A Topological analysis of an alternative to the PageRank algorithm in weighted directed graphs

NATALIA ANDREA LONDONO CASTRILLON
A Topological analysis of an alternative to the PageRank algorithm in weighted directed graphs

Natalia Andrea Londono Castrillon
Acknowledgements

I want to thank Martina Scolamiero, who was my guide and support during the thesis. She always made time for me and to help me figure out what the next step was.
I want to thank Professor Wojtek Chacholski for the support, the guidance and the faith in my project. I also want to thank him for introducing me to the world of topological data analysis.
I want to thank Professor Volker Betz for welcoming me to Darmstadt, for allowing me to play and edit the algorithm, for trusting me with this project and diving into an unknown topic with me.
Thank you to Lukas Roth for the help and to the Stochastic Processes department at the TU Darmstadt, and to the Lichtenberg Cluster for the computational power.
Abstract

We provide a first test and overview of an alternative to Google’s PageRank algorithm using persistent homology and topological data analysis. First, we present a data-set consisting of weighted, directed graphs. Then, we explain how we can transform said graphs to weighted, directed simplicial complexes. After that, we compute the persistent homology of the resulting complexes using Flagser by Daniel Luetgethmann. Once we have the starting homological data, we introduce the two versions of the algorithm. One version contracts and deletes every node visited. The other deletes only visited nodes with less than 3 neighbours. We run the algorithm and re-compute persistent homology. Using the results, we analyse changes in the homology to determine how the algorithm changes the topological structure of our data-sets. Subsequently, we try to answer the question on whether there is a correlation between the changes in the persistent homology and the effectiveness and speed of the algorithm.

Keywords — TDA, persistent homology, PageRank, weighted directed graphs
# Contents

1 Introduction ................................................. 7

2 Graphs .................................................... 9
   2.1 Defining graphs ....................................... 9
   2.2 Properties of graph .................................. 10

3 Simplicial Complexes ...................................... 13
   3.1 Defining simplicial complexes ....................... 13
   3.2 Homology ............................................ 14

4 Complexes of Graphs ...................................... 23
   4.1 From Graphs to Complexes ......................... 23
   4.2 Directed Complexes ................................ 24
   4.3 Weighted-directed complexes ....................... 25

5 Data-sets .................................................. 27
   5.1 Describing the data-sets ........................... 27
   5.2 Barcodes and filtration diagrams ................. 29
      5.2.1 Dimension Filtration ......................... 30
      5.2.2 Vietoris-Rips filtration ..................... 31
      5.2.3 Computation .................................. 35
      5.2.4 Graphing ..................................... 37
   5.3 The Algorithm ...................................... 39
   5.4 Its purpose ......................................... 39
   5.5 Our take on it ...................................... 39

6 Results .................................................... 47
   6.1 Running the algorithm .............................. 47
      6.1.1 Cleaning up the data .......................... 48
   6.2 Looking at the snapshots ............................ 48
      6.2.1 Greedy Algorithm results .................... 48
   6.3 Average Runs ....................................... 54
      6.3.1 Aggressive algorithm ......................... 54
      6.3.2 Less aggressive algorithm .................... 56
      6.3.3 Comparison .................................... 60

7 Conclusion and next steps .................................. 67

A About the original graphs ................................ 71
A.1 Degree histogram .................................................. 71
B Flagser commands .................................................. 75
C Cutting and editing the Datasets .................................. 77
  C.0.1 Computing Topological Properties ....................... 78
  C.0.2 Why we didn’t use these results ......................... 81
D Tables and graphs from the results .............................. 85
  D.1 Greedy algorithm .............................................. 85
    D.1.1 Cell count .................................................. 85
    D.1.2 Betti Numbers .............................................. 86
    D.1.3 Greedy algorithm ......................................... 86
Chapter 1

Introduction

The well known PageRank algorithm by [google], is to-date, the method Google uses for determining importance of websites. The algorithm uses of a random walker, which surfs the internet in search of the websites to rank. In general terms, the easier a website is to find, the higher the rank it receives. Because the internet is not always so easy to navigate, the random walker has some help. The walker can always, with a certain (fixed) probability jump anywhere (random) on the internet. That way, if the walker finds itself at a point with no edges pointing towards other nodes, or visiting the same nodes too often it will not remain there forever, but instead it will eventually jump to another random node and continue its walk. In stochastic processes, Page-Rank is referred to as a random walk, whose probabilities are dictated by a Markov Chain with a re-start.

This set up makes PageRank very efficient because the walker never gets stuck. However, it does jump randomly to places potentially far away from its goal. If there was a way the walker didn’t have to jump somewhere randomly on the internet, would it be more efficient? This master’s thesis aims to provide the first set of tests for an algorithm without random jumps \(^1\), with the ultimate goal of determining whether it could be more efficient than PageRank.

The paper takes seven data-sets, all of which are weighted directed graphs, and analyses them topologically by presenting them as flag complexes and computing the dimension of its homologies, and the cell count of directed simplices. Then, the algorithm is run on said data-sets, and its performance is examined. The topological properties of the data-sets are measured again after the algorithm has been run to determine the algorithm’s impact on the shape of the data. Based on these results, a first assessment will be provided on how effective/aggressive the algorithm is, and what changes could improve its results.

To do this, we first design an environment that will give us some rigorous structure to work with, and that allows us to talk about simplices, homologies, graphs, etc. This means a basic introduction to weighted directed graphs, followed by an introduction to simplicial complexes, and lastly our own definition of weighted directed complexes coming from weighted directed graphs. Then, there will be an overview of the topological properties we will measure given a simplicial complex, and some basic examples.

After the properties have been explained, the data-sets will be presented followed by a

\(^1\)Unless the jump is to another connected component
short explanation on how [flagser] was used to obtained the properties explained above, along with tables and graphs showing these results.

The next section explains the two versions of the algorithm we worked with. The first (greedy or aggressive) version deletes every node visited. The other (less aggressive or non-greedy) version one deletes only nodes which have less than two neighbours. Lastly, the results chapter explains in detail what we found in our experiments. The results show that the less-aggressive algorithm has a lower run-time and is therefore more efficient in this regard. We also see that in general, the greedy algorithm changes almost completely the structure of the data.

After the data has been run on this algorithm, it becomes very difficult to analyse, and very hard to walk through as there are many many edges (and therefore high dimensional simplices) with very small weights. This is also proven by the fact that most of the resulting graphs had to be modified in order to be run by Flagser on the super-computer. Both algorithms change the signature of the graphs noticeably, compared to the original graphs. The difference between the signatures among the resulting graphs of both algorithms is also noticeable as they group together in their respective groups according to the type of algorithm they were run on. This thesis also makes some recommendations as to what the next steps are in the project, such as testing algorithms with different criteria when it comes to which nodes to delete.
Chapter 2

Graphs

A graph is the most intuitive representation of a dataset in which there are connections between nodes. It is easy to visualize and gives a general idea of about how the structure of the data looks like.

In order to explain what we mean by representing the data as a graph, we must introduce the definitions of the concepts we will use.

2.1 Defining graphs

**Definition 2.1.1.** A graph $G$ is a a pair $(V, E)$ where $V$ is a finite set and $E$ is a subset of the set of non empty subsets of $V$ of size at most 2. Elements of $V$ are called vertices of $G$. Elements $\{v, w\}$ in $E$ are called edges, and we say an edge $\{v, w\}$ connects vertices $v$ and $w$. If $v = w$, then such an edge is called a loop.

**Example 1.** Let $W$ be a set of websites. Define the graph $G = (W, E)$ where $E$ consists of these subsets $\{x, y\}$ of $W$ for which there is a link between $x$ and $y$. It is unlikely that such graphs have loops. Since we are primarily interested in graphs representing portions of the internet our main focus is on graphs without loops.

It is also important to notice that given two websites $a, b$ there may be a link from $a$ pointing to $b$ but not from $b$ to $a$. In order to make this distinction we introduce the notion of directed graphs.

The following definition is taken from [Bluebraincomplexes].

**Definition 2.1.2.** A directed graph $G$ is a pair $(V, E)$ where $V$ is a finite set and $E$ is a subset of $V \times V$. As before, elements of $V$ are called vertices and elements of $E$ are called edges. Edges are pairs of elements $(a, b)$ in $V$. Such an edge is said to originate in $a$ and terminate in $b$.

**Remark.** Note that according to our definition of a directed graph, there can be at most one edge originating in one vertex and terminating in another one.

**Example 2.** Let $W$ be a set of websites, as in Example 1. Define the directed graph $G = (W, E)$ where $E$ consists of these pairs $(a, b)$ of elements in $W$ for which there is a link from $a$ to $b$.

Now we are getting closer to the real representation of the internet as a graph. But there
is still something missing. Suppose you are looking at a website, say a Google search result. The first links are big and easy to see, they are the first ten results of your search. However, there is many more links on this website. There are other results you will only see if you scroll down, and there are links to other pages of the google search. It would be ideal to be able to show that the first links (the first results of the search) are much more important than the links to the 9th and 10th page of the search results. We know the first get clicked on much more often than the second, and our definition of a graph should be able to account for that.

**Definition 2.1.3.** A weight on a (directed) graph is a function $\omega: E \rightarrow \mathbb{R}^+_0$. The value $\omega(e)$ is called the weight of $e$. A higher weight represents a higher probability of choosing that edge.

Usually weights are positive numbers, often with a certain meaning. For example, the importance of that edge. Following this convention, we decided to normalize the weights of our graphs so that the weight represents the probability of clicking on that given link. **Definition 2.1.4.** A normalized weighing on a graph $G$ is a weight on the graph, where the weight function is: $\hat{\omega}: E \rightarrow [0, 1]$ and such that given a node $v$, we have that for all $w \in V$

$$\hat{\omega}(v, w) = \frac{\sigma(v, w)}{\sum_{w \in V} \sigma(v, w)}$$

and so for all $v \in V$ we have as well that

$$\sum_{w \in V} \hat{\omega}(v, w) = 1$$

From this point on, whenever we talk about weighted graphs, we mean normalized weighted graphs.

### 2.2 Properties of graph

Our goal is to find out as much as possible about these graphs so that we can compare them. Therefore, we have selected some properties that we consider interesting, and we have measured them for every graph on our set. This will help us understand how each of these graphs looks like, how they are similar and how they are different. We will define those properties below, and give the results for our graphs in chapter 5. **Definition 2.2.1.** Given a graph $G = (V, E)$ its Fill is the number of edges $|E|$ over the amount of edges it could have. In an un-directed graph this number is $\frac{n(n-1)}{2}$ and in a directed graph it is $n(n-1)$. So in our case we will refer to the directed case only.

$$\text{Fill} = \frac{|E|}{n(n-1)} \quad (2.1)$$

**Definition 2.2.2.** Given a graph $G = (V, E)$, the **degree** degree of a node is the number of neighbours it has. Because we are talking about directed graphs, we could define outgoing degree and incoming degree as the amount of edges that go out of that node towards other nodes, and as the amount of edges pointing towards that node, respectively.
However, we will define the degree of our nodes as the sum of outgoing and incoming degrees.

\[ \text{deg}(u) = \sum_{(u,v) \in E} 1 + \sum_{(v,w) \in E} 1 \]  \hspace{1cm} (2.2)

**Definition 2.2.3.** Given a graph \( G = (V, E) \), its Avg. degree is the average of the degrees of its nodes. Similarly, its Max. degree is the maximum of the degrees of its nodes.

\[ \text{avg. deg} = \frac{1}{|V|} \sum_{v \in V} \text{deg}(v) \]  \hspace{1cm} (2.3)

\[ \text{Max. deg} = \max_{v \in V} \text{deg}(v) \]  \hspace{1cm} (2.4)
Chapter 3

Simplicial Complexes

3.1 Defining simplicial complexes

Even though graphs provide structure for data, we can take one step further and represent graphs as simplicial complexes. Simplicial complexes can be easily constructed from graphs, and we can find out more information about simplicial complexes and their properties, than of graphs. This is because we can think of simplicial complexes as higher dimensional objects, whereas graphs will always be two-dimensional. These extra dimensions give us room for extra structure, which evolves into more interesting properties, now in the field of topology.

Now that we have defined what graphs are and what type of graphs we have, we will define simplicial complexes, and then explain how we can view a graph as a simplicial complex.

1

Definition 3.1.1. Let S be a set. A simplicial complex X on S is a set of non empty finite subsets of S such that:

- for any s in S the set \{s\} belongs to X
- if \sigma belongs to X then so does any of its non-empty subsets.

An element \sigma in X is called a simplex of dimension |\sigma| - 1. The subset of n-dimensional simplices in X is denoted by X^n.

Example 3. The collection of all non-empty subsets of \{0, 1, \ldots, n\} is a simplicial complex and it is called the standard simplex of dimension n. We denote it by \Delta^n.

We can choose to give an ordering to the elements of a simplex, and we can order simplices in a simplicial complex. Having an order adds structure to simplices and complexes, and allows for new maps to be defined. The most important map being the face-map. However, ordered simplices can be difficult to work with computationally speaking. Furthermore, given the nature of directed graphs, they can not be represented as ordered simplices.

1Most of the following definitions are adaptations from [friedman2012], which I read and used for guidance as well as a mean of understanding simplicial complexes, and in order to create the definitions that are relevant for graphs.
Therefore, we will not work with ordered simplices or simplicial complexes. \(^2\) However, the notion of an ordered simplex helps understanding the definition of a face-map.

**Definition 3.1.2.** Given a fixed \(n\) and a simplex \([v_0, \ldots, v_n] \in X^n\) on an ordered complex \(X\), there is a collection of functions called face-maps \(d_0, \ldots, d_n : X^n \to X^{n+1}\) such that \(d_j[v_0, \ldots, v_n] = [v_0, \ldots, \hat{v}_j, \ldots, v_n]\). That is, the \(i\)th face-map of an \(n\) simplex is the \(n-1\) simplex resulting by removing the \(i\)th vertex of the original simplex (and its connections).

**Proposition 3.1.1.** Given a complex \(X\), \(d_jd_j = d_{j-1}d_j\) holds for any simplex in \(X\).

The proof of this theorem can be found in any algebra book and it is quite straightforward. Because I didn't come up with it myself, I will not write it down in this document.

**Proposition 3.1.2.** Given a simplex \(\sigma\) of dimension \(n\), a generalized face-map \(d : X^n \to X^j\) for any given \(0 \leq j < n\) can be obtained uniquely as a composition of (regular) face-maps \(d_0 \cdots d_n\), where \(d_{i_k} : m_k \to m_k - 1\) for some given \(m_k\), and where \(i_j < i_{j+1}\) holds for all \(j\).

### 3.2 Homology

In order to measure the topological structure of our complexes, we use topological metrics which come from a measure of topological complexity called homology. To know what this is, we need some background definitions. \(^3\)

**Definition 3.2.1.** A field is a set of elements together with addition and multiplication operations, both of which satisfy the following properties: associativity, commutativity, distributivity, identity, inverses.

**Example 4.** \(F_2 = \{0, 1\}\)

The computations in this thesis will be carried out using the field \(F_2 = \{0, 1\}\).

**Definition 3.2.2.** Let \(X\) be a simplicial complex, for each \(n \geq 0\), define \(C_n\) to be the \(F_2\) vector space which has \(X_n\) as its basis elements.

**Example 5.** Let \(X\) be the simplicial complex composed of three vertices, each connected to each other, and the filled in triangle as a simplex.

\[
X_0 = \{x, y, z\}, \; X_1 = \{xy, yz, zx\}, \; X_2 = \{xyz\}
\]

Then, we have that every element \(v_j \in C_0\) is a linear combination of \(x, y, z\) with either 0 or 1 as coefficients. That is every \(v_j\) is of the form \(v_j = a_1x + a_2y + a_3z\) where \(a_i \in \{0, 1\}\). Then, we say that \(C_0 \cong \mathbb{Z}_2^3\).

The same will be true for \(C_1\) because there are three edges in \(X_1\). So \(C_1 \cong \mathbb{Z}_2^3\).

Following the same logic it is easy to see that \(C_2 \cong \mathbb{Z}_2\) and lastly \(C_n \cong 0\) for all \(n \geq 3\).

\(^2\)Recall simplices and simplicial complexes are sets. An ordered simplex (or simplicial complex) is a simplex (or simplicial complex) with an ordering of its elements.

\(^3\)This is why it is useful to define an ordering on the simplices before defining a face-map. It helps us know which vertex is the \(i\)th vertex.

\(^4\)The definitions below were obtained from [Bluebraincomplexes] and [algebraictopbook] or constructed by me.
Now we define the following linear transformation.

**Definition 3.2.3.** Let $C_n$ be defined as above. Then, for each $n \geq 1$ the **differential** is defined as:

$$\delta_n : C_n \rightarrow C_{n-1} \quad \text{where} \quad \delta_n(\sigma) = \sum_{i=0}^{n} d_i(\sigma)$$

for every $\sigma \in C_n$. Here, $d_i(\sigma)$ is the $i$-th face of $\sigma$ as before.

In the case of $n = 0$ we simply say $\delta_0 : C_0 \rightarrow 0$ so $\delta_0(v_i) = 0$ for all vertices $v_i \in X_0$.

**Example 6.** Using the same complex as in Example 4, we can compute its relevant differentials.

$$\delta_1(xy) = y + x$$
$$\delta_1(yz) = z + y$$
$$\delta_1(zz) = x + z$$

Similarly $\delta_2(xyz) = yz + zx + xy$

**Remark.** One can prove that $C_n(K)$ is free abelian for any simplicial complex $K$.

And so, using this fact and given our definition of the differential function, we can define the homology group.

**Definition 3.2.4.** The **mod 2 homology group** of our simplicial complex $X$ is:

$$H_n(X, \mathbb{F}_2) = \begin{cases} 
\ker(\delta_n)/\operatorname{Im}(\delta_{n+1}) & \text{if } n \geq 1 \\
C_0/\operatorname{Im}(\delta_1) & \text{if } n = 0
\end{cases} \quad (3.1)$$

We will call the elements in the kernel of $\delta_n$ the **$n$-cycles** and denote them

$$\ker(\delta_n) := Z_n(X)$$

Similarly, we call the elements in the image of $\delta_n + 1$ the **$n$-boundaries** and denote them

$$\operatorname{Im}(\delta_{n+1}) := B_n(X)$$

**Example 7.** For the first computation we will focus again on the 2-dimensional simplex in Example 4. Based on the calculations in Example 5, we see have the following facts:

- $\operatorname{Im}(\delta_0) = 0$
- $\operatorname{Im}(\delta_1) = \langle y + x, z + y, x + z \rangle$
- $\operatorname{Im}(\delta_2) = \langle yz + zx + xy \rangle$

Note that $\ker(\delta_0) = C_0$ and so we can already compute

$$H_0(X, \mathbb{F}_2) = \langle x, y, z \rangle / \langle y + x, z + y, x + z \rangle \cong \mathbb{Z}_2 \quad (3.2)$$

Moving on to $H_1(X, \mathbb{F}_2)$ we will first compute $Z_1(X)$. An edge $e$ belongs to the kernel if its image under $\delta$ is zero. In Example 4, we say that all edges are a linear combination of our three basis elements with coefficients in our field.

Then, $e = a_1xy + a_2yz + a_3zx$ is in the kernel of $\delta_1$ if $\delta_1(e) = 0$ that is if:

$$\delta_1(a_1xy + a_2yz + a_3zx)$$
$$= a_1\delta_1(xy) + a_2\delta_1(yz) + a_3\delta_1(zx)$$
$$= a_1(x + y) + a_2(y + z) + a_3(z + x) = 0$$

and re-arranging

$$= (a_1 + a_3)x + (a_1 + a_2)y + (a_2 + a_3)z = 0$$

15
Then, we require \( a_1 = a_2 = a_3 \). Because, \( x \) will be zero any time \( a_1 = a_3 \) (either both 0 or both 1) and similarly for \( y \) and \( z \). Then, all elements in \( Z_1(K) \) are multiples of the basis element \( xy + yz + zx \) so the group is infinite cyclic and so \( Z_1(K) \cong \mathbb{Z}_2 \).

Then,

\[
H_1(X, \mathbb{F}_2) = \langle xy + yz + zx \rangle / \langle yz + zx + xy \rangle \cong 0
\]  

(3.4)

We are identifying all the basis element with itself so we get the trivial group here. Lastly, we look at \( H_2(X, \mathbb{F}_2) \). We have that \( \text{Im}(\delta_3) = 0 \). Now, the \( \ker(\delta_2) \) has elements such that their image is zero. But, given that \( C_2 \) is generated by a single element, and the image of that element and its generators is zero, then \( \ker(\delta_2) = 0 \) and similarly:

\[
H_2(X, \mathbb{F}_2) = 0
\]  

(3.5)

Now, we can talk about the intuition behind these computations. In particular, the dimension of each \( n \)-th homology group plays an important role in our computations and it is easy to picture what it means.

**Definition 3.2.5.** The \( n \)-th Betti Number \( \beta_n(X) = \dim(H_n(X, \mathbb{F}_2)) \) is the \( \mathbb{F}_2 \) vector-space dimension of the \( n \)-th homology group.

The Betti Number tells us how many \( n \)-dimensional cycles are not boundaries for higher dimensional simplices. That is, it tells us how many cycles are empty (or not filled). In other words, the Betti Number is the count of \( n \)-dimensional holes there are in our simplex. In order to see this, we will work through a more elaborate example going through all the steps we already computed for the 2-dimensional simplex above.

**Example 8.** Let \( K \) be the following simplicial complex:

As we saw before, for every \( n \) we can construct \( C_n \) to be the vector space spanned by our \( K_n \) listed above. Again, we choose the field to be \( \mathbb{F}_2 \), this means we don’t need an orientation for the complexes. To compute the homologies we need to compute several differentials. First, let’s understand how our spaces \( C_n \) look like. These are the spaces the maps \( \delta_n \) are acting on, so before computing the maps we should know where we are computing such maps. The sequence we are interested in looks like this:

\[
C_3 \xrightarrow{\delta_3} C_2 \xrightarrow{\delta_2} C_1 \xrightarrow{\delta_1} C_0 \xrightarrow{\delta_0} 0
\]

With this picture in mind, we can start the computation. Let us start with \( C_0 \). This vector space has the set \( K_0 = \{x, y, w, z, v\} \) as a basis. Our points are, of course, linearly independent from each other, so we can say that \( C_0 \cong \mathbb{Z}_2^5 \). The reason why we have \( \mathbb{Z}_2 \)

\(^5\text{Because in \( \mathbb{F}_2 \) we have } 1 = -1.\)
and not $\mathbb{Z}$ is because we are working on the field $\mathbb{F}_2$.\footnote{Some people define $C_n$ differently (as a space of functions) in order to get them to be isomorphic or equivalent to a product of $\mathbb{Z}$ but in our case we are using the field and so we will be working with $\mathbb{Z}_2$.}

Moving on to the computation of $C_1$ and $C_2$. Note that the elements of $K_1$ and $K_2$ are linearly independent as well, as it was the case in $K_0$. Therefore, we get that $C_1 \cong \mathbb{Z}_2^3$ and $C_2 \cong \mathbb{Z}_2^3$. Then, the sequence above is isomorphic to the following short exact sequence:

\[ 0 \longrightarrow \mathbb{Z}_2^3 \overset{\delta_1}{\longrightarrow} \mathbb{Z}_2^8 \overset{\delta_2}{\longrightarrow} \mathbb{Z}_2^5 \longrightarrow 0 \]

Now we can move on to the computation of the differentials.

Since $\delta_0$ is the zero map, we have that $\ker(\delta_0) = \langle x, y, z, w, v \rangle$ and $\operatorname{Im}(\delta_0) = 0$.

\[
\begin{align*}
\mathbb{Z}_0(K) & \cong 0 \\
\end{align*}
\]

Now for $\delta_1$, given that our $C_1$ has the edges in $K_1$ as a basis, and since $\delta_1$ is a group homomorphism, the image of $\delta_1$ is spanned by the image of our basis elements. i.e.

\[
\operatorname{Im}(\delta_1) = \langle w + x, w + z, y + z, y + x, v + x, v + y, z + v, w + v \rangle
\]

but here we have some elements that are not linearly independent. Remember, in our field we have $1 + 1 = 0$ so for example, $w + z + w + z = z + z$ therefore, we can simplify as:

\[
\operatorname{Im}(\delta_1) = \langle w + x, y + z, y + x, v + x, w + v \rangle
\]

That is,

\[
\begin{align*}
\mathbb{B}_0(K) & \cong \mathbb{Z}_2^5 \\
\end{align*}
\]

Using the information on equations 3.6 and 3.7 we can compute:

\[
\begin{align*}
\mathbb{H}_0(K, \mathbb{F}_2) = \mathbb{Z}_0(K) / \mathbb{B}_0(K) & \cong 0 / \mathbb{Z}_2^5 = 0 \\
\end{align*}
\]

The kernel of $\delta_1$ takes some more work to compute. An element in $C_1$ is of the form:

\[
e = \alpha(xw) + \beta(zw) + \gamma(xy) + \epsilon(xy) + \theta(vy) + \theta(vz) + \lambda(wv) + \mu(vz)
\]

and such an element $e$ gets sent to zero if $\delta_1(e) = 0$ that is if:

\[
\delta_1(e) = \delta_1 \left( \alpha(xw) + \beta(zw) + \gamma(xy) + \epsilon(xy) + \theta(vy) + \theta(vz) + \lambda(wv) + \mu(vz) \right) = 0
\]

Now, because $\delta_1$ is a group homomorphism \footnote{Recall, homomorphisms are linear maps. This is what allows us to do the next step.} we can re-write the equation as follows:
\[
\delta_1(e) = \alpha \cdot \delta_1(xw) + \beta \cdot \delta_1(zw) + \gamma \cdot \delta_1(zy) + \epsilon \cdot \delta_1(xy) \\
+ \eta \cdot \delta_1(vy) + \theta \cdot \delta_1(vz) + \lambda \cdot \delta_1(wu) + \mu \cdot \delta_1(vx) = 0
\]

and evaluating we get:

\[
\delta_1(e) = \alpha(x + w) + \beta(z + w) + \gamma(z + y) + \epsilon(x + y) \\
+ \eta(v + y) + \theta(v + z) + \lambda(w + v) + \mu(v + x) = 0
\]

We can re-write this one more time to group it in terms of variables:

\[
\delta_1(e) = (\alpha + \epsilon + \mu)x + (\gamma + \epsilon + \eta)y + (\beta + \gamma + \theta)z \\
+ (\alpha + \beta + \lambda)w + (\eta + \theta + \lambda + \mu)v = 0
\]

This gives us a system of equations which we need to solve. We can represent the system as a matrix.

\[
\begin{bmatrix}
\alpha & \beta & \gamma & \epsilon & \eta & \theta & \lambda & \mu \\
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w \\
v
\end{bmatrix} =
\begin{bmatrix}
0 \\
0 \\
0 \\
0 \\
0
\end{bmatrix}
\]

And we want it to be equal to zero. That is, we want to solve:

\[
\begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0
\end{bmatrix}
\]

But it is important to remember that we are working in \( F_2 \) so here \( 1 + 1 = 0 \). Reducing the matrix we get:

\[
\begin{bmatrix}
1 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\]

As we can see above, we have four pivots in our reduced matrix. That means we have four free variables, we will call them \( q, r, s, t \). So in terms of our greek letters, we have:
\[ \begin{align*}
\alpha &= q + t \\
\beta &= q + s + t \\
\gamma &= q + r + s + t \\
\epsilon &= q \\
\eta &= r + s + t \\
\theta &= r \\
\lambda &= s \\
\mu &= t
\end{align*} \]

This means that the kernel of \( \delta_1 \) is the span of:
\[
\begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} q + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} r + \begin{pmatrix} 0 \\ 1 \\ 1 \\ 1 \\ 0 \\ 1 \end{pmatrix} s + \begin{pmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{pmatrix} t
\]

This means that all the cycles in \( Z_1(K) \) are multiples of the following cycles:

\[
\begin{align*}
\alpha, \beta, \gamma, \epsilon &= < x, w, z, y > \\
\gamma, \eta, \theta &= < v, y, z > \\
\beta, \gamma, \eta, \lambda &= < w, z, y, v > \\
\alpha, \beta, \gamma, \eta, \mu &= < x, w, z, y, v >
\end{align*}
\]
Next on, $\text{Im}(\delta_2)$. As before, the image is spanned by the image of the basis elements. Therefore, we compute

\[
\delta_2(xuv) = uv + xv + wx \\
\delta_2(wuz) = vz + wz + wv \\
\delta_2(vxy) = xy + vy + vx
\]

Or if we write them as cycles then we have that:

\[
\delta_2(xuv) = < w, v, x > \\
\delta_2(wuz) = < v, z, w > \\
\delta_2(vxy) = < x, y, v >
\]

With this information, we can compute our first homology. We can do it geometrically, by looking at the picture and trying to identify the cycles in $Z_1(K)$ with cycles in $B_1(K)$. For example, if we take the green and blue dotted cycles (in the first picture) and add them together we get the teal (not dotted) cycle in $B_1(K)$. In a similar way, gray and red dotted together give us magenta (not dotted), and gray and green dotted give us yellow (not dotted).

So we have the following sums:
Therefore, we get that the homology is $4 - 3 = 1$ the total number of the cycles in $Z_1(K)$ minus the boundaries. We can also write this as a formal quotient.

$$H_1(K, F_2) = \langle xwzv, vyz, wzv, xwzv \rangle / \langle vuz, uzw, xyv \rangle$$

$$H_1(K, F_2) = \langle c_1, c_2, c_4 \rangle / \langle c_1 + c_2 + c_3, c_1 + c_4 \rangle \simeq Z_2$$

$$H_1(K, F_2) \simeq \frac{Z_1(K)}{B_1(K)}$$

Next we compute $\ker(\delta_2)$. An element $f \in C_2$ looks like $f = \alpha t_1 + \beta t_2 + \gamma t_3$ where, $t_1, t_2, t_3$ are the basis elements of $C_2$. In this example,

$$t_1 = xvw$$
$$t_2 = wvz$$
$$t_3 = zvy$$

Now, $f$ is in the kernel if $\delta_2(f) = 0$. That is, if

$$\delta_2(\alpha t_1 + \beta t_2 + \gamma t_3) = 0$$
$$\delta_2(\alpha t_1 + \delta_2(\beta t_2) + \delta_2(\gamma t_3) = 0$$
$$\alpha * \delta_2(t_1) + \beta * \delta_2(t_2) + \gamma * \delta_2(t_3) = 0$$

Evaluating,

$$\alpha(vw + zw + xv) + \beta(vz + wz + wv) + \gamma(vy + zy + zy) = 0$$

Because we are working in $F_2$ then we need $\alpha = \beta$ (so that $\alpha * vw + \beta * wv = 0$), and we need also $\beta = \gamma$ so that $\beta * vz + \gamma * vz = 0$. This means that $\alpha = \beta = \gamma$. But in order to make $\alpha xw$ disappear it must be that $\alpha = 0$. Then, from these requirements we get that $\ker(\delta_2) = 0$. Lastly, since our space $C_3 = 0$ then $\operatorname{Im}(\delta_2) = 0$ and so

$$H_2(K) = \ker(\delta_2) / \operatorname{Im}(\delta_3) \simeq 0$$

Now, the last property that we will talk about.
Definition 3.2.6. The Euler Characteristic of a simplex $\sigma$ is the alternate sum of its Betti numbers. That is

$$\chi(\sigma) = \sum_{i=0}^{\infty} (-1)^i \beta_n(X) \quad (3.11)$$

Remark. Even though the sum limit goes to infinity, it is finite because for every finite simplex, there exists an $N \in \mathbb{N}$ such that for all $n > N$ we have $\beta_n(X)$

Example 9. Taking the same simplex from the previous examples, we can compute its Euler characteristic.

$$\chi(X) = 0 - 1 + 0 = -1$$
Chapter 4

Complexes of Graphs

4.1 From Graphs to Complexes

How can we go from a graph to a simplicial complex or set? In particular, how can we use the definitions above to do so? Well, it depends on the graph we are using. Therefore, given an un-directed (loop-less) graph, and using definition 3.1.1 we can construct a simplicial complex given a graph in the following way.

**Definition 4.1.1.** Let $\mathcal{G} = (V, E)$ be a graph. Define $|\mathcal{G}|$ to be the collection of all finite non-empty subsets $\sigma$ of $V$ such that:

- For all $v \in V$, $\{v\} \in |\mathcal{G}|$

- A subset $\sigma$ in $V$ of cardinality bigger that 1 belongs to $|\mathcal{G}|$ if any of its two element subsets is an element in $E$, i.e., an edge in $\mathcal{G}$

**Proposition 4.1.1.** For any graph $\mathcal{G} = (V, E)$, the collection $|\mathcal{G}|$ is a simplicial complex.

*Proof.* Let $\sigma$ be in $|\mathcal{G}|$. Choose a subset $\tau \subset \sigma$. If $|\tau| = 1$ we are done because of the first part of Definition 4.1.1. If $|\tau| > 1$, then any two element subset of $\tau$ is also a two element subset of $\sigma$ and hence belongs to $E$. Thus $\tau$ satisfies the second requirement of Definition 4.1.1 and hence it belongs to $|\mathcal{G}|$. $\square$

This means that given a graph, every set of $n + 1$ vertices $\{v_0, \ldots, v_n\}$ for which its subgraph is a complete graph, becomes an $n$-simplex. Informally, every set that “could” be the skeleton of an $n$-simplex, is indeed an $n$-simplex as long as the edges are there.

**Example 10.** Take the following graph:

```
    v
   /|
  / \|
 /   \\
\     \\
Y-----x
    |
    |
    |
    |
    z
```

For example, in this graph, there is no way of knowing whether the triangle is “Filled” or not in advance. In flag complexes, the convention is to take all triangles to be filled. Therefore, that graph results in the following flag complex:
\[ X^0 = \{x, y, z, v\} \]
\[ X^1 = \{xy, yz, zx, yv\} \]
\[ X^2 = \{xyz\} \]

### 4.2 Directed Complexes

We would like to encode directionality when transforming directed graphs into geometric objects. Simplicial complexes as we defined them earlier fail to encode this information. Therefore, in this section we explain how to incorporate directionality in the form of so-called complexes of injective words.

For complexes of injective words, instead of subsets to encode simplices as it is done for simplicial complexes, we use functions:

**Definition 4.2.1.** An injective word of length \( n \) on a set \( X \), is an injective function \( \sigma : [n] = \{0, 1, 2, \ldots, n\} \rightarrow X \).

Since \( [n] \) is naturally ordered, an injective word induces an ordering of its values.

We often describe an injective word as a sequence \((x_0, x_1, x_2, \ldots, x_n)\) or word \( x_0 \cdots x_n \) of \( n + 1 \) different elements in \( X \).

**Definition 4.2.2.** A complex of injective words on a set \( X \) is a set \( K \) of injective words on \( X \), i.e., a set of injective functions \( \sigma : [n] \rightarrow X \) satisfying the following condition. If \( \sigma : [n] \rightarrow X \) is in \( K \), then so is the composition \( \sigma \alpha \) for every order preserving monomorphism \( \alpha : [m] \rightarrow [n] \). This means that if \( x = (x_0, x_1, x_2, \ldots, x_n) \) is a word in \( K \), then so is any of its sub-words.

Instead of simplicial complexes, to model ordered graphs we are going to use complexes of injective words:

**Definition 4.2.3.** Let \( G = (V, E) \) be a directed graph. Define \( |G| \) to be a collection consisting of these injective words \( \sigma : [n] = \{0, 1, 2, \ldots, n\} \rightarrow V \) such that for any \( i < j \) in \( [n] \), the pair \((\sigma(i), \sigma(j))\) is an edge in \( E \).

**Proposition 4.2.1.** For any directed graph \( G = (V, E) \), the collection \( |G| \) is a complex of injective words.

**Proof.** Let \( \sigma [n] \rightarrow V \) be an element in \( |G| \). Choose an order preserving monomorphism \( \alpha : [m] \rightarrow [n] \) and consider the composition \( \sigma \alpha : [m] \rightarrow V \). For any \( i < j \) in \( [m] \), since \( \alpha \) is order preserving \( \alpha(i) < \alpha(j) \). It follows that \((\sigma(\alpha(i)), \sigma(\alpha(j)))\) is in \( E \). Consequently the composition \( \sigma \alpha \) belongs to \( |G| \). \( \square \)

**Example 11.** Take the following directed graph:
Then, the directed word complex corresponding to the graph is made up of the following:

- Letters: \{x, v, u, y, z, w\}
- Two letter words: \{(xy), (xz), (xv), (yu), (yz), (uv), (zu), (uw), (wu)\}
- Three letter words: \{(xyz), (xyu), (xvz), (yuv), (vzu)\}

It is important to notice that not all paths of length 3 are words. For example, the path xyu is not a word as it doesn’t satisfy the condition of an edge xu. Similarly, there could be a four letter word xuvzy if there was an edge zy. However, the edge yz is not enough to satisfy this condition.

Now, in order to work with topological properties of directed graphs, we say that

**Definition 4.2.4.** the directed flag complex of a directed graph is the simplicial complex generated by taking every word in the directed word complex as a simplex.

### 4.3 Weighted-directed complexes

Now we have a structure that creates a complex based on directed graphs. The last step is to integrate the weight of these graphs to our structure, so that we can use these properties as well in topological computations. In the previous section we defined a weighted directed graph. Now, we can define a weighted, directed flag complex.

**Definition 4.3.1.** A weighted directed flag complex of a weighted directed graph, is a directed flag complex, where for all words, every pair of letters \((x_i, x_j)\) with \(i < j\) has a positive weight.

From this point on, we will work with flag complexes of graphs only, and we say that the elements of a flag complexes are called simplices.
Chapter 5

Data-sets

While the main goal of the algorithm is to rank websites of the internet, this project will mostly use other types of data for testing. The reason behind this is two-fold. Firstly, it is extremely expensive to store a data-set as huge as the internet. Furthermore, for the purposes of this thesis, we predict that the algorithm will stay within a certain "region" of the internet so taking a portion of the internet, or a smaller dataset would be enough. Secondly, because of this prediction, it becomes easy to find data-sets of the right size which have properties similar to those of the internet (weighted and directed networks) from reliable sources. Because this is the first testing stage of the algorithm, these graphs are a good starting point.

Once the algorithm is proven to work on these graphs, the next step would be to test the algorithm on a big portion of the internet, but this will not be done in this thesis.

5.1 Describing the data-sets

For this stage of testing, I chose seven data-sets. They were derived from three different well-known sources: [konectwebsite], [snapnets], and [Opsahl].

All of the networks are weighted and directed graphs. They were downloaded and there was a three-step cleaning process for the data. Firstly, any loops (edges whose source and goal node is the same) were removed. Then, weights were normalized. That is, the weights of each edge were edited so that the sum of weights of all edges coming out of any given node would add up to 1. Lastly, the graphs were cut in size. This step happened late in the project, and it was not planned. However, due to the amount of memory required to compute the topological properties of our graphs, we were forced to shrink our datasets.

Each dataset was cut to have 50 nodes, but we kept all the edges that were connecting these 50 nodes in the original graph. That is, we took the complete subgraph for these 50 nodes.

Below there are several tables, the first and second of which, list the number of n-dimensional simplices for the flag complexes associated to each graph. Recall that the

1The individual citations for every network can be found on the references section
2See the appendix, to read about the options we tried before making this decision, and why they didn’t work.
3To see the topological properties of the original (uncropped) datasets see the appendix.
simplices of zero-dimension are the nodes of a graph, so the number of 0-dimensional cells is the number of nodes in a graph. Likewise, the 1-dimensional simplices are the edges; in general, the number of n-dimensional simplices is the number of subsets of cardinality \( n + 1 \) \( \{x_0, x_1, \ldots, x_n\} \) such that for all \( i, j \in \mathbb{N} \), if \( 0 \leq i < j \leq n \) then \( (x_i, x_j) \) is a (directed) edge. The third and fourth tables, list the Betti numbers for every homology. Intuitively, \( \dim H_0 \) tells us the number connected components in the graph. \( \dim H_1 \) tells us the number of un-filled circles (holes) in the graph. That is, loops of size bigger than 3 that don't form a higher dimensional simplex. \( \dim H_2 \) tells us the number of empty spheres, and so on. Finally, the last table contains additional information about the graphs such as the fill percentage, average and maximum degree of nodes, and the Euler characteristic.

These computations were obtained by running "Flagser" by [flagser] on the Lichtenberg cluster at the Technical University of Darmstadt. For more information about how Flagser was used see the appendix.

<table>
<thead>
<tr>
<th>Graph</th>
<th>0-dim sim.</th>
<th>1-dim</th>
<th>2-dim</th>
<th>3-dim</th>
<th>4-dim</th>
<th>5-dim</th>
<th>6-dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advocato</td>
<td>51</td>
<td>173</td>
<td>402</td>
<td>1167</td>
<td>3088</td>
<td>5903</td>
<td>7266</td>
</tr>
<tr>
<td>Adol. Health</td>
<td>51</td>
<td>86</td>
<td>52</td>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>51</td>
<td>250</td>
<td>712</td>
<td>1315</td>
<td>1527</td>
<td>1059</td>
<td>400</td>
</tr>
<tr>
<td>BC OTC</td>
<td>51</td>
<td>289</td>
<td>926</td>
<td>1977</td>
<td>2214</td>
<td>900</td>
<td>0</td>
</tr>
<tr>
<td>BC Alpha</td>
<td>51</td>
<td>834</td>
<td>7749</td>
<td>46776</td>
<td>195787</td>
<td>578192</td>
<td>1179713</td>
</tr>
<tr>
<td>Open Flights</td>
<td>51</td>
<td>161</td>
<td>251</td>
<td>227</td>
<td>135</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Fb msg.</td>
<td>51</td>
<td>111</td>
<td>109</td>
<td>94</td>
<td>40</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.1.1: Number of n-simplices of Data (Part 1)

<table>
<thead>
<tr>
<th>Graph</th>
<th>7-dim sim.</th>
<th>8-dim</th>
<th>9-dim</th>
<th>10-dim</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advocato</td>
<td>5220</td>
<td>1560</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>62</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC Alpha</td>
<td>1551728</td>
<td>1050288</td>
<td>67200</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1.2: Number of n-simplices of Data (Part 2)

<table>
<thead>
<tr>
<th>Graph</th>
<th>( B_0 = \dim H_0 )</th>
<th>( B_1 )</th>
<th>( B_2 )</th>
<th>( B_3 )</th>
<th>( B_4 )</th>
<th>( B_5 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advocato</td>
<td>5</td>
<td>14</td>
<td>10</td>
<td>5</td>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>Adol. Health</td>
<td>9</td>
<td>7</td>
<td>8</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>4</td>
<td>8</td>
<td>5</td>
<td>12</td>
<td>18</td>
<td>3</td>
</tr>
<tr>
<td>BC OTC</td>
<td>2</td>
<td>20</td>
<td>33</td>
<td>197</td>
<td>460</td>
<td>253</td>
</tr>
<tr>
<td>BC Alpha</td>
<td>2</td>
<td>4</td>
<td>45</td>
<td>387</td>
<td>1373</td>
<td>8265</td>
</tr>
<tr>
<td>Open Flights</td>
<td>3</td>
<td>14</td>
<td>38</td>
<td>22</td>
<td>44</td>
<td></td>
</tr>
<tr>
<td>Fb msg.</td>
<td>2</td>
<td>18</td>
<td>12</td>
<td>7</td>
<td>6</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1.3: Betti Numbers of Data (Part 1)

Generally speaking, the numbers tell us how big the graph is, in terms of nodes and vertices, but also how deeply connected it is. Looking at these numbers make it easy to
<table>
<thead>
<tr>
<th>Graph</th>
<th>(B_6)</th>
<th>(B_7)</th>
<th>(B_8)</th>
<th>(B_9)</th>
<th>(B_{10})</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advogato</td>
<td>0</td>
<td>89</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adol. Health</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC Alpha</td>
<td>30987</td>
<td>101599</td>
<td>272668</td>
<td>5962</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1.4: Betti Numbers of data (Part 2)

<table>
<thead>
<tr>
<th>Graph</th>
<th>Source</th>
<th>Fill (%)</th>
<th>Avg. Deg</th>
<th>Max. Deg.</th>
<th>Euler Charact.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advogato</td>
<td>Konect</td>
<td>7.06</td>
<td>6.92</td>
<td>21</td>
<td>-96</td>
</tr>
<tr>
<td>Adol. Health</td>
<td>3.51</td>
<td>3.44</td>
<td>11</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>Wiki</td>
<td>Konect</td>
<td>10.2</td>
<td>10</td>
<td>42</td>
<td>4</td>
</tr>
<tr>
<td>BC OTC</td>
<td>SNAP</td>
<td>11.8</td>
<td>11.56</td>
<td>65</td>
<td>25</td>
</tr>
<tr>
<td>BC Alpha</td>
<td>SNAP</td>
<td>34</td>
<td>33.36</td>
<td>67</td>
<td>188858</td>
</tr>
<tr>
<td>Open Flights</td>
<td>Opsahl</td>
<td>6.57</td>
<td>6.44</td>
<td>26</td>
<td>49</td>
</tr>
</tbody>
</table>

Table 5.1.5: Other Characteristics

understand the structural differences among the data-sets/graphs. Even though we only took 50 nodes for every graph, we see that the differences in the n-dimensional simplices are considerable. This means some graphs are more deeply connected than others, regardless of their (original) size. Something else we notice just by looking at these tables is that in all graphs, we see that the number of n-dimensional simplices increase from \(n = 0\) until some given \(k\) and then decrease again until it reaches zero at some given dimension. These are a few of the comparisons we will take a look at when analyzing how the graphs change during and after the algorithm has been run. In the next section we explain which tools are most useful to see the change in the graphs, and what the best way to visualize these changes is.

### 5.2 Barcodes and filtration diagrams

The previous chapter explained how we can obtain information through several topological properties by describing a graph as a flag complex. However, sometimes it is useful to restrict the formation of simplices to obtain different complexes. This restriction is usually done through a “filtration”. A filtration of a flag complex is a sub-complex of the original where simplices which fail to satisfy a certain condition are removed. The study of the resulting complexes obtained by adjusting the filtration parameters (threshold), and their properties is known as persistence homology. Based on this intuition, the rigorous definition of filtration is as follows:

**Definition 5.2.1.** Given a simplicial complex \(K\), a filtration is an ordered set of simplicial complexes \(K^n\) of \(K\) such that for every \(n\) if \(i \leq j\) then \(K^i \subset K^j\). Each simplex \(K^n_i\) is denoted as an interval \([i, i+k]\) where \(i\) is the time the simplex appears in the filtration and \(i+k\) is the time where it becomes the face of a simplex that appears at time \(i+k\). We say the simplex was born at time \(i\), and died at time \(i+k\).
Remark. This means that a filtration is an ordering all of the sub-complexes of the flag complex. Furthermore, a filtration satisfies the condition that a simplex can only appear in the filtration once all its faces have appeared already.

While some theory defines a filtration as a completely ordered set, here we allow for simplices of the same dimension to appear in the filtration at the same time or without a specific order depending on our filtration algorithm.

Example 12. The zero filtration lists all the simplices possible, in a flag complex and they are all born at time zero and continue to live until "infinity".

Remark. The time at which each homology generator appears its called its birth-time and the time at which it stops to appear its called its death time.

Example 13. The zero filtration has as a filtration function, the zero function.

In this paper we will focus on two filtrations.

5.2.1 Dimension Filtration

This first way in which we will compute filtration homology is quite simple. This filtration function is very useful when dealing with unweighted graphs and it is good to get a basic understanding of how the graph looks like in general terms. In this filtration, the filtration function is defined as follows:

Definition 5.2.2. Given a complex $K$, the dimension filtration function

$$f : S \to \mathbb{N}$$

assigns all $n$-simplices a weight of $n$. That is, given a node $v$ (0-simplex), our function gives $f(v) = 0$. Given, an edge $e$ (1-simplex), we have $f(e) = 1$, and so on. (This definition taken from Daniel Lügehetmann. He implemented in [flagser])

When this filtration is applied to a weighted directed complex, we get filtration intervals of one of two forms: $[n, n+1]$ where $n \in \mathbb{N}$ and where $n$ is the dimension of the complex, or $[n, \infty]$ meaning the simplex never becomes a face of an $n + 1$-dimensional simplex.

Example 14. Take the graph $G$ below, and let $K$ be its corresponding flag complex.

![Diagram of a graph](image)

$$K = \{K^0, K^1, K^2\}$$

where

$$K^0 = \{x, v, w, z\}$$

$$K^1 = \{xv, xw, vw, wv\}$$

$$K^2 = \{xvw\}$$

When applying the dimension filtration to it, we get first a function and then the filtration itself. The function is the following:

$$f : K \to \mathbb{N}$$

where:
\[
\begin{align*}
f(x) &= 0 \\
f(v) &= 0 \\
f(w) &= 0 \\
f(z) &= 0 \\
f(xv) &= 1 \\
f(xw) &= 1 \\
f(vw) &= 1 \\
f(xwv) &= 2
\end{align*}
\]

The resulting persistence intervals are represented as follows:

<table>
<thead>
<tr>
<th>#dim_0</th>
<th>#dim_1</th>
<th>#dim_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,1]</td>
<td>[1,2]</td>
<td></td>
</tr>
<tr>
<td>[0,1]</td>
<td>[1,2]</td>
<td></td>
</tr>
<tr>
<td>[0,\infty]</td>
<td>[1,2]</td>
<td></td>
</tr>
<tr>
<td>[0,\infty]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

These intervals can be read like so: In the zero-dimension we see there are four intervals. That agrees with our graph since there are four nodes. This is always the case for filtrations; in dimension zero we should get as many intervals as we have nodes. From these four nodes, we see that two have death time 1, that means they join another simplex in dimension 1. Say x and v are the two first simplices with death 1 and that w makes it onto infinity along with z. In dimension 1, we can have at most two new intervals here since only two died in dimension 0 Since these represent the now 1-dimensional simplices arising from nodes x and v and connecting to w, we say that these represent the edges xv and vw. These two intervals die in dimension 2, since in dimension 2 the triangle xvw is born. The reason why the triangle is not listed on the dimension 2 list, is because this is technically the simplex shown under the list of dimension zero that made it to infinity, so it shouldn’t be listed twice. Note that at the end, we have two connected components, which are the two simplices that made it from [0, \infty] one is the node z which has dimension 0, and the other one is our 2-dimensional simplex xvw.

**Remark.** Note that the dimension filtration doesn’t show the creation of all edges. For example, if the edge vw didn’t exist, this wouldn’t be reflected in the persistence intervals.

### 5.2.2 Vietoris-Rips filtration

The second type of filtration, works perfectly in weighted graphs with very little adaptation. Originally created to filter according to distances, the Vietoris-Rips filtration is a continuous filtration (whereas the dimension filtration is a discrete filtration). Under this filtration, a simplex can be born or die at any given \( t \in \mathbb{R} \). This is different to the dimension filtration, where intervals are always natural numbers or infinity; \( t \in \mathbb{N} \cup \{ \infty \} \).

The Vietoris-Rips filtration is designed to show how a weighted graph would look like if we deleted (or filtered) all edges with distance less than \( t \in \mathbb{R} \). Therefore, in order to define the Vietoris-Rips filtration, one needs a notion of distance (or weight \(^4\)) between nodes. So to define the filtration, we first need a Vietoris-Rips complex, which we form from a graph \( G = (V, E) \), and its corresponding (pseudo-)distance function \( d : V \times V \to \mathbb{R}^+ \cup \{ \infty \} \).

**Definition 5.2.3.** Given a graph \( G = (V, E) \) with its corresponding (pseudo-)distance function \( d : V \times V \to \mathbb{R}^+ \cup \{ \infty \} \), and a given \( \varepsilon > 0 \) the associated \( \varepsilon \)-**Vietoris-Rips complex** of the graph is the simplicial complex constructed by

\(^4\)We explain later on how we can use this filtration when the values given to edges represent the weight of a connection instead of the distance between the nodes.
1. Creating a sub-graph \( \mathcal{H} = (V', E') \) of our graph \( \mathcal{G} \), where \( V' = V \) and

\[
E \supseteq E' = \{(v, w) \in E | d(v, w) \leq \varepsilon\}
\]

2. Taking our graph \( \mathcal{H} \) and creating the associated (weighted) \( K \) simplicial complex of our graph.

3. The resulting simplicial complex \( K \) is our \( \varepsilon \)-Vietoris-Rips complex.

The simplicial complex allows us to see how the complex would look like if only the stronger connections existed. That is, if only those connection with distances smaller than \( \varepsilon \) existed. However, it is even more interesting to see how the shape (and properties) of such \( \varepsilon \)-complexes change, as we let \( \varepsilon \) change. This is what the filtration explains.

**Definition 5.2.4.** Given a distance function \( d : V \times V \to \mathbb{R}^+ \cup \{\infty\} \), we define the **Vietoris-Rips filtration** as the resulting set of Vietoris Rips (sub-)complexes constructed by increasing the value of \( \varepsilon \) such that at its minimum, the resulting simplicial complex consists only of edges which connect nodes that are \( 0 \) distance apart, and such that at its maximum we get back our original graph \( \mathcal{G} \).

This definition is clear whenever we have a distance function. However, in our case we don’t have an explicit notion of distance between nodes, only edges with weights. So using these weights, we can aim to construct a notion of distance. It is important to recall that when standing at a certain node, the probability of choosing a given edge, and therefore going to the goal node is exactly the weight of the edge. Hence, if two nodes are connected through an edge with high weight, the distance between them should be small. This sounds intuitively correct, but an avid observer might spot one more hurdle to jump; we have directed graphs only. This means that the weights could vary depending on the direction of the edge; if they even exist on both directions. So just basing our distance function on the weights would lead to an asymmetric function, which would therefore not be a distance.

In order to prove this is not a problem, we need to go back and see the way we defined complexes given a weighted and directed graph. There are two facts in the definition that ensure that we can still define a pseudo-distance-like function that makes sense. Firstly, we must know that a word is a combination of letters in a specific order. This means that two words, containing the same letters, are different as long as the letters are in a different order. Secondly, every word may have each letter only once. These facts together mean that within a word, given two letters, we will see only one edge between them and for the purpose of that word, the other edge will be non-existent. This is easier to understand when we look at an example.

**Example 15.** Take the a directed graph with three vertices \( x, y, z \) and with edges in both directions between each pair of vertices. That is, we have a fully connected directed graph, and the edges are \( xy, yx, xz, zx, yz, zy \).

From the following graph we get the following topological properties:

- 0-simplices: 3
- 1-simplices: 6
- 2-simplices: 6
While the count of 0 and 1 simplices is completely uninteresting, in the 2-simplices we have the answer to our distance dilemma. Whenever we are connecting an point to each other we have two possibilities (one for each directed edge). And whenever we get a 2-simplex, it means we chose one edge for each pair of nodes and ignored the other edges. Therefore, we have six different simplices:

Figures 5.2.2 and 5.2.3 are two examples out of the six simplices that come from the graph in 5.2.1.

What these pictures mean to show is that, every simplex has only one edge in every direction. So even if in the original graph there are two values of distance between two nodes, for the purpose of the filtration there is only one edge per every two nodes. Another way of looking at it, is that in filtration's the node appears twice (or it has a copy), once with an edge in every direction. This is possible because we are working without weight on the nodes themselves. Therefore, the nodes appear always at time 0 regardless of the filtration, and the edges are the ones that dictate when the simplices appear. For example, if we say that edges with stronger weights appear first, then the word on the left (xzy) would appear first as it would be complete at $t = 0.5$ whereas the other would only be complete when going down until 0.3. But at no point during this filtration is there a conflict between the two words.

Now, we have explained why this works in terms of weight. So we need a function that represents the weight as a "distance", and larger weights give smaller "distances". First, lets look back at the rigorous definition of the weight function.

As said before, the weight represents the probability of going from the source to the goal
node. So depending on how we define it, we either get a function $f$ defined for all of $V \times V$, and with a range of $[0, 1]$ or a positive function only defined on $E$, mapping to $(0, 1]$. However, in the case of the Vietoris Rips filtration, it is convenient to work with the first definition of the weight distributing function. And this is why we defined it like so in the Defining Graphs chapter. With this definition, we have a candidate function $f' = 1 - f$ that is defined at zero as well; a perfect candidate for the distance function. With this distance function, we could start the filtration at zero with all the nodes being there, and the only edges being to themselves with weight zero. That is, all nodes are isolated and there are no issues with definiteness. Then, the filtration parameter $\epsilon$ can steadily increase, and only edges that have a distance shorter than $\epsilon$ would appear. Then, using our definition of word, any group of nodes and edges with distances $\leq \epsilon$ satisfying the conditions from the definition, would give rise to a simplex.

Let's look at the function and its impact on the following example.

**Example 16.** Take the weighted directed flag complex $K$, resulting from the following graph:

![Graph Image]

Here, all edges going out of a node add up to 1 and each weight represents the probability of going into that node. If we were to create the filtration only with the information directly from the weights we would get the following:

<table>
<thead>
<tr>
<th>Original weight function</th>
<th>Resulting inverse function</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f : V \times V \to [0, 1]$</td>
<td>$f^{-1} : [0, 1] \to V \times V$</td>
</tr>
<tr>
<td>$f(x, v) = 0.3$</td>
<td>$f^{-1}(1) = {(v, z), (w, x)}$</td>
</tr>
<tr>
<td>$f(x, y) = 0.7$</td>
<td>$f^{-1}(0.7) = {(x, y)}$</td>
</tr>
<tr>
<td>$f(y, v) = 0.4$</td>
<td>$f^{-1}(0.6) = {(y, z)}$</td>
</tr>
</tbody>
</table>

This means that using the inverse function as our filtration function, we would see that at zero only the nodes appear, but then we would see the nodes with lower weights appear first, and the ones with higher weights at last. We want the opposite from this because the filtration is supposed to be the inverse of a distance function. Therefore, we want to see nodes that are closer together connect first and increase the threshold gradually until we get the whole graph. In our case, we don't have a well-defined notion of distance since we have a directed graph. However, we can say that a measure of how close two edges would be can be relative to the strength of the path. That is, if there is an edge with a high weight between two nodes, we say they are “closer” to each other since it is “easier” to go from one node to the other. So as said before, we want the filtration function to be the inverse of $f(e) = 1 - f(e)$ where $f$ is the weight function of the edge $e$. That way, edges with higher weights would be treated as representing a shorter distance.
**Definition 5.2.5.** The weight-based closeness function $f'$ is a function defined to represent nodes with edges with high weights between them as being closer together.

$$f' = 1 - f$$

where $f$ is the weight distributing function. Two nodes are said to be far apart if $f'$ of the edge between them is large (1 being the maximum value, and meaning the two points are not connected by an edge).

**Remark.** We talk in terms of closeness and not distance or pseudo-distance due to the fact that in a directed graph the weights of the edges can be different in each direction. In order to keep things rigorous we cannot ignore this asymmetry, but for computations and filtrations, this function would serve as our distance function.

The values of the function can be easily represented in a matrix. Before, we can find our matrix let’s look at what a basic “adjacency matrix” is.

**Definition 5.2.6.** Let $G = (V, E)$ be an undirected graph. The adjacency matrix of a graph or system with $n$ nodes or states, is an $n \times n$ lower triangular matrix in which the entrance $i, j$ represents the distance between nodes $i$ and $j$.

**Remark.** The function is lower triangular because distance is symmetric, and therefore we can ignore the upper half. Furthermore, the diagonal is always zero.

We can think of the adjacency matrix, as an alternative representation to a distance function. That is, a function that instead of telling us how much an edge weight, it would tell us how “long” the edge is, or how far apart the two points are. The same idea can be applied to a directed graph, except now our distance matrix would not be necessarily symmetric.

**Definition 5.2.7.** Let $G = (V, E)$ be a directed graph. The directed adjacency matrix of a graph or system with $n$ nodes or states, is an $n \times n$ matrix in which the entrance $i, j$ represents the distance from node $i$ to node $j$. The matrix need not be symmetric, as the distance from node $j$ to node $i$ is represented in the entrance $j, i$ and may differ to the weight in the opposite direction.

If we wanted to represent weight, we could also have a weight matrix. Which took the values of the weight function and plotted them into the matrix.

**Definition 5.2.8.** Let $G = (V, E, \sigma)$ be a weighted un-directed graph. The weight matrix of a graph or system with $n$ nodes or states, is an $n \times n$ lower triangular matrix in which the entrance $i, j$ represents the weight of the link between nodes $i$ and $j$.

Similarly to the adjacency matrix, if the graph were un-directed, the matrix would be symmetric, but otherwise it wouldn’t need to be.

**Definition 5.2.9.** Let $G = (V, E, \sigma)$ be a directed, weighted graph. The directed weight matrix of a graph or system with $n$ nodes or states, is an $n \times n$ matrix in which the entrance $i, j$ represents the weight or “strength” of the link from node $i$ to node $j$. The matrix need not be symmetric, as the weight from node $j$ to node $i$ is represented in the entrance $j, i$ and may differ to the weight in the opposite direction.

### 5.2.3 Computation

Now that we have the definition of what a filtration is and how we will construct it here, we are ready for the computations. Since, the data-sets are so large, we will provide one
small example to give an intuition of what the filtration says and how it looks like. For
the following computations we used the --filtration option in Flagser.

Example 17. Using the same complex as in 16 we define its respective closeness function.
We already have a weight distributing function, we will list its non-zero values below.
Every other edge (or node pair) not included is in the kernel of the function. Using this
function we define the closeness function. Here, the elements not included are assumed to
have a maximum distance of 1, and were those in the kernel of the weight function.

\[
f: V \times V \to [0, 1]
\]
\[
f(w, x) = 1
\]
\[
f(x, v) = 0.3
\]
\[
f(x, y) = 0.7
\]
\[
f(y, v) = 0.4
\]
\[
f(y, z) = 0.6
\]
\[
f(v, z) = 1
\]
\[
f': V \times V \to [0, 1]
\]
\[
f'(w, x) = 0
\]
\[
f'(x, y) = 0.7
\]
\[
f'(y, v) = 0.6
\]
\[
f'(y, z) = 0.3
\]
\[
f'(v, z) = 0
\]

Having defined this function, we are ready to feed in Flagser a directed graph with a
distance notion for each edge. Given this information, flagser will compute the filtration
function for the Vietoris-Rips which would look like this.

\[
g: [0, 1] \to V \times V
\]
\[
g(0) = \{(w, x), (v, z)\}
\]
\[
g(0.3) = \{(x, y)\}
\]
\[
g(0.4) = \{(y, z)\}
\]
\[
g(0.6) = \{(y, v)\}
\]
\[
g(0.7) = \{(x, v)\}
\]

In the end, the filtration produced by Flagser (using this function) looks like this:

<table>
<thead>
<tr>
<th>#dim_0</th>
<th>#dim_1</th>
<th>#dim_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0, 0.3)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0, 0.4)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[0, \inf)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Now, we will provide a short explanation about these intervals. Because there are two
edges with weight 1, this translates into distance zero. Therefore, at time zero, there is
already only three connected components \{x, w\}, \{y\} and \{v, z\}. This is why we only see
three intervals in the dimension zero. Now, at time 0.3 our node z connects to node y and
so the first interval dies. We still have no triangles, just a chain of edges. Now, at time
0.4, y connects with z and our second interval dies, leaving us with only one connected
component, the last interval, which goes on until infinity. Because no new intervals are
born from the ones that die, this filtration is not very informative in this specific case. Note
that we have two 2-dimensional simplices and neither of them appears in the filtration.
Therefore, it is not a replacement for the Betti numbers. However, we do see how the
graph is getting connected and how its structure changes over time. Because it will be
useful for the next section, we will provide the dimension filtration for this graph as well:
#dim_0 #dim_1 #dim_2
[0,1) [1,2) 
[0,1) [1,2) 
[0,1) 
[0,1) [0,inf]

## 5.2.4 Graphing

After computing the filtration, there are a couple of ways in which we can show the results obtained. The first one is called a birth-death diagram. It has time on the x and y axes, and every interval is a point located at a coordinate (x, y) where x is its birth-time and y its death-time. Below is the birth-death diagram for the dimension filtration in our example.

Next, we have the diagram that helped give the name to this chapter; the barcodes. Although they present the same information as the birth-death diagram, in my opinion, it is easier to read information in this format when dealing with bigger graphs. The x-axis is time and each interval is a line that starts at its birth-time and stops at its death-time. In this specific case we see that the lines are red for the zero dimensional simplices and black for the one dimensional simplices. The same diagrams but for the Vietoris Rips filtration are presented below, with one additional diagram being presented; the lifetime diagram. In this diagram, the x axis is still the birth time, but the y axis represents the entire lifetime.
The last type of graph we will use is the signature of a homology. The signature, is a line plot that tells you the dimension of the homology (i.e. number of intervals of the filtration) of the graph at distance (or sometimes called time) $t$. The $x$ axis is the distance (or closeness-based weight function), and the $y$ axis is the dimension of the homology.

The signature has been given its name, due to the fact that on previous experiments$^5$, it has been quite different from one dataset to the next. Below, there are only one or two signatures per graph, but normally there are 9 or more since there is one for every dimension.

$^5$The experiments referenced above were done in Wojtek Chacholski's class Topological Data Analysis in 2018. For the experiment every dataset was a set of $x$ points and the points together formed a shape (with some noise). The signature (among other graphs) were computed, and the shape could be distinguished from the signature, regardless of the amount of points and the noise percentage.
5.3 The Algorithm

Throughout this paper, we have said that its goal is to analyze how the topological properties of the data-sets change in response to an algorithm. In order to justify why this is a valid question, we must explain the algorithm.

5.4 Its purpose

The algorithm was created to rank websites by their accessibility in a similar way Google PageRank does it. What PageRank does, is that it determines the relative importance of two websites by seeing how easily one can reach one website from the other. This is why it is useful to represent the internet as a weighted, directed graph. Every website is a node, and every link is a directed edge with a certain weight. However, there are some websites without “outgoing” edges, and so the random walker could get stuck at any of these given websites. To avoid this, Google’s algorithm has what we call a “random jump”. It means at every step, the walker has 15% chance of jumping to any node on the internet, whether it is connected to the current node or not. The purpose of the new algorithm is to eliminate or reduce the use of the random jump, in order to see if the algorithm then becomes more efficient.

5.5 Our take on it

With this goal in mind, we created the following algorithm. Again, we represented the internet as a directed weighted graph. However, we edited the graph somewhat before starting the actual ranking process. First, we made sure that the weight of all outgoing edges per node, were normalized to add up to 1. This way, every weight would represent the actual probability of choosing that edge. Then, for every edge \((i, k)\) on the graph, we searched to see if the opposite edge \((k, i)\) existed. If the mirror edge was not there, then we would add it to the graph with a very small weight, of 0.005, and re-normalize the graph. This was to avoid getting stuck on nodes with no outgoing edges. Then, we introduced a Random Walker, whose job is to walk through the graph. The walker starts at the source node (website) and travels randomly, using the edges (and its weights) through the internet, looking for the goal. Originally, the idea was to run the walk several times, and see how often would the walker find the goal first and how often would the walker get back to the source before reaching the goal. The ratio of successes (number of times the walker would find the goal first) to failures (number of times the walker would return to the source first) was to be recorded, and then the source and goal nodes would switch roles and the process would be repeated. The ratio of successes both ways would then be the relative importance of one node over the other. In the first version of the code, the walker would keep track of the websites it had visited before, and when it would determine it had been at a set of nodes for too long, the connections between these nodes would be change in order to help the walker escape. This algorithm worked but it was not efficient.

In hopes of improving it, we decided that instead of helping the walker after it had gotten stuck, we could simply make sure the walker never got stuck at all. We would still add the

---

6 such as PDFs or images
back-edges were they were not present, but the rest of the algorithm is quite different. It started at the source, and deleted every node the walker visited, but connected all of its neighbours to each other. This algorithm had a re-introduction of the random jump, but it was only to be used when the current node had no outgoing edges. However, because of the artificial back-edges, and because we are constantly connecting all neighbours of the current node with each other, the random jump was only necessary if we reached the last node of an isolated connected component of the graph. While this could happen with certain test data-sets, it was unlikely that we would find isolated regions of the internet.

We will refer to this algorithm as the Greedy algorithm or the Greedy version of the algorithm. Below is a pseudo-code that describes it:

This algorithm was more efficient than the first version, but returned very densely populated graphs. At the end of the algorithm, the graphs had significantly more edges and with very low weights. This is further explained in the results section.

Because having very many low weight edges, can make any search algorithm more inefficient, we would like to avoid this whenever possible, without spending too much time in certain nodes. This led us to tweak the algorithm further. We separated the WalkDelete function into two, a Walk function and a Delete function that would only be called when the current node satisfies a certain condition. Right now, the condition is that the given node has less than \( x \) neighbours. Nevertheless, more testing is encouraged in order to find out what is the optimal \( x \) or what is the optimal condition. This version of the algorithm looks as follows:
WalkDelete(graph, run_parameters) /* algorithm as a recursive function */
{
    while (walk_not_ended graph_size > 1)
    {
        if (current_node.has_neighbours())
        {
            choose a neighbour p using weights;
        }
        else
        {
            /* This only happens if the graph has more than one
               connected component, which is rare for the internet */
            choose a random node p on the network;
        }
        forall (neighbours e of the current node n) do
        {
            if (e == n) /* we get ready to delete this node, but
                           without losing its connections */
            {
                Delete e from neighbours;
            }
            forall (neighbours j of n) do
            {
                Weight(e,j) ← Weight(e,j) + Weight(e,n) *
                Weight(n,j) delete from neighbours j
            }
        }
    }
    /* Step to p; 
    Delete n from the Graph; 
    if (n == goal node OR graph_size == 1) 
    {
        End the walk;
        effective ← graph_size / (orig_Graph_size - 1)
    }
    return effective 
}
/* This is the definition of the delete function, called in the algorithm which is the Walk function. */
def DeleteNeighbours(graph, current_run_state)
{
    forall (neighbours e of the current node n)do
    {
        if (e == n) /* we get ready to delete this node, but without losing its connections */
        {
            Delete e from neighbours ;
        }
    }
    forall (neighbours j of n)do
    {
        Weight (e,j) ← Weight(e,j) + Weight(e,n) * Weight(n,j) delete from neighbours of j
    }
}

Walk(graph, run_parameters) /* The actual algorithm */
{
    while (walk_not_ended graph_size > 1)
    {
        if (current_node.has_neighbours())
        {
            choose a neighbour p using weights;
        }
        else
        {
            /* This only happens if the graph has more than one connected component, which is rare for the internet */
            /* choose a random node p on the network; */
            Step to p ;
        }
    if (previous_node.neighbours.size < x) /* Because we just took a step, then n is now the previous node */
    {
        DeleteNeighbours(graph, current_run_state) ;
        Remove n from the Graph;
    }
    if (n == goal node OR graph_size == 1)
    {
        End the walk ;
        effective ← graph.size / (orig.Graph.size - 1)
        return effective
    }
}
(a) Birth-death diagram

(b) Birth-death diagram zoomed in

Figure 5.2.4: 2 Figures side by side
Figure 5.2.5: Barcodes

(a) Birth-death diagram

(b) Lifetime diagram

Figure 5.2.6: Vietoris Rips filtration
Figure 5.2.7: Barcodes VR filtration

Figure 5.2.8: Signatures for the dimension filtration
Figure 5.2.9: Signature for VR filtration
Chapter 6

Results

We ran both algorithms on every dataset and during the process, we printed snapshots of the graphs to see how the structure changed during the run. We then took these in-between graphs, cleaned them up and computed the same topological properties we computed for the original graphs. We also ran the algorithm many more times just saving the graph that resulted at the end, with the hope of comparing just the signature among all runs and seeing whether there was a pattern.

The idea was to see a more aggressive change among the runs of the greedy algorithm, and a slower change when the other algorithm was ran. This is why we did the process mentioned below for both algorithms and then compared the results.

6.1 Running the algorithm

As said before, the graphs were normalized, and then the topological information was gathered on the normalized graphs. Afterwards, both algorithms were ran on every graph, and on every step (except the last one) the random walk went from node 4 to node 6. These were the steps we took:

1. The algorithms were ran 50 times without saving any information. This was only to check how many steps the process should take in average.

2. Then, snapshots of an “average” run were recorded for each graph for every algorithm. The goal was to have between 3 and 5 snapshots of every graph during the run in order to see the gradual changes of the graphs. ¹ These snapshots would serve as a visual aid of how each graph was affected by the algorithms.

3. We then ran flagser on the snapshots with the goal of recording the same topological information we had for the original graph and compared them.

4. Afterwards, we obtained the filtrations for the snapshots and compared the information with the original graph.

¹It was difficult to have the same amount of graphs per dataset because in the highly connected graphs, the algorithm reached the goal node in sometimes 5 or 7 steps, making it pointless to have more than 2 or 3 snapshots.
5. In addition to these snapshots, which mean to show how the graph evolves in the process, we also ran the algorithm 100 times per graph saving only the final version of the graph (no inbetween snapshots). We then computed the signatures and compared all of them with that of the original run.

6. We then ran the algorithm 30 more times on every graph, but each time with a different source and goal node. We then compared the signatures of these 30 graphs with the signature of the original graph.

6.1.1 Cleaning up the data

Before analysing any of the graphs obtained in the process above, we cleaned them. This meant, deleting edges that were loops. This was necessary because Flasger doesn’t support loops and in our case, we were only interested in edges with different source and goal nodes.

6.2 Looking at the snapshots

As said before, the first thing we did was record a single run for every graph in which we saw how the graph looked like at several stages of the run. We then observed how every topological property changed in every snapshot in comparison to the rest and to the original graph. The specific results are in the sections below. We expected the connectivity of the graph to increase pretty quickly, and the number of connected components as well, the latter especially with the greedy algorithm because every node visited would then be isolated.

6.2.1 Greedy Algorithm results

Using the cleaned snapshots, resulting from running the greedy algorithm on our original graphs, the idea was to compute the different filtrations and compare the resulting barcodes and signatures to those of the original graph. However, even before the comparison we got our first result. The increased number of edges also led most times to an exponential increase in the number of all simplices whose dimension was bigger than 0. The increase happened so quickly, that some snapshots of graphs such as Bitcoin Alpha and Adolescent health, which were previously small and quick to analyze, became too large for Flasger to compute its filtrations. Of course, with the bigger graphs, it was even worse. The bigger graphs that were strongly connected, were already being a challenge for Flasger, and it was no surprise to see that after the algorithm started being run, the resulting graphs were also too large to compute. Because Flasger yielded no results whatsoever, it is impossible to say how much larger the new graphs really were.

This meant, the same topological properties we computed in the beginning could not be

---

2This means the graphs with the full number of nodes and edges, before they were cut to a size of 50 nodes.
computed in the edited graphs, and because of this we decided to crop the graphs. ³ We cropped each graph to have a total of 50 nodes, plus the node zero which Flagser always counts. ⁴ We did this with the hope of being able to analyse the results of the greedy algorithm on Flagser.

Unsurprisingly, the drastic cropping, made Flagser calculations much quicker on the original graphs. The basic Flagser command, which before could take several hours for some of the graphs, was done in seconds for some of the cropped graphs and in less than an hour for all cropped graphs. As for the snapshots after the algorithm was ran, we were finally able to obtain results, of which two were very important. The first was that, regardless of which nodes were chosen as source and goal, the algorithm could always find the goal very quickly. The number of steps varied, but the computation time remained below the minute mark for all graphs. This was not the case for the entire graphs, where the running time could exceed five minutes. ⁵

The algorithm also had a relatively high success rate, with only one graph, Moreno, constantly passing the 40 steps mark.

However, after obtaining the snapshots of the graphs during the runs of the algorithm, Flagser still struggled to compute even the most basic properties for most snapshots. The Facebook graph was the only one for which Flagser could still compute all properties in all snapshots on the laptop without the help of the cluster. The Moreno was the only graph (aside from the Facebook graph) that returned results when ran on the cluster. This meant that even though the algorithm was relatively quick, it was increasing the dimensions of the homologies very quickly. Seeing this increase in running time for such small graphs, it occurred to us, that the algorithm was too aggressive. It was maybe unnecessary to compress at every step, and make the graph so highly connected so quickly. This was not only causing all edges to have very small weights, but also, creating almost fully connected graphs within a few steps. This is when we decided to try a less aggressive version of the algorithm, which would only compress certain nodes. This is less aggressive version is the second version seen in the Algorithm chapter. The hope was that this version would still be similarly quick, and that it would work on a less dense graph, keeping the weights of the edges significantly higher and keeping more of the original structure of the graph. The results of this less aggressive version are shown in the next section.

**Simplex Count**

While the more "aggressive" version of the algorithm didn't produce all the results we planned, it did produce a simplex count for all graphs, and full results for the Facebook and Moreno graphs. We obtained the simplex count for all snapshots using the -count

---

³This decision came as a result of trying several techniques to analyse results using Flagser and realizing none of them worked. See appendix 2.

⁴The node zero is also always completely isolated, and therefore the actual number of connected components is one less as our results state. We ignored the zero nodes and computed properties such as the fill assuming the number of nodes was 50.

⁵These running times are measured by running the algorithm on Xcode on a MacBook Pro 2015. Total running time could decrease if the algorithm were to be run in a computer with more RAM and more available disk space. However, for the purpose of consistency, when measuring time, we always ran the algorithm on this computer. Once the final version of the algorithm is developed, time tests in the cluster are recommended.
option on Flagsr. Even though we obtained results for all graphs, some graphs had to be altered in order to obtain such results. Some snapshots were so highly connected that we were not getting any results with the -count option, even when ran on the cluster. For some graphs, there would be no results, even after the cluster ran for 24 hours, and in some cases the cluster would also run out of memory. For those graphs for we couldn’t obtain any results, we started deleting edges with the smallest weight. This was the case for snapshots of the Advogato, Opsahl, wiki, and alpha graphs.

For every snapshot for which we couldn’t get any results we started by deleting all edges with weight less than 0.001. To do this, we deleted every line with the string 0.000 and re-normalized the graph. Then, we ran ./flagsr-count again. The decision to do this was justified by the fact that deleting an edge with less than 1% of possibility of being chosen will not affect the random walk of the algorithm very much. However, these small edges still contribute heavily towards the construction of simplices and slow the computations in Flagsr quite a lot. The deletion of such edges gave results for some snapshots, but for some others we had to delete edges with less than 0.002, 0.003, or 0.004 weight in order to obtain the -count results.

Below we will show the results for Facebook and Moreno, showing a table with the simplex count at each stage, but for the rest we will show the results only in the appendix.

Talking about Facebook and Moreno, in both graphs we saw a noticeable increase in the number of edges, but even more so in the number of higher dimensional simplices. As far as our results can tell 6, this happened in all datasets, during the first steps of the algorithm. This is because during the first steps, there is a higher creation than a loss of edges. You only lose the original edges of a node, but you connect all its neighbours to each other, increasing the simplex count quickly. Therefore, in graphs where the algorithm found the goal node relatively quickly, the final graph was a much higher connected graph, with many more simplices than the original. The Facebook graph is a clear example of this. Here, the algorithm would always find the goal fewer than 20 step, and the majority of times 7 in fewer than 10. When looking at the table, we see a very big increase between the original graph and the first snapshot, yet, the graph keeps growing quite noticeably afterwards as well. The number of 5 dimensional simplices is doubled between the first snapshot and the end of the graph, and the 7 dimensional simplices are more than tripled. Furthermore, we see the appearance of many 10 dimensional simplices, an indicator of very high connectivity, since every 10-simplex is a fully connected set of 11 nodes 8. So the graph only grew, and the process stopped running before the graph could begin to shrink.

Differently, in datasets where the algorithm had to run for longer, we started seeing a decrease in the number of simplices during the last steps, due to the high loss of nodes. In a highly connected graph like the one we get after a certain number of steps in the algorithm, a loss of one node results in a very high loss of edges, without necessarily a big gain (because all the neighbours are probably connected already) and this is when we start seeing a decrease in the number of edges. An example of this is the Moreno dataset run that we show in the second table below. By the time the algorithm was finished, it had deleted almost all nodes, with only less than 10 left. Because of this, we see a clear

---

6 As mentioned before, we couldn’t obtain all topological properties for some graphs, but we could see the simplex count for all
7 approximately 8 or 9 times out of every 10 runs
8 with directed edges of course
<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>snap 3</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>86</td>
<td>88</td>
<td>58</td>
<td>50</td>
<td>34</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>52</td>
<td>150</td>
<td>30</td>
<td>30</td>
<td>30</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>8</td>
<td>360</td>
<td>0</td>
<td>0</td>
<td>24</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>0</td>
<td>720</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>0</td>
<td>720</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2.2: Moreno

decrease in the number of n-dimensional simplices, for all dimensions as well as an increase in the number of connected components, which are all the isolated nodes and we see as Betti Numbers in the sub-section components below. Other examples of this are the advogato graph, which is practically empty at the end, as well as the Wiki, and Opsahl graphs. The wiki graph is also smaller than its original graph after the run, and the Opsahl graph is only somewhat higher connected, with newly formed 4 and 5 dimensional simplices but with less 1 dimensional simplices than its original counter-part.

<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>111</td>
<td>210</td>
<td>220</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>109</td>
<td>852</td>
<td>1122</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>94</td>
<td>5160</td>
<td>8088</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>40</td>
<td>30240</td>
<td>55560</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>0</td>
<td>151200</td>
<td>332640</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>0</td>
<td>604800</td>
<td>1663200</td>
</tr>
<tr>
<td>7-dim sim.</td>
<td>0</td>
<td>1814400</td>
<td>6652800</td>
</tr>
<tr>
<td>8-dim sim.</td>
<td>0</td>
<td>3628800</td>
<td>19958400</td>
</tr>
<tr>
<td>9-dim sim.</td>
<td>0</td>
<td>3628800</td>
<td>39916800</td>
</tr>
<tr>
<td>10-dim sim.</td>
<td>0</td>
<td>0</td>
<td>39916800</td>
</tr>
<tr>
<td>11-dim sim.</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 6.2.1: Facebook

As for the other graphs, the memory required to compute just the simplex count was already significant, but we did obtain results, and a similar pattern as in the two datasets above. The graphs that needed more steps to find the graphs were quite big and highly connected at first, and quite small and almost fully connected (between the nodes that remained) at last. An example of this is the advogato dataset where the first three snapshots are big and highly connected enough such that even the count command cannot run on the laptop, but the last three are smaller, with the last two being very quick to compute. This and the tables for the remaining datasets can be found on the appendix.
Betti Numbers

Directly related to the number of simplices in each graph we have the Betti Numbers. As new connections between nodes appear, and old connections disappear, we expect to see a change in the Betti Numbers. The most obvious change in this section is the change in the zeroth Betti Number, which is the number of connected components. Every step the greedy algorithm takes, leaves behind one more completely isolated node. This means an increase in the zeroth Betti Number of 1 per step. As for the rest of the Betti numbers, we see a pattern similar to that of the $n$-simplices, starting with a big increase, and for graphs where the algorithm didn't find the goal node so soon, a decrease close to the end of the run. An interesting feature of the more highly connected snapshots is the fact that there appears to be an increase in the lower Betti numbers, followed by many zeros in the next few Betti numbers, and then one non-zero Betti number at the end. This is the case for the Facebook run where the first snapshot and the end of the run show this characteristic, but we can also see it in the first snapshot of the Moreno run. The explanation for this is the following: Recall that the $n$th Betti number represents the number of $n$th dimensional holes in our surface. This means, the number of $n$-dimensional simplices that are not boundaries (faces) of an $n+1$-dimensional simplex. Therefore, when we take a snapshot, and see there are a lot of Betti Numbers which are zeros, and they are

\footnote{Therefore one can actually track how many steps the algorithm took just by looking at the difference in $B_0$ between the original graph and the step in question.}

\footnote{The node number 0 is not in any of our data-sets, since they all start from one, but flagser takes it into account, therefore we always see this one isolated component even from the beginning. That is also why the number of 0-dimensional simplices is 51 in all graphs instead of 50.}

\footnote{the highest one that is non-zero}

52
followed by a very large Betti Number, we can infer the following: All the $k$ dimensional simplices for which $B_k$ is zero, are boundaries of $k + 1$ simplices. Similarly, said $k + 1$ simplices are boundaries of $k + 2$ simplices and so on until we reach the $N$-dimensional simplices where the Betti Number is not zero anymore. Those simplices are not boundaries anymore, that is, they are not building higher dimensional simplices. So all the smaller ones, where coming together to build simplices of $N$ dimensions, but not bigger, and this is why we see that result.

As before we leave here the table for the Betti numbers of Facebook and Moreno as an example, and the rest of the tables together with the graphs will be in the appendix.

<table>
<thead>
<tr>
<th>Betti Number</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>2</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>$B_1$</td>
<td>18</td>
<td>37</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>$B_2$</td>
<td>12</td>
<td>23</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>$B_3$</td>
<td>7</td>
<td>58</td>
<td>33</td>
<td></td>
</tr>
<tr>
<td>$B_4$</td>
<td>6</td>
<td>0</td>
<td>53</td>
<td></td>
</tr>
<tr>
<td>$B_5$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$B_6$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$B_7$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$B_8$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>$B_9$</td>
<td>1334961</td>
<td>0</td>
<td></td>
<td>14684570</td>
</tr>
</tbody>
</table>

Table 6.2.3: Facebook: Betti Numbers

<table>
<thead>
<tr>
<th>Betti Number</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>snap 3</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>9</td>
<td>22</td>
<td>27</td>
<td>31</td>
<td>38</td>
</tr>
<tr>
<td>$B_1$</td>
<td>7</td>
<td>15</td>
<td>15</td>
<td>11</td>
<td>8</td>
</tr>
<tr>
<td>$B_2$</td>
<td>8</td>
<td>11</td>
<td>11</td>
<td>11</td>
<td>2</td>
</tr>
<tr>
<td>$B_3$</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td>9</td>
</tr>
<tr>
<td>$B_4$</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$B_5$</td>
<td>265</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 6.2.4: Moreno: Betti Numbers

**4th Result: Barcodes/Life-death diagrams**

Even though the tables give a great overview of how the graphs were changing, barcodes or life-death diagrams are valuable because we can see the change in the homology over "time". In the barcode of a graph we see when every single homology class is born, and when it dies, giving us a general overview of the trend of the graph. Of course, this means in order to have barcodes or life-death diagrams we need to have a filtration. The filtration is what tells us when the simplices are born and when they die. See the Data-sets Chapter for a detailed explanation. Once we choose a filtration, comparing barcodes or life-death
diagrams side by side, can tell us how the general trend of the simplices differ and how they are similar, independently of the end simplex count. Because, it is not useful to plot the barcodes of two graphs in the same diagram, each snapshot has to have its own barcode. This means we will have a lot of graphs and so here we will only show the barcodes for the facebook graphs and the rest will be in the appendix. In the Facebook barcodes, we can see how much the connection between the nodes is strengthened by the algorithm. At first, there seems to be a representative number of barcodes at all stages. But just in the first snapshot we see how many of the simplices are higher now, and how they all have much longer lives.

6.3 Average Runs

After having an idea about how the graph looks while the algorithm run, we wanted to see how the graphs looked after the algorithm had been ran. However, here computing just one run and analysing it wasn’t enough, because we knew that the end result of the graph, depended largely on how many steps the algorithm took to find its goal; especially in the greedy algorithm. This was because as we saw in the previous section, depending of the number of contractions (steps) the connectivity of the graph changed very quickly, increasing the graphs’ high dimensional simplices, up until it reached a "maximum" and from that point on, the simplices started decreasing quite rapidly. Therefore, running the algorithm once and getting a very small graph, didn’t rule out the possibility that a run that was even just a couple of steps quicker would result in a huge graph. Because of this, we decided to do thirty runs, and take this as a good enough sample for a first glance. As expected for all datasets except for one (moreno) the resulting graphs for the aggressive algorithm were much bigger than the original and even bigger than a graph obtained by running the algorithm without contracting any nodes. As for the results of the less aggressive algorithm, the differences were not as big, but nevertheless noticeable. In the next subsections we talk about each type of algorithm, and then compare the two of them.

6.3.1 Aggressive algorithm

The topological data on these graphs is difficult to compute due to the high connectivity of the resulting graphs. As we saw through the snapshots, if a random walk reached its goal node at a point where the graph had heavily expanded, the resulting graph was very heavily connected, and so running Flagser on such graphs would require more than 100GB of storage; making it impossible to find results for some graphs, even using the Cluster. Knowing this, before we tried to run Flagser on them, we tried a different approach to getting a first glance at how big the files were. We began by counting the number of lines in their text files. The number of lines was the total number of edges, and of course, the more edges a graph has the higher the chance of that graph having many highly-dimensional simplices. After seeing how many edges each graph had, we started running -Flagser on all files, starting with those with the least number of lines and increasing that until Flagser took too much memory on my computer and the computation wasn’t possible anymore.
The threshold for the computations was somewhere around 100 lines. There were files such as the 7th run of the wiki dataset, which had 93 lines, and flagser computed its filtration in a couple of minutes. However, there were files like the 13th run of OTC with 113 lines for which after an hour of run-time flagser still didn’t finish computing the filtration. Because of the lack of computational power, we limited the computation, focusing on getting the signatures of four runs per dataset. For those data-sets were all runs were too big to compute, we removed some edges according to the following procedure. First, we deleted all edges with weight less than 0.001 (0.1% probability) and checked again to see if graphs were small enough to compute their signatures. In the case of the Wikipedia and OTC data-sets the files had an average of 586 and 781 lines respectively. Because this was so far off the threshold, we deleted all files with weight less than 0.01 (1% probability). After each deletion of edges, we renormalized the weights so that the sum of all outgoing edges out of a node would be of 1. Using this method we obtained the signatures for four average runs per dataset. We then plotted these 4 signatures per dataset on the same plot together with the original graph, and the graph obtained by running the algorithm without contracting any nodes (we refer to the graph as the graph with no contractions).

The graphs of the Alpha data-set were also very large, and even though we obtained the filtrations using Flagser for four datasets, we couldn’t compute all the signatures together with the original and the graph with no contractions. So for this data-set we plotted just the four greedy runs. In the case of the wikipedia dataset, the fourth run was too large to plot, so we could only show three runs together with the original and the graph with no contractions. Below we will show the graphs for Facebook and Moreno, which are the graphs whose maximum non-zero homology was 8. The rest of the graphs had at least one run going over $H^8$ and their signatures will be shown in the appendix.

The first thing we notice when looking at the signatures, is that in all data-sets, the more lines a resulting graph has, the higher the dimension of the last non-zero homology. This is because, as more edges appear, they first fill out the smaller dimensional holes and create bigger (but still relatively small) dimensional simplices. Once there is enough of these smaller simplices, the higher dimensional holes start appearing, and so the homologies of higher dimensions, starts growing. This is why in the signatures we see that often the yellow line, the “largest” graph $^{13}$, is often the only one to have a non-zero homology in the highest dimension. Similarly, the grey and the blue ones, usually follow, while the red one is in all cases the first one to disappear. This is also why we can see a graph that has a very low homology (or even zero) throughout most dimensions (bigger than 1), and then at some point it goes up for one dimension, before disappearing back to zero. This is the case of the second largest graph (in gray) for OTC, where all its homologies have dimension zero at almost all times, before growing to a very high dimension of its 8th and 9th homology and then going back to zero. This means this graph has no holes in other dimensions, but just in the 8th and 9th dimensions. So this graph is full of very highly connected groups of nodes that are forming 8 or 9 dimensional spheres. We notice that the more edges the algorithm creates, higher the dimension, of the resulting spheres.

$^{12}$Because we had so many graphs, 30 per dataset, it wasn’t an option to run every graph on the Cluster. At the cluster I was using, the waiting time could be from 24 hours to a week per file, and the job itself could take up to 24 hours without any guarantee that the cluster would have enough memory to produce any results. Therefore, for the Average Runs, I ran everything from my Macbook Pro.

$^{13}$the graph with the most edges
6.3.2 Less aggressive algorithm

When running the less aggressive algorithm, it was very important to choose which nodes would be deleted and which wouldn't. It could be either done at random, every certain number of steps, or we could delete all nodes that fulfilled (or didn't fulfill) a certain requirement. To decide which method we would try first, we thought of how our deletion process works. Every time a node gets contracted, all of its neighbours get connected to each other if the connection doesn't already exist. If it does, then it is strengthened (by adding to its weight, the weight of the path going in between the deleted node). With this in mind, a good place to start is by only contracting nodes that are very cheap to contract, that is nodes with few neighbours. So this is the first requirement we test, and the only one we will talk about in this thesis. Other requirements should be tested in the future. So now that we knew we wanted to delete only nodes that had "few" neighbours, we needed to find out what few meant for every graph. To do this, we created a histogram of the
degrees of every graph (shown in the appendix) and chose a threshold on the maximum degree a node could have an still be deleted, so that the algorithm would contract no more than 30% of the nodes. Of course, this was an arbitrary decision, and just as a first test. The value can be changed at any time, and many values should be tested in the future.

Using the histograms, we found out that by deleting nodes with only 0, 1, 2 neighbours we were already deleting 30 – 50% of the nodes, depending on the graph. This means, 15 – 25 nodes could be deleted.

So we set this boundary for all graphs, and ran the algorithm 30 times per graph. We recorded the number of steps it took the algorithm to reach the goal node, the time it took, and how many contractions it did. We also then ran flagser on the graph, and then proceeded to compare the runs with the original graph, and afterwards with the greedy algorithm.
The first thing we saw is that the number of contractions was very low. This means that in the run we chose (node 4 as a source, and node 6 as a goal) our random walker seldom met nodes with 2 or less neighbours. In fact, three from our graphs Advo, Alpha, Wiki had no contractions at all i.e. they met no nodes with less than 3 neighbours. As for the other graphs, Opsahl had at least one contraction in every run, but only 1 contraction per run in 25 of the runs, and 2 contractions in the other 5 runs. Facebook also had contractions in every run, with 1 contraction in 14 of its runs, 2 contractions in 13 runs, and 3 contractions in 3 runs, which was the highest number of contractions per run for all graphs. OTC had only 3 runs with contractions, and in those contractions, there was only of 1 contraction per run. Lastly, Moreno had 8 runs with no contractions, 10 runs with 1 contraction, and 12 with 2 contractions.

Every contraction changed the topological structure of the data, but because every run had only a few contractions, the change in the overall topological structure was small as well. Of course, for the runs where no contractions happened, there were no changes in the structure and therefore we won’t be mentioning them here. It is interesting to mention that for all data-sets, all the runs with only 1 contraction presented a decrease of 2 in the count of one-dimensional simplices (that means that for all graphs with 1 contraction, the node deleted always had two edges) and there were no further changes in the simplex count. That is, all graphs with 1 contraction, presented no change in the number of x-dimensional simplices for x > 1. For the graphs with 2 contractions (from Facebook, Moreno, Opsahl) the 1-dimensional simplex count decreased by 2 more edges, and again there were no changes in the higher-dimensional simplices. Lastly, for Facebook, the runs with three more contractions showed the exact same behaviour. Again, a decrease of 2 more 1-dimensional simplices, and no change in the rest.

This means that the change in the simplex count was very similar throughout data-sets; namely, it increased linearly. Every contraction decreased the count of one-dimensional simplices, while the count of higher simplices remained unchanged.

Since these contractions didn’t alter the topological structure of our data-sets so drastically, we were still able to compute the Betti Numbers for every graph. The changes in the results for the Betti numbers were similar to those in the simplex count. Before we speak about them, we mention that for the purposes of running the calculations on Flagser we ran the software always with 51 nodes, that is we didn’t tell flagser our algorithm deleted nodes. We did that as a way to make sure all edges had been deleted.

As planned, we saw that each contraction represented an increase by 1 in the zeroth Betti number (i.e. number of connected components). That is, all the graphs with one contraction gained exactly one more connected component (namely the deleted node). Similarly, all the graphs with two and three contractions gained two and three connected components respectively (when compared to the original graph).

For the OTC graph, the contraction increased the number of connected components by one from 2 to 3, and the number of holes (the first Betti number) decreased by one as well from 22 to 21. In Opsahl, the original graph had 3 connected components. The graph after 1 contraction it had 4 and after 2 contractions it had 5. The first Betti number started at 18 in the original and decreased to 17 and 16 in the graphs with 1

\[\text{all data-sets refers to the data-sets that presented contractions in at least one of the thirty runs. That is: Fb, Opsahl, OTC, Moreno.}\]

\[\text{The node deleted could also have had only 1 edge}\]
and 2 contraction respectively. In Moreno we saw the exact same behaviour, except the original number of connected components went from 9, to 10 and 11 with one and two contractions respectively, and the number of holes started at 19, and went down to 18 and 17 respectively, with one and two contractions. Facebook like Moreno and Opsahl behaved the same, except this time, the number of connected components started at 2. It then went to 3,4,5 with 1,2,3 contractions respectively. The number of holes started at 61, and went down to 60,59,58 as the contractions went from 0 in the original, to 1,2, and 3. There were no other changes in the Betti Numbers. 16

Lastly, we computed the signatures for the graphs at every dimension and plotted them together. Recall that in order for us to compute the signatures, we first had to adapt our graph so that it showed distance between nodes instead of the weight of the links. As mentioned in the Dataset chapter, we did so by defining a pseudo-distance between nodes, which lets the distance between two nodes be $1-w$ where $w$ is the weight of the link between the nodes. That is, if two nodes are connected by an edge which has probability 1 of being used, then we say the distance between them is zero. Then, using this graph we computed its Vietoris-Rips filtration and from it we obtained the signatures of the graphs.

We computed the signatures of all graphs per dataset together, and because we could safely assume that every run that had presented a contraction had followed the same path (and thus deleted the same node), we took one graph for each total number of contractions, as well as the original graph and the graph that resulted from running the algorithm without contractions. So in each plot, we have the signatures of all dimensions, where every line is a type of graph (where the types are: original, no contractions, one contraction, two contractions, and three contractions). We did this for all four datasets which presented contractions in at least one of our thirty runs. That is, we have the results for the OTC signatures, followed by the Opsahl signatures, and lastly by the FB signatures.

Next we see the signatures for Facebook, where we had one more line in each plot, which represented the graph with three contractions. This signatures was plotted in color gray.

Recall that the x axis of these graphs, represents distance and not weight. So whenever there is an edge with probability 1, this will translate into distance zero. This is why, we see that for some graphs there are less than 51 connected components even at the beginning (see the $H_0$ plot of every data-set).

In terms of actual results, the first thing we see is that for all data-sets, the signatures of the lower dimensions remains relatively similar between graphs. This means, that despite the algorithm being run or not, or contracting nodes or not, the dimension of the zeroth homologies remains very similar for all “distances”. In the case of Opsahl, this is also the case for the first homology. However, for Facebook and Moreno, we see that the original graph is far apart from the rest of the graphs, with a notably lower dimension at all points in distance. This being said, the curve still behaves similarly, although at a lower scale.

As for the higher dimensions, all data-sets behave similarly, presenting very flat lines, and

---

16It is important to mention, that for every graph, the changes per number of contractions were always the same. That is, all the graphs with $x$ contractions of a certain dataset, always had the same topological structure, and the same differences with respect to the original graph. This means it is very likely that they deleted the same nodes, and therefore, took the same path.
a very big difference between the original graph and the rest. The difference between the rest of the graphs, is unnoticeable.

6.3.3 Comparison

In the previous subsections we saw how each of the two algorithms compared to the original graph, and to the graph with no contractions. Here we compare how the two algorithms did against each other, in terms of run time, number of steps, and see if there is any correlation between the changes in the homology and these two factors.
Figure 6.3.4: Opsahl signatures.

Signatures

First we plotted some of the signatures of the aggressive algorithm in the same graph as those of the less aggressive algorithm. That is, we took the three or four runs (depending on the dataset) from the aggressive runs, and plotted them in red, and on the same plot we plotted the resulting graphs of the less aggressive algorithm in red. Once again, we plotted the signature of the original graph and of the graph with no contractions as well, as a point of reference. By doing this we saw that the differences in the signatures was quite significant, and it was easy to tell what version of the algorithm produced which changes in the signatures.

These plots are shown below.

The graphs produced after running the greedy version of the algorithm have a higher number of connected components, which we can see from the zeroth homology, where the
red lines are always above the blue lines, and above the original graph and the graph with no contractions. We also see that the distance between components increases. In the original graph, the graph with no contractions and runs of the the less aggressive version of the algorithm, we see a steeper fall in the number of connected components, and we see it earlier. That is, components come faster together in these graphs, than in the “greedy” graphs, were the zero signature tends to be flatter and have the main fall in connected components at a higher distance. That is, there are more connected components and the connections between components is weaker after running the aggressive algorithm. We also see that for all the data-sets, the greedy graphs have a much lower first homology. That is, there are less holes in our graph than before. This makes sense, considering the nature of the algorithm. We can explain that with the following example.

Suppose the algorithm enters a hole like in step 1 of the picture above. The current node is the one in red in the first picture. The first part of the contraction is the connection of its three neighbours to each other. That is when the edges in the step 2 appear. Then, the
old edges will be deleted as shown in figure 3. Now, these three edges are a fully connected graph, creating a two-dimensional simplex as shown in figure 4. Then, the node is deleted, and the current node becomes the one in red in step 5. Once the node in red in step 5 is deleted, there will be a new triangle formed but instead of being composed of those three nodes, it will contain the node in red's right neighbour, and the current neighbour in red will be deleted. This continues to happen, eventually getting rid of the hole. Of course, the smaller the hole is, the less steps it requires to fill them out, and the more likely the algorithm is to fill it out.

This is why in the signatures, we see a decrease in the first homology at every distance. If the algorithm runs for long enough, this fill in the one dimensional holes can become a filling of the spheres or two-dimensional holes in the graph, leading to a decrease in the second homology, and so on. Then end result, becomes a graph with a lower homology in the smaller dimensions, but with the appearance of holes in higher dimensions, as all the filled holes in the lower dimensions come together to create new ones in higher dimensions.

Run time and number of steps

Above we saw that there are clear differences in the signatures between the graphs resulting from the greedy versus the non-greedy versions of the algorithm. Now we want to compare these two algorithms on their efficiency. To do this, for every run of every data-set, we recorded the algorithms' run time in microseconds as well as the number of steps required before walker arrived to its goal. That is, we recorded this information for 210 runs with the greedy algorithm and 210 for the non-greedy respectively.\footnote{Recall that the many of the runs of the less aggressive algorithm presented no contractions.}

The histograms also show a clear difference between both versions of the algorithms in both parameters; run-time and number of steps. The greedy algorithm had a much higher run-time than the non-greedy, with 90% of the runs taking more than 10,000 microseconds, compared with 20% of the non-greedy runs. The minimum run-time for the greedy algorithm was 2049 microseconds, while the minimum for the non-greedy version was of

\footnote{Recall that the many of the runs of the less aggressive algorithm presented no contractions.}
Figure 6.3.7: Run-time.

Figure 6.3.8: Number of steps.

less than half, totalling 797 microseconds. The average was also a lot lower for the less greedy version, coming to a bit over 12 thousand microseconds, while the average of the aggressive algorithm was of over 48 thousand microseconds. Still, even though the less aggressive version of the algorithm had much lower run-times, it had a much higher variance. For the less aggressive algorithm, almost every bin had a significant frequency, with
about 25% of all runs in every bin.

Given this result, the following step in testing the less aggressive algorithm would be a much larger test, in order to determine whether the distribution of the run-time leans towards a Gaussian distribution with an average between 5000 and 10,000 or, if it could be more accurately represented by a uniform distribution (or any other distribution).

Regarding the number of steps, the aggressive algorithm showed a much lower number of steps. This is to be expected given the nature of the aggressive algorithm. If every step taken is a node contracted, the maximum number of steps before finding the node, equals the total number of nodes in the graph. This happens when the goal node is found after all other nodes have been visited and deleted. So of course, for the aggressive version, no run went above 51 steps. The average number of steps was also lower in the aggressive version of the algorithm with 21 steps taken in average for said version versus 49 for the other. Nevertheless, we did see a much higher variance in the number of steps for the aggressive algorithm. So even though there were some runs in the non-greedy algorithm that took over a thousand steps, about 50% of the runs took less than 10 steps, and 60% took less than 20. This is a much more consistent result than for the greedy version, which had between 20 and 25% of its runs at every 10 step range.
Chapter 7

Conclusion and next steps

This thesis had as a goal to test and analyse the greedy algorithm implemented by [londono'adragna'2018] and proposed by [betzronix2016] as well as its alternative less aggressive version develop through the course of this project. The algorithm proposes an alternative to the current PageRank algorithm. An algorithm used by Google to rank its websites according to its importance. The algorithm being tested here, measures the relative importance of two websites according to how reachable the websites are. The algorithm measures such “reachability” by creating a random walker which walks through nodes using the links available and their respective weights to move from node to node until it reaches its goal. As opposed to the PageRank algorithm, our algorithm, only has the option to jump to a random node, whenever the node it is at has no neighbours whatsoever making it theoretically more efficient.

This jump, when it happens, happens only when the walker has visited (and contracted) an entire connected component of the graph without finding its goal, and it stands at the only node left. The walker can get stuck in such a node because (some) nodes visited are contracted, so it could be that all nodes in one component are deleted, and the walker is at the last one left. This contraction is done either at every step on the greedy version, or only when the node has less than two neighbours on the less greedy version. As for what it actually means to contract the nodes, the process is the following. Whenever a node gets contracted we want to “keep” its edges but get rid of the node itself. The way we do this, is by making sure the weight of its existing edges gets transferred to its neighbours. Suppose we want to delete node \( k \) which has out going edges to nodes \( r, s \) with the following weights \((k,r) = 0.3\) and \((k,s) = 0.7\). Say there are edges \((r,s)\) and \((s,r)\) with weights \(w_1, w_2\) respectively\(^1\). If we stand at node \( r \) (before node \( k \)'s contraction) we can go to node \( s \) either directly, or by going to node \( K \) and then to node \( s \). What we do is update the weight of edge \((r,s)\) so that it takes into account this probability of going through node \( k \) which won’t be there after the deletion of the node. That is we make the new weight, call it \( w'_1 \) as follows: \( w'_1 = w_1 + (r,k) * (k,s) = (r,s) + (r,k) * (k,s), \) that way we preserve the weight ranking after contracting node \( k \). Preserving these probabilities is important, as this is what will help us rank a node’s “reachability”.

To test how these contraction techniques perform, this thesis takes seven different datasets of weighted directed graphs, and defines a mathematical framework under which

\(^1\)these weights could be zero
such graphs can be viewed as "weighted-directed" complexes. After, we have define such complexes, we view our data-sets as such and measure their homologies. That is, using Flagser [flagser], we count the number of directed simplices of every dimension, for every data-set as well as the dimension of every homology. Afterwards, we run the algorithm on every data-set, while taking snapshots of the graphs at several stages during the run. We then proceed to run Flagser on these snapshots and compare how the properties measure before change as the algorithm runs. We find that the graphs grow very quickly, in terms of number of edges and simplices. This fast growth happens within a few steps and it makes it impossible to compute their homologies due to the amount of memory required to perform this action.

After having an idea of how the graphs evolve through the run, we run both versions of the algorithm 30 times per graph to get an idea of how the resulting graphs look like in every data-set and for both versions of the algorithm. Based on such experiments, we determine that the aggressive version of the algorithm changes the topological structure of all data-sets, and that the resulting graphs all share similar characteristics. Resulting graphs of the aggressive algorithm have a much higher number of edges and of connected components. This means that the higher number of edges is distributed among less nodes, which leads to an increase in the count of fully connected k-graphs for different values of k (usually k between 6 and 10). Topologically, this means a decrease in the homologies of the lower dimensions, (typically we see a decrease in the first and second dimensions) and an increase in the higher dimensions (usually increases from five on), with mixed results in dimensions 3 and 4 for most data-sets. We also see a decrease in the average weight per edge, which is a result of the increase in number of edges (recall edges are normalized to add up to 1).

The results above refer to the aggressive algorithm. As for the less aggressive version, we contract only nodes that have less than 3 neighbours. This is because nodes that are not highly connected are faster to contract. With this restriction, we hoped to decrease total run-time of the algorithm.

When comparing run-time of both versions of the algorithm, we do see a lower average run-time for the less aggressive algorithm. As mentioned in the previous section, only 19% of the less aggressive runs exceeded the 10000 micro-seconds mark, compared to over 90% of the runs of the aggressive runs. However, it is important to mention that we also found that most of the runs in the less aggressive version of the algorithm didn't present any contractions whatsoever, and run-time varied greatly even without contractions, making it difficult to say whether less contractions do make the algorithm faster indeed. The goal of the less aggressive test was to see contractions of about 30% of the nodes, but in reality most runs had no contractions, with only 2 runs out of 210 having a contraction as high as 5% of its nodes (3 nodes contracted on two FB runs). Because of this, further tests are required, in order to determine whether there is any run-time optimization when contracting, and on whether contracting poorly connected nodes is more efficient than contracting nodes at random. However, partial results suggest that little to no-contractions will have a shorter run-time in average.

As for the topological structure of the graphs, resulting graphs of the less aggressive version of the algorithm, don't contain such a high number of high-dimensional holes as the resulting graphs of its greedier version. For both versions of the algorithm, plotting all the signatures of one version results in grouping. It is easy to distinguish between a signature produced by a greedy run versus a non-greedy. These differences are also easy
to spot when compared to the original graph and to the graph with no contractions.

The next steps for those working in developing this algorithm further, are to try different criteria of contraction as an alternative to the first version of the less aggressive algorithm (which was used in this thesis). From the tests ran here, it seems like there is no clear additional benefit from contracting at every step, and the right choice of nodes to contract could make the algorithm much more efficient without changing the topological properties of the graph too dramatically. Whether contracting at all is more efficient must be also tested, and running tests against the original PageRank algorithm (perhaps changing the random jump probability) are advised. Further, a goal of the algorithm should be to use (un)supervised learning so that the algorithm could adapt the contraction strategies according to the starting topology of the graph. We have seen that the amount of contractions, run-time and number of steps depend largely on the starting data-set. Therefore, it would be most efficient to have an algorithm, which based on previous experience could determine what type of strategy would help the walker maximize its chances of a quick success.
Appendix A

About the original graphs

A.1 Degree histogram

Figure A.1.1: Advogato histogram
Figure A.1.2: Moreno histogram

Figure A.1.3: Alpha histogram
Figure A.1.4: Facebook histogram

Figure A.1.5: Opsahl histogram
Figure A.1.6: OTC histogram

Figure A.1.7: Wiki histogram
Appendix B

Flagser commands

Here we will give a detailed explanation of how flagser was used and what every command means. Use this as a reference when reading through the results. Flagser is a software created by [Flagser]. It computes directed simplicial complexes of graphs, and their homologies. It can also compute filtrations when given weighted graphs. The basic command is

```
./flagser --out outputfilename inputfilename
```

Listing 1: Flagser command 1 (standard)

It counts the simplicial complexes and computes the number of holes per Homology, as well as the Euler characteristic of the input file given, and creates a new file with the name given as output file where it lists the results.

However, the computation of the Betti Numbers requires a lot of memory, and therefore for some data-sets, Flagser kills the computations when the memory usage is too large. In order to avoid this, we tried using the option `--memory` which allows Flagser to use more memory in its computations.

```
./flagser-memory --out outputfilename inputfilename
```

Listing 2: Flagser command 2 (memory)

The next option, restricts the minimum or maximum homology to be computed. This can be useful if we are focusing on simplicial complexes of only a certain dimension, or above (below) a certain dimension. Below we show the example for which we restrict the program to computing the homology dimension only above 7, but `--min-dim` can be replaced by `--max-dim` to restrict the dimension to only those below the number indicated afterwards.
Listing 3: Flagser command 3 (minimum dimension)

The following command counts only the number of simplices per dimension and Euler characteristics without computing the dimension of the homology. It is useful for very large graphs, for which it is very expensive to determine when a simplex is a boundary of another of larger dimension. It has the quickest running time, but it provides the least amount of information.

Listing 4: Flagser command 4 (just count simplices)

The last option we will talk about is the --approximate option. It is a way of obtaining an overall view of all of a graph's topological properties without consuming too much memory. Instead of restricting the dimension by a maximum or a minimum, or stopping to compute the dimension of its homologies, it does everything but it ignores simplices that lead to very large computations. [Flagser] says that the command ignores simplices that give more than n entries in the computation matrix, where n is the number followed by --approximate. Usually, the impact in precision is very low. See the next chapter in the appendix for an example.

Listing 5: Flagser command 5 (approximate results)
Appendix C

Cutting and editing the Datasets

As mentioned on the Dataset chapter, we were forced to cut the datasets late in the project. In order to show the process we lived with the dataset, and to give a bigger overall view of how we treated the datasets, I decided to leave a section of a previous version of the chapter below. First, we see the original topological properties of the complete datasets.

<table>
<thead>
<tr>
<th>Graph</th>
<th>Source</th>
<th>Size</th>
<th>Volume</th>
<th>Avg. Deg.</th>
<th>fill</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advogato</td>
<td>Konect</td>
<td>6541</td>
<td>51127</td>
<td>15.6328</td>
<td>0.00119498</td>
</tr>
<tr>
<td>Adolescent Health (moreno)</td>
<td>Konect</td>
<td>2539</td>
<td>12969</td>
<td>10.2158</td>
<td>0.002012575</td>
</tr>
<tr>
<td>U.S. Airports</td>
<td>Konect</td>
<td>1574</td>
<td>28236</td>
<td>35.8780</td>
<td>0.0114043286</td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>Konect</td>
<td>7118</td>
<td>103675</td>
<td>29.1304</td>
<td>0.002046247</td>
</tr>
<tr>
<td>Bitcoin OTC</td>
<td>SNAP</td>
<td>5881</td>
<td>35592</td>
<td>12.1041</td>
<td>0.001029257</td>
</tr>
<tr>
<td>Bitcoin Alpha</td>
<td>SNAP</td>
<td>3783</td>
<td>24186</td>
<td>12.7867</td>
<td>0.001690465</td>
</tr>
<tr>
<td>Open Flights</td>
<td>Opsahl</td>
<td>7976</td>
<td>30501</td>
<td>7.6482</td>
<td>0.0004795106</td>
</tr>
<tr>
<td>Facebook-like S.N.</td>
<td>Opsahl</td>
<td>1899</td>
<td>20296</td>
<td>21.3755</td>
<td>0.005631049</td>
</tr>
<tr>
<td>C elegans</td>
<td>Opsahl</td>
<td>306</td>
<td>2345</td>
<td>15.3268</td>
<td>0.025125898</td>
</tr>
</tbody>
</table>

Table C.0.1: Graph Characteristics of data 1

<table>
<thead>
<tr>
<th>Graph</th>
<th>Max. Deg.</th>
<th>LCC</th>
<th>NCC</th>
<th>LSCC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advogato</td>
<td>943</td>
<td>5042</td>
<td>0</td>
<td>3140</td>
</tr>
<tr>
<td>Adol. Health</td>
<td>36</td>
<td>2539</td>
<td>1</td>
<td>2155</td>
</tr>
<tr>
<td>U.S. Airports</td>
<td>596</td>
<td>1572</td>
<td>0</td>
<td>1402</td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>1167</td>
<td>7066</td>
<td>0</td>
<td>1300</td>
</tr>
<tr>
<td>Bitcoin OTC</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Bitcoin Alpha</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Open Flights</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Facebook msg.</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table C.0.2: Graph Characteristics of data 2

77
### Table C.0.3: Number of n-simplices of Data (Part 1)

<table>
<thead>
<tr>
<th>Graph</th>
<th>0-dim sim.</th>
<th>1-dim sim.</th>
<th>2-dim sim.</th>
<th>3-dim sim.</th>
<th>4-dim sim.</th>
<th>5-dim sim.</th>
<th>6-dim sim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advocato</td>
<td>6542</td>
<td>47135</td>
<td>188022</td>
<td>696594</td>
<td>2944859</td>
<td>14088057</td>
<td>69259929</td>
</tr>
<tr>
<td>Adol. Health</td>
<td>2540</td>
<td>12969</td>
<td>10502</td>
<td>6886</td>
<td>4915</td>
<td>3087</td>
<td>972</td>
</tr>
<tr>
<td>U.S. Airports</td>
<td>1575</td>
<td>28236</td>
<td>1013262</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>7119</td>
<td>103617</td>
<td>745129</td>
<td>3650763</td>
<td>13452427</td>
<td>40061359</td>
<td>99386593</td>
</tr>
<tr>
<td>BC OTC</td>
<td>6006</td>
<td>35592</td>
<td>125886</td>
<td>506933</td>
<td>1780830</td>
<td>4952733</td>
<td>10260674</td>
</tr>
<tr>
<td>BC Alpha</td>
<td>7605</td>
<td>24186</td>
<td>88753</td>
<td>358619</td>
<td>1230365</td>
<td>3278886</td>
<td>6447341</td>
</tr>
<tr>
<td>Open Flights</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fb msg.</td>
<td>1900</td>
<td>20296</td>
<td>39982</td>
<td>33159</td>
<td>13640</td>
<td>2840</td>
<td>656</td>
</tr>
</tbody>
</table>

### Table C.0.4: Number of n-simplices of Data (Part 2)

<table>
<thead>
<tr>
<th>Graph</th>
<th>7-dim sim.</th>
<th>8-dim sim.</th>
<th>9-dim sim.</th>
<th>10-dim sim.</th>
<th>11-dim sim.</th>
<th>12-dim sim.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advocato</td>
<td>318675823</td>
<td>1284428343</td>
<td>4331113643</td>
<td>11790038786</td>
<td>25125968411</td>
<td>40727877713</td>
</tr>
<tr>
<td>Adol. Health</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U.S. Airports</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Wiki Vote</td>
<td>205513052</td>
<td>347650344</td>
<td>468192797</td>
<td>484730551</td>
<td>365960990</td>
<td>185989553</td>
</tr>
<tr>
<td>BC OTC</td>
<td>14153693</td>
<td>10555723</td>
<td>2320246</td>
<td>8988</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>BC Alpha</td>
<td>8419527</td>
<td>5801340</td>
<td>988041</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Open Flights</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Fb msg.</td>
<td>287</td>
<td>472</td>
<td>1020</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

However, you might notice some of the graphs here are incomplete. The text below explains how we obtained these results, and they explain why not everything is filled in. The text is a section cut out of a previous version of the paper.

### C.0.1 Computing Topological Properties

In order to find the topological properties of the data, we ran Flagser on every dataset. First, we computed the results shown on the tables called “Number of n-simplices” at the end of the chapter. For most data-sets (those small enough), we computed these results using command 1 in the previous chapter.

However, the computation of the Betti Numbers requires a lot of memory, and therefore for some data-sets, *(Advocato, US Flights, and Open Flights)*, Flagser would kill the computations when the memory usage was too large. In order to avoid this, we tried using the memory option (command 2) which allows Flagser to use more memory in its computations.

Unfortunately, this modification alone didn’t yield any additional results. The programs were still stopped either by the program (Flagser) itself or sometimes by the cluster when the node would run out of memory. Whenever possible, the experiments were ran again expanding the memory usage of the cluster, but this did not help because each node would
still run out of memory. ¹ After realizing that carrying out all the computations at once would be too memory expensive (200+ GB) per data-set we decided to divide the problem. Therefore, we put an upper limit on the amount of homologies we ran, in an effort to get partial results, which could be later put together.

Take *Advogato* for example. With this data-set, even on the super computer, the algorithm was always stopped after the sixth homology (*H₀*) when ran with command 1. Therefore, we thought of limiting the computation to start at the seventh homology, and putting these partial results together with those previously obtained. So we ran the minimum dimension command 3.

In most cases, limiting the dimension for a maximum wouldn’t work. The command would also run out of memory unless the minimum dimension was small enough. But when put to the test with a higher minimum it would use all the memory just as if it had started from zero.

Because neither of these modifications where giving good results, the next step was downsizing further. The next option we tried was two fold.

¹The main problem here is that the computation of such simplices is not parallelizable, and therefore, we could only use one node per computation.
<table>
<thead>
<tr>
<th>Graph</th>
<th>$B_6$</th>
<th>$B_7$</th>
<th>$B_8$</th>
<th>$B_9$</th>
<th>$B_{10}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Advogato</td>
<td>26536</td>
<td>-</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Adol. Health</td>
<td>60</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>U.S. Airports</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Wiki Vote</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BC OTC</td>
<td>356315</td>
<td>912713</td>
<td>1972402</td>
<td>552640</td>
<td>0</td>
</tr>
<tr>
<td>BC Alpha</td>
<td>195832</td>
<td>443846</td>
<td>977004</td>
<td>156539</td>
<td>0</td>
</tr>
<tr>
<td>Open Flights</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fb msg</td>
<td>74</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table C.0.7: Betti Numbers of data (Part 2)

Counting and approximating

The first part of the downsizing process for large graphs was using Flagser’s option count. When this option is activated, Flagser computes only the number of cells per homology. That is, it tells you only how many simplices there are without seeing how they connect. It can, therefore, not tell you how many dimensions every homology has. We used command 4 in order to finish collecting information about the n-simplices of data.

This command gave us the biggest improvement in terms of results obtained. While the first three commands took between 24 to 48 hours to run, the just-count command could help us fill in the Number of simplices tables in less than ten minutes for Advogato and Wiki vote.

However, we still had no information about Betti numbers of these datasets. This is where the second part came to play. The last option we considered for this part of the information collection was approximate n. In the documentation, [flagser] describes it as a computation process that skips all cells which create a reduction matrix with more than n entries. Due to this fact, the documentation suggests validating the results later on with longer computation time. However, we knew this was not possible given our computation limits. Furthermore, we are interested in how the graph changes, more so than in the graphs themselves, and therefore an approximation for larger graphs is enough. To compute this we used command 5.

At first, the dataset **Advogato** was run with $n = 100,000$ but after the program timed out again, we reduced it significantly to $n = 10,000$ in order to see if we could get some preliminary results and maybe increase $n$ gradually for accuracy.

To give an idea how $n$ affects the results, we ran the approximate command on the **Adolescent Health** dataset to get an idea of how it changed the results but the dataset was too small and so no simplices were skipped. Therefore, we decided to run it on **Bitcoin Alpha**, where the command did skip some simplices. The results where as follows:

As you can see here, even with such a small upper bound for the columns skipped, the difference was quite small, even when many columns were indeed skipped, such as in the sixth or seventh homologies. Given the difference was so small in a smaller dataset, we decided to run the approximation with the same $n = 10,000$ for **Advogato** and **Wikipedia**. The results were then used to complete the information about Betti numbers. Numbers in blue have been approximated and detailed information about the approximation is
<table>
<thead>
<tr>
<th>Betti Number</th>
<th>Full Command</th>
<th>n = 10k</th>
<th>difference No.</th>
<th>difference %</th>
<th>Skipped cols.</th>
</tr>
</thead>
<tbody>
<tr>
<td>$B_0$</td>
<td>3827</td>
<td>3872</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$B_1$</td>
<td>5310</td>
<td>5301</td>
<td>-9</td>
<td>-0.1%</td>
<td>16</td>
</tr>
<tr>
<td>$B_2$</td>
<td>6657</td>
<td>6654</td>
<td>-3</td>
<td>-0.04%</td>
<td>35</td>
</tr>
<tr>
<td>$B_3$</td>
<td>11585</td>
<td>11571</td>
<td>-14</td>
<td>0.1%</td>
<td>80</td>
</tr>
<tr>
<td>$B_4$</td>
<td>31038</td>
<td>31003</td>
<td>-35</td>
<td>0.1%</td>
<td>210</td>
</tr>
<tr>
<td>$B_5$</td>
<td>90933</td>
<td>90697</td>
<td>-236</td>
<td>0.25%</td>
<td>782</td>
</tr>
<tr>
<td>$B_6$</td>
<td>195832</td>
<td>195648</td>
<td>-184</td>
<td>0.09%</td>
<td>1394</td>
</tr>
<tr>
<td>$B_7$</td>
<td>443846</td>
<td>444059</td>
<td>+213</td>
<td>0.04%</td>
<td>1405</td>
</tr>
<tr>
<td>$B_8$</td>
<td>977004</td>
<td>977823</td>
<td>+819</td>
<td>0.08%</td>
<td>1</td>
</tr>
<tr>
<td>$B_9$</td>
<td>156539</td>
<td>156539</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table C.0.8: 10k approximation comparison Bitcoin Alpha

Presented in the appendix.

Remark: Due to the fact that both datasets, Advogato and Wikipedia elections are considerably bigger than Bitcoin Alpha the difference in accuracy could be bigger, but because of the low percentages of inaccuracy, even an increase in accuracy should remain relatively low in general terms. Of course, this is only our intuition and it was not proven.

C.0.2 Why we didn’t use these results

The beginning of this section marks the end of the snippet taken out of the previous version of this paper. While, the results were relatively promising, and there was no apparent reason to shrink the datasets any further, this changed when we ran the algorithm on the datasets. The first dataset we ran the algorithm on was the Facebook messenger dataset. This was the smallest, and less densely connected dataset that I was working on, and I could run every flagger calculation on my personal laptop. However, after one run of the algorithm, the dataset became so densely connected, that not even the super computer could run most flagger commands, leaving us unable to compare the before and after properties, and defeating the entire purpose of the project. We had a fear this would happen, and it is explained in detail in the result section, and so in order to obtain some kind of results in this experiment, we decided to simply cut the graphs. Because the algorithm was increasing density quite quickly, we decided to make the graphs of size 50, obtain some results, and if possible (time-wise and computational-wise) increase the sizes and so on.
<table>
<thead>
<tr>
<th>Betti Number</th>
<th>Full Command</th>
<th>( n = 10k )</th>
<th>skipped</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_0 )</td>
<td>1444</td>
<td>1444</td>
<td>0</td>
</tr>
<tr>
<td>( B_1 )</td>
<td>4522</td>
<td>4497</td>
<td>33</td>
</tr>
<tr>
<td>( B_2 )</td>
<td>6591</td>
<td>6573</td>
<td>39</td>
</tr>
<tr>
<td>( B_3 )</td>
<td>8140</td>
<td>8120</td>
<td>42</td>
</tr>
<tr>
<td>( B_4 )</td>
<td>10218</td>
<td>10209</td>
<td>47</td>
</tr>
<tr>
<td>( B_5 )</td>
<td>12157</td>
<td>12121</td>
<td>205</td>
</tr>
<tr>
<td>( B_6 )</td>
<td>26536</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_7 )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_8 )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_9 )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{10} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{11} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{12} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{13} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{14} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{15} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{16} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>( B_{17} )</td>
<td>-</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

Table C.0.9: 10k approximation comparison Advogato

<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>Snapshot 1:</th>
<th>Snapshot 2:</th>
<th>Snapshot 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-sim</td>
<td>2540</td>
<td>2540</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1-sim</td>
<td>12969</td>
<td>20923</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-sim</td>
<td>10502</td>
<td>28274</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-sim</td>
<td>6886</td>
<td>31284</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4-sim</td>
<td>4915</td>
<td>39120</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5-sim</td>
<td>3087</td>
<td>48360</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6-sim</td>
<td>972</td>
<td>40320</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7-sim</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table C.0.10: Moreno comparison: Cell count

<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>Snapshot 1:</th>
<th>Snapshot 2:</th>
<th>Snapshot 3:</th>
</tr>
</thead>
<tbody>
<tr>
<td>( B_0 )</td>
<td>2</td>
<td>2</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_1 )</td>
<td>4848</td>
<td>8332</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_2 )</td>
<td>1168</td>
<td>6589</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_3 )</td>
<td>436</td>
<td>5211</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_4 )</td>
<td>198</td>
<td>5268</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_5 )</td>
<td>157</td>
<td>5360</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_6 )</td>
<td>60</td>
<td>16731</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( B_7 )</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table C.0.11: Moreno comparison: Betti Numbers

82
Table C.0.12: Comparison, graph theoretical properties
Appendix D

Tables and graphs from the results

D.1 Greedy algorithm

D.1.1 Cell count

<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>snap 3</th>
<th>snap 4</th>
<th>snap 5</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>173</td>
<td>403</td>
<td>368</td>
<td>260</td>
<td>160</td>
<td>72</td>
<td>6</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>402</td>
<td>402</td>
<td>3750</td>
<td>2550</td>
<td>1350</td>
<td>504</td>
<td>6</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>1167</td>
<td>39559</td>
<td>34149</td>
<td>24257</td>
<td>11880</td>
<td>3024</td>
<td>0</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>3088</td>
<td>34835</td>
<td>266644</td>
<td>20408</td>
<td>95040</td>
<td>15120</td>
<td>0</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>5903</td>
<td>2708159</td>
<td>1789582</td>
<td>1485554</td>
<td>665280</td>
<td>60480</td>
<td>0</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>7266</td>
<td>18364266</td>
<td>10367887</td>
<td>9258204</td>
<td>3991680</td>
<td>181440</td>
<td>0</td>
</tr>
<tr>
<td>7-dim sim.</td>
<td>5220</td>
<td>106838848</td>
<td>51890467</td>
<td>48712243</td>
<td>19958400</td>
<td>362880</td>
<td>0</td>
</tr>
<tr>
<td>8-dim sim.</td>
<td>1560</td>
<td>522088128</td>
<td>223947085</td>
<td>212538309</td>
<td>79833600</td>
<td>362880</td>
<td>0</td>
</tr>
<tr>
<td>9-dim sim.</td>
<td>0</td>
<td>2087827632</td>
<td>830052569</td>
<td>751705970</td>
<td>0</td>
<td>239500800</td>
<td>0</td>
</tr>
<tr>
<td>10-dim sim.</td>
<td>0</td>
<td>6609321504</td>
<td>2622810709</td>
<td>2094011650</td>
<td>0</td>
<td>479001600</td>
<td>0</td>
</tr>
<tr>
<td>11-dim sim.</td>
<td>0</td>
<td>15830812800</td>
<td>6971164044</td>
<td>4428441840</td>
<td>0</td>
<td>479001600</td>
<td>0</td>
</tr>
<tr>
<td>12-dim sim.</td>
<td>0</td>
<td>26790094656</td>
<td>15226215054</td>
<td>6771369360</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13-dim sim.</td>
<td>0</td>
<td>28327283712</td>
<td>26292717000</td>
<td>6972435360</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>14-dim sim.</td>
<td>0</td>
<td>13940035200</td>
<td>33650882568</td>
<td>4261831200</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>15-dim sim.</td>
<td>0</td>
<td>0</td>
<td>28331912592</td>
<td>1163289600</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>16-dim sim.</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>11742248448</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table D.1.1: Advogato cell count

85
<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>250</td>
<td>445</td>
<td>297</td>
<td>154</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>712</td>
<td>4093</td>
<td>2030</td>
<td>474</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>1315</td>
<td>32922</td>
<td>12081</td>
<td>1053</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>1527</td>
<td>229478</td>
<td>60375</td>
<td>1419</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>1059</td>
<td>1370952</td>
<td>244734</td>
<td>810</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>400</td>
<td>6887620</td>
<td>764190</td>
<td>90</td>
</tr>
<tr>
<td>7-dim sim.</td>
<td>62</td>
<td>28315128</td>
<td>1690920</td>
<td>0</td>
</tr>
<tr>
<td>8-dim sim.</td>
<td>0</td>
<td>91710432</td>
<td>2270520</td>
<td>0</td>
</tr>
<tr>
<td>9-dim sim.</td>
<td>0</td>
<td>221860800</td>
<td>1285200</td>
<td>0</td>
</tr>
<tr>
<td>10-dim sim.</td>
<td>0</td>
<td>370557504</td>
<td>136080</td>
<td>0</td>
</tr>
<tr>
<td>11-dim sim.</td>
<td>0</td>
<td>377384832</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>12-dim sim.</td>
<td>0</td>
<td>185863680</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>13-dim sim.</td>
<td>0</td>
<td>207360000</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table D.1.2: Wiki cell count

### D.1.2 Betti Numbers

#### Tables

#### Graphs

### D.1.3 Greedy algorithm

*Remark.* The Alpha runs contained the longest filtrations, and therefore the heaviest files. Therefore, we couldn’t get a plot of the runs together with the original graph and the graph with no contractions. In the figures D.1.10 and D.1.11 we see only the resulting graphs of some of the runs after the greedy algorithm.

---

1At the time the results for "snap 1" came out my storage space on the cluster had been limited to a minimum. There is reason to believe there was not enough space to save the entire file and therefore there could be dimensions missing. However, the calculation was not ran again.
<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>161</td>
<td>360</td>
<td>-</td>
<td>48</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>251</td>
<td>4118</td>
<td>-</td>
<td>216</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>227</td>
<td>43119</td>
<td>-</td>
<td>840</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>135</td>
<td>407327</td>
<td>-</td>
<td>2520</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>0</td>
<td>3438180</td>
<td>-</td>
<td>5040</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>0</td>
<td>25609218</td>
<td>-</td>
<td>5040</td>
</tr>
<tr>
<td>7-dim sim.</td>
<td>0</td>
<td>165702276</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>8-dim sim.</td>
<td>0</td>
<td>912753360</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>9-dim sim.</td>
<td>0</td>
<td>4166951040</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>10-dim sim.</td>
<td>0</td>
<td>15190912800</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>11-dim sim.</td>
<td>0</td>
<td>41881613760</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>12-dim sim.</td>
<td>0</td>
<td>80145313920</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>13-dim sim.</td>
<td>0</td>
<td>91757836800</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>14-dim sim.</td>
<td>0</td>
<td>48498912000</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>15-dim sim.</td>
<td>0</td>
<td>15567552000</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>16-dim sim.</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>

Table D.1.3: Opsahl cell count

<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>834</td>
<td>931</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>7749</td>
<td>12714</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>46776</td>
<td>130559</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>195787</td>
<td>1080038</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>578192</td>
<td>7641708</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>1179713</td>
<td>47927926</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7-dim sim.</td>
<td>1551728</td>
<td>267031482</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8-dim sim.</td>
<td>1050288</td>
<td>1291165136</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9-dim sim.</td>
<td>67200</td>
<td>5250994700</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10-dim sim.</td>
<td>0</td>
<td>17374811366</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11-dim sim.</td>
<td>0</td>
<td>45229416902</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12-dim sim.</td>
<td>0</td>
<td>89039073344</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13-dim sim.</td>
<td>0</td>
<td>124869571382</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14-dim sim.</td>
<td>0</td>
<td>111620984466</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15-dim sim.</td>
<td>0</td>
<td>47780234460</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16-dim sim.</td>
<td>0</td>
<td>0</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table D.1.4: Alpha cell count
<table>
<thead>
<tr>
<th>n-simplex</th>
<th>Original Graph</th>
<th>snap 1</th>
<th>snap 2</th>
<th>End of run</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-dim sim.</td>
<td>51</td>
<td>51</td>
<td>51</td>
<td>51</td>
</tr>
<tr>
<td>1-dim sim.</td>
<td>289</td>
<td>582</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2-dim sim.</td>
<td>926</td>
<td>6690</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3-dim sim.</td>
<td>1977</td>
<td>67984</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4-dim sim.</td>
<td>2214</td>
<td>605154</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5-dim sim.</td>
<td>900</td>
<td>4694318</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>6-dim sim.</td>
<td>0</td>
<td>31516848</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>7-dim sim.</td>
<td>0</td>
<td>181442751</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>8-dim sim.</td>
<td>0</td>
<td>885278127</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9-dim sim.</td>
<td>0</td>
<td>3609314398</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>10-dim sim.</td>
<td>0</td>
<td>12094003321</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>11-dim sim.</td>
<td>0</td>
<td>32677676924</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>12-dim sim.</td>
<td>0</td>
<td>69691904497</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>13-dim sim.</td>
<td>0</td>
<td>114587926579</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>14-dim sim.</td>
<td>0</td>
<td>141607416522</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>15-dim sim.</td>
<td>0</td>
<td>127889593232</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>16-dim sim.</td>
<td>0</td>
<td>81187129192</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>17-dim sim.</td>
<td>0</td>
<td>33896229420</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>18-dim sim.</td>
<td>0</td>
<td>8042603040</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>19-dim sim.</td>
<td>0</td>
<td>697737024</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table D.1.5: OTC cell count

![Figure D.1.1: Moreno cell count](image)

Figure D.1.1: Moreno cell count
Figure D.I.2: Facebook cell count (scale 1:10,000,000)

Figure D.I.3: Advogato cell count (scale 1:1,000,000,000)
Figure D.1.4: Wikipedia cell count (scale 1:100,000,000)

Figure D.1.5: Opsahl cell count
Figure D.1.6: Betti Numbers Facebook

Figure D.1.7: Betti Numbers Moreno
Figure D.1.8: Advogato signatures 1.

Figure D.1.9: Advogato signatures 2 ($H9$).
Figure D.1.10: Alpha signatures 1.

Figure D.1.11: Alpha signatures 2
Figure D.1.12: Opsahl signatures 1.
Figure D.1.13: Opsahl signatures 2.
Figure D.1.14: OTC signatures 1.

Figure D.1.15: OTC signatures 2.
Figure D.1.16: Wiki signatures 1.
Figure D.1.17: Wiki signatures 2.