COMPUTING PERSISTENT HOMOLOGY OF DIRECTED FLAG COMPLEXES

DANIEL LÜTGEHETMANN, DEJAN GOVC, JASON P. SMITH, AND RAN LEVI

ABSTRACT. We present a new computing package FLAGSER, designed to construct the directed flag complex of a finite directed graph, and compute persistent homology for flexibly defined filtrations on the graph and the resulting complex. The persistent homology computation part of FLAGSER is based on the program RIPSER [2], but is optimised specifically for large computations. The construction of the directed flag complex is done in a way that allows easy parallelisation by arbitrarily many cores. FLAGSER also has the option of working with undirected graphs. For homology computations FLAGSER has an Approximate option, which shortens compute time with remarkable accuracy. We demonstrate the power of FLAGSER by applying it to the construction of the directed flag complex of digital reconstructions of brain microcircuitry by the Blue Brain Project and several other examples. In some instances we perform computation of homology. For a more complete performance analysis, we also apply FLAGSER to some other data collections. In all cases the hardware used in the computation, the use of memory and the compute time are recorded.

1. INTRODUCTION

In an ongoing collaboration with the Blue Brain Project [5] and the Laboratory for Topology and Neuroscience [15], we study certain ordered simplicial complexes (Definition 2.3) arising from directed graphs that model brain microcircuitry reconstructions created by the Blue Brain Project team. These complexes, which we refer to as *directed flag complexes*, generalise the usual concept of the flag complex (a.k.a. clique complex) associated to a graph, and prove particularly useful in studying directed networks in general and neural networks in particular.

Over the past decade, homology and persistent homology – particularly in the form of persistent homology of Vietoris-Rips complexes of point clouds – entered the field of data analysis. Therefore, its efficient computation is of great interest, which led to several computational tools like GUDHI [17], PHAT [4] and quite a number of other packages. However, working with networks, and specifically with neural networks at this order of magnitude and level of detail, present certain challenges that can not be addressed by existing software.

One immediate problem is that network data is typically given in the form of a graph, rather than a point cloud. For instance, data emerging from the Blue Brain Project is typically encoded as a weighted directed graph. Since neurons and synaptic connections between them are modeled with great precision, if one wishes to exploit the full potential in the data, then considering the direction and weight of connections between nodes (neurons) is natural and in fact essential. To date we are not aware of any highly efficient software packages that are designed to work with directed graphs. The very impressive GUDHI package, for instance, is designed to work with simplicial complexes. The directed flag complex and numerous other topological constructions one may associate with a directed graph are not simplicial complexes. Hence to deal with networks that are directed by nature, one would need software capable of efficiently constructing various complexes one can associate with a directed graph. Furthermore, since the direction of connection does play an important role in the analysis, such software should be capable of incorporating information about direction as part of the data to be processed, beyond the mere construction of the complex.

A second issue arises from the size of the networks in question. In our earlier work we considered networks in the order of magnitude of thirty thousand vertices and eight million directed edges (reconstructed neocortical column of a 14 days old rat [6]). Software was designed as part of the work that lead to the publication [19], and is freely available [9]. However, at that time the magnitude of the

resulting complexes - typically 6-dimensional with about 160 million simplices across all dimensions - limited our ability to conduct certain deeper investigations, as even the construction of the complexes required a considerable number of compute hours on a large HPC node, and homology computations with the full complex were completely out of reach. More recent reconstructions by the BBP team produced much larger and more complex models (See for instance the model of the full mouse cortex [7]). While advanced computational packages such as GUDHI and PHAT are perfectly capable of dealing with very large simplicial complexes due to the reduction techniques they use, they too fail to produce satisfactory results on networks of the size we are interested in, within reasonable compute time.

Finally, we required a package that will allow us to flexibly associate weights to simplices in our complexes so that we can examine data with respect to a variety of filtrations. Inspired by our main object or study - neural networks - we had to take into account the fact that neurons have a very rich structure, as do connections between them. Hence associating weights to vertices and edges in the representing graph, and furthermore, using those values to induce meaningful weights on higher dimensional simplices typically involves non-obvious choices and implementations that are not catered for in existing software packages.

In this article we present a new software package – FLAGSER – that was developed as part of an EPSRC funded project, *Topological Analysis of Neural Systems*. FLAGSER and its derivatives (see [21]) address the issues listed above, and shows remarkable performance in many relevant cases. While FLAGSER is designed specifically for dealing with the kind of constructions and high performance computations required in this project, it is potentially applicable in a variety of other areas. The rest of the introduction is dedicated to a short description of the computational properties, advantages and limitations of FLAGSER.

FLAGSER generates the (directed or undirected) flag complex of a finite (directed or undirected) graph from its connectivity matrix. The construction algorithm is parallelisable to an arbitrary number of cores, which proves extremely useful when working with very large graphs (See Figure 11). To save storage space and increase speed, only the graph is loaded into memory, while by default the much larger directed flag complex is never stored. This enables a very fast construction of the directed flag complex, and allows FLAGSER to deal with graphs of remarkable size (See Figure 10).

The homology computation algorithm of FLAGSER is based in part on a rather new addition to the topological data analysis software collection, RIPSER by Ulrich Bauer [2]. However, since FLAGSER is designed to work primarily with graph data structures, it includes the capability to generate the flag complex of a finite graph from its connectivity matrix and perform computations on the complex without storing its simplices in memory. In addition the reduction algorithm and coboundary computation methods were modified and optimized for use with very large sparse graphical datasets. We performed a number of comparisons of FLAGSER with GUDHI, which showed that FLASGER outperforms GUDHI when the dataset is large and sparse, while GUDHI performs better on denser datasets. However, some of the graphs we examined were so large that non of the software we tested, including FLAGSER, were capable of finishing a precise computation (see Figure **??**).

To address this problem, FLASGER includes an extremely useful feature: The *Approximate* function. In general when one performs data analysis computation, the order of magnitude of a specific homology module is much more important that the precise value of the Betti number in question. The Approximate function allows the homology computation part of the program to compute Betti numbers and persistent homology within a user defined accuracy, and as a result dramatically reduce compute time. With FLAGSER we are able to perform the construction of the directed flag complex of the neocortical column of a rat [6] on a laptop within less than 30 seconds, and compute a good approximation for it's homology within a few of hours on an HPC cluster (Figure 13). Such computations were completely out of reach before. The accuracy parameter for the Approximate function can be modified to suit requirements. Experimentation suggests (although we cannot prove it in general) that the actual accuracy of computations that use the Approximate function is much higher than the accuracy theoretically guaranteed (see Figure 7).

FLAGSER has a built-in flexible way of associating weights to vertices and/or edges in a given graph, and the capability to have user defined weight functions that allow filtering the complex in question in a variety of ways. In particular this flexibility allows combining weights that arise from different

features of the data to create a single filtration on the complex. Flagser also contains the capability of computing homology with coefficients in odd characteristics.

The directed flag complex is only one of the topological constructions one may associate with a directed graph. Motivated by more recent ideas we developed modifications of Flagser: Flagser-count, Tournser, and Deltser (See Section 7 for detail).

The usefulness of Flagser and its derivatives is clearly not limited to neuroscience. Topological analysis of graphs and complexes arising in both theoretical problems and applicable studies have become quite prevalent in recent years [8, 20]. Hence Flagser, in its original or modified form, could become a major computational aid in many fields.

In the following, we discuss the main design choices and their impact on computations. Section 6 contains real-world examples demonstrating the usefulness of the software package. In Section **??** we compare some performance aspects of Flagser to those of Gudhi and point out advantages and disadvantages. Flagser is an open source package, licensed under the LGPL 3.0 scheme (See Section 7).

2. Mathematical background

In this section, we give a brief introduction to directed flag complexes of directed graphs, and to persistent homology. For more detail, see [18], [12] and [19].

Definition 2.1. *A* graph *G is a pair (V, E), where V is a finite set referred to as the* vertices *of G, and E is a subset of the set of unordered pairs $e = \{v, w\}$ of distinct points in V, which we call the* edges *of G. Geometrically the pair $\{u, v\}$ indicates that the vertices u and v are adjacent to each other in G. A* directed graph, *or a* digraph, *is similarly a pair (V, E) of vertices V and edges E, except the edges are ordered pairs of distinct vertices, i.e. the pair (v, w) indicates that there is an edge from v to w in G. In a digraph we allow reciprocal edges, i.e. both (u, v) and (v, u) may be edges in G, but we exclude loops, i.e. edges of the form (v, v).*

Next we define the main topological objects considered in this article.

Definition 2.2. *An* abstract simplicial complex on a vertex set *V is a collection X of non-empty finite subsets $\sigma \subseteq V$ that is closed under taking non-empty subsets. An* ordered simplicial complex on *a vertex set V is a collection of non-empty finite* ordered *subsets $\sigma \subseteq V$ that is closed under taking non-empty ordered subsets. Notice that the vertex set V does not have to have any underlying order. In both cases the subsets σ are called the* simplices *of X, and σ is said to be a k-simplex, or a k-dimensional* simplex, *if it consists of $k + 1$ vertices. If $\sigma \in X$ is a simplex and $\tau \subseteq \sigma$, then τ is said to be a* face *of σ. If τ is a subset of V (ordered if X is an ordered simplicial complex) that contains σ, then τ is called a* coface *of σ. If $\sigma = (v_0, \ldots, v_k)$ is a k-simplex in X, then the i-th face of σ is the subset $\partial_i(\sigma) = (v_0, \ldots, \hat{v}_i, \ldots, v_k)$ where by $\hat{v}_i$ we mean omit the i-th vertex. For $d \geq 0$ denote by X_d the set of all d-dimensional simplices of X.*

Notice that if X is any abstract simplicial complex with a vertex set V, then by fixing a linear order on the set V, we may think of each one of the simplices of X as an ordered set, where the order is induced from the ordering on V. Hence any abstract simplicial complex together with an ordering on its vertex set determines a unique ordered simplicial complex. Clearly any two orderings on the vertex set of the same complex yield two distinct but isomorphic ordered simplicial complexes. Hence from now on we may assume that any simplicial complex we consider is ordered.

Definition 2.3. *Let $G = (V, E)$ be a directed* graph. *The* directed flag complex dFl(*G) is defined to be the ordered simplicial complex whose k-simplices are all* ordered *$(k + 1)$-cliques, i.e. $(k + 1)$-tuples $\sigma = (v_0, v_1, \ldots, v_k)$, such that $v_i \in V$ for all i, and $(v_i, v_j) \in E$ for $i < j$. The vertex v_0 is called the* initial vertex *of σ or the* source *of σ, while the vertex v_k is called the* terminal vertex *of σ or the* sink *of σ.*

Let $\mathbb{F}$ be a field, and let X be an ordered simplicial complex. For any $k \geq 0$ let $C_k(X)$ be the $\mathbb{F}$-vector space with basis given by the set X_k of all k-simplices of X. If $\sigma = (v_0, \ldots, v_k)$ is a k-simplex in X, then the i-th face of σ is the subset $\partial_i(\sigma) = (v_0, \ldots, \hat{v}_i, \ldots, v_k)$, where $\hat{v}_i$ means that the vertex

v_i is removed to obtain the corresponding $(k-1)$-simplex. For $k \geq 1$, define the *k-th boundary map* $d_k \colon C_k(X) \to C_{k-1}(X)$ by

$$d_k(\sigma) = \sum_{i=0}^{k} (-1)^i \partial_i(\sigma) \in C_{k-1}(X).$$

The boundary map has the property that $d_k \circ d_{k+1} = 0$ for each $k \geq 1$ [12]. Thus $\mathrm{Im}(d_{k+1}) \subseteq \mathrm{Ker}(d_k)$ and one can make the following fundamental definition:

Definition 2.4. *Let X be an ordered simplicial complex. The k-th homology group of X (over $\mathbb{F}$) denoted by $H_k(X) = H_k(X; \mathbb{F})$ is defined to be*

$$H_k(X) = \mathrm{Ker}(d_k)/\mathrm{Im}(d_{k+1}).$$

If X is an ordered simplicial complex, and $Y \subseteq X$ is a subcomplex, then the inclusion $\iota \colon Y \to X$ induces a *chain map* $\iota_* \colon C_*(Y) \to C_*(X)$, i.e. a homomorphism $\iota_k \colon C_k(Y) \to C_k(X)$ for each $k \geq 0$, such that $d_k \circ \iota_k = \iota_{k-1} \circ d_k$. Hence one obtains an induced homomorphism for each $k \geq 0$,

$$\iota_* \colon H_k(Y) \to H_k(X).$$

This is a particular case of a much more general property of homology, namely its functoriality with respect to so called simplicial maps. We will not however require this level of generality in the discussion below.

Definition 2.5. *A filtered simplicial complex is a simplicial complex X together with a filter function $f \colon X \to \mathbb{R}$, i.e. a function satisfying the property that if σ is a face of a simplex τ in X, then $f(\sigma) \leq f(\tau)$. Given $b \in \mathbb{R} \sqcup \{\infty\}$, define the sublevel complex $X[b]$ to be the inverse image $f^{-1}((-\infty, b])$.*

Notice that the condition imposed on the function f in Definition 2.5 ensures that for each $b \in \mathbb{R} \sqcup \{\infty\}$, the sublevel complex $X[b]$ is a subcomplex of X, and if $b \leq b'$ then $X[b] \subseteq X[b']$. By default we identify X with $X[\infty]$. Thus a filter function f as above defines an increasing family of subcomplexes $X[b]$ of X that is parametrized by the real numbers.

If the values of the filter function f are a finite subset of the real numbers, as will always be the case when working with actual data, then by enumerating those numbers by their natural order we obtain a finite sequence of subspace inclusions

$$X[1] \subseteq X[2] \subseteq \cdots \subseteq X[n] = X.$$

For such a sequence one defines its *persistent homology* as follows: for each $i \leq j$ and each $k \geq 0$ one has a homomorphism

$$h_k^{i,j} \colon H_k(X[i]) \to H_k(X[j]),$$

induced by the inclusion $X[i] \to X[j]$. The image of this homomorphism is a k-th persistent homology group of X with respect to the given filtration, as it represents all homology classes that are present in $H_k(X[i])$ and carry over to $H_k(X[j])$. To use the common language in the subject, $\mathrm{Im}(h_k^{i,j})$ consists of the classes that were "born" at or before filtration level i and are still "alive" at filtration level j. The rank of $\mathrm{Im}(h_k^{i,j})$ is a *persistent Betti number* for X and is denoted by $\beta_k^{i,j}$.

Persistent homology can be represented as a collection of *persistence pairs*, i.e., pairs (i,j), where $0 \leq i < j \leq n$. The *multiplicity* of a persistence pair (i,j) in dimension k is, roughly speaking, the number of linearly independent k-dimensional homology classes that are born in filtration i and die in filtration j. More formally,

$$\mu_k^{i,j} = (\beta_k^{i,j-1} - \beta_k^{i-1,j-1}) - (\beta_k^{i,j} - \beta_k^{i-1,j}).$$

Persistence pairs can be encoded in *persistence diagrams* or *persistence barcodes*. See [10] for more detail.

FLAGSER computes the persistent homology of filtered directed flag complexes. The filtration of the flag complexes is based on filtrations of the 0-simplices and/or the 1-simplices, from which the filtration values of the higher dimensional simplices can then be computed by various formulas. When using the trivial filtration algorithm of assigning the value 0 to each simplex, for example, FLAGSER simply computes the ordinary simplicial Betti numbers. The software uses a variety of tricks to shorten the computation time (see below for more details), and outperforms many other software packages, even for the case of unfiltered complexes (i.e. complexes with the trivial filtration). In the following,

we mainly focus on (persistent) homology with coefficients in finite fields, and if we don't mention it explicitly we consider the field with two elements $\mathbb{F}_2$.

3. Representing the directed flag complex in memory

An important feature of Flagser is its memory efficiency which allows it to carry out rather large computations on a laptop with a mere 16GB of RAM (see controlled examples in Section 6). In order to achieve this, the directed flag complex of a graph is generated on the fly and not stored in memory. An n-simplex of the directed flag complex $\mathrm{dFl}(G)$ corresponds to a directed $(n+1)$-clique in the graph $G = (V, E)$. Hence each n-simplex is identified with its list of vertices ordered by their in-degrees in the corresponding directed clique from 0 to n. The construction of the directed flag complex is performed by a double iteration:

(I) For each vertex of the graph we iterate over all simplices which have this vertex as the initial vertex.
(II) Iterate (I) over all vertices.

This procedure enumerates all simplices exactly once.

In more detail, consider a fixed vertex v_0. The first iteration of Step (I) starts by finding all vertices w such that there exists a directed edge from v_0 to w. This produces the set $S_1^{v_0}$ of all 1-simplices of the form (v_0, w), i.e. all 1-simplices that have v_0 as their initial vertex. In the second iteration of Step (I), construct for each 1-simplex $(v_0, v_1) \in S_1^{v_0}$ obtained in the first iteration, the set $S_2^{v_0, v_1}$ by finding the intersection of the two sets of vertices that can be reached from v_0 and v_1, respectively. In other words,

$$S_2^{v_0, v_1} = \{w \in V \mid (v_0, w) \in S_1^{v_0}, \text{ and } (v_1, w) \in S_1^{v_1}\}.$$

This is the set of all 2-simplices in $\mathrm{dFl}(G)$ of the form (v_0, v_1, w). Proceed by iterating Step (I), thus creating the prefix tree of the set of simplices with initial vertex v_0. The process must terminate, since the graph G is assumed to be finite. In Step II the same procedure is iterated over all vertices in the graph (but see (c) below.)

This algorithm has three main advantages:

(a) The construction of the directed flag complex is very highly parallelizable. In theory one could use one CPU for each of the vertices of the graph, so that each CPU computes only the simplices with this vertex as initial vertex.
(b) Only the graph is loaded into memory. The directed flag complex, which is usually much bigger, does not have any impact on the memory footprint.
(c) The procedure skips branches of the iteration based on the prefix. The idea is that if a simplex is not contained in the complex in question, then no simplex containing it as a face can be in the complex. Therefore, if we computed that a simplex $(v_0, \ldots, v_k)$ is not contained in our complex, then we don't have to iterate Step (I) on its vertices to compute the vertices w that are reachable from each of the v_i. This allows us to skip the full iteration branch of all simplices with prefix $(v_0, \ldots, v_k)$. This is particularly useful for iterating over the simplices in a subcomplex.

In the code, the adjacency matrix (and for improved performance its transpose) are both stored as bitsets[1]. This makes the computation of the directed flag complex more efficient. Given a k-simplex $(v_0, \ldots, v_k)$ in $\mathrm{dFl}(G)$, the set of vertices

$$\{w \in V \mid (v_0, \ldots, v_k, w) \in \mathrm{dFl}(G)\}$$

is computed as the intersection of the sets of vertices that the vertices $v_0, \ldots, v_k$ have a connection to. Each such set is given by the positions of the 1's in the bitsets of the adjacency matrix in the rows corresponding to the vertices $v_0, \ldots, v_k$, and thus the set intersection can be computed as the logical "AND" of those bitsets.

---

[1]Bitsets store a single bit of information (0 or 1) as one physical bit in memory, whereas usually every bit is stored as one byte (8 bit). This representation improves the memory usage by a factor of 8, and additionally allows the compiler to use much more efficient bitwise logical operations, like "AND" or "OR". Example of the bitwise logical "AND" operation: `0111 & 1101 = 0101`.

In some situations, we might have enough memory available to load the full complex into memory. For these cases, there exists a modified version of the software that stores the tree described above in memory, allowing to

- perform computations on the complex such as counting, or computing co-boundaries etc., without having to recompute the intersections described above, and
- associate data to each simplex with fast lookup time.

The second part is useful for assigning unique identifiers to each simplex. Without the complex in memory a hash map with custom hashing function has to be created for this purpose, which slows down the lookup of the global identifier of a simplex dramatically.

4. Computing the coboundaries

In RIPSER, [2], cohomology is computed instead of homology. The reason for this choice is that for typical examples, the coboundary matrix contains many more columns that can be skipped in the reduction algorithm than the boundary matrix [3].

If σ is a simplex in a complex X, then σ is a canonical basis element in the chain complex for X with coefficients in the field $\mathbb{F}$. The hom-dual σ^*, i.e. the function that associates $1 \in \mathbb{F}$ with σ and 0 with any other simplex, is a basis element in the cochain complex computing the cohomology of X with coefficients in $\mathbb{F}$. The differential in the cochain complex requires considering the coboundary simplices of any given simplex, i.e. all those simplices that contain the original simplex as a face of codimension 1.

In RIPSER, the coboundary simplices are never stored in memory. Instead, the software enumerates the coboundary simplices of any given simplex by a clever indexing technique, omitting all coboundary simplices that are not contained in the complex defined by the filtration stage being computed. However, this technique only works if the number of theoretically possible coboundary entries is sufficiently small. In the *directed* situation, and when working with a directed graph with a large number of vertices, this number is typically too large to be enumerated by a 64-bit integer. Therefore, in FLAGSER we precompute the coboundary simplices into a sparse matrix, achieving fast coboundary iteration at the expense of longer initialization time and more memory load. An advantage of this technique is that the computation of the coboundary matrices can be fully parallelized, again splitting the simplices into groups indexed by their initial vertex.

Given an ordered list of vertex identifiers $(v_0, \ldots, v_k)$ characterising a directed simplex σ, the list of its coboundary simplices is computed by enumerating all possible ways of inserting a new vertex into this list to create a simplex in the coboundary of σ. Given a position $0 \leq p \leq k+1$, the set of vertices that could be inserted in that position is given by the intersection of the set of vertices that have a connection from each of the vertices $v_0, \ldots, v_{p-1}$ and the set of vertices that have a connection to each of the vertices $v_p, \ldots, v_k$. This intersection can again be efficiently computed by bitwise logical operations of the rows of the adjacency matrix and its transpose (the transpose is necessary to efficiently compute the incoming connections of a given vertex).

With the procedure described above, we can compute the coboundary of every k-simplex. The simplices in this representation of the coboundary are given by their ordered list of vertices, and if we want to turn this into a coboundary matrix we have to assign consecutive numbers to the simplices of the different dimensions. We do this by creating a hash map, giving all simplices of a fixed dimension consecutive numbers starting at 0. This numbering corresponds to choosing the basis of our chain complex by fixing the ordering, and using the hashmap we can lookup the number of each simplex in that ordering. This allows us to turn the coboundary information computed above into a matrix representation. If the flag complex is stored in memory, creating a hash map becomes unnecessary since we can use the complex as a hash map directly, adding the dimension-wise consecutive identifiers as additional information to each simplex. This eliminates the lengthy process of hashing all simplices and is therefore useful in situations where memory is not an issue.

5. Performance considerations

FLAGSER is optimized for large computations. In this section we describe some of the algorithmic methods by which this is carried out.

5.1. **Sorting the columns of the coboundary matrix.** The reduction algorithm iteratively reduces the columns in order of their filtration value (i.e. the filtration value of the simplex that this column represents). Each column is reduced by adding previous (already reduced) columns until the pivot element does not appear as the pivot of any previous column. While reducing all columns the birth and death times for the persistence diagram can be read off: roughly speaking, if a column with a certain filtration value f_0 ends up with a pivot that is the same as the pivot of a column with a *higher* filtration value f_1, then this column represents a class that is born at time f_0 and dies at time f_1. Experiments showed that the order of the columns is crucial for the performance of this reduction algorithm, so FLAGSER sorts the columns that have the same filtration values by an experimentally determined heuristic (see below). This is of course only useful when there are many columns with the same filtration values, for example when taking the trivial filtration where every simplex has the same value. In these cases, however, it can make very costly computations tractable.

The heuristic to sort the columns is applied as follows:
(1) Sort columns in ascending order by their filtration value. This is necessary for the algorithm to produce the correct results.
(2) Sort columns in ascending order by the number of non-zero entries in the column. This tries to keep the number of new non-zero entries that a reduction with such a column produces small.
(3) Sort columns in descending order by the position of the pivot element of the unreduced column. This means that the larger the pivot (i.e. the higher the index of the last non-trivial entry), the lower the index of the column is in the sorted list. The idea behind this is that "long columns", i.e. columns whose pivot element has a high index, should be as sparse as possible. Using such a column to reduce the columns to its right is thus more economical. Initial coboundary matrices are typically very sparse. Thus ordering the columns in this way gives more sparse columns with large pivots to be used in the reduction process.
(4) Sort columns in descending order by their "pivot gap", i.e. by the distance between the pivot and the next nontrivial entry with a smaller row index. A large pivot gap is desirable in case the column is used in reduction of columns to its right, because when using such a column to reduce another column, the added non-trivial entry in the column being reduced appears with a lower index. This may yield fewer reduction steps on average.

Remark 5.1. *We tried various heuristics—amongst them different orderings of the above sortings— but the setting above produced by far the best results. Running a genetic algorithm in order to find good combinations and orderings of sorting criteria produced slightly better results on the small examples we trained on, but it did not generalize to bigger examples. It is known that finding the perfect ordering for this type of reduction algorithm is NP-hard [23].*

5.2. **Approximate computation.** Another method that speeds up computations considerably is the use of approximate computation. This simply means allowing FLAGSER to skip columns that need many reduction steps. The reduction of these columns takes the most time, so skipping even only the columns with extremely long reduction chains gives significant performance gains. By skipping a column, the rank of the matrix can be changed by at most one, so the error can be explicitly bounded. This is especially useful for computations with trivial (i.e. constant) filtrations, as the error in the resulting Betti numbers is easier to interpret than the error in the set of persistence pairs.

Experiments on smaller examples show that the theoretical error that can be computed from the number of skipped columns is usually much bigger than the actual error. Therefore, it is usually possible to quickly get rather reliable results that afterwards can be refined with more computation time. See Figure 1 for an example of the speed-up gained by approximate computation. The user of FLAGSER can specify an approximation level, which is given by a number (defaulting to infinity). This number determines after how many reduction steps the current column is skipped. The lower the number, the more the performance gain (and the more uncertainty about the result). At the current time it is not possible to specify the actual error margin for the computation, as it is very complicated to predict the number of reductions that will be needed for the different columns.

5.3. **Dynamic priority queue.** When reducing the matrix, RIPSER stores the column that is currently reduced as a priority queue, ordering the entries in descending order by filtration value and in

Approximation	none	10^4	10^3	10^2	10^1
Theoretical error	0	322	1380	7849	192638
Actual error	0	0	19	58	65
Computation time	113s	49s	43s	41s	39s

FIGURE 1. Theoretical and actual error in the computation of the first Betti number by skipping the reduction of columns that require certain numbers of reduction steps. The cell counts are 4656, 35587 and 1485099 in dimensions $0, 1$ and 2 respectively, and the first Betti number is 3321. The computations were made as averages over ten runs on a MacBook Pro, 2.5 GHz Intel Core i7.
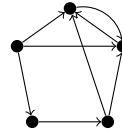

FIGURE 2. A digraph for which using apparent pairs to compute the homology of the directed flag complex gives the wrong results.

ascending order by an arbitrarily assigned index for entries with the same filtration value. Each index can appear multiple times in this queue, but due to the sorting all repetitions are next to each other in a contiguous block. To determine the final entries of the column after the reduction process, one has to loop over the whole queue and sum the coefficients of all repeated entries.

This design makes the reduction of columns with few reduction steps very fast but gets very slow (and memory-intensive) for columns with a lot of reduction steps: each time we add an entry to the queue it has to "bubble" through the queue and find its right place. Therefore, FLAGSER enhances this queue by dynamically switching to a hash-based approach if the queue gets too big: after a certain number of elements were inserted into the queue, the queue object starts to track the ids that were inserted in a hash map, storing each new entry with coefficient 1. If a new entry is then already found in the hash map, it is not again inserted into the queue but rather the coefficient of the hash map is updated, preventing the sorting issue described above. When removing elements from the front, the queue object then takes for each element that is removed from the front of the queue the coefficient stored in the hash map into account when computing the final coefficient of that index.

5.4. **Apparent pairs.** In RIPSER, so-called *apparent pairs* are used in order to skip some computations completely. These rules are based on discrete Morse theory [11], but they only apply for *non-directed* simplicial complexes. Since we consider directed graphs, the associated flag complex is a semi-simplicial rather than simplicial complex, so we cannot use this simplification. Indeed, experiments showed that enabling the skipping of apparent pairs yields wrong results for directed flag complexes of certain directed graphs. For example, when applied to the directed flag complex of the graph in Figure 2, using apparent pairs gives $\beta_1 = \beta_2 = 0$ and not using apparent pairs gives $\beta_1 = \beta_2 = 1$, which is visibly the correct answer.

Experiments showed that when applied to directed flag complexes that arise from graphs without reciprocal connections (so that the resulting directed flag complex is a simplicial complex), enabling the skipping of apparent pairs does not give a big performance advantage. In fact, we found a significant reduction in run time for subgraphs of the connectome of the reconstruction of the neocortical column based data averaged across the five rats that were used in [19], the Google plus network and the Twitter network from KONECT [14].

6. SAMPLE COMPUTATIONS

In this section we present some sample computations carried out using FLAGSER. Almost all computations were done on a laptop PC with 32GB of RAM and a 2-core processor. The data files were either taken from online public domain sources or generated for the purpose of computation, as detailed below.

Dataset	Vertices	Edges	Density	Count	Homology
BE	43	336	18.172	0.01s, 3.32MB	0.08s, 3.81MB
GP	23.6K	39.2K	0.007	0.50s, 564.40MB	2.09s, 569.54MB
IN	410	2765	1.644	0.04s, 3.68MB	1.27s, 9.13MB
JA*	198	2742	6.994	49.63s, 3.57MB	10h44m3.00s, 75.79GB
MQ	62	1187	30.879	0.09s, 3.44MB	9.41s, 50.22MB
PR	2239	6452	0.129	0.04s, 8.63MB	0.07s, 9.17MB
FN	40	760	47.500	2m23.26s, 3.46MB	Unfinished in 24h
BB†*	31.3K	7.8M	0.793	23.76s, 1.08GB	12h15m3.00s, 52.96 GB
BA	280	1911	2.437	0.00s, 3.52MB	0.03s, 3.84MB
CE	279	2194	2.819	0.02s, 3.55MB	0.12s, 4.55KB
ER†	31.3K	7.9M	0.804	15.06s, 1.08GB	1h30m51.00s, 4.98GB

FIGURE 3. Performance of FLAGSER on a variety of datasets: BE = Windsurfers, GP = Google+, IN = Infectious, JA = Jazz, MQ = Macaques, PR = Protein, FN = 19th step of the Rips filtration of the 40-cycle graph, BB = Blue Brain Project neocortical column, BA = Barabasi-Albert Graph, CE = C-Elegans, ER = Erdős-Rényi Graph. In all cases Density is given as connection probability as percentage.

In the Figure 13, the dagger symbol on the dataset abbreviation means that the approximate function of FLAGSER was used with approximation parameter 100000 (the code stops reducing a column after 100000 steps). This allows for computation of homology in a reasonable compute time. The asterisk stands for a homology computation that was made on a single core of an HPC cluster. This was done because the memory requirements of the homology computation were too large to perform it on a laptop with 16 GB of RAM.

The first six datasets were taken from the public domain Koblenz Network Collection KONECT [14, 13]. We chose the following datasets (with no intention of any analysis but the performance of FLAGSER):

- BE = HumanContact/Windsurfers - An undirected network containing interpersonal contacts between windsurfers in southern California during the fall of 1986,
- GP = Social/Google+ - A directed network contains Google+ user to user links,
- IN = HumanContact/Infectious - face-to-face behavior of people during the exhibition INFEC-TIOUS: STAY AWAY in 2009 at the Science Gallery in Dublin,
- JA = HumanSocial/Jazz musicians - collaboration network between Jazz musicians,
- MQ = Animal/Macaques - A directed network containing dominance behaviour in a colony of 62 adult female Japanese macaques,
- PR = Metabolic/Human protein (Figeys) - a network of interactions between proteins in Humans.

See Figure 13 for detail on performance.

One remarkable dataset among those is the set JA encoding a collaboration network between jazz musicians (an edge between two musicians if they played together). While this network is rather small and not particularly dense, computations with it required markedly more time and resources. This could be explained by the observation that the associated directed flag complex is 30-dimensional, with more that 150 million 14-dimensional simplices and similarly large numbers in neighbouring dimensions. Homology computation that took almost 11 hours and required more than 75GB of memory yielded a rather curious result with Betti numbers 1, 6, 1 in dimensions 0, 1, 2, respectively. This indicates homological simplicity is not advantageous in computation time. Two of the networks came as weighted graphs, which allowed us to compute persistent homology in those two cases (Figure 4). The filtration was determined by assigning to each simplex the maximal weight of an edge it contains.

The last five datasets are:

- 19th step of the Rips filtration of the 40-cycle graph (equipped with the graph metric), i.e., the graph with 40 vertices, where each vertex is connected to all others except the vertex with graph distance exactly 20 from itself,
- a sample of a Blue Brain Project reconstruction of the neocortical column or a rat [19],

Dataset	Persistence	
BE	0.09s,	3.91MB
MQ	11.47s,	69.54MB

FIGURE 4. Performance of FLAGSER computing persistence.

- an Erdős-Rényi graph with a similar number of vertices and connection probability as such a network,
- a C. Elegans brain network [22], and
- a Barabási-Albert graph with a similar number of vertices and connection probability as a C. Elegans brain network.

The set FN is again an example of how large dimensionality effects computation time. In this case one knows by general theory [1] that the resulting flag complex is a model for the 19-sphere, and so the nontrivial Betti numbers occur only in dimensions 0 and 19, and they are 1 in both. However, due to the size of the complex (2^{20} 19-simplices and nearly 200 million 9-simplices) we were not able to obtain a homology computation within 24 hours.

Next, we tested the performance of FLAGSER with respect to several values of the Approximate parameter. The performed the computation on two graphs: a) a Blue Brain Project neocortical column reconstruction, denoted above by BB (Figure 5), and an Erdős-Rényi graph with the same number of vertices and connection probability (Figure 8). These computations were carried out on a single core of an HPC cluster.

Approx.	Time	Memory	β_1	β_2	β_3	β_4	β_5
1	40m44s	11.77GB	14,640	9,831,130	30,219,098	513,675	785
10	38m44s	11.96GB	15,148	14,831,585	12,563,309	13,278	40
100	40m29s	12.69GB	15,188	13,956,297	7,171,621	9,898	37
1000	55m40s	15.42GB	15,283	14,598,249	5,780,569	9,822	37
10000	2h24m17s	23.78GB	15,410	14,872,057	5,219,666	9,821	37
100000	12h15m3s	52.96GB	15,438	14,992,658	4,951,674	9,821	37

FIGURE 5. Performance of FLAGSER on the Blue Brain Project graph with respect to different values of the Approximate parameter. The Betti numbers computed are an approximation to the true Betti numbers. See Figure 6 for accuracy.

Approx.	δ_1	δ_2	δ_3	δ_4	δ_5
1	7,724,934	56,746,602	19,092,109	664,481	4,162
10	6,714,811	15,944,792	796,559	372	0
100	179,794	4,107,019	6,414	0	0
1000	14,209	1,902,229	42	0	0
10000	1,002	1,054,397	0	0	0
100000	27	664,857	0	0	0

FIGURE 6. Number of skipped columns of the coboundary matrix for the Blue Brain Project graph with respect to different values of the Approximate parameter. The approximation accuracy of β_i depends theoretically on the number of columns skipped in the coboundary matrices for δ_{i-1} and δ_i.

Consider for example β_4. By Figures 5 and 6 the true value is $\beta_4 = 9821$, since in that case no columns are skipped in δ_3 and δ_4. Thus perfect accuracy is achieved with Approximate parameter 10000. However, with Approximate parameters 1000 and 100 one has a theoretical error of 42 and 6414 respectively, whereas the actual error is 1 and 77 respectively. In the case of β_2 we do not know the actual answer. However, since the total number skipped with Approximate parameter 100000 is 664884, and the approximate value computed for β_2 is 14992658, the calculation is accurate within

less than $\pm 5\%$. Similarly, β_1 is computed to be 15438 with 27 skipped columns, which stands for an accuracy at least as good as $\pm 0.2\%$. For β_3 the theoretical bound is clearly not as good, but here as in all other cases where precise computation was not carried out, the trajectory of the approximated Betti numbers as the approximation parameter grows suggests that the number computed is actually much closer to the real Betti numbers than the theoretical error suggests (Figure 7).
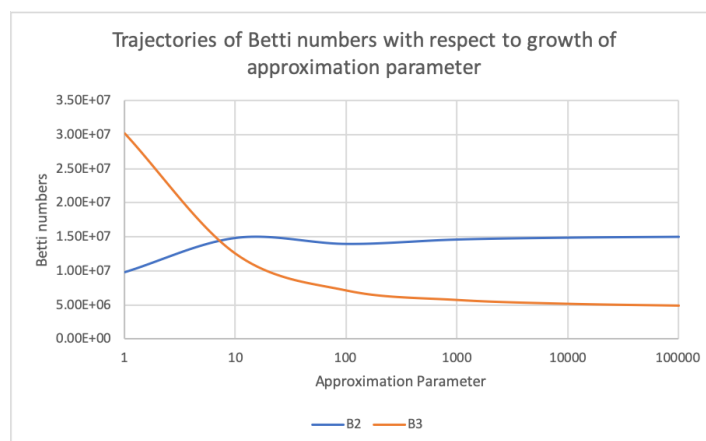

FIGURE 7. Trajectories of β_2 and β_3 as a function of the approximation parameter. A change in order of magnitude in the approximation parameter results in a small change in the Betti number computed.

We performed a similar computation with an Erdős-Rényi random directed graph with the same number of vertices as the Blue Brain Project column and the same average probability of connection (Figures 8 and 9). Notice that there is a very substantial difference in performance between the Blue Brain Project graph and a random graph with the same size and density parameters. The difference can be explained by the much higher complexity of the Blue Brain Project graph, as witnessed by the Betti numbers.

Approx.	Time	Memory	β_0	β_1	β_2	β_3
1	4m45s	3.25GB	1	19,675	14,675,052	40
10	4m31s	3.10GB	1	19,981	9,204,135	4
100	4m44s	3.18GB	1	19,981	8,074,149	4
1000	5m48s	3.33GB	1	19,981	7,921,730	4
10000	14m18s	3.72GB	1	19,982	7,876,858	4
100000	1h30m51s	4.98GB	1	19,984	7,857,917	4

FIGURE 8. Performance of FLAGSER on an Erdős-Rényi graph with respect to different values of the Approximate parameter. The Betti numbers range from dimension 0 to dimension 3.

A remarkable feature of FLAGSER is its parallelisable construction algorithm of the directed flag complex. To demonstrate the capability of the software, we computed some large scale examples. In particular, we computed the cell counts of two sample circuits produced by the Blue Brain Project. The first is a reconstruction of the somatosensory cortex of a rat (Figure 10), and the second is the so called PL region of the reconstruction of the full neocortex of a mouse (Figure 11). We also computed the simplex counts on some large non-biological networks (Figure 12).

The somatosensory cortex (Figure 10) is a digraph with about 1.7 million vertices and 1.1 billion edges (average connection probability 0.04%). The Blue Brain Project model of the full mouse cortex is available online [7]. The full reconstruction consists of approximately 10 million neurons. Its synaptic connections fall into three categories. Local (short distance) connections, intermediate distance connections and long distance connections. We computed the cell count of the PL region of the left

Approx.	δ_1	δ_2	δ_3
1	7,692,010	719,041	124
10	1,501,909	3	0
100	371,920	0	0
1000	219,501	0	0
10000	174,628	0	0
100000	155,685	0	0

FIGURE 9. Number of skipped columns of the coboundary matrix for an Erdős-Rényi graph with respect to different values of the Approximate parameter in dimensions from 1 to 3.
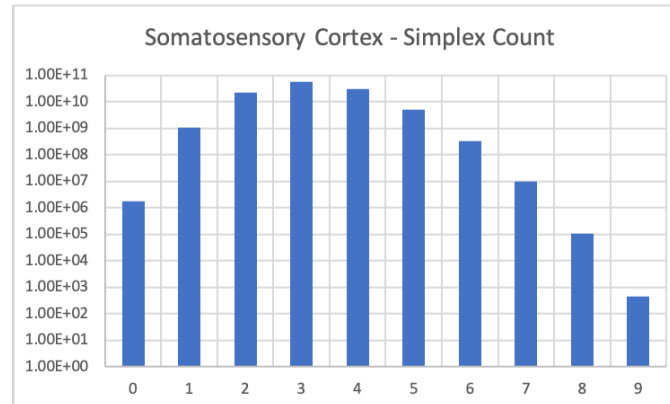

FIGURE 10. Cell count in the Blue Brain reconstruction of the somatosensory cortex. Computation was run on an HPC cluster using 256 CPUs, required 55.69GB of memory and took 7.5 hours to complete.
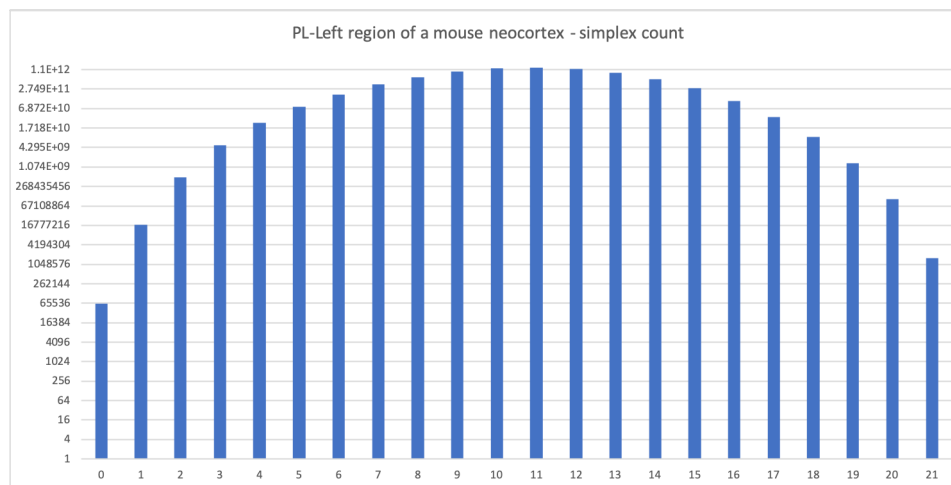

FIGURE 11. The simplex count for the PL-Left region of a mouse neocortex (local connections only). The complex is 21-dimensional with more than 1.2 trillion 11-dimensional simplices. Computation was run in parallel on two nodes of an HPC cluster with 256 CPUs each, required 1GB of memory and took 5 days to complete (run as 10 different jobs of 24 hours each).

hemisphere in the reconstruction, consisting of approximately 64,000 neurons and 17.5 million local

(A) Runtime: 48m45s, Memory: 2.5GB

(B) Runtime: 6h24m50s, Memory: 40.5GB

(C) Runtime: 3m49s, Memory: 1GB
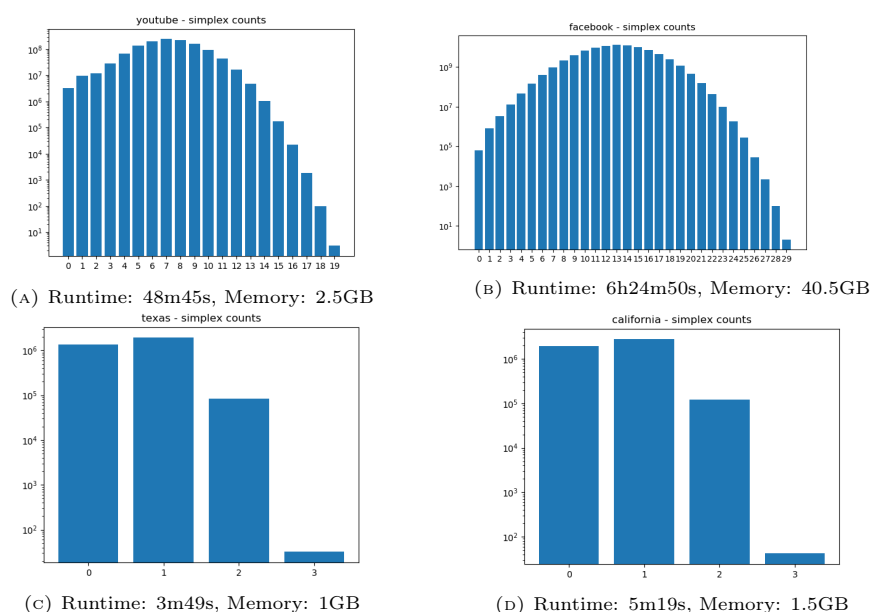

(D) Runtime: 5m19s, Memory: 1.5GB

FIGURE 12. Four large datasets taken from the Koblenz Network Collection [14]. (a) Social network of YouTube users and their friendship connections, (b) Friendship data of Facebook users, (c) Road network of the State of Texas, a node is an intersection between roads or road endpoint and edges are road segments and (d) Road network of the State of California. All computations were carried out on an HPC cluster using 256 cores.

connections (average connection probability 0.4%). We did not consider any connections but the local ones.

Interestingly the number of neurons in the model of the PL region [7] is roughly twice that in the reconstruction of the neocortical column studied in [19], and the probability of connection is roughly half. Hence we expected a comparable performance of FLAGSER. The results are therefore quite surprising. The likely reason for the difference in computation speeds is the higher dimensionality of the model of neocortex of the mouse, resulting in a gigantic number of simplices in the middle dimensions (approximately 1.1 trillion 11-dimensional simplices), compared to the neocortical column of the rat, with the former containing simplices of dimension 21 (see Figure 11) and the latter only containing simplices up to dimension 7.

These computation were done on an adapted version of FLAGSER available at [21]. This alternate version has been adapted in the following ways: the code to compute homology is removed as this is not needed, the option to load the graph from compressed sparse matrix format is included as this speeds up the time required to read the graph, the graph can be stored using Google's dense hash map to reduce the memory needed for very large graphs, and partial results are printed during run time which allows for the computation to be resumed if it is stopped (for example, if the process is killed due to time limitations on HPC).

6.1. **Comparisons.** We end this section by comparing the performance of FLAGSER in homology computations to that of GUDHI, which is recognised as another state of the art computational software package. In doing so we are forced to restrict attention to undirected graphs, since GUDHI is designed to work only with simplicial complexes. In addition, while FLAGSER takes a directed graph as input and produces the complex in unison with the computation of homology, GUDHI must be given the complex as its input on which it performs its computation. Thus we combine GUDHI with the python package NETWORKX which contains state of the art clique finding algorithms for undirected graphs. More specifically, we apply the *find_cliques* method of NETWORKX to get the maximal cliques of the graph and inserting these into a GUDHI *SimplexTree*, and then compute the homology using GUDHI.

Furthermore, as NETWORKX does not allow for parallel enumeration of cliques, for a fair comparison we only enable FLAGSER to use a single core.

In Figure 13 we compare the run time of GUDHI+NETWORKX with FLAGSER. We can see that for small examples such as C. Elegans and BBP layer 2 the times and memory usage are fairly similar, although GUDHI+NETWORKX is a bit faster on BBP layer 2. We can also see that GUDHI+NETWORKX was faster on the PGP graph, with FLAGSER not finishing without the max dimension option. Although for this graph we can see that GUDHI+NETWORKX required 92GB of memory, whereas using max dimension with FLAGSER only required 5.4GB. For the OpenFlights and SimpleWiki graphs neither GUDHI+NETWORKX or FLAGSER finished. However, if we use the max dimension option with FLAGSER we can compute the homology of OpenFlights (upto dimension 6) in under four hours using only 5.2GB of memory. Similarly, for SimpleWiki if we use both the approximate and max dimension option, then we can get results in under 24 hours, on a graph with over 100k vertices.

Dataset	Vertices	Edges	GUDHI+NETWORKX	Flagser	Flagser (Approx. 100000)	Betti Numbers	Approx. Betti Numbers
C. Elegans	279	2194	1.229s, 585MB	0.705s, 589MB	0.701s, 524MB	[1, 162, 83]	[1, 162, 83]
BBP Layer 2	3416	132309	11s, 919MB	145s, 745MB	129s, 612MB	[1, 5290, 106360, 5, 0, 0]	[1, 5288, 106381, 5]
BBP Layer 5	6114	756107	DNF	DNF	13h4m45s, 1.4GB	Unknown	[1, 1222, 1427597, 1133]
PGP	10680	24316	3h6m20s, 92GB	3h25m46s, 5.4GB (MD=6)	3h30m32s, 5.3B (MD=6)	[1, 1093, 22, 1]	[1, 1093, 22, 1, 0, 0]
OpenFlights	2939	19256	DNF	3h45m18s, 5.2GB (MD=6)	3h37m19s, 5.3GB (MD=6)	[24354, 310, 107, 53, 2, 2, 0]	[24354, 309, 108, 53, 4, 2, 0]
SimpleWiki	100312	824968	DNF	DNF	15h7m41s, 21.0GB (MD=6)	Unknown	[304, 125571, 45306, 31866, 10486, 319, 6]

FIGURE 13. A comparison of the run times of FLAGSER and GUDHI+NETWORKX to compute the homology of the undirected flag complex of six undirected graphs. Where DNF denotes the computation did not finish in under 24 hours, Approx. 100000 denotes the use of the approximate function of FLAGSER with value 100000, and MD=6 denotes the use of the max dimension option of FLAGSER to compute only up to dimension 6. In the cases where the max dimension option was used, the computation did not finish within 24 hours without it. All computations were done on an HPC cluster using a single CPU. The BBP layers are taken from the Blue Brain Project website [5] and the other circuits are taken from KONECT [14]. We consider the graphs as simple undirected graphs for these computations, as such the edge counts in the table above may differ from the original graphs as reciprocal edges and loops were removed.

Dataset	δ_1	δ_2	δ_3	δ_4	δ_5	δ_6	δ_7
C. Elegans	0	0	0	0	0	0	0
BBP2	0	23	0	0	0	0	0
BBP5	0	32	49614	0	0	0	0
PGP							
OpenFlights	0	2	0	2	0	0	0
SimpleWiki	0	56	1357	1280	63	0	0

FIGURE 14

To conclude we add that although GUDHI+NETWORKX is the closest state of the art comparison to FLAGSER we are aware of, it does not allow for the use of some of the key advantages of FLAGSER, such as parallelisation and working with directed graphs and ordered simplicial complexes. Moreover, we are unaware of any faster algorithm for the enumeration of directed cliques than the breadth first search approach applied by FLAGSER.

7. AVAILABILITY OF SOURCE CODE

FLAGSER was developed as a part of an EPSRC funded project on applications of topology in neuroscience. As such FLAGSER is an open source package, licensed under the LGPL 3.0 scheme. The code can be found on the GitHub page of Daniel Lütgehetmann who owns the copyright for the code and will maintain the page [16].

In the course of using FLAGSER in the project three adapted versions were developed, and can be found in the GitHub page of Jason Smith [21], who will maintain the page. The following modifications are available.

- FLAGSER-COUNT: A modification of a basic procedure in FLAGSER, optimised for very large networks.
- DELTSER: A modification of FLAGSER that computes persistent homology on any finite delta complex (like a simplicial complex, except more than one simplex may be supported on the same set of vertices).
- TOURNSER: The *flag tournaplex* of a directed graph is the delta complex whose simplices are all tournaments that occur in the graph. TOURNSER is an adaption of FLAGSER which computes (persistent) homology of the tournaplex of a finite directed graph. Tournser uses two filtrations that occur naturally in tournaplexes, but can also be used in conjunction with other filtrations.

REFERENCES

[1] M. ADAMASZEK, *Special cycles in independence complexes and superfrustration in some lattices*, Topology Appl., 160 (2013), pp. 943–950, `https://doi.org/10.1016/j.topol.2013.03.009`.

[2] U. BAUER, *Ripser: A lean C++ code for the computation of Vietoris-Rips persistence barcodes.* `http://ripser.org`, 2015-2018. Accessed: 29-08-2018.

[3] U. BAUER, *Ripser: Efficient computation of Vietoris-Rips persistence barcodes.* `http://ulrich-bauer.org/ripser-talk.pdf`, 2018. Accessed: 29-08-2018.

[4] U. BAUER, M. KERBER, J. REININGHAUS, AND H. WAGNER, *Phat – Persistent homology algorithms toolbox*, Journal of Symbolic Computation, 78 (2017), pp. 76 – 90, `http://www.sciencedirect.com/science/article/pii/S0747717116300098`.

[5] BLUE BRAIN PROJECT, `https://www.epfl.ch/research/domains/bluebrain` (accessed 2019/07/02).

[6] BLUE BRAIN PROJECT, *Digital reconstruction of neocortical microcircuitry*, 2019. Data retrieved from, `https://bbp.epfl.ch/nmc-portal/downloads`.

[7] BLUE BRAIN PROJECT, *Mouse whole-neocortex connectome model*, 2019. Data retrieved from, `https://portal.bluebrain.epfl.ch/resources/models/mouse-projections/`.

[8] A. COSTA AND M. FARBER, *Large random simplicial complexes, I*, J. Topol. Anal., 8 (2016), pp. 399–429, `https://doi.org/10.1142/S179352531650014X`.

[9] P. DŁOTKO, *Topological neuroscience software*, `http://neurotop.gforge.inria.fr` (accessed 2019/07/02).

[10] H. EDELSBRUNNER AND J. HARER, *Persistent homology—a survey*, in Surveys on discrete and computational geometry, vol. 453 of Contemp. Math., Amer. Math. Soc., Providence, RI, 2008, pp. 257–282, `https://doi.org/10.1090/conm/453/08802`.

[11] R. FORMAN, *Morse theory for cell complexes*, Advances in Mathematics, 134 (1998), pp. 90 – 145, `https://doi.org/10.1006/aima.1997.1650`.

[12] A. HATCHER, *Algebraic topology*, Cambridge: Cambridge University Press, 2002.

[13] J. KUNEGIS, *Konect – the Koblenz Network Collection.* `http://konect.uni-koblenz.de/`.

[14] J. KUNEGIS, *Konect – the Koblenz Network Collection*, Proceedings of the 22nd International Conference on World Wide Web Companion, (2013), pp. 1343–1350, `http://konect.uni-koblenz.de/`.

[15] LABORATORY FOR TOPOLOGY AND NEUROSCIENCE, `https://hessbellwald-lab.epfl.ch/` (accessed 2019/07/02).

[16] D. LÜTGEHETMANN, *Computing homology of directed flag complexes.* `https://github.com/luetge/flagser`, 2019.

[17] C. MARIA, J.-D. BOISSONNAT, M. GLISSE, AND M. YVINEC, *The Gudhi Library: Simplicial Complexes and Persistent Homology*, Research Report RR-8548, INRIA, June 2014, `https://hal.inria.fr/hal-01005601`.

[18] J. R. MUNKRES, *Elements of algebraic topology*, Addison-Wesley, 1984.

[19] M. W. REIMANN, M. NOLTE, M. SCOLAMIERO, K. TURNER, R. PERIN, G. CHINDEMI, P. DŁOTKO, R. LEVI, K. HESS, AND H. MARKRAM, *Cliques of neurons bound into cavities provide a missing link between structure and function*, Frontiers in Computational Neuroscience, 11 (2017), p. 48, `https://www.frontiersin.org/article/10.3389/fncom.2017.00048`.

[20] R. J. SANCHEZ-GARCIA, M. FENNELLY, S. NORRIS, N. WRIGHT, G. NIBLO, J. BRODZKI, AND J. W. BIALEK, *Hierarchical spectral clustering of power grids*, IEEE Transactions on Power Systems, 29 (2014), pp. 2229–2237.

[21] J. P. SMITH, *flagser-adaptions.* `https://github.com/JasonPSmith/flagser-adaptions`, 2019.

[22] L. R. VARSHNEY, B. L. CHEN, E. PANIAGUA, D. H. HALL, AND D. B. CHKLOVSKII, *Structural properties of the caenorhabditis elegans neuronal network*, PLoS Computational Biology, 7 (2011), `http://dblp.uni-trier.de/db/journals/ploscb/ploscb7.html#VarshneyCPHC11`.

[23] M. YANNAKAKIS, *Computing the minimum fill-in is NP-complete*, SIAM Journal on Algebraic Discrete Methods, 2 (1981), pp. 77–79, `https://doi.org/10.1137/0602010`.

Institute of Mathematics, University of Aberdeen
E-mail address: luetgehetmann@gmail.com

Institute of Mathematics, University of Aberdeen
E-mail address: dejan.govc@abdn.ac.uk

Institute of Mathematics, University of Aberdeen
E-mail address: jason.smith@abdn.ac.uk

Institute of Mathematics, University of Aberdeen
E-mail address: r.levi@abdn.ac.uk