Keeping it sparse: Computing Persistent Homology revisited

Department of Mathematics, TUM School of CIT, Technical University of Munich, Germany, Germany

Department of Science and Technology, Linköping University, Sweden

Department of Mathematics, Graz University of Technology, Austria, and Department of Mathematics and Statistics, SUNY - University at Albany, NY, USA

École Polytechnique, France

Department of Mathematics, Graz University of Technology, Austria

Computer Science Department, Ben Gurion University, Israel

Abstract

In this work, we study several variants of matrix reduction via Gaussian elimination that try to keep the reduced matrix sparse. The motivation comes from the growing field of topological data analysis where matrix reduction is the major subroutine to compute barcodes, the main invariant therein. We propose two novel variants of the standard algorithm, called swap and retrospective reductions. We test them on a large collection of data against other known variants to compare their efficiency, and we find that sometimes they provide a considerable speed-up. We also present novel output-sensitive bounds for the retrospective variant which better explain the discrepancy between the cubic worst-case complexity bound and the almost linear practical behavior of matrix reduction. Finally, we provide several constructions on which one of the variants performs strictly better than the others.

Keywords and phrases Barcode algorithm, Topological Data Analysis, Matrix reduction, Sparse matrices

Digital Object Identifier 10.57717/cgt.v3i1.50

Funding *Ulrich Bauer*: DFG Collaborative Research Center SFB/TRR 109 “Discretization in Geometry and Dynamics”
Talha Bin Masood: Swedish Research Council (VR) grant number 2023-04806
Barbara Giunti: Austrian Science Fund (FWF) grant number P 29984-N35 and P 33765-N
Michael Kerber: Austrian Science Fund (FWF) grant number P 29984-N35 and P 33765-N
Abhishek Rathod: DFG Collaborative Research Center SFB/TRR 109 “Discretization in Geometry and Dynamics”

1 Introduction

1.1 Motivation

Persistent homology is arguably the most important tool in the thriving area of topological data analysis. The presence of efficient algorithms for computing the **barcode**, its main invariant, has been an important contributing factor to its success. In the so-called **standard algorithm** [15], this computation boils down to a “restricted” Gaussian elimination of a

© CC-BY
licensed under Creative Commons License CC-BY 4.0

boundary matrix of a filtered simplicial complex: no swapping of rows or columns, only column operations are allowed, and the additions are only from left to right.

Since Gaussian elimination has cubic worst-case complexity in the size of the matrix, barcodes can be computed in polynomial time. This makes it already efficient compared to many other topological invariants, which are usually (NP-)hard to compute or not computable at all. In practice, however, the performance is even better: we observe a close-to-linear practical behavior that allows computing the barcode for matrices with billions of columns [5, 34]. This happens because the matrices to be reduced are **sparse** (that is, the number of nonzero entries per column is a small constant) and tend to remain so during the reduction.

Gaussian elimination on sparse matrices is cheaper because column additions can be performed more efficiently with appropriate choices of sparse matrix data structures [5]. However, the sparsity of the input matrix does not alter the worst-case cubic bounds for matrix reduction. Indeed, carefully crafted constructions force a matrix to become dense in the elimination process, making later column additions expensive [31]. On the other hand, such situations seem to be pathological and do not happen in practice (and also not on average [20]). Therefore, the practical efficiency of reduction procedures can be linked to the preservation of the sparsity of the matrix during the elimination process.

The standard algorithm has been optimized in several ways, exploiting the special structure of boundary matrices. This led to significant further improvements in practice (which are discussed below). However, the impact of sparsity on the reduction has not been adequately studied. We pose the following questions in this paper: are there variants of the standard algorithm, possibly performing different operations, that keep the matrix sparser than the standard one, and do these variants exhibit better practical behavior than existing methods?

1.2 Results

We first observe that some of the restrictions imposed on the Gaussian elimination can be relaxed: any operation preserving the ranks of certain submatrices is allowed (Corollary 1). This observation enables us, for example, to swap certain columns and to perform some right-to-left column operations.

With this insight, we then introduce two new variants of the standard algorithm. The first one, called the **swap algorithm**, introduces one extra rule: before adding a column c_1 to a column c_2, it swaps c_1 and c_2 if c_2 is sparser (i.e., has fewer nonzero entries). Note that checking the size of a column, as well as swapping two columns, requires constant time for most matrix representation types, and hence the overhead of this variant is negligible as long as the column data structure allows for easy retrieval of the size. We show in extensive experimental tests that the swap algorithm is usually competitive with the fastest known algorithms and sometimes leads to significant speedup.

The other variant is called the **retrospective algorithm**. It is based on the (well-known) idea that, once the **pivot** of a column has been found, we can perform additional column operations to further eliminate entries in the column (sometimes, this is called the "full" or the "exhaustive" reduction). The retrospective variant pushes this idea further: it eliminates entries also via right-to-left additions of newly reduced columns. We show significant speedup over the state-of-the-art by experimental comparison for this variant as well. Moreover, the retrospective strategy links the non-zero entries of a column with the (persistent) homology classes at that step, providing complexity bounds that depend on the topology of the underlying data set and are therefore output-sensitive.

For practical efficiency, both variants are combined with existing improvements of the standard algorithm, namely the **clear** and **compress optimization** [3] which typically save a lot of operations in matrix reduction.

We also show that none of the proposed algorithms is strictly better than the others in the following sense. Having chosen one of the three algorithms (standard, swap, and retrospective reduction), we can find a family of inputs for which the chosen algorithm performs a linear number of operations whereas the other two have quadratic complexity.

We also investigate other variants to ensure sparsity. For instance, during the exhaustive reduction, we generate several intermediate columns which are all valid representations for the rest of the algorithm, and we pick the sparsest column among these. Remarkably, whilst this strategy appears to improve on both the standard and the exhaustive variants, in practice it performs worse than both of them. This shows that ensuring sparsity is not the only reason for the good practical performance of computing barcodes.

1.3 Related work

The PHAT library [5] contains a collection of algorithms and matrix representation types to test various approaches for Gaussian elimination on boundary matrices in a unified framework. Our work contributes several new algorithms to the PHAT library. We confirm the earlier observation that the quality of different elimination strategies significantly depends on the chosen data structure.

There are numerous other libraries to compute the barcode; we just refer to some comparative studies [29, 34, 42]. We point out that, in there, all tested libraries include further functionalities, in particular, generating a boundary matrix out of a point cloud, whereas PHAT, as well as the present paper, focuses entirely on the Gaussian elimination step. PHAT is among the most efficient libraries for this substep, as demonstrated, for instance, in [5] and [34].

Even more efficient algorithms have been developed for special cases of simplicial complexes, for instance, Vietoris–Rips complexes [2, 24] and cubical complexes [22, 25, 38, 39]. The improvements in these specialized implementations are not based on optimized reduction strategies such as the ones considered in the present paper. However, not all of the reduction strategies considered here are compatible with these implementations, which is why we focus on the general-purpose framework PHAT for our comparisons. There have also been several approaches that have focused specifically on parallel and distributed computation [3, 4, 28, 40] for performance gains. Another approach, taken for example in [9, 21], is to simplify the filtration without modifying the homology. This approach effectively downsizes the matrix but does not sparsify it.

The best worst-case complexity for computing the barcode is $O(n^\omega)$, where ω is the matrix multiplication constant [30]. However, this approach is not based on Gaussian elimination and is not competitive in practice. There is no sub-cubic complexity bound known for any barcode algorithm based on Gaussian elimination. The output-sensitive bounds that we derive still lead to cubic worst-case bounds, but can be tighter depending on the topological properties of the input. These bounds refine the bound by Chen and Kerber [10].

Computing homology with respect to $\mathbb{Z}$ coefficients requires computing the Smith normal form using integer Gaussian elimination [32, Chapter 1]. In that context, fill-in of matrices is a minor concern, whereas a significant challenge comes from the fact that the size of intermediate entries during matrix reduction can grow exponentially [19].

2 Preliminaries

2.1 Matrix reduction

The algorithms presented in Section 3 can be stated and implemented for other field coefficients with minor changes. However, for simplicity, throughout the paper, we work with $\mathbb{Z}_2$ coefficients. Given a matrix M with entries in $\mathbb{Z}_2$, M^i denotes its i-th row, M_j its j-th column, M^i_j its element in position i, j, and $\#M_j$ the number of nonzero entries in M_j. N denotes the number of columns of M.

The **pivot** of a column, denoted by $\text{piv}(M_j)$, is the (row) index of the lowest nonzero element in M_j. A **left-to-right column operation** is the addition of M_j to M_i with $j < i$. A matrix is **reduced** if all its nonzero columns have pairwise distinct pivots. The process of obtaining a reduced matrix using column additions is called **matrix reduction**. A **pivot pair** is a pair of indices (i, j) such that $i = \text{piv}(M_j)$ in the reduced matrix. Algorithm 1 reduces the columns from left to right in order and is usually referred to as the **standard algorithm** for matrix reduction.

Algorithm 1 Standard reduction

Input: Boundary matrix D
Output: Reduced matrix R
1 $R = D$
2 **for** $j = 1, \ldots, N$
3 **while** $\text{piv}(R_{j'}) = \text{piv}(R_j) \neq 0$ *for* $j' < j$
4 add $R_{j'}$ to R_j

2.2 Filtered simplicial complex and boundary matrix

We apply matrix reduction on a class of matrices that arise from computational topology.

A **simplicial complex** K over a finite set V is a collection of subsets (called **simplices**) of V closed under inclusion, i.e. with the property that if $\sigma \in \mathsf{K}$ and $\tau \subset \sigma$, also $\tau \in \mathsf{K}$. A simplex with $(k+1)$-elements is called ***k*-simplex** and its dimension, $\dim(\sigma)$, is k. The dimension of K is the maximal dimension of its simplices. For $k = 0, 1, 2$ the terms **vertices**, **edges**, and **triangles** are also used, respectively. For a k-simplex $\sigma \in \mathsf{K}$, we call a $(k-1)$-simplex τ with $\tau \subset \sigma$ a **facet** of σ. The set of facets of σ is called its **boundary**.

A **simplexwise filtered simplicial complex** is a sequence of nested simplicial complexes $\emptyset = \mathsf{K}_0 \subseteq \cdots \subseteq \mathsf{K}_N = \mathsf{K}$ such that $\mathsf{K}_i \setminus \mathsf{K}_{i-1} = \{\sigma_i\}$ for all $i = 1, \ldots, N$. We denote the dimension of a simplex σ_i by d_i. The **boundary matrix** D of a filtered simplicial complex is the $(N \times N)$-matrix such that $D^i_j = 1$ if σ_i is a facet of σ_j, and 0 otherwise. In other words, the j-th column of D encodes the boundary of the j-th simplex of the filtration. Note that the boundary of a k-simplex consists of exactly $k+1$ facets, so under the reasonable assumption that the maximal dimension of a simplicial complex is a small constant, D has only a constant number of nonzero entries in each column.

2.3 Persistence pairs

Matrix reduction on boundary matrices reveals topological properties of the underlying filtered simplicial complex. We use standard notations for the necessary concepts that originate from (persistent) homology theory. We also informally describe their topological

meaning, although no deeper understanding of these concepts is required for the results of the paper.

Fixing a filtration boundary matrix D, matrix reduction yields a collection of pivot pairs (i, j). The corresponding pair of simplices (σ_i, σ_j) is called a **persistence pair**. For a persistence pair, $\dim(\sigma_j) = \dim(\sigma_i) + 1$. Informally, the meaning of a persistence pair is that when σ_i is added to the filtered simplicial complex (at step i), it gives rise to a new "hole" in the complex (more precisely, a homology class). This hole disappears when σ_j enters the filtered simplicial complex (e.g., σ_j fills up that hole). For formal definitions of these concepts, see [14, Sec. VII].

Pivot pairs of boundary matrices have special properties that are not true for other types of matrices: first of all, every pivot pair (i, j) satisfies $i < j$, because a filtration boundary matrix is necessarily upper-triangular, and this property is preserved by matrix reduction. Moreover, every index j appears in at most one pivot pair: this is based on the fact that inserting a k-simplex into a simplicial complex either creates a homology class in dimension k or kills a homology class in dimension $k - 1$ [14, Pag. 154, see also Sec. V.4]. This allows us to classify simplices of the filtered simplicial complex into three types: we call a simplex **positive** if it appears as the first entry in a persistence pair, **negative** if it appears as the second entry in a persistence pair, and **essential** if it does not appear in any persistence pair. In topological terms, essential simplices create a hole that is not filled up during the course of the filtration.

In what follows, we blur the difference between pivot pairs (of indices) and persistence pairs (of simplices) and identify σ_i, its index i in the filtration, and the i-th column/row of the filtration boundary matrix. Hence, whenever convenient, we also talk about positive/negative indices and rows/columns.

2.4 Clear and compress

The special structure of boundary matrices allows for simple but effective speedups of matrix reduction. We describe two speedup heuristics that are relevant in this work, discussed extensively in [3]. Both are based on the observation that every index appears in at most one pivot pair.

For the first heuristic, let us fix a negative index i and a column D_j with $D_j^i \neq 0$. It is then easy to see that i cannot become the pivot of D_j during the reduction process (because then, either D_j itself or another column must end up with i as the pivot, contradicting the assumption that i is negative). Hence, we can simply remove the index i from D_j without changing the pivot pairs. We call the process of removing all negative row indices from a column **compressing a column**. For the second heuristic, let us fix a positive index i and consider its column D_i. It can be readily observed that in the reduced matrix, D_i cannot have a pivot because that would imply that i is negative. Therefore, D_i can just be set to zero without changing the pivot pairs. We call this step **clearing a column**.

Note that to make use of clearing, the simplices of the simplicial complex have to be processed in decreasing dimensions; we refer to this variant of matrix reduction with clearing as **twist reduction**; the pseudocode can be obtained from Algorithm 2 by removing Lines 6 and 7. On the other hand, using compression requires proceeding in increasing dimensions, so clear and compress mutually exclude each other, except for more sophisticated approaches [3]. As shown in [5], the twist reduction has a very satisfying practical performance and is the default choice in the PHAT library.

2.5 Column representations

A crucial design choice when implementing matrix reduction is how to store the columns of the matrix. Since boundary matrices are initially sparse, and usually do not fill up too much in the reduction process, a dense vector over $\mathbb{Z}_2$ is a bad choice for its memory consumption. A data structure whose size is proportional to the number of nonzero entries in a column is preferred. A natural choice is to simply store the indices of nonzero entries in a sorted dynamic array (vector). Adding two such columns requires a merge of the two arrays canceling double-occurrences and is therefore proportional to the combined size of both columns. Storing the indices in heap order instead, we can realize the addition of M_i to M_j by inserting every entry of M_i into the heap of M_j which is possible in logarithmic time per entry in M_i. This approach does not eliminate double-occurrences of entries, but their removal can be delayed to a later point when either a pivot is queried or sufficiently many operations have been performed on a column such that a linear scan of the heap is affordable. Using such a "lazy-heap", the amortized cost of adding M_j to M_i is proportional to the size of M_i plus logarithmic overhead. We remark that sorted linked lists and balanced binary search trees can be used instead of vectors and heaps, respectively, to obtain similar performance guarantees in theory, but these data structures suffer from the lack of locality in storing the data which results in many cache misses and makes them significantly slower in most application scenarios.

Furthermore, while storing the non-zero indices of a column as a vector (or heap) is an efficient choice, it can be worthwhile to transform a column to a different data structure when reducing it. For instance, while storing every column as 0-1-vector is prohibitive because of the memory consumption, it can be beneficial to "expand" the dense array of non-zero indices to a long array of 0, and 1, perform column additions on this column, and transform it back into a dense column once the column is reduced. An alternative is to use a lazy heap solely for the column to be reduced, and sorted arrays for all other columns.

The software library PHAT [5] implements the all aforementioned data representations and several additional ones – we refer to the paper for more extensive explanations of the data structures. We emphasize that the performance of matrix reduction depends not only on the reduction algorithm but on combining that algorithm with a suitable column representation; see [5, Tables 1 and 4].

2.6 Dualization

Given a simplex σ, the collection of all the simplices that have σ as a facet is the **coboundary** of σ. As we did for the boundary, we can define the filtration **coboundary matrix**. The crucial observation is that the two matrices are (almost) anti-transposes of each other, and their pivot pairs are in bijection. We do not enter the details here, referring to [12] for the precise statements, but this observation has important consequences in practice. As already observed [5, 34], it is much faster to reduce the coboundary matrix than the boundary matrix for some inputs, in particular, for Vietoris–Rips filtrations. An explanation of this phenomenon is given in [2]. Anti-transposing the matrix is called the **dualization process**, and it adds another degree of freedom when comparing the efficiency of the reductions.

3 Sparsification variants

The Pairing Lemma [14, Pag. 154] shows that the presence of a pivot pair (i, j) is related to an inclusion-exclusion formula of ranks of $D[\geq *, \leq \bullet]$, submatrices of D given by the last $*$

rows of the first $\bullet$ columns. It is usually used to prove the correctness of Algorithm 1, but it is much more general, as it implies:

▶ **Corollary 1.** *Any matrix reduction algorithm that preserves the ranks of the submatrices $D[\geq i, \leq j]$, for all $i, j \in \{1, \ldots, N\}$ is a valid barcode algorithm.*

In other words, any reduction whose operations do not alter the ranks of lower-left submatrices will result in the same barcode decomposition.

Proof. Consider a reduction algorithm satisfying the hypothesis, and assume it obtains the pivot pair (i, j). By the Pairing Lemma, (i, j) is a pivot pair if and only if $\operatorname{rank}(D[\geq i, \leq j]) - \operatorname{rank}(D[\geq i+1, \leq j]) + \operatorname{rank}(D[\geq i+1, \leq j-1]) - \operatorname{rank}(D[\geq i, \leq j-1]) = 1$. Since all these ranks are preserved by the reduction, the claim follows. ◀

Notably, this interpretation of matrix reduction generalizes the common assumption that the reduced matrix R is obtained from the original boundary matrix D by left-to-right column additions, or equivalently, by multiplication with an invertible rank upper-triangular matrix. While this restriction ensures that the reduction data determines a decomposition of the filtered chain complex (see, e.g., [2, 16]), the above observation shows that a weaker condition is sufficient if one is only interested in the barcode itself and not in the representative cycles or cocycles. This insight opens the possibility of many new variants of the barcode algorithm that go beyond the use of left-to-right column additions. We now present some that try to keep the matrix sparse during the reduction.

3.1 Swap reduction

Our first major variant is based on the following simple idea: assume that the standard algorithm adds column R_i to R_j (hence $i < j$ and R_i and R_j have the same pivot). Before doing so, we can check first whether R_j has fewer entries than R_i; in this case, we swap columns R_i and R_j first and perform the addition afterward (which still results in replacing R_j with $R_i + R_j$). This swap is not only profitable in the column additions performed in this step, but also in every later column addition that involves column R_i.

We call this variant the **swap reduction**. This variant appeared in Schreiber's PhD Thesis [36, p. 77] as a tool to control the size of the boundary matrix in a theorem on average complexity of matrix reduction; see also [27]. The first indications of its practical performance appeared in [35]. We also point out that the swap reduction can easily be combined with the clearing optimization; see Algorithm 2 for the pseudocode for this variant.

For correctness, assume that columns $1, \ldots, i-1$ are already reduced and that column i has the same pivot of a previous column, call it i'. Then the ranks of all lower-left submatrices up to i are unchanged if we swap i and i'. The correctness follows from Corollary 1.

3.2 Exhaustive reduction

We review the **exhaustive reduction**, discussed in [17], even if the idea was already present in [18, 41]. The idea is that after the pivot of the reduced matrix has been identified, further (left-to-right) column additions are performed to eliminate nonzero entries with indices smaller than the pivot. Note that this algorithm produces the lexicographically smallest possible representative for the column given by left-to-right column additions. We omit the pseudocode for brevity (see [17]). The exhaustive reduction is combined with the compress-optimization (i.e. removing negative entries from a column before processing

Algorithm 2 Swap reduction

Input: Boundary matrix D of a simplicial complex of dimension d
Output: Reduced matrix R

```
1  R = D
2  for δ = d, d − 1, ..., 0
3      for j = 1, ..., N
4          if R_j has simplex-dimension δ
5              while piv(R_j') = piv(R_j) ≠ 0 for j' < j
6                  if #R_j < #R_j'
7                      swap R_j and R_j'
8                  add R_j' to R_j
9              if R_j ≠ 0
10                 Set R_i to 0 for i = piv(R_j)
```

it) [41]. In this way, the exhaustive reduction guarantees that the number of nonzero entries in R_ℓ after reduction is at most the number of homological classes in K_ℓ.

3.3 Retrospective reduction

Our second major variant is the **retrospective reduction**, based on the idea of using (previous and subsequent) pivots to eliminate entries in a column when it needs to be added. An entry in R_k^i is **unpaired at ℓ** if there does not exist a pivot pair (i, j) with $j \leq \ell$, and **paired at ℓ** otherwise. Whenever we add a column R_ℓ to R_k, we first update R_ℓ by removing through appropriate column additions all entries that have been paired meanwhile. Note that, if the addition of R_m to R_ℓ is needed for this purpose, then R_m has to be updated first, so the step is recursive. The recursion stops because the pivot of a column is strictly decreasing in every recursive call. Since these right-to-left column additions involve only entries whose index is smaller than the pivot in the respective column, none of them changes the rank of any lower-left submatrix. Therefore, this reduction is correct by Corollary 1.

The retrospective algorithm has the property that whenever R_j gets added to another column during iteration k, its size is at most the number of homological classes persisting from j to k (see Lemma 5). That is, it tries to sparsify columns “that matter”, i.e. the columns that get added to other columns.

3.4 Representative cycles

A **representative cycle** is a set of simplices that loop around a hole in the complex, and its computation is often of interest, in addition to the one of the associated persistence pair [2, 13, 23, 29, 33]. In the standard algorithm, at the end of the reduction, the nonzero column providing the persistence pair (i, j) encodes such a representative cycle directly; this is not true in the swap and in the retrospective reductions. However, it holds that during the execution of either algorithm, once the persistence pair (i, j) is identified (i.e., before any swapping of column j or any right-to-left column additions on column j), the column represents a valid representative for the homology class. So, while the representatives are not encoded in the final matrix, they can be stored with little extra effort.

Algorithm 3 Retrospective reduction

```
   Input: Boundary matrix D
   Output: Reduced matrix R
 1 Procedure Main(D)
 2     R = D, P = ∅
 3     for j = 1, ..., N
 4         Remove the negative entries from R_j
 5         Reduce(j)
 6 Procedure Reduce(j)
 7     while ∃ paired entries in R_j
 8         Let ℓ be the largest index for which R_j^ℓ is paired
 9         Add Reduce(P[ℓ]) to R_j
10     if R_j ≠ 0
11         P[Pivot(R_j)] ← j
12     return R_j
```

3.5 Further variants

There are numerous alternatives to obtain reduced columns of (potentially) smaller size. We mention two more variants: recall that the exhaustive algorithm performs a sequence of further column additions after the pivot has been determined. In this process, it computes a sequence of columns $c_1, \ldots, c_r$, all with the same pivot and therefore being valid choices for the reduced matrix. In the **mixed strategy**, we simply remember which column has the smallest size and use its reduced column. Since this variant "locally" improves the size of a reduced column compared to both the standard and exhaustive variant, one could hope that the mixed strategy improves on both of them.

A (perhaps obvious) further variant is to compute the column with the smallest size among all possible alternatives. This problem can be re-phrased as follows. Given a vector W and n vectors $U_1, \ldots, U_n$ in $\mathbb{Z}_2^m$, find $a_1, \ldots, a_n$ in $\mathbb{Z}_2$ such that $W + a_1 U_1 + \cdots + a_n U_n$ has the minimum number of nonzero coefficients. This problem is called the Nearest Codeword Problem (NCW), which is known to be NP-hard and NP-hard to approximate within any constant factor [1]. For completeness, we show a simple reduction from MaxCut.

▶ **Proposition 2.** *NCW is NP-hard.*

Proof. Let $G = (V, E)$ be a graph, with set of vertices $V = (v_1, \ldots, v_n)$ and edges $E = (e_1, \ldots, e_m)$, and $M(G) \in \mathbb{Z}_2^{m \times n}$ its vertex-edge incidence matrix. Set $W = [1, \ldots, 1]$, and $U_1, \ldots, U_n$ as the columns of $M(G)$. Given $a_1, \ldots, a_n \in \mathbb{Z}_2$, let $W' = W + a_1 U_1 + \cdots + a_n U_n$. For $e_j = (v_i, v_k)$, we have that $W'[j] = 0$ if and only if $a_i = 0$ and $a_k = 1$, or $a_i = 1$ and $a_k = 0$. Therefore, setting $A = \{i \colon a_i = 0\}$ and $B = \{i \colon a_i = 1\}$, we have

$$|\{j \colon W'[j] = 0\}| = |\{e = (v_i, v_k) \in E \colon (i \in A \wedge k \in B) \vee (k \in A \wedge i \in B)\}|$$

Thus, maximizing the number of zeros of W' is equivalent to finding a maximum cut of G, so we can reduce MaxCut to NCW, establishing the NP-hardness of NCW [26]. ◀

4 Experiments

We have implemented our new algorithmic variants (**swap**, **retrospective**, and **mix**) as an extension of the publicly available PHAT library [5]. We also implemented the exhaustive reduction for comparison. All our algorithms are implemented such that they can be combined with any of the data structures provided by PHAT (which required minor extensions of the interface), and are included in version 1.7 of the library. Moreover, we included a branch called `keeping_it_sparse_experiments` that contains further modifications and test scripts required to reproduce the experiments of this section. Finally, the datasets that were used in our experiments together with the required scripts to produce them are available in a public repository [8].

We address three questions in our experimental evaluation:

- To what extent do our novel approaches really sparsify the reduced boundary matrix, and does this sparsification lead to a reduction in the number of matrix operations performed?
- What are the most appropriate data structures to represent columns for our novel approaches?
- How do the best combinations perform in comparison with the default options of PHAT?

For our tests, we ran our experiments on a workstation with an Intel Xeon E5-1650v3 CPU and 64 GB of RAM, running Ubuntu 18.04.6 LTS, with gcc version 9.4.0 and optimization flags -O3 -DNDEBUG. The implementation is not parallelized.

4.1 Datasets

We cover different types of filtered simplicial complexes to investigate the performance in a broader context. In particular, we used **Vietoris–Rips filtrations** of high-dimensional point clouds, taken from the benchmark set in [34] (in all cases, we restricted to the 2-skeleton without imposing a limit on the edge length), we generated **alpha shape filtrations** of random points clouds on a cube, a swissroll and a torus (generated with [37]) and **lower star filtrations** generated from publicly available three-dimensional scalar fields [11]. The latter are not simplicial but cubical complexes – all concepts in this paper carry over to this case without difficulty.

We also include the **shuffled filtration**: it is obtained by adding n vertices, then all $\binom{n}{2}$ edges in random order, and finally all $\binom{n}{3}$ triangles in random order. Such filtrations tend to perform significantly more column operations than the standard examples. Therefore, we consider them as a “stress-test” for challenging reduction tasks that have recently shown up in the context of image persistence [7] and two-parameter persistence computation using cohomology [6].

4.2 Sparsity and bitflips

We examine the number of nonzero entries of each variant’s final reduced boundary matrices, called the **fill-in**. Moreover, we count the number of column additions for each variant to see the effect on efficiency. For a more detailed picture, we also count the number of **bitflips** of the algorithm: when adding a column R_k to R_ℓ, the size of R_k equals the number of entries in R_ℓ that needs to be flipped, and the number of bitflips is the accumulated number of such flips over all column additions. We remark that the name “bitflip” is a slight abuse, as it refers to a model where each boundary matrix entry is stored as 1 bit, which is not necessarily what happens in practice, but it is nevertheless a descriptive name, which is why we chose it. We expect the number of bitflips to be a good indicator of the practical

	Cube			Swissroll			Torus		
Algorithm	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips
twist	0.56M	55,613	0.23M	0.62M	40,257	0.15M	1.10M	0.10M	0.50M
twist*	0.95M	45,019	0.66M	1.16M	41,285	0.74M	1.95M	39,186	1.05M
swap	0.56M	55,560	**0.22M**	0.62M	**40,243**	**0.14M**	1.08M	0.10M	**0.47M**
swap*	0.85M	**44,547**	0.54M	1.02M	41,293	0.61M	1.74M	**39,184**	0.84M
retro	0.17M	0.52M	0.76M	0.20M	0.59M	0.87M	**0.31M**	1.03M	1.39M
retro*	**0.16M**	0.38M	0.45M	**0.18M**	0.42M	0.47M	0.36M	0.72M	0.98M
exhaust	0.19M	0.81M	1.42M	0.21M	0.99M	1.69M	0.32M	1.67M	2.68M
exhaust*	0.20M	0.42M	0.60M	0.21M	0.45M	0.56M	0.38M	0.82M	1.22M
mix	0.18M	1.08M	1.97M	0.21M	1.35M	2.41M	0.32M	2.27M	3.89M
mix*	0.20M	0.56M	0.88M	0.21M	0.57M	0.79M	0.38M	3.07M	5.72M

	Random 50			Random 100			Senate		
Algorithm	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips
twist	3,728	0.28M	0.91M	14,975	5.35M	19.02M	15,683	1.66M	5.05M
twist*	62,780	**101**	6,514	0.51M	**222**	38,342	0.54M	**124**	24,729
swap	3,691	0.27M	0.85M	14,887	5.19M	17.29M	15,673	1.66M	5.00M
swap*	62,420	**101**	**6,154**	0.51M	224	**33,830**	0.54M	**124**	**16,655**
retro	**1,274**	57,677	60,124	**5,049**	0.48M	0.49M	**5,355**	0.53M	0.54M
retro*	0.29M	2,466	0.62M	6.41M	9,944	14.32M	1.66M	10,421	3.38M
exhaust	1,326	61,555	67,968	5,172	0.51M	0.55M	5,377	0.54M	0.56M
exhaust*	0.28M	2,503	0.63M	5.36M	10,025	13.67M	1.66M	10,430	3.40M
mix	1,326	64,008	72,874	5,172	0.52M	0.57M	5,377	0.55M	0.59M
mix*	58,693	22,753	1.12M	0.49M	0.22M	21.84M	0.52M	0.19M	18.83M

	Nucleon			Fuel			Tooth		
Algorithm	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips
twist	1.52M	**0.26M**	**1.22M**	30.15M	13.29M	53.28M	34.70M	4.90M	29.43M
twist*	1.55M	0.33M	2.19M	31.56M	13.86M	89.69M	33.18M	4.92M	30.07M
swap	1.51M	0.27M	1.23M	30.14M	13.29M	53.28M	33.61M	**4.83M**	**26.79M**
swap*	1.45M	0.31M	1.71M	24.94M	**13.21M**	78.44M	31.59M	4.86M	27.15M
retro	0.41M	1.04M	2.16M	1.51M	16.73M	49.55M	8.13M	21.51M	34.70M
retro*	**0.30M**	1.00M	1.41M	**1.03M**	16.09M	**29.64M**	**7.35M**	21.94M	31.73M
exhaust	0.59M	1.28M	3.15M	14.89M	30.53M	106.36M	11.07M	27.03M	52.18M
exhaust*	0.53M	1.25M	2.21M	14.31M	29.13M	69.12M	11.05M	28.55M	53.08M
mix	0.46M	10.89M	22.25M	2.25M	111.02M	260.13M	10.85M	120.02M	239.17M
mix*	0.43M	3.79M	7.40M	2.02M	48.46M	108.28M	10.79M	76.24M	150.20M

Table 1 Fill-in, number of column operations and number of bitflips for (top) alpha shapes filtrations of 10000 points, sampled respectively from a cube, a swissroll, and a torus in $\mathbb{R}^3$, generated using [37] (each value is the average of 5 random samplings from each shape); (middle) a Vietoris–Rips filtration up to degree 2 of 50 random points in $\mathbb{R}^{16}$, 100 random points in $\mathbb{R}^4$, and 102 points in $\mathbb{R}^{60}$ of the senate dataset (taken from [34]); (bottom) the lower star filtrations of the nucleon ($41 \times 41 \times 41$ voxels, 68 KB), fuel ($64 \times 64 \times 64$ voxels, 256 KB), and tooth ($103 \times 94 \times 161$ voxels, 1.5 MB) image from [11]. "M" stands for millions, $*$ for the dualized matrix. The best performance in each column is in bold.

performance of an algorithm, as the bulk of the running time of matrix reduction is usually spent on column additions.

	50 points			75 points			100 points		
Algorithm	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips	fill-in	Col.ops	Bitflips
twist	13,933	1.48M	40.61M	56,424	12.13M	794.68M	0.16M	53.91M	6,459.57M
twist*	1.10M	11,510	21.07M	9.58M	58,050	497.59M	44.11M	184,441	4,532.40M
swap	3,799	0.74M	3.41M	8,610	4.36M	28.19M	15,401	15.81M	154.84M
swap*	0.57M	**8,283**	9.06M	5.05M	**39,071**	249.32M	22.49M	**0.12M**	2,334.65M
retro	**1,274**	81,191	**0.28M**	**2,849**	0.34M	**3.28M**	**5,049**	0.97M	**20.89M**
retro*	3.81M	15,563	70.71M	32.70M	76,790	1,582.02M	147.23M	0.25M	13,599.27M
exhaust	12,735	1.10M	22.16M	60,825	9.58M	507.11M	0.19M	44.10M	4,576.32M
exhaust*	1.48M	12,708	39.14M	12.13M	53,649	782.60M	53.92M	0.16M	6,405.81M
mix	8,278	1.38M	23.00M	35,264	11.48M	520.24M	0.11M	51.59M	4,681.67M
mix*	1.07M	30,183	36.07M	9.43M	0.15M	741.30M	43.60M	0.45M	6,151.18M

Table 2 Fill-in, number of column operations, and number of bitflips for the shuffled filtrations on 50, 75, and 100 points. Each value is the average of 5 iterations. "M" stands for millions, $*$ for the dualized matrix. The best performance in each column is in bold.

The choice of the column representation has no influence on this experiment. On the contrary, the number changes dualizing the input matrix. Therefore, we tested each algorithm both on the primal and on the dual matrix.

Tables 1 and 2 show the outcome for one instance per filtration type. We can see that swap reduction consistently leads to smaller reduced matrices and also reduces the number of column operations and bitflips, compared to twist reduction (with the exception of dualized Vietoris–Rips filtrations, where the number of column operations is very small), although the difference is sometimes marginal. For most datasets, we can see that the retrospective, followed closely by the mix, outputs much sparser matrices, with the notable exception of dualized Vietoris–Rips. However, the number of column operations and bitflips generally does not decrease. The exhaustive reduction is also producing similarly sparse matrices, but for shuffled filtrations. However, the number of bitflips seems to be generally higher than for the retrospective. Note that in some examples, the ratio of column operations and bitflips for exhaustive and retrospective is close to one (e.g., in the middle part of Table 1), meaning that the majority of column operations add columns with a single non-zero entry. This shows that the compression strategy is particularly beneficial for these instances. Finally, the numbers indicate that mix reduction is not a successful strategy: even if it manages to obtain sparse reduced matrices, often sparsifying comparably to the retrospective, it requires many more column operations and many more bitflips (to an extent that surprised the authors).

Hence, our novel variants do improve sparsity quite consistently, but this does not automatically lead to improved performances. According to these experiments, there is no direct correlation between sparser matrices and fewer bitflips.

4.3 Data structures

We consider the runtime next. In particular, we look at the influence of the column representation on the performance of the algorithm. For that, we run each of our algorithms with each of the 8 available representations in PHAT (we refer to [5] for an extensive description of the data structures). We show the running times for an alpha shape filtration and a Vietoris–Rips filtration in Table 3, for a lower star filtration and shuffled filtration in Table 4. Note that these tables show only the running time for the matrix reduction; the time to read the input file into memory and to (potentially) dualize the matrix (both of

which usually take more time than the reduction itself) is not shown.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
twist	0.3	0.1	0.2	0.2	0.2	0.2	0.2	**0.1**
twist*	7.2	0.3	0.6	0.4	0.4	0.5	0.3	0.2
swap	0.3	**0.1**	0.2	0.4	0.4	0.2	0.2	0.4
swap*	9.4	0.3	0.6	21.4	10.0	0.5	0.3	5.5
retro	0.7	0.6	0.8	1.0	0.7	0.7	0.7	0.7
retro*	0.7	**0.5**	0.8	0.8	0.6	0.6	0.6	0.6
exhaust	0.7	0.5	0.9	0.8	0.8	0.7	0.7	0.7
exhaust*	0.7	**0.4**	0.7	0.6	0.6	0.6	0.5	0.5
mix	0.9	0.6	1.1	1.0	1.0	1.0	1.0	1.0
mix*	0.8	**0.5**	1.0	0.8	0.8	0.7	0.8	0.8

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
twist	3.9	1.5	3.4	3.8	3.5	3.5	2.2	2.4
twist*	0.7	**0.1**	**0.1**	0.3	**0.1**	0.2	**0.1**	**0.1**
swap	4.0	1.6	3.4	5.1	4.1	3.6	2.5	9.0
swap*	0.8	**0.1**	**0.1**	1.2	0.8	0.2	**0.1**	0.5
retro	1.7	**1.4**	1.9	3.3	2.0	2.5	1.8	2.0
retro*	73.9	6.1	42.8	155.8	191.7	237.3	120.6	38.7
exhaust	1.3	**0.9**	1.5	1.8	1.7	1.4	1.3	1.5
exhaust*	46.0	1.7	12.1	13.5	12.6	11.7	5.7	2.3
mix	1.3	**0.9**	1.4	1.8	1.4	1.4	1.4	1.4
mix*	+5m	186.1	81.1	+5m	+5m	65.0	20.5	+5m

Table 3 (Top) Alpha filtrations on 40,000 points on a cube, average reduction time over 5 random samplings. (Bottom) Vietoris–Rips filtration, 297 points in ambient dim 202 up to degree 2 (celegans dataset from [34]). All timings are in seconds but for the timeout (minutes), $*$ stands for the dualized matrix, and the best runtime(s) per algorithm (both over primal and dual input) is in bold.

First of all, the tables confirm the earlier findings in [5]: the performance of the twist algorithm highly varies depending on the chosen data structure, and the best results are achieved using the A-Bit-Tree representation, which is also the default in PHAT. Moreover, for Vietoris–Rips filtrations, it is highly beneficial to dualize the matrix.

The swap reduction generally performs similarly to twist, working fast with dualization on Vietoris–Rips complexes. Remarkably, it performs much better if run with A-Full on the lower star filtration. Swap is generally slower on the data structures Heap, A-Heap, and A-Bit-Tree. The explanation lies in the way how we use these data structures to represent columns, which does not permit a constant-time access to the size of a column. As the size is queried before every column addition, this results in a considerable slow-down for the swap algorithm.

We observe that the performance for the retrospective algorithm seems more stable across different data structures than for other algorithms on alpha and lower star filtrations. We speculate that the reason for this general stability is the general sparsity of the columns, which reduces the importance of how the entries are stored in memory. We also observe

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
twist	+5m	12.9	3.0	3.0	3.0	3.4	2.3	2.0
twist*	+5m	+5m	4.8	4.8	4.0	4.2	2.5	**1.9**
swap	+5m	12.9	2.8	+5m	+5m	3.4	**2.4**	+5m
swap*	+5m	+5m	4.8	+5m	+5m	4.2	2.6	+5m
retro	5.6	**4.7**	6.2	9.1	6.9	7.2	6.6	7.1
retro*	6.8	4.9	7.1	9.4	7.2	7.7	7.0	7.5
exhaust	5.8	**4.1**	6.7	6.6	6.9	6.6	5.9	6.3
exhaust*	7.8	4.7	8.4	7.1	7.3	7.2	6.3	6.7
mix	14.0	9.0	15.8	15.6	16.2	15.6	13.9	18.6
mix*	14.0	**8.1**	15.2	13.8	13.4	13.0	11.7	14.5

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
twist	28.6	6.8	58.2	59.4	54.8	46.8	6.8	2.1
twist*	47.4	4.4	69.2	55.9	52.6	41.1	4.7	**1.3**
swap	2.8	**0.5**	2.4	6.9	4.9	1.7	0.6	3.4
swap*	28.3	1.9	38.0	86.5	56.0	22.9	6.7	14.2
retro	0.2	**0.1**	0.3	0.7	0.6	0.7	0.4	0.2
retro*	100.1	17.2	257.9	432.7	452.0	420.5	271.6	55.7
exhaust	17.8	4.4	36.6	36.4	33.8	29.3	4.4	**1.4**
exhaust*	51.9	6.5	85.6	87.5	82.7	64.5	8.0	2.2
mix	20.5	4.8	37.8	37.8	35.0	30.3	4.8	**1.7**
mix*	143.4	7.9	117.8	211.9	158.0	65.4	89.2	29.7

Table 4 (Top) Lower star filtration for the tooth image from [11] ($104 \times 91 \times 161$ voxels, 1.5 MB). (Bottom) Average reduction times for 5 iteration of a shuffled filtration on 75 points. All timings are in seconds but for the timeout (minutes), $*$ stands for the dualized matrix, and the best runtime per algorithm (both over primal and dual input) is in bold.

that it sometimes, but not always, improves on the exhaustive algorithm. Remarkably, the retrospective is competitive in practice (for example, on the alpha filtration) even if it is not implemented with the clearing optimization, but uses the compress optimization instead. We are not aware of other variants with this characteristic.

As expected from Tables 1 and 2, the mix algorithm generally has the poorest practical performance among all tested algorithms. We therefore leave it out in further comparisons.

Based on our experiments, we identified that twist works most efficiently in combination with A-Bit-Tree (as previously known), and the retrospective and the exhaustive algorithms work best with the Vector representation but for the shuffled filtrations, where exhaustive should be paired with A-Bit-Tree. The swap reduction sometimes works best with Vector and sometimes with A-Full. Since the advantage of A-Full was generally more significant, we chose A-Full for subsequent experiments.

4.4 Performance on large datasets

We now compare the performance on larger instances. We focus on the combinations that were identified to be most efficient in the previous experiments. We run each algorithm on

the original matrix and its dual and pick the better of the two runtimes.

The results are displayed in Table 5. We observe that retrospective is systematically by 1 or 2 orders of magnitude better than twist for shuffled, and worse by one or two orders of magnitude for Vietoris–Rips filtrations. Also, it is interesting that whenever twist is faster after dualizing, retrospective is faster without dualization and vice versa. We observe that swap is always as fast or slightly slower than the twist on the first three filtrations. It is decidedly faster only on shuffled filtrations; however, it is not as fast as retrospective. Moreover, swap and twist mostly perform better on the same type of matrices: original for alpha shape and lower star filtrations, and dual on Vietoris–Rips. Remarkably, swap performs better on shuffled filtrations without dualization, unlike the twist reduction. Interesting is the case of lower star filtrations: while here it is clear that swap and exhaustive are not the best performing, it is more difficult to choose between twist and retrospective. Indeed, while retrospective is generally better than twist, sometimes the latter takes only half the time. This behavior does not correlate with the input sizes. We theorize that it depends on the number of short-lived bars: when there are more, the sparsification of the retrospective pays off (as it happens in the *bonsai* image), while when there are fewer, like in the *skull* image, it does not. Finally, the exhaustive scales as retrospective on alpha shape filtrations but is slower by two orders of magnitude on the shuffled filtration. Moreover, exhaustive performs slightly better than retrospective on Vietoris–Rips filtrations, even if not as good as swap and twist.

In general, we see that our variants keep the matrix sparse for the shuffled filtration and this leads to a significant improvement in efficiency. Likely, this happens because, in the reduction of random filtrations, each column is added several times. Thus, keeping it sparse pays off.

4.5 Memory consumption

We also tested the memory peak consumption of the algorithms (Table 6). For the alpha shape filtrations, the twist and the swap are better than the retrospective and exhaustive. On the other three filtrations, the exhaustive is generally better than any other algorithms. Twist and swap are quite similar on alpha and Vietoris–Rips filtrations, but swap uses considerably less memory on shuffled, possibly because it does not need to dualize. In these experiments, the memory overhead for dualizing the input matrix is included in these numbers, which might partially explain why the dualized instances usually take more memory.

4.6 Summary of the experiments

The experiments show that, while the sparsity of the matrix is a necessary condition for efficiency, it is, perhaps surprisingly, not always a sufficient one. The general behavior appears to be that the more structured the data, the less it pays off to sparsify.

For highly structured data, such as Vietoris–Rips and alpha filtrations, the price of keeping the matrix sparse exceeds its advantages. Indeed, the only competitive sparsifying algorithm in this situation is the swap algorithm, which does only a few extra operations and does not sparsify aggressively.

On the other hand, for random data such as the shuffled filtration, sparsifying is a winning strategy. In this situation, the retrospective, which tries most aggressively to sparsify the matrix, is by two orders of magnitude faster than the twist, and the swap outperforms the twist by one order of magnitude.

Algorithm	Cube 40K	Cube 80K	Cube 160K	Swissroll 40K	Swissroll 80K	Swissroll 160K	Torus 40K	Torus 80K	Torus 160K
twist+A-Bit-Tree	**0.1**	**0.3**	**0.8**	**0.1**	**0.2**	**0.5**	**0.2**	**0.4**	**0.8**
swap+A-Full	0.2	0.4	0.9	**0.1**	**0.2**	**0.5**	0.3	0.6	1.5
retro+Vector	*0.5	*1.2	*2.7	*0.5	*1.2	*2.7	*1.0	*2.3	*5.4
ex+Vector	*0.4	*1.0	*2.3	*0.4	*1.0	*2.3	*0.8	*1.8	*4.2

Algorithm	Hydrogen	Shockwave	Lobster	MRI Head	Engine	Statue Leg	Bonsai	Skull
twist+A-Bit-Tree	12.3	12.2	**25.8**	**13.8**	54.5	53.9	177.9	**24.2**
swap+A-Full	17.3	15.3	37.8	14.9	74.3	71.0	264.4	25.0
retro+Vector	***10.9**	***10.4**	*30.3	*30.3	***54.4**	***50.8**	***150.6**	*62.4
ex+Vector	*28.6	*21.9	*62.2	*29.3	*239.2	*102.5	*+5m	*58.3

Algorithm	Vietoris–Rips 297	Vietoris–Rips 300	Vietoris–Rips 445	Vietoris–Rips 512	Vietoris–Rips 1000	Vietoris–Rips 1000	Shuffled 100	Shuffled 125	Shuffled 150
twist+A-Bit-Tree	***0.1**	***0.1**	***0.3**	***0.5**	***3.2**	***3.0**	*8.4	*39.2.5	*154.4
swap+A-Full	***0.1**	***0.1**	***0.3**	*0.8	*4.4	*3.6	2.7	8.1	23.4
retro+Vector	1.6	2.1	8.1	15.4	170.7	171.0	**0.3**	**0.9**	**2.9**
ex+Vector/A-Bit-Tree	0.9	1.2	4.8	9.3	121.2	119.5	10.8	50.0	192.8

Table 5 Running times (in seconds) on alpha shape filtrations on 40K, 80K, and 160K points sampled from a cube, a swissroll, and a torus data sets, (top) and various data sets for the lower star, Vietoris–Rips, and shuffled filtrations. The images are: hydrogen ($128 \times 128 \times 128$ voxels, 2 MB), shockwave ($64 \times 64 \times 512$, 2.0 MB), lobster ($301 \times 324 \times 56$ voxels, 5.2 MB), MRI head ($256 \times 256 \times 124$ voxels, 7.8 MB), engine ($256 \times 256 \times 128$ voxels, 8 MB), statue leg ($341 \times 341 \times 93$ voxels, 10.3 MB), bonsai ($256 \times 256 \times 256$ voxels, 16 MB), and skull ($256 \times 256 \times 256$ voxels, 16.0 MB) from [11]. The Vietoris–Rips datasets are the celegans, vicsek 1, house, fractal linear edge, 1000 random points in R^8, and dragon from [34], ordered by their numbers of points displayed at the top. Each value for the shuffled filtration is the average over 5 samplings. The * signals that the running time was achieved using the dual matrix. The best performance per input is highlighted in bold.

Another emerging behavior is that exhaustive performs quite badly across all input data. Possibly, this depends on the fact that while it does many operations, it keeps the matrix less sparse than other sparsifying algorithms. In other words, it pays a price to sparsify but the payoff is lower.

Last but not least, the experiments on the lower star filtrations indicate that the winning strategy is to use either the twist or the retrospective, but there is no emerging trend indicating when to use which. Even more, usually the retrospective is faster, but when it is not it performs much worse than the twist. A possible explanation is that this depends on the number of shorter bars: when there are more, and therefore more computation is required, sparsifying pays off.

5 Output-sensitive bounds

The idea of the retrospective reduction is to keep reducing the columns-to-be-added using the newly found pivots. This, together with the fact that pivots encode information about persistence pairs, allows us to bound the number of bitflips with the persistence Betti

	Cube			Swissroll			Torus		
Algorithm	40K	80K	160K	40K	80K	160K	40K	80K	160K
twist+A-Bit-Tree	0.10G	0.20G	0.40G	0.13G	0.26G	0.54G	0.23G	0.48G	1.01G
swap+A-Full	0.13G	0.25G	0.50G	0.16G	0.32G	0.67G	0.28G	0.60G	1.27G
retro+Vector	*0.18G	*0.35G	*0.71G	*0.23G	*0.47G	*0.97G	*0.40G	*0.86G	*1.86G
exhaustive+Vector	*0.18G	*0.35G	*0.71G	*0.23G	*0.47G	*0.97G	*0.40G	*0.86G	*1.86G

Algorithm	Hydrogen	Shockwave	Lobster	MRI Head	Engine	Statue Leg	Bonsai	Skull
twist+A-Bit-Tree	3.17G	3.20G	6.70G	5.78G	11.88G	13.44G	34.99G	12.02G
swap+A-Full	3.55G	3.56G	7.63G	7.16G	13.32G	15.33G	37.91G	14.94G
retro+Vector	*9.16G	*4.66G	*9.11G	*10.31G	*15.65G	*18.47G	*42.06G	*21.55G
ex+Vector	*2.78G	*2.84G	*6.92G	*10.31G	*10.73G	*13.68G	*26.48G	*21.55G

	Vietoris–Rips						Shuffled		
Algorithm	297	300	445	512	1000	1000	100	125	150
twist+A-Bit-Tree	*0.66G	*0.68G	*2.22G	*3.41G	*25.18G	*25.18G	*0.37G	*1.09G	*2.85G
swap+A-Full	*0.76G	*0.78G	*2.54G	*3.87G	*28.82G	*28.82G	22.51M	40.86M	72.05M
retro+Vector	0.45G	0.46G	1.50G	2.28G	16.93G	16.93G	31.47M	62.66M	118.13M
ex+Vector/A-Bit-Tree	0.35G	0.36G	1.15G	1.75G	13.05G	13.05G	19.17M	33.69M	56.10 M

Table 6 Memory peak consumption for the best-performing combination of algorithms, data structures, and dualization. The * signals that the running time was achieved using the dual matrix. "G" and "M" stand for gigabyte and megabyte, respectively. The inputs are the datasets from Table 5.

numbers. To prove the bounds, we group the column additions into complementary classes: forward/backward, depending on if the addition is left-to-right or right-to-left, and non-/ interval, according to whether the column indices are outside or inside some persistence interval. The first bound is then obtained by counting the bitflips from forward additions directly and the ones from backward using interval additions. The second bound follows by counting the bitflips from non-interval additions directly and the ones from intervals using forward and backward additions.

Let us write P for the set of pivot pairs of a filtered simplicial complex and

$$\overline{P} := P \cup \{(i, N+1) \mid \sigma_i \text{ is essential}\}\,.$$

The **Betti number** β_k for $1 \le k \le N$ is defined as $\beta_k := \#\{(i,j) \in \overline{P} \mid i \le k < j\}\}$. The topological interpretation is that β_ℓ is the number of holes in the complex K_ℓ. The **persistent Betti number** $\beta_{k,\ell}$, for $1 \le k \le \ell \le N$ is defined as

$$\beta_{k,\ell} := \#\{(i,j) \in \overline{P} \mid i \le k,\ j > \ell\}$$

and gives the number of holes that are persistently present in all complexes $\mathsf{K}_k, \ldots, \mathsf{K}_\ell$. In Algorithm 3, R_k is **pivoted** after the procedure `Reduce` is invoked with k as the argument in the **for** loop of the procedure `Main`.

▶ **Observation 3.** *By construction of the* `Reduce` *procedure, for any step ℓ, after any invocation of* `Reduce`(j)*, the entries above* $\operatorname{piv}(R_j)$ *are unpaired at ℓ. In particular, after the first invocation of* `Reduce`(j)*, the number of entries above* $\operatorname{piv}(R_j)$ *is bounded above by β_j.*

The addition of R_k to R_ℓ is called **forward** if $k < \ell$ and **backward** if $k > \ell$. A **forward** (resp. **backward**) **bitflip** is a bitflip resulting from a forward (backward) addition. Given

$(i,j) \in \overline{P}$ and two integers k, ℓ, the bitflip of R_ℓ^i when adding R_k to R_ℓ is an **interval bitflip** if $k, \ell \in \{i+1, \ldots, j\}$, and **non-interval** otherwise. For the retrospective algorithm, the interval bitflips dominate the cost. Moreover, this cost can be bounded in terms of persistence Betti numbers and index persistence.

▶ **Observation 4.** *For every $k = 1, \ldots, N$, the first iteration of* `Reduce`(k) *finds the pivot, and the subsequent iterations perform only backward additions to R_k.*

▶ **Lemma 5.** *If $k < \ell$ and R_ℓ is added to R_k, all the resulting bitflips are interval.*

Proof. By Observation 4, if the addition of R_ℓ to R_k flips the i-th entry, then $i \leq \text{piv}(R_k)$. Thus, $i < k < \ell$. For the other inequality, by Observation 3, it follows that when R_ℓ is added to R_k the entries in R_ℓ are not paired with indices less than ℓ. ◀

▶ **Lemma 6.** *Any two columns are added to each other at most once backward and at most once forward.*

Proof. Fix $1 \leq k < \ell \leq N$. R_k is added to R_ℓ only once when R_ℓ is being pivoted. R_ℓ is added to R_k only if ℓ is paired to j and $R_k^j \neq 0$. Once R_k^j is eliminated from R_k, it is not reintroduced, since j is now paired, and therefore R_ℓ is added to R_k at most once. ◀

▶ **Lemma 7.** *Let (i,j) be a pivot pair. Once R_j is pivoted, $\#R_j$ is always $\leq 1 + \beta_{i,j}$.*

Proof. By Observation 3, immediately after R_j is pivoted, the entries above $i = \text{piv}(R_j)$ are unpaired at j, and hence $\#R_j \leq 1 + \beta_{i,j}$. Subsequently, R_ℓ is added to R_j only if $\ell > j$ and $\text{piv}(R_\ell) < i$. Moreover, when R_ℓ is added to R_j, by Observation 3, the entries above $\text{piv}(R_\ell)$ are unpaired at ℓ. Hence, $\#R_j \leq 1 + \beta_{i,j}$ is maintained in subsequent additions into R_j. ◀

As an immediate corollary, we have:

▶ **Corollary 8.** *Let $(i,j) \in P$. The number of bitflips of an addition to or from R_j is $\leq 1 + \beta_{i,j}$.*

▶ **Proposition 9.** *The total number of bitflips in Algorithm 3 is bounded by either of the following sums:*

$$\sum_{(i,j)\in P} (\beta_{i,j} + 1)^2 + \sum_{k=1}^{N} (d_k + 1)(\beta_k + 1) \; , \tag{1}$$

$$\sum_{(i,j)\in P} (\beta_{i,j} + 1) \cdot \{j - i + 1\} + \sum_{k=1}^{N} (d_k + 1)(\beta_k + 1) \; . \tag{2}$$

Proof. Fix a pivot pair (i,j). By Corollary 8, any additions involving R_j has at most $1 + \beta_{i,j}$ bitflips, giving a multiplicand of $(1 + \beta_{i,j})$ inside the summation for the first term of bounds (1) and (2).

The first term in bounds (1) and (2) is then obtained by bounding the number of backward bitflips in two fashions: by bounding column additions *into* and *from* R_j, respectively.

1. We first bound how many columns are added to R_j. Since backward additions into R_j are executed to zero out a formerly unpaired entry that is now paired, at most $\beta_{i,j} + 1$ entries from R_j need to be zeroed out. Multiplying the bound on the number of added columns to R_j with the bound on the number of bitflips from each column addition, and then summing over all the persistence pairs gives the first term of (1).

2. Next, we bound how many columns R_j is added to. By Lemma 5, R_j is added only to columns R_ℓ for $i+1 \leq \ell \leq j-1$, and by Lemma 6, the backward additions happen at most once. Again, multiplying the bound on the number of added columns R_j is added to with the bound on the number of bitflips from each column addition, and then summing over all the persistence pairs gives the first term of (2).

The second term in both bounds (1) and (2) accounts for forward bitflips. Fix $\ell < k$. By Observation 3, before R_ℓ is added to R_k, $\#R_\ell \leq 1 + \beta_k$. The paired entries in R_k before R_k is pivoted is bounded by $d_k + 1$. Hence, $d_k + 1$ is the maximum number of columns that need to be added to R_k to zero out, and each is added only once (Lemma 6). So, the total number of forward bitflips in R_k is bounded by $(d_k + 1)(\beta_k + 1)$. ◀

▶ **Proposition 10.** *The total number of bitflips required to reduce D in Algorithm 3 is bounded by*

$$\sum_{(i,j)\in\overline{P}} (j-i)^2 \;+\; \sum_{k=1}^{N}(d_k+1)\,.$$

Proof. The two addends are bound, respectively, by the interval and non-interval bitflips.

Let $(i,j) \in \overline{P}$. For $i < k < \ell \leq j$, by Lemma 6, the elements R_k^i and R_ℓ^i are added to each other at most once. So, the total number of forward and backward interval bitflips in row i is bounded by $(j-i)^2/2$ each, and the first term follows.

By Lemma 5, all non-interval bitflips are forward bitflips. By Observation 3, if R_k is added to R_ℓ for $k < \ell$, the entries above $\operatorname{piv}(R_k)$ are not paired until ℓ, and lead to interval bitflips in R_ℓ. As a result of adding R_k, the only non-interval bitflips in R_ℓ occur in the row index of $\operatorname{piv}(R_k)$. Since D is a boundary matrix, at the beginning every column k has at most $d_k + 1$ unpaired entries which need to be zeroed out, proving the claim. ◀

If we restrict the matrix D to the p-simplices of the complex, we obtain the p-dimensional boundary matrix $D^{(p)}$. We then have a finer analysis of Proposition 10. The bound for the number of bitflips required to reduce $D^{(p)}$ is

$$\sum_{(i,j)\in\overline{P},\; d_j=p} (j-i)^2 \;+\; N(p+1)\,.$$

Using Observation 3 and Lemma 5, the maximum number of entries in R_i for $i = 1,\ldots,N$ during the course of the algorithm is $d_i + \beta_i + 1$. Let $\beta_{\max} = \max_{i=1,\ldots,N} \beta_i$. Then the peak memory consumption for Algorithm 3 is bounded by $O(N(\max_i d_i + \beta_{\max}))$.

6 Differentiating examples

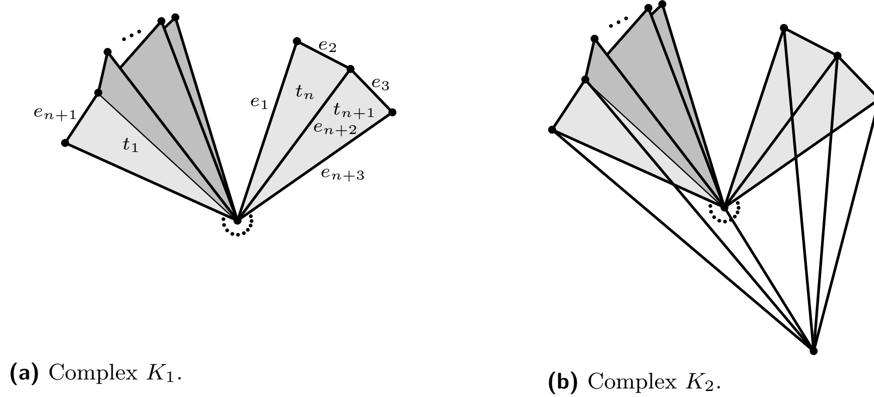
Our experiments have shown that, in practice, retrospective and swap reductions have the potential to run faster than twist. However, there exist constructions for which either of the three mentioned algorithms performs asymptotically better than the other two. Specifically:

▶ **Proposition 11.** *Let $A \in \{twist, swap, retrospective\}$. Then there exists an infinite family of filtered simplicial complexes with increasing size n, such that the number of bitflips for matrix reduction using A is bounded by $O(n)$, where the number of bitflips for the other two algorithms is $\Omega(n^2)$.*

We prove this statement by four constructions of filtered simplicial complexes with the following properties:

- Complex K_1 causes $O(n)$ bitflips for retrospective, and $\Omega(n^2)$ bitflips for twist and swap,
- Complex K_2 causes $O(n)$ bitflips for twist and swap, and $\Omega(n^2)$ bitflips for retrospective.
- Complex K_3 causes $O(n)$ bitflips for swap, and $\Omega(n^2)$ bitflips for twist.
- Complex K_4 causes $O(n)$ bitflips for twist, and $\Omega(n^2)$ bitflips for swap.

The statement follows directly from these 4 constructions.



(a) Complex K_1.

(b) Complex K_2.

Figure 1 Depiction of K_1 and K_2.

Each construction consists of two parts: the complexity study of the reduction of a specific boundary matrix, and the existence of a simplicial complex realizing said boundary matrix.

Existence of simplicial complex K_1. The complex is depicted in Figure 1a. We start with a structure that we call an (open) wheel: it consists of n triangles incident to a **wheel center** vertex such that subsequent triangles share an edge, but the first and last triangles do not share an edge. The edges incident with the wheel center are called **spoke edges**. The two that are part of only one triangle are called the **initial** and the **final** spoke edge, respectively. The edges not-incident with the wheel center are called **tire edges**. We enumerate these edges as follows: first the initial spoke edge, then all the tire edges along the wheel starting with the one adjacent to the initial spoke, and finally all the remaining spoke edges following the wheel starting with the spoke forming a triangle with the initial spoke edge. We then sort the triangles following the wheel starting from the final spoke edge (that is, in the opposite direction w.r.t. the spoke edges). Note that by design, the tire edges are merging components in the filtration and are, therefore negative.

Next, we attach a **fan** of size n to the final spoke edge. This means we introduce n additional vertices $v_1, \ldots, v_n$ and form n **fan triangles**, each joining one v_i with the final spoke edge. The two edges of a fan triangle, not being the final spoke edge, are called **fan edges**. We call the fan edge incident to the center **center fan edge**, and the other one **outer fan edge**. We sort the edges of the filtration by letting the center fan edges come after the tire edges, followed by the outer fan edges, followed by the final spoke edge. Finally, the fan triangles come after the wheel triangles.

Reduction complexity of K_1. This construction yields a filtered simplicial complex whose boundary matrix contains the matrix of Figure 2 as submatrix. Note that Figure 2 is also a submatrix of the construction in [31]. The first half of the matrix contains a “staircase” of columns with decrementing pivots. The staircase is of size 4 in Figure 2 but can easily be extended to an arbitrary n in the obvious way. The second half consists of columns that all have the same pivot, equal to the lowest step of the staircase. When reducing the matrix (using the standard or twist algorithm), the reduction of each column in the second half

$$
\begin{array}{c}
\text{wheel triangles} \quad \text{fan triangles} \\
\left(\begin{array}{cccc|cccc}
* & * & * & * & * & * & * & * \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
* & * & * & * & * & * & * & * \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}\right)
\end{array}
$$

Figure 2 (Sub)matrix of K_1

causes the algorithm to add each column of the first half to it in order. In total, this causes a quadratic number of column operations. The swap algorithm has the same complexity since here no swapping happens.

The retrospective algorithm, however, sparsifies the first column before it gets added to the second half. This simplification requires linear time (by one iteration through the staircase) and results in a unit vector. In all additions to the second half, the cost is therefore constant. This leads to linear complexity. It is important to note that the third nonzero entry for the left-hand-side columns comes from a tire edge, which is negative. Hence, these indices will be removed when reducing wheel triangles. Therefore, ignoring these indices, we observe that indeed the first column turns into a unit vector. Reducing the fan triangles with the retrospective algorithm thus results in removing the final spoke edge, which makes the outer fan edge the pivot, and the reduction of the column stops after one bitflip.

It remains to be argued that the reduction of the edges in the filtration is linear as well for the retrospective reduction. However, this is simple to see, putting an appropriate order of the vertices in the complex. We omit the details.

Existence of simplicial complex K_2. For K_2, we extend the complex K_1 by adding one more vertex, called the **apex**, and connecting it with every vertex of the wheel via an edge (see Figure 1b). We call these edges **apex edges**. We sort the edges of K_2 in the following order: apex edges, center fan edges, initial spoke edge, tire edges, inner spoke edges, outer fan edges, and final spoke edge. The triangles remain in the same order as in K_1.

The two major differences to the situation of K_1 are: the tire edges are now positive edges, so the compression will not remove these entries anymore. Moreover, shifting the outer fan edges later in the filtration creates a block of n edges between the inner spoke edges and the final spoke edge. The filtration boundary matrix therefore looks as depicted in Figure 3, where the just mentioned block is given between the two horizontal lines.

Reduction complexity of K_2. We see now that twist and swap reduction only cause one column addition for every column on the right because the entries in the newly inserted block prevent the algorithm from doing further reductions. Importantly, each column operation only causes a constant number of bitflips, so that the complexity is linear in the end. Again, it can easily be argued that the reduction of the edges for the twist and swap algorithm requires only linear time.

For the retrospective reduction, the addition of the first column to the second half causes a “sparsification”, as in the previous example. However, in this case, this sparsification actually turns the first column into a column with n nonzero entries because it collects all indices of tire edges while iterating through the staircase. Since we then add this column n times (once to every column on the right) and each addition causes n bitflips, we get quadratic complexity.

Reduction complexity of K_3. For K_3, we build up a filtration boundary matrix as depicted in Figure 4. For reference, we call the horizontal blocks in figure block 1 to block 5,

	wheel triangles				fan triangles			
	0	0	0	1	0	0	0	1
	0	0	1	0	0	0	1	0
	0	1	0	0	0	1	0	0
apex edges, ...	1	0	0	0	1	0	0	0
inner spoke edges	0	0	0	1	0	0	0	0
	0	0	1	1	0	0	0	0
	0	1	1	0	0	0	0	0
	1	1	0	0	0	0	0	0
	0	0	0	0	1	0	0	0
outer fan	0	0	0	0	0	1	0	0
edges	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	1
final spoke	1	0	0	0	1	1	1	1

Figure 3 Matrix for K_2 (with $n = 4$)

starting from the bottom.

The twist algorithm applied to this boundary matrix reduces the fifth column by adding all columns on the left to it. This results in a fill-in of row indices in block 4, and the pivot being the unique row index of block 2. Since all columns on the right have the same pivot, the reduced fifth column with n entries is added to every column to the right, resulting in quadratic complexity.

In the swap reduction, the fifth column is reduced in the same way. However, when added to the first column to the right, a swap happens so that in the reduction of the subsequent columns, the sixth column is used. We can observe by the block structure in block 3 that all further columns are reduced after one column addition. Also, column six has only 3 nonzero entries, so the total complexity is linear.

Existence of simplicial complex K_3. To realize the depicted matrix as the boundary matrix of a simplicial complex, we again construct an open wheel. We attach one triangle to the final spoke edge, joining it with a new vertex (represented by the middle column of the matrix). Then, on the edge of that triangle not incident to the wheel center, we attach a fan of n triangles. It is easily possible to sort the edges of this complex in a way that we get the depicted block structure.

	*	*	*	*	*	*	*	*	*
block 5	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
	*	*	*	*	*	*	*	*	*
	0	0	0	1	0	0	0	0	0
block 4	0	0	1	0	0	0	0	0	0
	0	1	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0
	0	0	0	0	0	1	0	0	0
block 3	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	0	0	1	0
	0	0	0	0	0	0	0	0	1
block 2	0	0	0	0	1	1	1	1	1
	0	0	1	1	0	0	0	0	0
block 1	0	1	1	0	0	0	0	0	0
	1	1	0	0	0	0	0	0	0
	1	0	0	0	1	0	0	0	0

Figure 4 Matrix for K_3 (with $n = 4$)

Reduction complexity of K_4. For K_4, we build a boundary matrix as in Figure 5. For general n, the matrix has n columns on the left (group A), n columns on the right (group C), and exactly 3 columns in the middle (group B).

For the twist reduction, the 2-nd column in group B gets reduced with one column addition, resulting in a column with 4 nonzero entries. The 3-rd column in group B then

has the same pivot as the just-reduced column in the middle, and the reduction of the 3-rd column requires the addition of all columns in group A. Still, this process only requires a linear amount of bitflips. All columns in group C get added from the 2-nd column in group B, and because of their entries in the 3-rd row block, the reduction stops after one addition. In total, the twist reduction needs only linear time.

In the swap reduction, the difference is that at the beginning of the reduction of the 3-rd column in group B, a swap happens (as the 3rd column has only 3 nonzero entries, the 2-nd column has 4 entries). That means that the 3-rd column in the middle gets added to all columns in group C. Consequently, for the reduction of every column on the right, the reduction adds all the groups on the left to it, resulting in a quadratic number of column additions.

Existence of simplicial complex K_4. The construction of a complex K_4 that realizes this boundary matrix can be done as follows: Similarly to K_3, we start with an open wheel, attach one new triangle (that is the 3-rd column in group B), and put a fan of n triangles at its outer edge (these are the columns forming group C). To one of these triangles (that is, the 1-st column in group B) we attach another triangle (the 2-nd column in group B). The edge that is shared among the last described triangles is the last row in the matrix. The edges can easily be sorted to yield the matrix of Figure 5.

$$
\begin{array}{cccc|ccc|cccc}
\multicolumn{4}{c}{\text{group A}} & \multicolumn{3}{c}{\text{group B}} & \multicolumn{4}{c}{\text{group C}} \\
\end{array}
$$

$$
\left(\begin{array}{cccc|ccc|cccc}
* & * & * & * & * & * & * & * & * & * & * \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
* & * & * & * & * & * & * & * & * & * & * \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
\hline
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
\hline
0 & 0 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}\right)
$$

Figure 5 Matrix for K_4 (with $n = 4$)

7 Conclusion and Discussion.

In this work, we analyzed how the sparsity of the reduced matrix correlates with the efficiency of the reduction by comparing different algorithms that keep the matrix sparse(r). The experiments show that there is no direct relation, as algorithms resulting in less sparse matrices were faster than others that aggressively sparsify. Nevertheless, the idea of keeping the matrix sparse has led us to novel reduction strategies that improve upon state-of-the-art reductions. Hence, sparsity is an important factor in fast matrix reduction.

The retrospective algorithm often achieves comparable or even better performance than the twist reduction without clearing columns. Specifically, it outperforms all other tested methods for shuffled filtration. Up to our knowledge, this is the first time that a method without clearing has been proven competitive in practice, which is remarkable as the clearing is the standard optimization that consistently leads to improved performances. In our experiments over a wide range of datasets, the retrospective method has regularly low fill-in

compared to the other methods. We believe that the superior performance of the retrospective method is rooted in its sparsity-preserving property.

As indicated in Section 6, there is no strategy that is strictly better than others, so the best choice of reduction for a specific type of input has to be determined by comparison. We have integrated our novel variants into the PHAT library to facilitate further comparisons.

References

1 Sanjeev Arora, László Babai, Jacques Stern, and Z Sweedyk. The hardness of approximate optima in lattices, codes, and systems of linear equations. *Journal of Computer and System Sciences*, 54(2):317–331, 1997.

2 Ulrich Bauer. Ripser: efficient computation of Vietoris–Rips persistence barcodes. *Journal of Applied and Computational Topology*, pages 1–33, 2021.

3 Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological Methods in Data Analysis and Visualization III, Theory, Algorithms, and Applications*, pages 103–117. Springer, 2014.

4 Ulrich Bauer, Michael Kerber, and Jan Reininghaus. Distributed computation of persistent homology. In *2014 proceedings of the sixteenth workshop on algorithm engineering and experiments (ALENEX)*, pages 31–38. SIAM, 2014.

5 Ulrich Bauer, Michael Kerber, Jan Reininghaus, and Hubert Wagner. PHAT–persistent homology algorithms toolbox. *Journal of symbolic computation*, 78:76–90, 2017.

6 Ulrich Bauer, Fabian Lenzen, and Michael Lesnick. Efficient two-parameter persistence computation via cohomology. In *39th International Symposium on Computational Geometry*, volume 258 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 15, 17. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2023.

7 Ulrich Bauer and Maximilian Schmahl. Efficient computation of image persistence. In *39th International Symposium on Computational Geometry*, volume 258 of *LIPIcs. Leibniz Int. Proc. Inform.*, pages Art. No. 14, 14. Schloss Dagstuhl. Leibniz-Zent. Inform., Wadern, 2023.

8 Talha Bin Masood, Barbara Giunti, and Michael Kerber. Benchmark datasets for Keeping it sparse: Computing Persistent Homology revisited, May 2024. `doi:10.3217/hht7z-8ek20`.

9 Jean-Daniel Boissonnat and Siddharth Pritam. Edge Collapse and Persistence of Flag Complexes. In Sergio Cabello and Danny Z. Chen, editors, *36th International Symposium on Computational Geometry (SoCG 2020)*, volume 164 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 19:1–19:15, Dagstuhl, Germany, 2020. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

10 Chao Chen and Michael Kerber. Persistent homology computation with a twist. In *Proceedings 27th European Workshop on Computational Geometry*, 2011.

11 Open Scientific Visualization Datasets. `https://klacansky.com/open-scivis-datasets/`.

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

13 Tamal Krishna Dey and Yusu Wang. *Computational Topology for Data Analysis.* Cambridge University Press, 2022.

14 Herbert Edelsbrunner and John Harer. *Computational topology: an introduction.* American Mathematical Soc., 2010.

15 Herbert Edelsbrunner, David Letscher, and Afra Zomorodian. Topological persistence and simplification. In *Proceedings 41st annual symposium on foundations of computer science*, pages 454–463. IEEE, 2000.

16 Herbert Edelsbrunner and Dmitriy Morozov. Persistent homology. In *Handbook of Discrete and Computational Geometry*, pages 637–661. Chapman and Hall/CRC, 2017.

17 Herbert Edelsbrunner and Katharina Ölsböck. Tri-partitions and bases of an ordered complex. *Discrete & Computational Geometry*, 64(3):759–775, 2020.

18 Herbert Edelsbrunner and Afra Zomorodian. Computing linking numbers of a filtration. *Homology, Homotopy and Applications*, 5(2):19–37, 2003.

19 Xin Gui Fang and George Havas. On the worst-case complexity of integer gaussian elimination. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 28–31, 1997.

20 Barbara Giunti, Guillaume Houry, and Michael Kerber. Average complexity of matrix reduction for clique filtrations. In *Proceedings of the 2022 on International Symposium on Symbolic and Algebraic Computation*, ISSAC '22, pages 1–8, New York, NY, USA, 2022. Association for Computing Machinery.

21 Marc Glisse and Siddharth Pritam. Swap, Shift and Trim to Edge Collapse a Filtration. In Xavier Goaoc and Michael Kerber, editors, *38th International Symposium on Computational Geometry (SoCG 2022)*, volume 224 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:15, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

22 Pierre Guillou, Jules Vidal, and Julien Tierny. Discrete morse sandwich: Fast computation of persistence diagrams for scalar data - an algorithm and A benchmark, 2022. Preprint, available at arXiv:2206.13932.

23 Haibin Hang, Chad Giusti, Lori Ziegelmeier, and Gregory Henselman-Petrusek. U-match factorization: sparse homological algebra, lazy cycle representatives, and dualities in persistent (co)homology, 2021. Preprint, available at arXiv:2108.08831.

24 Gregory Henselman and Robert Ghrist. Matroid filtrations and computational persistent homology, 2016. Preprint, available at arXiv:1606.00199.

25 Shizuo Kaji, Takeki Sudo, and Kazushi Ahara. Cubical ripser: Software for computing persistent homology of image and volume data, 2020. Preprint, available at arXiv:2005.12692.

26 Richard M. Karp. Reducibility among combinatorial problems. *Complexity of Computer Computations*, pages 85–103, 1972.

27 Michael Kerber and Hannah Schreiber. Barcodes of towers and a streaming algorithm for persistent homology. *Discrete & Computational Geometry*, 61(4):852–879, 2019.

28 Ryan Lewis and Dmitriy Morozov. Parallel computation of persistent homology using the blowup complex. In *Proceedings of the 27th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 323–331, 2015.

29 Lu Li, Connor Thompson, Gregory Henselman-Petrusek, Chad Giusti, and Lori Ziegelmeier. Minimal cycle representatives in persistent homology using linear programming: An empirical study with user's guide. *Frontiers in Artificial Intelligence*, 4, 2021.

30 Nikola Milosavljević, Dmitriy Morozov, and Primož Škraba. Zigzag persistent homology in matrix multiplication time. In *Proceedings of the Twenty-Seventh Annual Symposium on Computational Geometry*, SoCG '11, page 216–225, New York, NY, USA, 2011. Association for Computing Machinery.

31 Dmitriy Morozov. Persistence algorithm takes cubic time in worst case. *BioGeometry News, Dept. Comput. Sci., Duke Univ*, 2, 2005.

32 James R Munkres. *Elements of algebraic topology.* CRC press, 2018.

33 Ippei Obayashi. Volume-optimal cycle: Tightest representative cycle of a generator in persistent homology. *SIAM Journal on Applied Algebra and Geometry*, 2(4):508–534, 2018.

34 Nina Otter, Mason A Porter, Ulrike Tillmann, Peter Grindrod, and Heather A Harrington. A roadmap for the computation of persistent homology. *EPJ Data Science*, 6:1–38, 2017.

35 Dominik Schmid. A modification of the persistence reduction algorithm (unpublished). Master's thesis, Graz University of Technology, 2020.

36 Hannah Schreiber. *Algorithmic Aspects in standard and non-standard Persistent Homology.* PhD thesis, Graz University of Technology, 2019.

37 TaDAset - a scikit-tda project. `https://github.com/scikit-tda/tadasets`, 2018.

38 Hubert Wagner. Slice, Simplify and Stitch: Topology-Preserving Simplification Scheme for Massive Voxel Data. In *39th International Symposium on Computational Geometry (SoCG 2023)*, pages 60:1–60:16, 2023.

39 Hubert Wagner, Chao Chen, and Erald Vuçini. *Efficient Computation of Persistent Homology for Cubical Data*, pages 91–106. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

40 Simon Zhang, Mengbai Xiao, and Hao Wang. GPU-accelerated computation of Vietoris–Rips persistence barcodes, 2020. Preprint, available at arXiv:2003.07989.

41 A. Zomorodian and G. Carlsson. Computing persistent homology. *Discrete & Computational Geometry*, 33:249–274, 2005.

42 Matija Čufar and Žiga Virk. Fast computation of persistent homology representatives with involuted persistent homology. *Foundations of Data Science*, 5(4):466–479, 2023.