \hookrightarrow \hat{\mathbb{K}}_1 \hookrightarrow \dots \hookrightarrow \hat{\mathbb{K}}_m$ *, where the inclusions are not necessarily elementary, such that* $\mathcal{T}$ *and* $\mathcal{F}$ *have the same barcode and the width of the filtration* $|\hat{\mathbb{K}}_m|$ *is at most* $O(\Delta \cdot n \log n_0)$*. Moreover,* $\mathcal{F}$ *can be computed from* $\mathcal{T}$ *with a streaming algorithm in* $O(\Delta \cdot |\hat{\mathbb{K}}_m| \cdot C_\omega)$ *time and space complexity* $O(\Delta \cdot \omega)$*, where* C_ω *is the cost of an operation in a dictionary with* ω *elements.*

The remainder of the section is organized as follows. We define $\mathcal{F}$ in Section 3.1 and prove that it yields the same barcode as $\mathcal{T}$ in Section 3.2. In Section 3.3, we prove the upper bound on the width of the filtration. In Section 3.4, we explain the algorithm to compute $\mathcal{F}$ and analyze its time and space complexity.

3.1 Active and small coning

Coning. We briefly revisit the *coning strategy* introduced by Dey, Fan and Wang [13]. Let $\phi : \mathbb{K} \to \mathbb{L}$ be an elementary contraction $(u, v) \rightsquigarrow u$ and define

$$\mathbb{L}^* = \mathbb{K} \cup \big(u * \overline{\mathrm{St}}(v, \mathbb{K})\big) \text{ (see Figure 1).}$$

Dey et al. show that $\mathbb{L} \subseteq \mathbb{L}^*$ and that the map induced by inclusion is an isomorphism between $H(\mathbb{L})$ and $H(\mathbb{L}^*)$. By applying this result at any elementary contraction, this implies that every zigzag tower can be transformed into a zigzag filtration with identical barcode.

Given a tower $\mathcal{T}$, we can also obtain an non-zigzag filtration using coning, if we continue the operation on $\mathbb{L}^*$ instead of going back to $\mathbb{L}$. More precisely, we set $\tilde{\mathbb{K}}_0 := \mathbb{K}_0$ and if ϕ_i is an inclusion of simplex σ, we set $\tilde{\mathbb{K}}_{i+1} := \tilde{\mathbb{K}}_i \cup \{\sigma\}$. If ϕ_i is a contraction $(u, v) \rightsquigarrow u$, we set $\tilde{\mathbb{K}}_{i+1} = \tilde{\mathbb{K}}_i \cup \big(u * \overline{\mathrm{St}}(v, \tilde{\mathbb{K}}_i)\big)$. Indeed, it can be proved that $(\tilde{\mathbb{K}}_i)_{i=0,\dots,m}$ has the same barcode as $\mathcal{T}$. However, the filtration will not be small, and we will define a smaller variant now.

Our new construction yields a sequence of complexes $\hat{\mathbb{K}}_0, \dots, \hat{\mathbb{K}}_m$ with $\hat{\mathbb{K}}_i \subseteq \hat{\mathbb{K}}_{i+1}$. During the construction, we maintain a flag for each vertex in $\hat{\mathbb{K}}_i$, which marks the vertex as *active* or *inactive*. A simplex is called *active* if all its vertices are active, and *inactive* otherwise. For a vertex u and a complex $\hat{\mathbb{K}}_i$, let $\mathrm{Act}\overline{\mathrm{St}}(u, \hat{\mathbb{K}}_i)$ denote its *active closed star*, which is the set of active simplices in $\hat{\mathbb{K}}_i$ in the closed star of u.

The construction is inductive, starting with $\hat{\mathbb{K}}_0 := \emptyset$. If $\mathbb{K}_i \xrightarrow{\phi_i} \mathbb{K}_{i+1}$ is an elementary inclusion with $\mathbb{K}_{i+1} = \mathbb{K}_i \cup \{\sigma\}$, set $\hat{\mathbb{K}}_{i+1} := \hat{\mathbb{K}}_i \cup \{\sigma\}$. If σ is a vertex, we mark it as active. It remains the case that $\mathbb{K}_i \xrightarrow{\phi_i} \mathbb{K}_{i+1}$ is an elementary contraction of the vertices u and v. If $|\mathrm{Act}\overline{\mathrm{St}}(u, \hat{\mathbb{K}}_i)| \leq |\mathrm{Act}\overline{\mathrm{St}}(v, \hat{\mathbb{K}}_i)|$, we set

$$\hat{\mathbb{K}}_{i+1} = \hat{\mathbb{K}}_i \cup \left(v * \mathrm{Act}\overline{\mathrm{St}}(u, \hat{\mathbb{K}}_i)\right)$$

and mark u as inactive. Otherwise, we do the same by inverting the role of u and v in the construction. This ends the description of the construction. We write $\mathcal{F}$ for the filtration $(\hat{\mathbb{K}}_i)_{i=0,\dots,m}$.

There are two major changes compared to the construction of $(\tilde{\mathbb{K}}_i)_{i=0,\dots,m}$. First, to counteract the potentially large growth of the involved cones, we restrict coning to active simplices. We will show below that the subcomplex of $\hat{\mathbb{K}}_i$ induced by the active vertices is isomorphic to $\mathbb{K}_i$. As a consequence, we add the same number of simplices by passing from $\hat{\mathbb{K}}_i$ to $\hat{\mathbb{K}}_{i+1}$ as in the approach by Dey et al. does when passing from $\mathbb{K}$ to $\mathbb{L}^*$.

A second difference is that our strategy exploits that an elementary contraction of two vertices u and v leaves us with a choice: we can either take u or v as the representative of the contracted vertex. In terms of simplicial maps, these two choices correspond to setting $\phi_i(u) = \phi_i(v) = u$ or $\phi_i(u) = \phi_i(v) = v$, if ϕ_i is the elementary contraction of u and v. It is obvious that both choices yield identical complexes $\mathbb{K}_{i+1}$ up to renaming of vertices. However, the choices make a difference in terms of the size of $\hat{\mathbb{K}}_{i+1}$, because the active closed star of u to v in $\hat{\mathbb{K}}_i$ might differ in size. Our construction simply choose the representative which causes the smaller $\hat{\mathbb{K}}_{i+1}$.

3.2 Topological equivalence

We assume w.l.o.g. that the vertices in $\mathbb{K}_i$ are named such that, whenever our construction encounters an elementary contraction ϕ_i of u and v and turns v inactive, we have $\phi_i(u) = \phi_i(v) = u$. With this convention, $\mathbb{K}_i$ is the subcomplex of $\hat{\mathbb{K}}_i$ induced by the active vertices.

▶ **Lemma 2.** *A simplex σ is in $\mathbb{K}_i$ if and only if σ is an active simplex in $\hat{\mathbb{K}}_i$.*

The proof works by induction on i, analyzing carefully the effect of a contraction on $\hat{\mathbb{K}}_i$ and $\mathbb{K}_i$. Moreover, when an elementary contraction of u and v turns v inactive, every simplex $\sigma = \{v, v_1, \dots, v_d\}$ that becomes inactive in $\hat{\mathbb{K}}_i$ has a corresponding simplex $\tau = \{u, v, v_1, \dots, v_d\}$ that also becomes inactive. The pairs (σ, τ) can be arranged in collapsible pairs, which implies with an inductive argument:

▶ **Lemma 3.** *For every $0 \leq i \leq m$, the complex $\hat{\mathbb{K}}_i$ collapses to $\mathbb{K}_i$.*

▶ **Proposition 4.** *$\mathcal{T}$ and $\mathcal{F}$ have the same barcode.*

Proof Sketch. Let $\hat{\phi}_i : \hat{\mathbb{K}}_i \to \hat{\mathbb{K}}_{i+1}$ and $\mathrm{inc}_i : \mathbb{K}_i \to \hat{\mathbb{K}}_i$ denote inclusion maps. Lemma 3 implies that the induced homology map $\mathrm{inc}_i^* : H(\mathbb{K}_i) \to H(\hat{\mathbb{K}}_i)$ is an isomorphism for all

$0 \leq i \leq m$. The following diagram connects the persistence modules induced by $\mathcal{T}$ and $\mathcal{F}$:

$$\begin{array}{ccccccc} H(\mathbb{K}_0) & \xrightarrow{\phi_0^*} & H(\mathbb{K}_1) & \xrightarrow{\phi_1^*} & \ldots & \xrightarrow{\phi_{m-1}^*} & H(\mathbb{K}_m) \\ \downarrow{\scriptstyle \mathrm{inc}_0^*} & & \downarrow{\scriptstyle \mathrm{inc}_1^*} & & & & \downarrow{\scriptstyle \mathrm{inc}_m^*} \\ H(\hat{\mathbb{K}}_0) & \xrightarrow{\hat{\phi}_0^*} & H(\hat{\mathbb{K}}_1) & \xrightarrow{\hat{\phi}_1^*} & \ldots & \xrightarrow{\hat{\phi}_{m-1}^*} & H(\hat{\mathbb{K}}_m) \end{array} \tag{1}$$

Our result follows from the *Persistence Equivalence Theorem* [15, p.159] which asserts that $(\mathbb{K}_j)_{j=0,\ldots,m}$ and $(\hat{\mathbb{K}}_j)_{j=0,\ldots,m}$ have the same barcode if (1) commutes, that is, if $\mathrm{inc}_{i+1}^* \circ \phi_i^* = \hat{\phi}_i^* \circ \mathrm{inc}_i^*$, for all $0 \leq i < m$. The latter statements follows from the fact that $\mathrm{inc}_{i+1} \circ \phi_i$ and $\hat{\phi}_i \circ \mathrm{inc}_i$ are contiguous maps. ◀

3.3 Size analysis

The contracting forest. We associate a rooted labeled forest $\mathcal{W}_j$ to a prefix $\emptyset = \mathbb{K}_0 \xrightarrow{\phi_0} \ldots \xrightarrow{\phi_{j-1}} \mathbb{K}_j$ of $\mathcal{T}$ inductively as follows: For $j = 0$, $\mathcal{W}_0$ is the empty forest. Let $\mathcal{W}_{j-1}$ be the forest of $\mathbb{K}_0 \to \ldots \to \mathbb{K}_{j-1}$. If ϕ_{j-1} is an elementary inclusion of a d-simplex, we have two cases: if $d > 0$, set $\mathcal{W}_j := \mathcal{W}_{j-1}$. If a vertex v is included, $\mathcal{W}_j := \mathcal{W}_{j-1} \cup \{x\}$, with x a single node tree labeled with v. If ϕ_{j-1} is an elementary contraction contracting two vertices u and v in $\mathbb{K}_{j-1}$, there are two trees in $\mathcal{W}_{j-1}$, whose roots are labeled u and v. In $\mathcal{W}_j$, these two trees are merged by making their roots children of a new root, which is labeled with the vertex that u and v are mapped to. So $\mathcal{W}_j$ is *full*, that is, every node has 0 or 2 children.

Let $\mathcal{W} := \mathcal{W}_m$ denote the forest of the tower $\mathcal{T}$. Let Σ denote the set of all simplices that are added at elementary inclusions in $\mathcal{T}$, and recall that $n = |\Sigma|$. For a node x in $\mathcal{W}$, we denote by $E(x) \subseteq \Sigma$ the subset of simplices with at least one vertex that appears as label in the subtree of $\mathcal{W}$ rooted at x. If y_1 and y_2 are the children of x, the following follows at once:

$$|E(x)| \quad \geq \quad |E(y_1)| + |E(y_2) \setminus E(y_1)|\,. \tag{2}$$

We say that the set N of nodes in $\mathcal{W}$ is *independent*, if there are no two nodes $x_1 \neq x_2$ in N, such that x_1 is an ancestor of x_2 in $\mathcal{W}$. A vertex in $\mathbb{K}_i$ appears as label in at most one $\mathcal{W}$-subtree rooted at a vertex in the independent set N. Thus, a d-simplex σ can only appear in up to $d+1$ E-sets of vertices in N. That implies:

▶ **Lemma 5.** *Let N be an independent set of vertices in $\mathcal{W}$. Then, $\sum_{x \in N} |E(x)| \leq (\Delta+1) \cdot n$.*

The cost of contracting. In order to bound the total size of $\hat{\mathbb{K}}_m$, we need to bound the number of simplices added in all these contractions. We define the *cost* of a contraction ϕ_i as $|\hat{\mathbb{K}}_{i+1} \setminus \hat{\mathbb{K}}_i|$. Since each contraction corresponds to a node x in $\mathcal{W}$, we can associate these costs to the internal nodes in the forest, denoted by $c(x)$. The leaves get cost 0.

▶ **Lemma 6.** *For an internal node x of $\mathcal{W}$ with children y_1, y_2, $c(x) \leq 2 \cdot |E(y_1) \setminus E(y_2)|$.*

Proof Sketch. Let $\phi_i : \mathbb{K}_i \to \mathbb{K}_{i+1}$ denote the contraction that is represented by the node x, and let w_1 and w_2 the labels of its children y_1 and y_2, respectively. By construction, w_1 and w_2 are vertices in $\mathbb{K}_i$ that are contracted by ϕ_i. Let $C_1 = \overline{\mathrm{St}}(w_1, \mathbb{K}_i) \setminus \overline{\mathrm{St}}(w_2, \mathbb{K}_i)$ and $C_2 = \overline{\mathrm{St}}(w_2, \mathbb{K}_i) \setminus \overline{\mathrm{St}}(w_1, \mathbb{K}_i)$. By Lemma 2, $\overline{\mathrm{St}}(w_1, \mathbb{K}_i) = \mathrm{Act}\overline{\mathrm{St}}(w_1, \hat{\mathbb{K}}_i)$, and the same for w_2. So, because the common simplices of the two active closed stars do not influence the cost of the contraction, we have $c(x) \leq \min\{|C_1|, |C_2|\}$, because the contraction is defined such that the resulting complex is as small as possible.

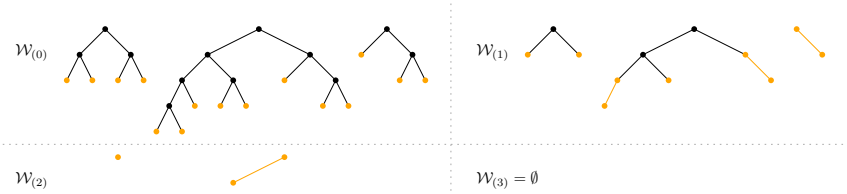

Figure 2 Iterations of the pruning procedure. The only-child-paths are marked in color.

In particular, $c(x) \leq |C_1|$. It is left to show that $|C_1| \leq 2 \cdot |E(y_1) \setminus E(y_2)|$. We do this by a simple charging scheme which attributes the existence of a simplex in C_1 to a simplex in $E(y_1) \setminus E(y_2)$ such that no simplex in the latter set is charged more than twice. ◀

An *ascending path* $(x_1, ..., x_L)$, with $L \geq 1$, is a path in a forest such that x_{i+1} is the parent of x_i, for $1 \leq i < L$. We call L the *length* of the path and x_L its *endpoint*. For ascending paths in $\mathcal{W}$, the *cost* of the path is the sum of the costs of the nodes. The set P of ascending paths is *independent*, if the endpoints in P are pairwise different and form an independent set of nodes. We define the *cost* of P as the sum of the costs of the paths in P.

▶ **Lemma 7.** *An ascending path with endpoint x has cost at most $2 \cdot |E(x)|$. An independent set of ascending paths in $\mathcal{W}$ has cost at most $2 \cdot (\Delta + 1) \cdot n$.*

Proof. For the first statement, let $p = (x_1, ..., x_L)$ be an ascending path with $v_L = v$. Without loss of generality, we can assume the the path starts with a leaf x_1, because otherwise, we can always extend the path to a longer path with at least the same cost. We let $p_i = (x_1, \ldots, x_i)$ denote the sub-path ending at x_i, for $i = 1, \ldots, L$, so that $p_L = p$. We let $c(p_i)$ denote the cost of the path p_i and show by induction that $c(p_i) \leq 2 \cdot |E(x_i)|$. For $i = 1$, this follows because $c(p_1) = 0$. For $i = 2, \ldots, L$, x_i is an internal node, and its two children are x_{i-1} and some other node x'_{i-1}. Using induction and Lemma 6, we have that

$$c(p_i) = c(p_{i-1}) + c(x_i) \leq 2 \cdot (|E(x_{i-1})| + |E(x'_{i-1}) \setminus E(x_{i-1})|) \leq 2 \cdot |E(x_i)|,$$

where the last inequality follows from (2). The second statement follows from Lemma 5 because the endpoints of the paths form an independent set in $\mathcal{W}$. ◀

Ascending path decomposition. An *only-child-path* in a binary tree is an ascending path starting in a leaf and ending at the first encountered node that has a sibling, or at the root of the tree. Consider the following pruning procedure for a full binary forest $\mathcal{W}$. Set $\mathcal{W}_{(0)} \leftarrow \mathcal{W}$. In iteration i, we obtain the forest $\mathcal{W}_{(i)}$ from $\mathcal{W}_{(i-1)}$ by deleting the only-child-paths of $\mathcal{W}_{(i-1)}$. We stop when $\mathcal{W}_{(i)}$ is empty. Figure 2 shows the pruning procedure on an example. We define the following integer valued function for nodes in $\mathcal{W}$:

$$r(x) = \begin{cases} 1, & \text{if } x \text{ is a leaf,} \\ r(y_1) + 1, & \text{if } x \text{ has children } y_1, y_2 \text{ and } r(y_1) = r(y_2), \\ \max\{r(y_1), r(y_2)\}, & \text{if } x \text{ has children } y_1, y_2 \text{ and } r(y_1) \neq r(y_2). \end{cases}$$

With two simple inductive arguments, we can show the next two lemmas:

▶ **Lemma 8.** *A node x of a full binary forest $\mathcal{W}$ is deleted in the pruning procedure during the $r(x)$-th iteration.*

▶ **Lemma 9.** *For a node x in a full binary forest, let $s(x)$ denote the number of nodes in the subtree rooted at x. Then $s(x) \geq 2^{r(x)} - 1$. In particular, $r(x) \leq \log_2(s(x) + 1)$.*

With that, we can bound the size of the constructed filtration.

▶ **Proposition 10.** *$|\hat{\mathbb{K}}_m| \leq n + 2 \cdot (\Delta + 1) \cdot n \cdot (1 + \log_2(n_0)) = O(n \cdot \Delta \cdot \log_2(n_0))$, where n_0 is the number of vertices included in $\mathcal{T}$.*

Proof Sketch. The first summand counts the number of elementary inclusions, the second one the total cost of the contractions. The costs of all nodes removed in one iteration of the pruning procedure are at most $2 \cdot (\Delta + 1) \cdot n$ by Lemma 7 because the considered only-child-paths form an independent set of ascending paths. By Lemma 9, all nodes have been considered after $1 + \log_2(n_0)$ iterations. ◀

3.4 Algorithm

A *dictionary* is a data structure that stores a set of *items* of the form `(k,v)`, where `k` is called the *key* and is unique and `v` is called the *value* of the item. The dictionary supports three operations: `insert(k,v)` adds a new item, `delete(k)` removes the item with key `k` and `search(k)` returns the item with key `k`, or returns that no such item exists. Common realizations are balanced binary search trees [11, §12] and hash tables [11, §11].

Simplicial complexes by dictionaries. The main data structure of our algorithm is a dictionary D that represents a simplicial complex. Every item stored in the dictionary represents a simplex, whose key is the list of its vertices. Every simplex σ itself stores an dictionary CoF_σ. Every item in CoF_σ is a pointer to another item in D, representing a cofacet τ of σ. The key of the item is a vertex identifier (e.g., an integer) for v such that $\tau = v * \sigma$. Assuming that the size of a dictionary is linear in the number n of stored elements, the size of D is in $O(n\Delta)$, if Δ is the dimension of the represented complex. With the right `search(k)` function and key encoding, each simplex insertion and deletion requires $O(\Delta)$ dictionary operations. In what follows, it is convenient to assume that dictionary operations have unit costs; we multiply the time complexity with the cost of a dictionary operation at the end to compensate for this simplification.

The conversion algorithm. We assume that the tower $\mathcal{T}$ is given to us as a stream where each element represents a simplicial map ϕ_i in the tower: an element starts with a token {`I`, `C`} that specifies the type of the map and ends with the identifiers of the involved vertices. The algorithm outputs a stream of simplices specifying the filtration $\mathcal{F}$: while handling the i-th input element, it outputs the simplices of $\hat{\mathbb{K}}_{i+1} \setminus \hat{\mathbb{K}}_i$ in increasing order of dimension. We use an initially empty dictionary D and maintain the invariant that after the i-th iteration, D represents the active subcomplex of $\hat{\mathbb{K}}_i$, which is equal to $\mathbb{K}_i$ by Lemma 2.

If the algorithm reads an inclusion of a simplex σ from the stream, it simply adds σ to D and writes σ to the output stream. If the algorithm reads a contraction of two vertices u and v, from $\mathbb{K}_i$ to $\mathbb{K}_{i+1}$, we let $c_i = |\hat{\mathbb{K}}_{i+1} \setminus \hat{\mathbb{K}}_i|$ denote the cost of the contraction. The first step is to determine which of the vertices has the smaller closed star in $\mathbb{K}_i$. The size of the closed star of a vertex v could be computed by a simple graph traversal in D, starting at a vertex v and following the cofacet pointers recursively. However, we want to identify the smaller star with only $O(c_i)$ operations. Therefore, we change the traversal in several ways: First of all, observe that $|\overline{\mathrm{St}}(u)| \leq |\overline{\mathrm{St}}(v)|$ if and only if $|\mathrm{St}(u)| \leq |\mathrm{St}(v)|$. Now define $\mathrm{St}(u, \neg v) := \mathrm{St}(u) \setminus \mathrm{St}(v)$. Then, $|\mathrm{St}(u)| \leq |\mathrm{St}(v)|$ if and only if $|\mathrm{St}(u, \neg v)| \leq |\mathrm{St}(v, \neg u)|$,

because we subtracted the intersection of the stars on both sides. Finally, note that $\min\{|\mathrm{St}(u,\neg v)|, |\mathrm{St}(v,\neg u)|\} \leq c_i$. Moreover, we can count the size of $\mathrm{St}(u,\neg v)$ by a cofacet traversal from u, ignoring cofacets that contain v in $O(|\mathrm{St}(u,\neg v)|)$ time. Finally, we count the sizes of $\mathrm{St}(u,\neg v)$ and $\mathrm{St}(v,\neg u)$ at the same time by a simultaneous graph traversal of both, terminating as soon as one of the traversal stops. The running time is then proportional to $2 \cdot \min\{|\mathrm{St}(u,\neg v)|, |\mathrm{St}(v,\neg u)|\} = O(c_i)$, as required. Assume w.l.o.g. that $|\overline{\mathrm{St}}(u)| \leq |\overline{\mathrm{St}}(v)|$. Also in time $O(c_i)$, we can obtain $\mathrm{St}(u,\neg v)$. We sort its elements by increasing dimension, which can be done in $O(c_i + \Delta)$ using integer sort. For each simplex $\sigma = \{u, v_1, \ldots, v_k\} \in \mathrm{St}(u,\neg v)$ in order, we check whether $\{v, v_1, \ldots, v_k\}$ is in D. If not, we add it to D and also write it to the output stream. Then, we output $\{u, v, v_1, \ldots, v_k\}$. At the end of the loop, we wrote exactly the simplices in $\mathbb{K}_{i+1} \setminus \mathbb{K}_i$ to the output stream. It remains to maintain the invariant on D. Assuming still that $|\overline{\mathrm{St}}(u)| \leq |\overline{\mathrm{St}}(v)|$, u turns inactive in $\hat{\mathbb{K}}_{i+1}$. We simply traverse over all cofaces of u and remove all encountered simplices from D.

Complexity analysis. By applying the operation costs on the above described algorithm, we obtain the following statement. Combined with Propositions 4 and 10, it completes the proof of Theorem 1.

▶ **Proposition 11.** *The algorithm requires $O(\Delta \cdot \omega)$ space and $O(\Delta \cdot |\hat{\mathbb{K}}_m| \cdot C_\omega)$ time, where $\omega = \max_{i=0,\ldots,m} |\mathbb{K}_i|$ and C_ω is the cost of an operation in a dictionary with ω elements.*

Proof Sketch. By the above description, the cost of a contraction can be bounded by $O(\Delta(c_i + d_i)C_\omega)$, where c_i is the number of simplices added, and d_i the number of simplices that become inactive in the i-th iteration. Because $\sum c_i$ and $\sum d_i$ are both in $O(|\mathbb{K}_m|)$, the result follows. ◀

Using balanced binary trees as dictionary, we get $C_\omega = O(\Delta \log \omega)$ because comparing two keys costs $O(\Delta)$. Using hash tables, the expected complexity is $C_\omega = O(\Delta)$.

Experimental results. The following tests where made on a 64-bit Linux (Ubuntu) HP machine with a 3.50 GHz Intel processor and 63 GB RAM. The programs were all implemented in C++ and compiled with optimization level –O2.

To test the performance of our algorithm, we compared it to the software Simpers (downloaded in May 2016)[2], which is the implementation of the Annotation Algorithm from Dey, Fan and Wang described in [13]. Simpers computes the persistence of the given filtration, so we add to our time the time the library PHAT (version 1.4.1) needs to compute the persistence from the generated filtration (with default parameters).

The results of the tests are in Table 1. The timings for File IO are not included in the process time of PHAT and Simpers. The memory peak was obtained via the '/usr/bin/time –v' Linux command. The first three towers in the table, data1-3, were generated incrementally on a set of n_0 vertices: In each iteration, with 90% probability, a new simplex is included, that is picked uniformly at random among the simplices whose facets are all present in the complex, and with 10% probability, two randomly chosen vertices of the complex are contracted. This is repeated until the complex on the remaining k vertices forms a $k-1$-simplex, in which case no further simplex can be added. The remaining data was generated from the SimBa (downloaded in June 2016) library with default parameters using the point clouds from [14]. To obtain the towers that SimBa computes internally, we included a print command at a suitable location in the SimBa code.

[2] `http://web.cse.ohio-state.edu/~tamaldey/SimPers/Simpers.html`

Table 1 Experimental results. The symbol ∞ means that the calculation time exceeded 12 hours.

						Alg1 + PHAT			Simpers	
	c	n	n_0	Δ	ω	filtration size	time (s)	mem. peak (kB) Alg1 / total	time (s)	mem. peak (kB)
data1	495	4 833	500	4	2 908	19 747	0.12	4 644 / 7 040	2.49	10188
data2	795	7 978	800	4	4 816	35 253	0.20	5 424 / 10 228	13.97	20308
data3	794	8 443	800	5	5 155	38 101	0.22	5 744 / 10 916	19.29	24 924
GPS	1 746	8 585	1 747	3	1 747	9 063	0.07	4 072 / 5 292	0.35	6 064
KB	22 499	95 019	22 500	3	22 500	133 433	0.50	10 520 / 18 712	2.83	24 460
MC	23 074	143 928	23 075	3	28 219	185 447	0.72	14 636 / 25 272	4.12	26 020
S3	252 995	1 473 580	252 996	4	252 996	1 824 461	10.09	94 020 / 221 636	49.86	239 404
PC25	14 999	10 246 125	15 000	3	2 191 701	12 283 003	135.02	1 029 680 / 2 223 544	∞	–

To verify that the space consumption of our algorithm does not dependent on the length of the tower, we constructed an additional example whose size exceeds our RAM capacity, but whose width is small: we obtained a tower of length about $3.5 \cdot 10^9$ which has a file size of about 73 GB, but only has a width of 367. Our algorithm took about 2 hours to convert this tower into a filtration of size roughly $4.6 \cdot 10^9$. During the conversion, the virtual memory used was constantly around 22 MB and the resident set size about 3.8 MB only, confirming the theoretical prediction that the space consumption is independent of the length.

4 Persistence by Streaming

If the original tower is small, we want to be able to compute its persistence even if the tower is extremely long. So we design here a streaming variation of the reduction algorithm that computes the barcode of filtrations with a more efficient memory use than the standard algorithm. More precisely, we will prove the following theorem:

▶ **Theorem 12.** *With the same notation as in Theorem 1, we can compute the barcode of a tower $\mathcal{T}$ in worst-case time $O(\omega^2 \cdot \Delta \cdot n \cdot \log n_0)$ and space complexity $O(\omega^2)$*

Algorithmic description. The input to the algorithm is a stream of elements, each starting with a token {`Add`, `I`} followed by an identifier which represents a simplex σ. In the `Add` case, this is followed by the identifiers of the facets of σ. The `I` means that σ has become inactive.

The algorithm uses a matrix data type M as its main data structure. We realize M as a dictionary of columns, indexed by a simplex identifier. Each column is a sorted linked list of identifiers corresponding to the non-zero row indices of the column. In particular, we can access the pivot of the column in constant time and we can add two columns in time proportional to the maximal size of the involved lists. There are two secondary data structures that we mention briefly. Firstly, a dictionary where the keys represents row indices r and their corresponding value is the column that has r as pivot. Secondly, a dictionary representing the set of active simplex identifiers, plus a positive/negative flag. It is straight-forward to maintain these structures during the algorithm, and we will omit the description of the required operations.

The algorithm uses two subroutines. The first one, called `reduce_column`, takes a column identifier j as input and iterate through the non-zero row indices of j: if an index i is the index of an inactive and negative column in M, remove the entry from the column j (cf. to "compression" at end of Section 2). Afterwards, while the column is non-empty, and its

pivot i is the pivot of another column $k < j$ in the matrix, add column k to column j. The second subroutine, `remove_row`, takes a index ℓ as input: let j be the column with ℓ as pivot. Traverse all non-zero columns of the matrix except column j. If a column $i \neq j$ has a non-zero entry at row ℓ, add column j to column i. After traversing all columns, remove column j from M.

The main algorithm can be described easily now: if we add a simplex, we add the column to M and call `reduce_column` on it. If at the end of that routine, the column is empty, it is removed from M. If the column is not empty and has pivot ℓ, we report (ℓ, j) as a persistence pair and check whether ℓ is active. If not, we call `remove_row(`ℓ`)`. If the input stream specifies that simplex ℓ becomes inactive, we check whether ℓ appears as pivot in the matrix and call `remove_row(`ℓ`)` in this case.

▶ **Proposition 13.** *The algorithm computes the correct barcode.*

Proof Sketch. First, note that removing a column from M within the procedure `remove_row` does not affect further reduction steps. Then, `remove_row` might also include right-to-left column additions. But we can easily show that a column reduced with right-to-left additions can also be expressed by a sequence of left-to-right column additions, and thus yields the same pivot as in the standard algorithm. ◀

Complexity analysis. We analyze how large the structure M can become during the algorithm. After every iteration, the matrix represents the reduced boundary matrix of some intermediate complex $\hat{\mathbb{L}}$ with $\hat{\mathbb{K}}_i \subseteq \hat{\mathbb{L}} \subseteq \hat{\mathbb{K}}_{i+1}$ for some $i = 0, \dots, m$. Moreover, the active simplices define a subcomplex $\mathbb{L} \subseteq \hat{\mathbb{L}}$ and there is a moment during the algorithm where $\hat{\mathbb{L}} = \hat{\mathbb{K}}_i$ and $\mathbb{L} = \mathbb{K}_i$, for every $i = 0, \dots, m$. We call this the *i-th checkpoint*.

▶ **Lemma 14.** *At every moment, the number of columns stored in M is at most 2ω.*

This come from the fact that, throughout the algorithm, a column is stored in M only if it has an active pivot. The number of rows is more difficult to bound because we cannot guarantee that each column in M corresponds to an active simplex. Still, the number of rows is asymptotically the same:

▶ **Lemma 15.** *At every moment, the number of rows stored in M is at most 4ω.*

Proof Sketch. Consider a row index ℓ and a time in the algorithm where M represents $\hat{\mathbb{L}}$. There are at most 2ω active row indices at any time. Moreover, following the algorithm, ℓ cannot represent an inactive negative simplex neither an active one that was paired with another index. Therefore, we restrict our attention to the remaining case, that ℓ is inactive, positive and has not been paired so far. It is well-known that in this case, ℓ is the generator of an homology class of $\hat{\mathbb{L}}$. Let $\beta(\hat{\mathbb{L}}) := \sum_{i=0}^{\Delta} \beta_i(\hat{\mathbb{L}})$ denote the sum of the Betti numbers of the complex. Then, it follows that the number of such row indices is at most $\beta(\hat{\mathbb{L}})$. We have that $\beta(\hat{\mathbb{K}}_i) = \beta(\mathbb{K}_i)$ by Lemma 3, and since $\mathbb{K}_i$ has at most ω simplices, $\beta(\mathbb{K}_i) \leq \omega$. Since we add at most ω simplices to get from $\hat{\mathbb{K}}_i$ to $\hat{\mathbb{L}}$, and each addition can increase β by at most one, we have that $\beta(\hat{\mathbb{L}}) \leq 2\omega$. ◀

▶ **Proposition 16.** *The algorithm runs in time $O(\omega^2 \cdot \Delta \cdot n \cdot \log n_0)$ with $O(\omega^2)$ space.*

Proof. The space complexity is immediately clear from the preceding two lemmas, as M is the dominant data structure in terms of space consumption. For the time complexity, we observe that both subroutines `reduce_column` and `remove_row` need $O(\omega)$ column additions and $O(\omega)$ dictionary operations in the worst case. A column addition costs $O(\omega)$, and a

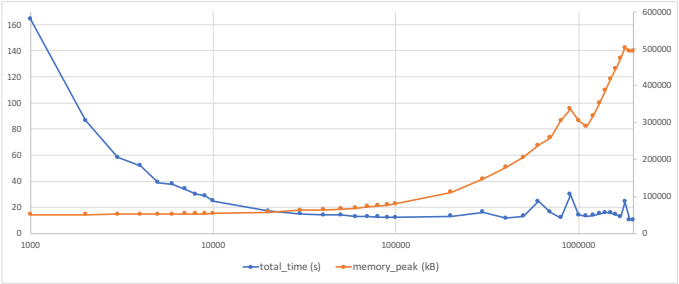

Figure 3 Evolution of processing time (left Y-axis in sec) and process memory peak (right Y-axis in kB) depending on the chunk size (logarithmic X-axis).

dictionary operation is not more expensive So, the complexity of both methods is $O(\omega^2)$. Since each routine is called at most once per input element, and there are $O(\Delta \cdot n \cdot \log n_0)$ elements by Theorem 1, the bound follows. ◀

Implementation. The algorithm just described is not efficient in practice, partially because `remove_row` scans the entire matrix M which should be avoided. We outline a variant that behaves better in practice. The idea is to perform a "batch" variant of the previous algorithm: We define a *chunk size* C and read in C elements from the stream; we insert added columns in the matrix, but not reducing the columns yet. After having read C elements, we start the reduction of the newly inserted columns using the *clearing optimization*: that is, we go in decreasing dimension and remove a column as soon as its index becomes the pivot of another column; see [8] for details. After the reduction ends, except for the last chunk, we go over the columns of the matrix and check for each pivot whether it is active. If it is, we traverse its row entries in decreasing order, but skipping the pivot. Let ℓ be the current entry. If ℓ is the inactive pivot of some column j, we add j to the current column. If ℓ is inactive and represents a negative column, we delete ℓ from the current column. After performing these steps for all remaining columns of the matrix, we go over all columns again, deleting every column with inactive pivot.

How to choose the parameter C? The chunk provides a trade-off between time and space efficiency. Roughly speaking, the matrix can have up to $O(\omega + C)$ columns during this reduction, but the larger the chunks are, the more benefit one can draw from clearing.

Experimental evaluation. The tests were made with the same setup as in Section 3.4. Figure 3 shows the effect of the chunk size parameter C on the runtime and memory consumption of the algorithm. The data used is `S3` (see Section 3.4); we also performed the tests on the other examples from Table 1, with similar outcome. The File IO operations are included in the measurements. Confirming the theory, as the chunk size decreases, our implementation needs less space but more computation time (while the running time seems to increase slightly again for larger chunk sizes).

For the $4.6 \cdot 10^9$-inclusions-tower from Section 3.4, with $C = 200\,000$, the algorithm took around 4.5 hours, the virtual memory used was constantly around 68 MB and the resident set size constantly around 49 MB, confirming the theoretical statement that the memory size does not depend on the length of the filtration.

References

1 U. Bauer, M. Kerber, and J. Reininghaus. Clear and Compress: Computing Persistent Homology in Chunks. In *Topological Methods in Data Analysis and Visualization III*, Mathematics and Visualization, pages 103–117. Springer, 2014.

2 U. Bauer, M. Kerber, and J. Reininghaus. Distributed Computation of Persistent Homology. In *Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 31–38, 2014.

3 U. Bauer, M. Kerber, J. Reininghaus, and H. Wagner. Phat – Persistent Homology Algorithms Toolbox. *Journal of Symbolic Computation*, 78:76–90, 2017.

4 J.-D. Boissonnat, T. Dey, and C. Maria. The Compressed Annotation Matrix: An Efficient Data Structure for Computing Persistent Cohomology. In *European Symp. on Algorithms (ESA)*, pages 695–706, 2013.

5 M. Botnan and G. Spreemann. Approximating Persistent Homology in Euclidean space through collapses. *Applied Algebra in Engineering, Communication and Computing*, 26:73–101, 2015.

6 G. Carlsson. Topology and Data. *Bulletin of the AMS*, 46:255–308, 2009.

7 G. Carlsson, V. de Silva, and D. Morozov. Zigzag Persistent Homology and Real-valued Functions. In *ACM Symp. on Computational Geometry (SoCG)*, pages 247–256, 2009.

8 C. Chen and M. Kerber. Persistent Homology Computation With a Twist. In *European Workshop on Computational Geometry (EuroCG)*, pages 197–200, 2011.

9 C. Chen and M. Kerber. An output-sensitive algorithm for persistent homology. *Computational Geometry: Theory and Applications*, 46:435–447, 2013.

10 A. Choudhary, M. Kerber, and S. Raghvendra. Polynomial-Sized Topological Approximations Using The Permutahedron. In *32nd Int. Symp. on Computational Geometry (SoCG)*, pages 31:1–31:16, 2016.

11 T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to algorithms*. The MIT press, 3rd edition, 2009.

12 V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Dualities in persistent (co)homology. *Inverse Problems*, 27:124003, 2011.

13 T. Dey, F. Fan, and Y. Wang. Computing Topological Persistence for Simplicial Maps. In *ACM Symp. on Computational Geometry (SoCG)*, pages 345–354, 2014.

14 T. Dey, D. Shi, and Y. Wang. SimBa: An efficient tool for approximating Rips-filtration persistence via Simplicial Batch-collapse. In *European Symp. on Algorithms (ESA)*, pages 35:1–35:16, 2016.

15 H. Edelsbrunner and J. Harer. *Computational Topology: an introduction*. American Mathematical Society, 2010.

16 H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological Persistence and Simplification. *Discrete & Computational Geometry*, 28:511–533, 2002.

17 H. Edelsbrunner and S. Parsa. On the Computational Complexity of Betti Numbers: Reductions from Matrix Rank. In *ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 152–160, 2014.

18 M. Kerber. Persistent Homology: State of the art and challenges. *Internationale Mathematische Nachrichten*, 231:15–33, 2016.

19 M. Kerber and H. Schreiber. Barcodes of Towers and a Streaming Algorithm for Persistent Homology. *arXiv*, abs/1701.02208, 2017. URL: `http://arxiv.org/abs/1701.02208`.

20 M. Kerber and R. Sharathkumar. Approximate Čech Complex in Low and High Dimensions. In *Int. Symp. on Algortihms and Computation (ISAAC)*, pages 666–676, 2013.

21 C. Maria, J.-D. Boissonnat, M. Glisse, and M. Yvinec. The Gudhi Library: Simplicial Complexes and Persistent Homology. In *Int. Congress on Mathematical Software (ICMS)*, volume 8592 of *Lecture Notes in Computer Science*, pages 167–174, 2014.

22 C. Maria and S. Oudot. Zigzag Persistence via Reflections and Transpositions. In *ACM-SIAM Symp. on Discrete Algorithms (SODA)*, pages 181–199, 2015.

23 N. Milosavljevic, D. Morozov, and P. Skraba. Zigzag persistent homology in matrix multiplication time. In *ACM Symp. on Computational Geometry (SoCG)*, pages 216–225, 2011.

24 N. Otter, M. Porter, U. Tillmann, P. Grindrod, and H. Harrington. A roadmap for the computation of persistent homology. *arXiv*, abs/1506.08903, 2015.

25 S. Oudot. *Persistence theory: From Quiver Representation to Data Analysis*, volume 209 of *Mathematical Surveys and Monographs*. American Mathematical Society, 2015.

26 D. Sheehy. Linear-size approximation to the Vietoris-Rips Filtration. *Discrete & Computational Geometry*, 49:778–796, 2013.

27 A. Zomorodian and G. Carlsson. Computing Persistent Homology. *Discrete & Computational Geometry*, 33:249–274, 2005.