Persistent Homology Computation with a Twist

Chao Chen* Michael Kerber†

Abstract

The persistence diagram of a filtered simplicial complex is usually computed by reducing the boundary matrix of the complex. We introduce a simple optimization technique: by processing the simplices of the complex in decreasing dimension, we can "kill" columns (i.e., set them to zero) without reducing them. This technique completely avoids reduction on roughly half of the columns. We demonstrate that this idea significantly improves the running time of the reduction algorithm in practice. We also give an output-sensitive complexity analysis for the new algorithm which yields to sub-cubic asymptotic bounds under certain assumptions.

1 Introduction

Persistent homology is a quickly-growing area of research in the analysis of topological spaces. Substantial progress, both theoretical and practical, has been made during the last decade; we refer to [3] for a recent textbook on the topic.

The classical way of computing the persistence of a simplicial complex (or more precisely, a filtration of it) is by reduction [4]: the boundary matrix of the complex is transformed by column operations until each column either turns zero, or has a unique lowest nonzero entry. For a complex with n simplices, the algorithm runs in cubic time in the worst case; an example where this bound is actually achieved has been presented in [6]. However, it has been observed that the algorithm rather behaves linear in practical applications [7, 1].

We present an optimization of the persistence algorithm. The idea is that the reduction of a column that corresponds to a negative simplex (one that destroys an homology class) also reveals the simplex that creates the corresponding class. Since that column is known to be zero after reduction, we can simply set it to zero ("kill it") without applying any column operation on it. We change the order of column reduction in the algorithm to profit from this trick as much as possible.

Our algorithm permits an output-sensitive complexity analysis. Writing d for the dimension of the complex, P for the persistence pairs of the filtration (that is, the indices of creator/destroyer pairs), and E for the set of homology classes of the underlying simplicial complex, we obtain a bound of

$$O\left(d \log n (\#E \cdot \sum_{(i,j)\in P} (j-i) + \sum_{(i,j)\in P} (j-i)^2)\right).$$

Although our variant does not improve the worst case bound (and even worsens it by a factor of $d \log n$), the bound still shows that the algorithm can perform sub-cubic, and even sub-quadratic if the total index persistence, namely, the squared sum of indices differences of persistence pairs, is small.

We further investigate the case of cubical data, which is the common format in computer vision and visualization: the space is a cubical subset of Euclidean space tiled into unit cubes; the filtration is done with respect to a function f that assigns a value to the center of each unit cube. Based on our complexity bound above, we can prove a running time of $O(c^2 n \log n)$ for computing persistence on such data, where c is the number of critical points of f. We also compare the practical performance of our implementation with the classical approach as well as with the sophisticated cohomology algorithm of the Dionysus library and observe a better running time of our optimization for cubical data in $\mathbb{R}^3$.

2 Background

Let $K = \{\sigma_1, \ldots, \sigma_n\}$ denote a simplicial complex of dimension d. We assume an ordering on the simplices such that for each $i \leq n$, $K_i := \{\sigma_1, \ldots, \sigma_i\}$ is a simplicial complex again. The chain $\emptyset = K_0 \subset K_i \ldots \subset K_n = K$ is called a *filtration* of K. Normally, such a filtration is defined according to a function $f : K \to \mathbb{R}$ that orders the simplices of K by function value.

For $0 \leq i \leq n$ and $0 \leq p \leq d$, we denote the p-th homology group of K_i by $\mathsf{H}_p(K_i)$, and we write $\mathsf{H}(K_i)$ for the direct sum of all homology groups of K_i. The rank of $\mathsf{H}_p(K_i)$ is called the p-th Betti number $\beta_p(K_i)$. We obtain K_i by adding the simplex σ_i into K_{i-1}. Denoting d_i as the dimension of σ_i, there are two possible changes of the homology due to the addition of σ_i, either a class of dimension d_i is created or a class of dimension $d_i - 1$ is destroyed. In the former case, we call σ_i a *positive* simplex, or a *creator*; in the latter case, it is called *negative*, or a *destroyer*.

To each negative simplex σ_j, we can associate a unique positive simplex σ_i with $i < j$ such that σ_j

*IST Austria, Klosterneuburg, Austria; Vienna University of Technology, Vienna, Austria, chao.chen@ist.ac.at

†IST Austria, Klosterneuburg, Austria, mkerber@ist.ac.at

destroys the homology class that was created when σ_i was added in the filtration (see [3] for a precise definition). We call (σ_i, σ_j) a *persistence pair* and $j-i$ its *index persistence*. Note that there are simplices that are not paired, namely those (positive) simplices that create homology classes of the simplicial complex K; we call them *essential* simplices. By definition, every simplex of K either belongs to a persistence pair or is essential.

Reduction Algorithm. The boundary of a simplex σ in dimension d_σ is the set of its faces in dimension $d_\sigma - 1$. For $d_\sigma = 0$, the boundary is empty, otherwise, it consists of exactly $d_\sigma + 1$ simplices. The boundary matrix $\partial \in (\mathbb{Z}_2)^{n\times n}$ of a filtered complex K is a $n \times n$ matrix where the j-th column represents the boundary of σ_j, that is $\partial_{i,j} = 1$ if and only if σ_i belongs to the boundary of σ_j. In this case, σ_i must belong to the complex before σ_j is added, so ∂ is an upper-triangular matrix.

For $0 \neq M_j = (m_1, \ldots, m_n) \in \mathbb{Z}_2^n$, we set $\mathrm{low}(M_j) := \max\{i = 1, \ldots, n \mid m_i = 1\}$. For $M_j = 0$, $\mathrm{low}(M_j)$ is undefined. A column operation of the form $M_j \leftarrow M_j + M_k$ is called *reducing* if $k < j$ and $\mathrm{low}(M_j) = \mathrm{low}(M_k)$. A matrix is called *reduced* if no reducing column operation can be performed on it. We call a matrix R a *reduction* of M if R is reduced and arises from a sequence of reducing column operations from M.

A reduction R of the boundary matrix ∂ as above yields the complete information about the persistent homology of the complex. Define

$$\begin{aligned} P &:= \{(i,j) \mid R_j \neq 0 \wedge i = \mathrm{low}(R_j)\} \\ E &:= \{i \mid R_i = 0 \wedge \mathrm{low}(R_j) \neq i \forall j = 1, \ldots, n\}. \end{aligned}$$

Then, $(i,j) \in P$ if and only if (σ_i, σ_j) form a persistence pair of the underlying filtrated complex. Moreover, $i \in E$ if and only if σ_i is essential [3]. In particular, this information does not depend on the choice of the reduction. The simplest way of reducing ∂ is to process from left to right and to reduce a column completely by adding columns from its left (Algorithm 1). A lookup table can be used to identify the next column to be added in constant time. It can be observed immediately that the running time is cubic in n, and cubic running time is indeed necessary for certain input filtrations as presented in [6].

Algorithm 1 Left-to-right persistence computation

```
1: procedure PERSISTENCE_LEFT_RIGHT(∂)
2:     R ← ∂; L ← [0, ..., 0]                    ▷ L ∈ Z^n
3:     for j = 1, ..., n do
4:         while R_j ≠ 0 ∧ L[low(R_j)] ≠ 0 do
5:             R_j ← R_j + R_{L[low(j)]}
6:         if R_j ≠ 0 then L[low(R_j)] ← j
7:     return R
```

3 Reduction by killing

Let $R_j \neq 0$ a column of a reduction of ∂ and $i = \mathrm{low}(R_j)$. Recall that σ_j kills the homology class that is created by σ_i. The key observation exploited in our new algorithm is that in this case, R_i must be zero, since σ_i is in particular a positive simplex. So, we can just set the i-th column of ∂ to zero and have it reduced, without doing any further column operation on it. We say in this case that we *kill* column i.

Reconsidering Algorithm 1, the killing idea does not save any column operations, because whenever a column is identified as positive, it is already reduced due to the left-to-right traversal strategy. Therefore, we change the reduction order of the columns and proceed in decreasing dimensions: setting $d := \dim K$, we first reduce columns that correspond to d-simplices (from left to right), then columns that correspond to $(d-1)$-simplices, and so on. Indeed, if the reduced j-th column has lowest entry i, the corresponding simplex σ_i has dimension $\dim \sigma_j - 1$, so in the moment when ∂_j (namely, the j-th column of ∂) is reduced, ∂_i has not been processed yet. Algorithm 2 summarizes the new method. The dimension of the complex K is passed as a second argument. We also assume that each column stores the dimension of the simplex that it represents in an additional data field. We call this the *simplex-dimension* of a column. By an inductive argument on the simplex dimension, one can prove that Algorithm 2 and Algorithm 1 return the same reduction matrix R.

Algorithm 2 Decreasing dimension persistence computation

```
1: procedure PERSISTENCE_DECR_DIM(∂, d)
2:     R ← ∂; L ← [0, ..., 0]                    ▷ L ∈ (Z)^n
3:     for δ = d, ..., 1 do
4:         for j = 1, ..., n do
5:             if M_j has simplex-dimension δ then
6:                 while R_j ≠ 0 ∧ L[low(R_j)] ≠ 0 do
7:                     R_j ← R_j + R_{L[low(j)]}
8:                 if R_j ≠ 0 then
9:                     i ← low(R_j)
10:                    L[i] ← j
11:                    R[i] ← 0                   ▷ Kill column i
12:    return R
```

4 Analysis

The same analysis as before also shows a worst-case bound of $O(n^3)$ for Algorithm 2. The example from [6] can be adapted to show that cubic running time can also be achieved for this algorithm. However, we will derive a complexity bound that depends on the index persistence of the pairs in the complex, and the number of essential classes of the complex.

We store a matrix $M \in (\mathbb{Z}_2)^{n\times n}$ as an array of

size n, each entry representing a column. A column is stored as a balanced binary search tree storing the indices of nonzero entries as nodes. This way, an operation of the form $M_j \leftarrow M_i + M_j$ can be performed with $O(\#M_i \log(\#M_i + \#M_j)) = O(\#M_i \log n)$ operations, where $\#M_i$ is the number of nonzero entries in column i. Compared to the usual list-representation of columns, the worst case complexity worsens by a logarithmic factor when using trees. However, the complexity of an operation only depends logarithmically on the column to be reduced; this is advantageous when a column accumulates more and more entries by adding small columns during the reduction.

Lemma 1 *Algorithm 2 (as well as Algorithm 1) reduces the j-th column in time $O(\sum_{i=1}^{j-1} \#R_i \log n) = O(\#R \log n)$, where $\#R$ is the total number of nonzero entries in the final reduced matrix R.*

Proof. In the reduction process for a single column, we add only columns from the left to it, and each one at most once. Moreover, all columns which can be added are already reduced. □

The next goal is to bound $\#R$. The crucial observation for that is the following:

Lemma 2 *Let $R_j \neq 0$ be a reduced column and $i = \mathrm{low}(R_j)$. Then, R_j is a linear combination of the columns $\partial_{i+1}, \ldots, \partial_j$. In particular, $\#R_j \leq (d+1)(j-i)$.*

Proof. Assume that the claim is true for any nonzero column with index smaller than j. Initially, R_j is set to ∂_j which is clearly a linear combination. During the reduction, a column R_k is added to R_j only if $k < j$ and also only if $\ell := \mathrm{low}(R_k) > i$, because otherwise $\mathrm{low}(R_j) < i$ at the end. It follows that $i < \ell < k < j$ and so, R_k is by induction a linear combination of $\partial_{\ell+1}, \ldots, \partial_k$. The number of 1's in any column ∂_k is bounded by $d+1$. Thus, only up to $(d+1)(j-i)$ 1's can appear in R_j. □

Corollary 3 $\#R \leq (d+1) \sum_{(i,j)\in P} (j-i)$

We could simply multiply the bound from Lemma 1 by n to obtain an output sensitive bound both for Algorithm 1 and 2. However, we can further improve on this by exploiting that we zero out many columns in Algorithm 2 instead of reducing them. As a first step, we refine the argument from Lemma 2 to derive a more adaptive bound for non-zero columns of R:

Corollary 4 *Let $R_j \neq 0$ be a reduced column and $i = \mathrm{low}(R_j)$. Algorithm 2 (as well as Algorithm 1) reduces ∂_j to R_j in $O(d \log n (j-i)^2)$.*

Proof. Recall from the proof of Lemma 2 that we only add R_k to R_j if $\ell = \mathrm{low}(R_k)$ satisfies $i < \ell < k < j$. So, the number of ones in R_k is bounded by $(d+1)(k-\ell) \leq (d+1)(j-i)$. Since at most $(j-i)$ column operations of that form must be performed for R_j, the bound follows. □

Theorem 5 *Algorithm 2 has a total running time of*

$$O(d \log n (\sum_{(i,j)\in P} (j-i)^2 + \#E \sum_{(i,j)\in P} (j-i))).$$

Proof. Note that the columns of ∂ are in one-to-one correspondence to the simplices of K. It thus makes sense to talk about negative, positive and essential columns of ∂. We divide the columns in three different classes: for negative columns, we can bound the cost by $O(d \log n \sum_{(i,j)\in P} (j-i)^2)$ by Corollary 4. For essential columns, Lemma 1 yields a total cost of $O(\#E \log n \#R) = O(d \log n \#E \sum_{(i,j)\in P} (j-i))$. The remaining columns are positive, but inessential columns. By the killing idea, they are zeroed out before any operation is performed on them, thus, the cost for them is zero. □

5 Cubical Complex

We give an example of how this bound leads to subcubic bounds for special input types. We refer to [3, §VI] for definitions of concepts introduced in this section. Let d be a fixed constant from now and consider a regular cubical grid in d dimensions. We let f denote a function that assign a (different) real value to each grid point. By consistently triangulating each d-cube of the grid without introducing new vertices (e.g., as done in [1]), we can extend f to a piecewise linear function. Depending on its *lower star*, each vertex is either regular or critical; we let c denote the number of critical points of f. We filter the complex according to the *lower star filtration* with respect to f. Due to the cubical structure, the number of simplices in each lower star is bounded by a constant (which is exponential in d).

The final complex is homeomorphic to a d-ball, so there are no essential classes except for the connected component; hence we only have to consider the cost of negative simplices. We distinguish two cases: for persistence pairs that are created and destroyed within the same lower star, the index persistence is bounded by a constant, and since there are up to $n/2$ such pairs, the total cost for them is $O(n \log n)$ with Corollary 4 (with n being the size of the complex). Second, we consider pairs that span over more than one lower star. There are $O(c)$ such pairs, and by Lemma 1 and Corollary 3, we can reduce them in $O(c\#R \log n)$ operations, where $\#R$ is the number of ones in the boundary matrix. Note that pairs that live in only one lower star cause at most a constant number of

ones in R, and pairs that span over more lower stars cause up to n ones in R. This yields a bound of $\#R = O(n + cn)$, and thus a total running time of $O(c^2 n \log n)$ for the reduction algorithm.

Our bound shows that when the complex is subdivided regularly (for instance, by a barycentric subdivision), the running time of computing persistence scales with a $n \log n$ factor which matches general observations in practice. However, our idealistic analysis can only be a first step to investigate this behavior, because introducing noise usually increases the number of critical points in the complex.

6 Experiments

We verify our optimization in practical data. In specific, we focus on cubical data, namely, when the topological space of interest is a subset of Euclidean space (e.g. a hypercube), and the function is sampled uniformly. This class of data is common in image processing and visualization. In specific, we use 3-dimensional data from the *Volvis voxel data repository*[1]. A sample of the results are shown in Table 1.

We triangulate the domain with the *Freudenthal triangulation* [5]. In this case, the lower star of each vertex has a bounded size, which is exponential to the dimension of the domain. Although originally given as $256 \times 256 \times 256$, we downsized the data to $100 \times 100 \times 100$, which leads to simplicial complexes with 25.4 million simplices.

We compare our method (KIL) with two other implementations, the standard reduction algorithm (STA), and the cohomology-reduction algorithm (COH) in the *Dionysus*[2] implementation by Morozov [2]. We compare in terms of both execution time and number of basic add operations. The execution time includes both matrix reduction time and the time for building boundary matrices based on given filtrations. We use `std::vector` instead of a binary search tree to represent each column because the former has shown better practical behavior.

The testing platform of our experiments is a six-core AMD Opteron(tm) processor 2.4GHz with 512KB L2 cache per core, and 66GB of RAM, running Linux. The code runs on a single core.

The experiments show a significant improvement of our method over the standard algorithm and over cohomology reduction. We also notice that the factor of improvement highly depends on the particular input instance. We remark that cohomology reduction is even slower than the standard algorithm in some cubical data. Although cohomology reduction appears to have fewer add operations, it incurs more overhead due to involved data structures in Dionysus.

We conclude by reporting the performance on two non-cubical data by Morozov [2], namely, the alpha complex of uniform samples of a torus embedded in 3-dimensional Euclidean space (0.6 million simplices), and the 4-skeleton of the rips complex of the mumford data (2.4 million simplices). Although our algorithm again improves over the standard persistence algorithm, it is slower than cohomology reduction algorithm on this data. Hence, we observe that the two methods (KIL and COH) have different behaviors over different data; a deeper understanding of this is needed and will be our future work.

Execution time (minute)				
	aneurism	bonsai	foot	skull
KIL	1.02	1.10	1.16	1.45
STA	6.95	7.63	6.54	9.45
COH	1.47	19.76	59.20	95.09
Number of Add Operations (million)				
	aneurism	bonsai	foot	skull
KIL	472.8	104.2	439.9	692.1
STA	24436.8	33664.6	27410.6	66693.3
COH	56.2	3276.1	7064.2	15362.3

Table 1: Performance on cubical data.

	Time(minute)		Add Operations(million)	
	Torus	Mumford	Torus	Mumford
KIL	0.17	0.68	1224.6	1459.8
STA	7.55	0.74	73926.3	1524.5
COH	0.04	0.26	3.8	0.004

Table 2: Performance on non-cubical data.

Acknowledgement

The authors thank Dmitriy Morozov for helpful discussion and answering our questions concerning Dionysus in depth. We thank Herbert Edelsbrunner for helpful comments.

The first author's work is partially supported by the Austrian Science Fund under grant P20134-N13.

References

[1] P. Bendich, H. Edelsbrunner, and M. Kerber. Computing robustness and peristence for images. *IEEE Transactions on Visualization and Computer Graphics*, 16:1251–1260, 2010.

[2] V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Dualities in persistent (co)homology. Manuscript, 2010.

[3] H. Edelsbrunner and J. Harer. *Computational Topology, An Introduction*. American Mathematical Society, 2010.

[4] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28(4):511–533, 2002.

[5] H. Freudenthal. Simplizialzerlegungen von beschränkter Flachheit. *Annals of Mathematics*, 43(3):580–582, 1942.

[6] D. Morozov. Persistence algorithm takes cubic time in the worst case. In *BioGeometry News*. Duke Computer Science, Durham, NC, 2005.

[7] A. Zomorodian and G. Carlsson. Computing persistent homology. *Discrete & Computational Geometry*, 33(2):249–274, 2005.

[1] `http://www.volvis.org/`

[2] `http://www.mrzv.org/software/dionysus/`