PHAT – Persistent Homology Algorithms Toolbox

Ulrich Bauer[1], Michael Kerber[2], Jan Reininghaus[1], and Hubert Wagner[3]

[1] IST Austria, Klosterneuburg, Austria
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany
[3] Jagiellonian University, Poland

Abstract. PHAT is a C++ library for the computation of persistent homology by matrix reduction. We aim for a simple generic design that decouples algorithms from data structures without sacrificing efficiency or user-friendliness. This makes PHAT a versatile platform for experimenting with algorithmic ideas and comparing them to state of the art implementations.

1 Introduction

Persistent homology is one of the most widely applicable tools in the emerging field of computational topology. Intuitively, persistent homology tracks the topological features in a growing sequence of shapes. For a comprehensive introduction to the theory and some applications, see [4, 5].

Computing persistent homology for a given data set requires the construction of a *filtered cell complex*, i.e., an ordered list of cells such that every prefix forms a subcomplex. We represent a filtered cell complex by its *boundary matrix*, a square matrix whose indices correspond to the ordering of the cells, and whose entries encode the boundary relation of the complex. Since we only consider homology with coefficients in $\mathbb{Z}_2$, the entries of the boundary matrix are in $\mathbb{Z}_2$. Given a boundary matrix, computing persistent homology amounts to transforming it into a *reduced form* using certain elementary matrix operations, similar to Gaussian elimination. A boundary matrix is called *reduced* iff the column pivots, i.e., the maximal indices of the nonzero column entries, are disjoint.

The purpose of PHAT[4] is to provide a platform for comparative evaluation of new and existing algorithms and data structures for matrix reduction. PHAT is among the fastest codes for computing persistent homology currently available [1] and can be obtained under the GNU Lesser General Public License.

While the worst case computational complexity is cubic for all combinations of algorithms and data structures, the actual running time for real world data sets can differ drastically. We demonstrate that there are three key ideas that can have a tremendous impact on the running time: the *clearing* optimization suggested in [2], the computation of persistent cohomology as proposed in [3], and the use of an efficient data structure for column additions.

[4] `http://phat.googlecode.com`

2 Design

PHAT is a C++ library consisting of about 3200 lines of code. Its main purpose is the computation of the persistent homology of a boundary matrix in an extensible, simple, and efficient way. A boundary matrix can be passed to PHAT through an input file that encodes the boundary in a compact form; we refer to the README file of the library for details on the file format. Internally, the matrix is accessed through a template class called `boundary_matrix` with the following interface:

```
template< typename Representation >
class boundary_matrix  {
    Representation rep;
    int get_max_index( int idx );
    void add_to( int source, int target );
    ...
};
```

A `boundary_matrix` stores an internal object of the supplied `Representation` type and forwards all matrix access and manipulation requests to this object. Furthermore, `boundary_matrix` implements several functions independent of the representation, such as reading from and writing to a file. This way, the required interface of the representation class is kept as small as possible. We remark that dynamic polymorphism would give similar advantages (collecting the interface of `boundary_matrix` in an abstract base class and implementing it in several subclasses). We decided for the templated version for efficiency reasons: in a polymorphic implementation, every call to a matrix operation requires a lookup in the virtual function table, and most of the execution time is in fact spent for such low-level matrix operations.

The main function of the library has the following signature:

```
template< typename ReductionAlgorithm, typename Representation >
void compute_persistence_pairs( persistence_pairs& pairs,
    boundary_matrix< Representation >& boundary_matrix );
```

It takes a `boundary_matrix` as input and transforms it into reduced form. Moreover, it computes the persistence pairs (as pairs of indices) from the reduced matrix and stores them in the container `pairs`. The function has two template parameters: `Representation` defines the data structure for the boundary matrix (see above), and `ReductionAlgorithm` specifies which method is used for matrix reduction. The template class `ReductionAlgorithm` must yield *function objects*, i.e., it must implement the `()` operator. This operator is assumed to take a boundary matrix as an argument and transform it into reduced form. The template parameter `ReductionAlgorithm` thereby specifies *in which order* columns of the matrix are accessed, while the template parameter `Representation` specifies *how* columns of the matrix are accessed. As shown in Section 5, both aspects are equally important for an efficient implementation, and we describe the options provided by the PHAT library below.

3 Algorithms

We use the following notation throughout this paper. The *pivot index* of a column in the matrix is the largest index of any nonzero entry. All our reduction algorithms perform left-to-right column additions until no two columns have the same pivot index. A matrix with this property is called *reduced.*

PHAT 1.4 provides five different choices of reduction strategies, two of which have a parallel implementation using OpenMP.

Sequential algorithms. The `standard` algorithm for reducing boundary matrices [6] traverses the columns from left to right and reduces a column completely (through left-to-right column additions) before proceeding with the next one. The reduction of a column is complete once either its pivot index does not appear as a pivot index in a previous column or the column becomes zero.

The algorithm `twist` [2] is based on the standard algorithm and exploits the observation that a column will eventually be reduced to an empty column if its index appears as the pivot of another column. By reducing columns in decreasing order of the dimensions of the corresponding cells, we can explicitly *clear* the columns corresponding to pivot indices. Since the omitted column operations often constitute the bulk of the column operations in the standard algorithm, the clearing optimization can have a tremendous impact on practical performance. It is therefore also used in all other algorithms described below. Due to its simplicity and efficiency, the `twist` algorithm is the default in PHAT.

The idea behind the `row` algorithm [3] is to traverse the columns from right to left. Whenever the pivot of a newly inspected column A equals the pivot of a column B to its right, we add A to B.

Parallel algorithms. The algorithm `spectral_sequence` decomposes the boundary matrix into blocks and processes them in diagonals from the main diagonal outward. The reduction of the individual blocks in a single diagonal is then independent and can be performed in parallel.

The `chunk` algorithm [1] begins with the reduction of two diagonals of blocks, as in the spectral sequence algorithm. In a second step, it simplifies the partially reduced boundary matrix by eliminating the indices of the already found pairs from the matrix. Finally, the simplified matrix is reduced using the twist algorithm. The chunk algorithm is a generalized version of the approach in [8] to compute persistence using discrete Morse theory [7]. The first two steps of the algorithm be run in parallel.

Dualization. Every algorithm for persistent homology also yields an algorithm for persistent cohomology by applying it to the corresponding coboundary matrix. This matrix is given by the *anti-transpose* of the boundary matrix D, obtained by swapping $D_{i,j}$ with $D_{n+1-j,n+1-i}$. Reducing the coboundary matrix yields the same persistence pairs, up to reindexing. In some cases, this *dualization* yields significant speed-ups [3], in particular for Vietoris–Rips complexes. PHAT contains a method to anti-transpose a boundary matrix, thus providing the option to dualize every algorithm proposed in this section.

4 Data Structures

All boundary matrix data structures currently implemented in PHAT use a `vector` containing the individual columns. The column type is defined by the representation. There are two kinds of representations in PHAT, direct and accelerated. A direct representation makes use of a single column type for storage and computation, whereas an accelerated representation provides an additional type that is optimized for fast column additions. To simplify notation, we refer to the number of nonzero entries of a column as its *length*.

Direct representations. The class `vector_list` represents a column as a doubly-linked list (`list<int>`) storing the indices of nonzero entries in increasing order, as suggested in [5]. Adding two columns of lengths k and m can therefore be performed in time $O(k + m)$ by computing the symmetric difference of the lists. The pivot of a column can be found in $O(1)$ by querying the last element of the list. The representation `vector_vector` is analogous, using a dynamically growing array (`vector<int>`) instead of a linked list. This representation is more machine friendly, since it makes use of a contiguous memory region. However, both representations have the disadvantage that column additions are expensive when a small column is added to a large column.

An alternative representation is `vector_set`, where columns are stored as balanced binary search trees (`set<int>`). Adding a column A of length k to a column B of length m can be performed as follows. We iterate through the entries of A, removing the entry from B if it is already present, and inserting it otherwise. The complexity of such an addition is $O(k \log(k + m))$, which can be much better than $O(k + m)$ when $k \ll m$. The pivot of a column can be found in $O(1)$.

The representation `vector_heap` combines the advantages of contiguous storage and efficient column addition. Columns are again stored as `vector<int>`, but the indices are now arranged in heap order. Adding a column A of length k to a column B of length m can be lazily performed by inserting the indices of A into B in amortized time $O(k \log(k + m))$. This implies that an index may temporarily appear multiple times in the heap. The symmetric difference operation is delayed until a certain number of insertions is exceeded or the content of the column is queried. This allows for the pivot of a column of length k to be found in amortized time $O(1)$.

Accelerated representations. Similar to `vector_heap`, accelerated representations also aim at combining contiguous storage with efficient column additions. An accelerated representation is a subclass derived from `vector_vector` that contains an instance of a specialized column type, supporting fast column additions and pivot queries. This column is used as a cache for the *active column*, i.e., the last column modified by the algorithm. For a net gain in performance, efficient conversion between `vector<int>` and cache type, column additions, and pivot queries are required. Moreover, the employed algorithm needs to exhibit a cache-friendly access pattern. This is the case for all algorithms of Section 3 except from the `row` algorithm.

A simple yet efficient choice for the column cache is `set<int>`. The resulting representation is called `sparse_pivot_column`. Another option is the use of a heap as explained in the description of `vector_heap`. The corresponding representation is called `heap_pivot_column`.

Alternatively, `full_pivot_column` uses a bit array of size n to store the entries of the column explicitly. Adding a column of length k to this representation still takes time $O(k \log(k + m))$ due to some additional structure required to quickly access the pivot of a column. We store all modified indices in a heap called *history*, and extract elements from the history until we find an index with nonzero column entry in the bit array. To ensure that no element is inserted into the history multiple times, we also maintain a bit array representing its content. When converting the bit array back into a `vector<int>`, we repeatedly extract and remove the pivot in order to re-initialize it to zero for further use.

The `bit_tree_pivot_column` representation stores the column explicitly in a tree structure. Internally, an 64-ary tree is used, which is encoded implicitly in a bit array. It not only supports fast insertions, deletions and lookup of entries, but also maximum, minimum, successor, and predecessor queries, all in time $O(\log n)$. Furthermore, the structure can be traversed and cleared in time proportional to the number of nonzero entries.

5 Experiments

To evaluate the performance of the different algorithms and data structures, we perform computational experiments using two data sets. The running times are measured on a workstation with two Intel Xeon E5645 CPUs using the integrated benchmark utility of PHAT v1.4. All data sets are available on the project homepage.

The first data set is the 3-skeleton of the Vietoris-Rips filtration of a point cloud generated by a uniform random sample of the 2-sphere. Using 64 points, the resulting boundary matrix has 679,120 columns and 2,670,528 nonzero entries. The running times in seconds for the matrix reduction of this data set are shown in Table 1, where we denote the dualized algorithms by $(\cdot)^*$. It can be observed that the combination of dualization with algorithms employing the clearing optimization leads to drastically shorter running times compared to other choices. To admit a meaningful comparison for these fast algorithms, we repeat the experiment using 192 points, resulting in a boundary matrix with 56,050,288 columns and 223,002,432 nonzero entries. The running times in Table 2 show that the more sophisticated data structures significantly improve the running time. Moreover, the simple sequential `twist` algorithm is about as fast as the parallel algorithms on this example.

The second data set is a lower-star Morse filtration of a cubical complex generated from a 3D image that indicates separation behavior in a vector field [9]. Using a 64^3 sub-region of the image, we get a boundary matrix with 2,048,383

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
standard	15.5	2.7	6.5	5.6	5.2	7.7	2.3	1.6
standard*	2353.4	160.3	15.9	13.4	13.5	15.1	4.1	0.6
twist	15.1	2.4	6.4	5.7	5.3	7.1	2.0	1.4
twist*	0.2	0.0	0.0	0.0	0.0	0.0	0.0	0.0
row	44.1	4.5	19.0	7.4	20.7	34.3	15.4	13.4
row*	0.2	0.1	0.1	0.1	0.1	0.1	0.1	0.0
chunk	0.6	0.2	0.5	0.5	0.3	0.5	0.2	0.2
chunk*	2.8	0.3	0.1	0.1	0.1	0.1	0.1	0.0
spectral sequence	9.7	1.7	4.1	3.3	3.4	4.2	1.5	1.1
spectral sequence*	0.3	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Table 1. Running times (in seconds) for the 3-skeleton of the Vietoris-Rips filtration of 64 points on a 2-sphere, using different combinations of algorithms and data structures. The prefix "A-" refers to accelerated representations, while $(\cdot)^*$ denotes dualization.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
twist*	2635.4	339.9	4.9	2.0	2.5	6.1	2.1	1.0
row*	2842.3	434.6	5.6	4.0	59.4	107.9	45.9	17.3
chunk*	24391.6	3276.2	25.2	14.2	14.0	20.7	8.7	4.0
spectral sequence*	2644.8	349.2	5.2	1.9	3.3	6.6	3.1	1.0

Table 2. Running times for the 3-skeleton of the Vietoris-Rips filtration of 192 points on a 2-sphere. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
standard	144.8	15.2	27.9	19.5	16.6	18.6	13.6	9.5
standard*	461.0	39.1	33.1	22.3	22.1	21.4	12.4	14.9
twist	9.6	0.6	0.4	0.1	0.1	0.1	0.1	0.1
twist*	343.0	19.1	1.0	0.5	0.6	0.7	0.2	0.2
row	10.3	1.1	0.4	0.4	1.5	2.1	1.2	0.7
row*	339.9	32.9	1.1	1.0	27.0	44.6	17.8	7.5
chunk	1.8	0.2	0.2	0.1	0.1	0.1	0.1	0.1
chunk*	5.7	0.5	0.3	0.2	0.2	0.2	0.2	0.2
spectral sequence	9.6	0.8	0.2	0.1	0.1	0.1	0.1	0.1
spectral sequence*	338.1	21.4	0.9	0.7	0.8	0.8	0.3	0.1

Table 3. Running times for a 64^3 image data set. See Table 1 for details.

	List	Vector	Set	Heap	A-Heap	A-Set	A-Full	A-Bit-Tree
twist	2080.2	101.7	26.4	11.3	11.1	12.3	10.4	8.8
row	3201.8	210.4	52.1	32.5	1778.7	3437.1	1244.1	734.3
chunk	894.5	156.1	9.8	6.5	6.3	6.2	5.6	4.7
spectral sequence	1197.7	261.7	11.9	8.5	8.5	8.4	7.1	6.1

Table 4. Running times for a 256^3 image data set. See Table 1 for details.

columns and 6,096,762 nonzero entries. The running times for this data set are shown in Table 3. We observe that homology computation is generally faster than cohomology computation for this data set, and the clearing optimization is again crucial for a fast algorithm. To investigate the performance behavior further, we also apply a subset of the algorithms to the full data set consisting of 256^3 voxels – the corresponding boundary matrix has 133,432,831 columns and 399,515,130 nonzero entries. The results in Table 4 again demonstrate the usefulness of the complex data structures introduced in Section 4. In contrast to the first data set, the parallel algorithms can outperform the sequential methods in this case.

Acknowledgements

This research is partially supported by the Toposys project FP7-ICT-318493-STREP, the Max Planck Center for Visual Computing and Communication, and the Foundation for Polish Science IPP Programme *Geometry and Topology in Physical Models.*

References

1. U. Bauer, M. Kerber, and J. Reininghaus. Clear and compress: Computing persistent homology in chunks. In *Topological Methods in Data Analysis and Visualization III*, Mathematics and Visualization, pages 103–117. Springer, 2014.
2. C. Chen and M. Kerber. Persistent homology computation with a twist. In *27th European Workshop on Computational Geometry (EuroCG)*, pages 197–200, 2011.
3. V. de Silva, D. Morozov, and M. Vejdemo-Johansson. Dualities in persistent (co)homology. *Inverse Problems*, 27(12):124003+, 2011.
4. H. Edelsbrunner and J. Harer. Persistent homology — a survey. In *Surveys on Discrete and Computational Geometry: Twenty Years Later*, Contemporary Mathematics, pages 257–282. 2008.
5. H. Edelsbrunner and J. Harer. *Computational Topology. An Introduction.* American Mathematical Society, 2010.
6. H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. *Discrete & Computational Geometry*, 28(4):511–533, 2002.
7. R. Forman. Morse theory for cell complexes. *Advances in Mathematics*, 134(1):90–145, 1998.
8. D. Günther, J. Reininghaus, H. Wagner, and I. Hotz. Efficient computation of 3D Morse –Smale complexes and persistent homology using discrete Morse theory. *The Visual Computer*, 28(10):959–969, 2012.
9. J. Kasten, J. Reininghaus, W. Reich, and G. Scheuermann. Toward the extraction of saddle periodic orbits. In *Topological Methods in Data Analysis and Visualization III*, Mathematics and Visualization, pages 55–69. Springer, 2014.