Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Vlad Sebastian Velici
April 28, 2015

Similar nodes in large graphs

Project Supervisor: Dr. Adam Prügel-Bennett
Second Examiner: Dr. Sasan Mahmoodi

A project report submitted for the award of
MEng Computer Science with Artificial Intelligence

Contents

Abstract

Given a large graph, in the range of millions or billions of nodes, it is difficult to efficiently find similar nodes. Examples of applications of finding similar nodes in data represented as graphs are all around the web nowadays: *people you may know* and *who to follow* suggestions on social networks, or topics that may interest you on different websites.

This project will explore ways to obtain meaningful information of this nature from large datasets represented as graphs.

The goals of this project are developing efficient ways to find similar nodes in large graphs for different use cases and implementing an open-source software library to provide everyone with this functionality.

1 Introduction

Plenty of datasets are or can be represented as graphs where vertices represent entities and edges represent relationships between entities. A problem of interest is to find entities that are similarly connected. Example instances of this problem are finding *people you may know* in a social network, people with common interests from research publications repositories or identifying possible duplicates in a dataset.

It is possible to calculate cloud vectors that represent the neighbourhoods of vertices in a graph only by looking at its structure. Then if a cloud vector is close to another it means the vertices they represent belong to a similar neighbourhood. Using the dot product of cloud vectors we can compute different similarity measures between vertices. Cosine similarity and Euclidean distance are studied in this report.

It is computationally too expensive to compute all the cloud vectors and then the dot products or the Euclidean distances. This project investigates methods for approximating the dot products, different similarity measures based on the dot products, and the evaluation of these measures on real world datasets.

2 Related work

The work in [4] presents a way to characterise similarities between vertices of weighted undirected graphs based on a Markov-chain model of random walks. It uses the leading eigenvectors to approximate the Laplacian matrix of the graph and its pseudo-inverse to provide a similarity measure. The model is used as a recommender system in the work.

A similar diffusion process is used in [13] for an algorithm to find communities (parts of the graph with vertices connecting to each other more than they connect to the rest of the graph) in graphs, which has various applications in graph visualisation, recommender systems, analysis of networks, especially social networks, etc.

A model that works for both directed and undirected graphs is presented in this report. It uses the normalised adjacency matrix to model a diffusion process and obtain similarity measures between vertices.

3 Cloud vectors and similarity measures

The cloud vector of a vertex represents the way this vertex is connected to the rest of the graph. In other words, it represents an extended neighbourhood of the node. It is created by a process similar to diffusion, as shown in *Figure 1*.

Given a graph $\mathcal{G}(\mathcal{V},\mathcal{E})$ with adjacency matrix $\mathbf{M}$. The number of nodes is $N = |v|$ and the number of edges is $|\epsilon|$.

The cloud vectors represent the neighbourhoods of vertices of the graph $\mathcal{G}$, thus looking at the number of walks of different lengths from a vertex i to all the other vertices in the graph is a promising start.

The adjacency matrix has the property that if it is raised to the k^{th} power, the (i, j) element of it, $\mathbf{M}^k_{ij}$, is the number of walks of length k from vertex i to vertex j. Therefore the i^{th} column of $\mathbf{M}^k$ is a vector that represents the way vertex i is connected to the rest of the graph by walks of length k.

The could vectors should include the walks of all lengths with a preference towards the shorter walks over the longer walks. Therefore the adjacency matrix should be normalised such that all rows sum up to one. Let

$$\mathbf{D}_{ij} = \frac{\mathbf{M}_{ij}}{\sum_{k=1}^{N} \mathbf{M}_{ik}} \tag{1}$$

be the normalised adjacency matrix. A matrix that includes walks of all lengths as described above can be written as the sum

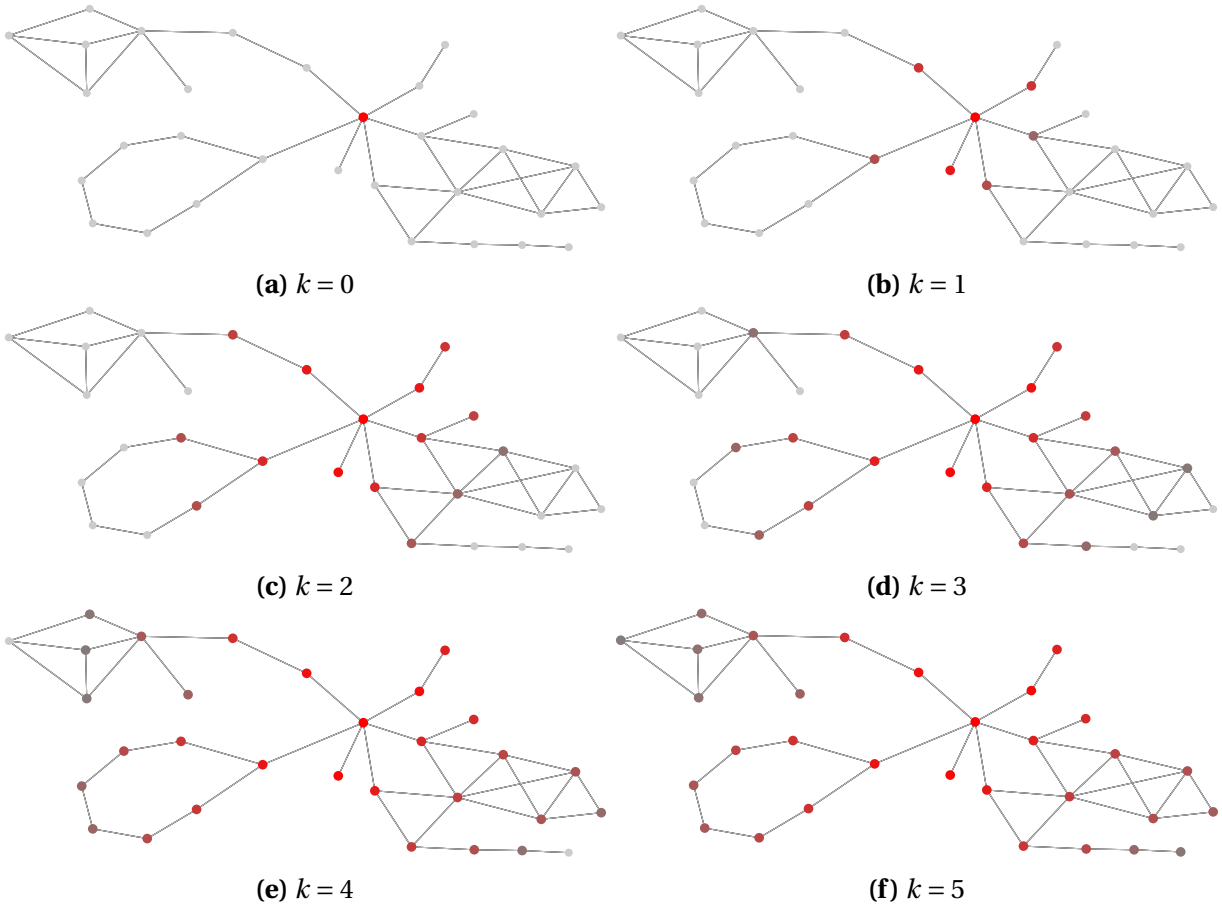
$$\sum_{t}^{\infty} \mu^t \mathbf{D}^t, \tag{2}$$

where $\mu \in (0,1)$ is a penalising factor. For each vertex of the graph, $i \in \mathcal{V}$, the cloud vector is the i^{th} column of the matrix in *Equation 2*, or mathematically

$$\boldsymbol{c}_i = \sum_{t=0}^{\infty} \mu^t \mathbf{D}^t \boldsymbol{\delta}_i \tag{3}$$

where $\boldsymbol{\delta}_i$ is the i^{th} column of the $N \times N$ identity matrix.

Figure 1 shows the diffusion process of computing $\boldsymbol{c}_i$ up to $k = 0,1,2,3,4,5$. The coloured vertices are used in the computation of $\boldsymbol{c}_i$ up to that k. The saturation of the colour of a vertex



(a) $k = 0$ **(b)** $k = 1$ **(c)** $k = 2$ **(d)** $k = 3$ **(e)** $k = 4$ **(f)** $k = 5$

Figure 1: Computation of $\boldsymbol{c}_i = \sum_{t=0}^{k} 0.5^t \mathbf{D}^t \boldsymbol{\delta}_i$ up to different k. Vertex i is the only red vertex in **(a)**. The saturation of the colour of a vertex, j, represents the connection strength between vertices i and j (red means strong connection and light grey no connection).

j represents the j^{th} value of a normalised $\boldsymbol{c}_i$, such that red represents the highest value in $\boldsymbol{c}_i$ and light grey represents values of 0. In other words, the saturation of the colour of a vertex j represents the strength of the connection between vertex i and vertex j. When $k = 0$, only the first vertex, i, counts. As k gets larger, more and more vertices are considered into the cloud vector $\boldsymbol{c}_i$, making it more accurate.

3.1 Similarity between two vertices

Two vertices of a graph are similar if they are in a similar neighbourhood. The cloud vector of a vertex represents the neighbourhood of that vertex, therefore the similarity between two vertices can be obtained by comparing their cloud vectors.

One similarity measure is the Euclidean distance between cloud vectors. For vertices i and j it is $\|\boldsymbol{c}_i - \boldsymbol{c}_j\|$. The smaller the distance is, the more similar the vertices are. The minimum distance is 0 and it is the similarity between a vertex and itself.

Another similarity measure is the cosine similarity, which is the cosine of the angle between the two vectors. It has values in the interval $[-1, 1]$, with the property that $\cos(0) = 1$ and $\cos(a) < \cos(b) \leq 1$, where $\pi \geq a > b \geq 0$. A higher value means higher similarity, with 1 being the maximum. The cosine similarity ignores the magnitudes of the cloud vectors and only considers the angle between them, therefore two cloud vectors that are not identical but have the same direction (angle between them 0) have the cosine similarity 1.

4 Naive approach

A naive implementation that does not scale to large graphs is to compute and store all could vectors $\boldsymbol{c}_i, \forall i \in \mathcal{V}$. Observe that *Equation 3* is a geometric series and can be computed as

$$\boldsymbol{c}_i = (\mathbf{I} - \mu\mathbf{D})^{-1}\boldsymbol{\delta}_i, \tag{4}$$

so every $\boldsymbol{c}_i$ vector is the i^{th} column in the matrix $(\mathbf{I} - \mu\mathbf{D})^{-1}$.

The naive approach was implemented and run on a small graph designed by hand and it gave meaningful results (vertices connected to the graph in a similar way have a small $\|\boldsymbol{c}_i - \boldsymbol{c}_j\|^2$) but it does not scale for large datasets.

Large graphs are expected to have sparse adjacency matrices, where the number of edges is significantly lower then the number of vertices squared ($|\mathcal{E}| \ll N^2$). The inverse $(\mathbf{I} - \mu\mathbf{D})^{-1}$ is not sparse and it takes significantly more memory than the adjacency matrix. If the graph is large enough, it can easily take more memory than the available RAM on an average computer. Another problem is that computing the matrix inverse is a computationally costly operation of complexity $O(N^3)$.

5 Approximative approach

There are two similarity measures that should be computed, namely the Euclidean distance between two cloud vectors and the cosine of the angle between them.

Geometrically, if vectors $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$ start from the origin, then $\boldsymbol{c}_i$, $\boldsymbol{c}_j$ and $\boldsymbol{c}_i - \boldsymbol{c}_j$ describe a triangle. Let θ be the angle between $\boldsymbol{c}_i$ and $\boldsymbol{c}_j$. From the Law of Cosines

$$\|\boldsymbol{c}_i - \boldsymbol{c}_j\|^2 = \|\boldsymbol{c}_i\|^2 + \|\boldsymbol{c}_j\|^2 - 2\|\boldsymbol{c}_i\|\|\boldsymbol{c}_j\| \cos\,\theta. \tag{5}$$

Take the dot product $\boldsymbol{c}_i^T \boldsymbol{c}_j = \|\boldsymbol{c}_i\|\|\boldsymbol{c}_j\| \cos\,\theta$. The angle between $\boldsymbol{c}_i$ and itself is 0 and $\cos\,0 = 1$, thus $\|\boldsymbol{c}_i\|^2 = \boldsymbol{c}_i^T \boldsymbol{c}_i$. Now *Equation 5* becomes

$$\|\boldsymbol{c}_i - \boldsymbol{c}_j\|^2 = \boldsymbol{c}_i^T \boldsymbol{c}_i + \boldsymbol{c}_j^T \boldsymbol{c}_j - 2\boldsymbol{c}_i^T \boldsymbol{c}_j, \tag{6}$$

from which the Euclidean distance can be computed by taking the square root of the result.

The cosine similarity can be computed using the same dot products

$$\cos\,\theta = \frac{\boldsymbol{c}_i^T \boldsymbol{c}_j}{\|\boldsymbol{c}_i\|\|\boldsymbol{c}_j\|} = \frac{\boldsymbol{c}_i^T \boldsymbol{c}_j}{\sqrt{\boldsymbol{c}_i^T \boldsymbol{c}_i}\,\sqrt{\boldsymbol{c}_j^T \boldsymbol{c}_j}}. \tag{7}$$

Therefore, it is sufficient to be able to compute all dot products $\boldsymbol{c}_i^T \boldsymbol{c}_j$ (for all $i, j \in v$) to compute the similarities between all vertices of the graph. Computing and storing all $\boldsymbol{c}_i^T \boldsymbol{c}_j; i, j \in v$ takes too much time and memory for a very large dataset, so we will make a trade-off between the accuracy of the result and the computational complexity.

An approximative algorithm to efficiently compute the dot products $\boldsymbol{c}_i^T \boldsymbol{c}_j$ will be described.

The algorithm is slightly different for directed and undirected graphs, because undirected graphs have symmetric adjacency matrices. Symmetric matrices have the advantage of having real eigenvalues and orthogonal eigenvectors, whereas non-symmetric matrices can have both real and complex eigenvalues and the eigenvectors need not be orthogonal.

Both versions of the algorithm use the normalised adjacency matrix (from *Equation 1*), which

can be vectorised as

$$\mathbf{D} = \mathbf{WM}, \tag{8}$$

where $\mathbf{W}$ is the diagonal matrix

$$\mathbf{W}_{ii} = \frac{1}{\sum_{j=1}^{N} \mathbf{M}_{ij}}. \tag{9}$$

5.1 Undirected graphs

In this subsection, assume that the graph $\mathcal{G}$ is undirected and, as a consequence, the adjacency matrix $\mathbf{M}$ is a real symmetric matrix.

The normalised adjacency matrix $\mathbf{D}$ is not symmetric as it is the product of a diagonal matrix ($\mathbf{W}$) and a symmetric matrix ($\mathbf{M}$). Knowing we are interested in $\mathbf{D}^t$, define a new symmetric matrix

$$\mathbf{A} = \mathbf{W}^{1/2}\mathbf{M}\mathbf{W}^{1/2} \tag{10}$$

that can be used in

$$\mathbf{D}^t = (\mathbf{WM})^t = \mathbf{W}^{1/2}\mathbf{A}^t\mathbf{W}^{-1/2}. \tag{11}$$

The matrices $\mathbf{W}^{1/2}$ and $\mathbf{W}^{-1/2}$ are easy to compute because $\mathbf{W}$ is a diagonal matrix. Apply the operation only on the diagonal elements and obtain $(\mathbf{W}^{1/2})_{ii} = (\mathbf{W}_{ii})^{1/2}$ and $(\mathbf{W}^{-1/2})_{ii} = (\mathbf{W}_{ii})^{-1/2}$.

Let λ_i be the i^{th} eigenvalue of $\mathbf{A}$ and $\mathbf{v}^{(i)}$ its corresponding eigenvector ($\forall i \in \{1,2,...,N\}$). $\mathbf{V}$ is a matrix of all N eigenvectors (such that $\mathbf{v}^{(i)}$ is the i^{th} column of $\mathbf{v}$), and $\mathbf{\Lambda}$ is a diagonal matrix of all N eigenvalues

$$\mathbf{v} = \begin{bmatrix} | & | & & | \\ \mathbf{v}^{(1)} & \mathbf{v}^{(2)} & \dots & \mathbf{v}^{(N)} \\ | & | & & | \end{bmatrix}, \qquad \mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & & \lambda_N \end{bmatrix}. \tag{12}$$

The diagonalisation of the matrix $\mathbf{A}$ is $\mathbf{A} = \mathbf{v}\mathbf{\Lambda}\mathbf{v}^{-1}$. $\mathbf{A}$ is a real symmetric matrix thus its eigenvectors are orthogonal, which makes $\mathbf{v}$ an orthogonal matrix, therefore $\mathbf{v}^{-1} = \mathbf{v}^T$ [15].

The diagonalisation of $\mathbf{A}$ is then

$$\mathbf{A} = \mathbf{v}\mathbf{\Lambda}\mathbf{v}^T \tag{13}$$

$$\Rightarrow \mathbf{A}^t = \left(\mathbf{v}\,\mathbf{\Lambda}\,\mathbf{v}^T\right)^t = \mathbf{v}\mathbf{\Lambda}^t\mathbf{v}^T \tag{14}$$

$$= \sum_{a=1}^{N} \lambda_a^t\, \mathbf{v}^{(a)}\, \mathbf{v}^{(a)T}. \tag{15}$$

From *Equation 11* and *Equation 15*

$$\mathbf{D}^t = \mathbf{W}^{1/2} \sum_{a=1}^{N} \lambda_a^t\, \mathbf{v}^{(a)}\, \mathbf{v}^{(a)T} \mathbf{W}^{-1/2}. \tag{16}$$

Plug $\mathbf{D}^t$ from *Equation 16* into *Equation 3*

$$\boldsymbol{c}_i = \sum_{t=0}^{\infty} \mu^t \mathbf{W}^{1/2} \sum_{a=1}^{N} \lambda_a^t\, \mathbf{v}^{(a)}\, \mathbf{v}^{(a)T} \mathbf{W}^{-1/2} \boldsymbol{\delta}_i, \tag{17}$$

rearrange the equation and obtain

$$\boldsymbol{c}_i = \mathbf{W}^{1/2} \sum_{a=1}^{N} \sum_{t=0}^{\infty} \mu^t \lambda_a^t\, \mathbf{v}^{(a)}\, \mathbf{v}^{(a)T} \mathbf{W}^{-1/2} \boldsymbol{\delta}_i. \tag{18}$$

Note that $\sum_{t=0}^{\infty} \mu^t \lambda_a^t$ is a sum of a geometric series with the first term 1 and ratio $\mu\lambda_a$, therefore

$$\sum_{t=0}^{\infty} \mu^t \lambda_a^t = \frac{1}{1-\mu\lambda_a}. \tag{19}$$

Observe $\mathbf{v}^{(a)T}\mathbf{W}^{-1/2}\boldsymbol{\delta}_i$ is a scalar

$$\mathbf{v}^{(a)T}\mathbf{W}^{-1/2}\boldsymbol{\delta}_i = \mathbf{v}_i^{(a)}\mathbf{W}_{ii}^{-1/2}. \tag{20}$$

Substitute *Equation 20* and *Equation 19* back into *Equation 18* and obtain

$$\boldsymbol{c}_i = \mathbf{W}^{1/2} \sum_{a=1}^{N} \frac{1}{1-\mu\lambda_a}\, \mathbf{v}^{(a)}\, \mathbf{v}_i^{(a)} \mathbf{W}_{ii}^{-1/2}. \tag{21}$$

An approximation of the vectors $\boldsymbol{c}_i$ $(\forall i \in \mathcal{V})$ can be obtained by only using the k leading eigenvectors and eigenvalues of $\mathbf{A}$ instead of all of them. It is expected that the $\mathbf{A}$ is large and sparse, because in a typical dataset the number of relationships is significantly smaller than the number of data points squared. In terms of the undirected graph $\mathcal{G}(\mathcal{V},\mathcal{E})$, it is expected that $\|\mathcal{E}\| \ll \|\mathcal{V}\|^2$. The Lanczos Method is an efficient iterative algorithm to compute the leading eigenvalues of large sparse real symmetric matrices [3] and it is implemented in both Matlab[16] and Python (SciPy).

Compute the largest k eigenvalues and eigenvectors using the Lanczos Method and define $\hat{\mathbf{v}}$ to be the matrix of the k leading eigenvectors and $\hat{\mathbf{\Lambda}}$ to be the diagonal matrix of the leading k eigenvalues

$$\hat{\mathbf{v}} = \begin{bmatrix} | & | & & | \\ \mathbf{v}^{(1)} & \mathbf{v}^{(2)} & \dots & \mathbf{v}^{(k)} \\ | & | & & | \end{bmatrix}, \qquad \hat{\mathbf{\Lambda}} = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & & \lambda_k \end{bmatrix}. \tag{22}$$

Let $\hat{\boldsymbol{c}}_i$ be an approximation of $\boldsymbol{c}_i$

$$\hat{\boldsymbol{c}}_i = \mathbf{W}^{1/2} \sum_{a=1}^{k} \frac{1}{1-\mu\lambda_a} \mathbf{v}^{(a)} \mathbf{v}_i^{(a)} \mathbf{W}_{ii}^{-1/2}. \tag{23}$$

To simplify the equation, let $\mathbf{Z}$ be a $N \times k$ matrix such that

$$\mathbf{Z}_{ij} = \frac{\mathbf{v}_i^{(j)} \mathbf{W}_{ii}^{-1/2}}{1-\mu\lambda_j}, \tag{24}$$

or vectorised using a new $k \times k$ diagonal matrix $\mathbf{R}$,

$$\mathbf{Z} = \mathbf{W}^{-1/2} \hat{\mathbf{v}} \mathbf{R}, \qquad \mathbf{R}_{ii} = \frac{1}{1-\mu\lambda_i}. \tag{25}$$

Substitute $\mathbf{Z}$ in *Equation 23* to get

$$\hat{\boldsymbol{c}}_i = \mathbf{W}^{1/2} \sum_{a=1}^{k} \mathbf{Z}_{ia} \mathbf{v}^{(a)}. \tag{26}$$

The dot product between $\hat{\boldsymbol{c}}_i$ and $\hat{\boldsymbol{c}}_j$ becomes

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \left(\mathbf{W}^{1/2} \sum_{a=1}^{k} \mathbf{Z}_{ia} \mathbf{v}^{(a)}\right)^T \left(\mathbf{W}^{1/2} \sum_{a'=1}^{k} \mathbf{Z}_{ia'} \mathbf{v}^{(a')}\right) \tag{27}$$

$$= \sum_{a=1}^{k} \mathbf{Z}_{ia} \mathbf{v}^{(a)T} \mathbf{W} \sum_{a'=1}^{k} \mathbf{Z}_{ia'} \mathbf{v}^{(a')} \tag{28}$$

$$= \sum_{a=1}^{k} \sum_{a'=1}^{k} \mathbf{Z}_{ia} \mathbf{Z}_{ja'} \mathbf{v}^{(a)T} \mathbf{W} \mathbf{v}^{(a')} \tag{29}$$

Define a new $k \times k$ matrix $\mathbf{Q}$ such that

$$\mathbf{Q}_{ij} = \mathbf{v}^{(i)T} \mathbf{W} \mathbf{v}^{(j)}, \tag{30}$$

or vectorised

$$\mathbf{Q} = \hat{\mathbf{v}}^T \mathbf{W} \hat{\mathbf{v}}. \tag{31}$$

$\mathbf{Z}_i$ is the i^{th} row of the matrix $\mathbf{Z}$, as a column vector. Substitute $\mathbf{Q}$ into *Equation 29* and obtain

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \sum_{a=1}^{k} \sum_{a'=1}^{k} \mathbf{Z}_{ia} \mathbf{Z}_{ja'} \mathbf{Q}_{aa'} \tag{32}$$

$$= \mathbf{Z}_i^T \mathbf{Q} \mathbf{Z}_j \tag{33}$$

A Hermitian, positive-definite matrix can be factorised into a lower (or upper) triangular matrix and its transpose. The process is called Cholesky decomposition. The matrix $\mathbf{Q}$ only has real elements, and

$$\mathbf{Q}^T = (\hat{\mathbf{v}}^T \mathbf{W} \hat{\mathbf{v}})^T = \hat{\mathbf{v}}^T (\hat{\mathbf{v}}^T \mathbf{W})^T = \hat{\mathbf{v}}^T \mathbf{W} \hat{\mathbf{v}} = \mathbf{Q}, \tag{34}$$

thus $\mathbf{Q}$ is a real symmetric matrix, so Hermitian. We expect all the weights in the graph to be positive, so all the diagonal elements in $\mathbf{W}$ are positive, therefore we can say that

$$\forall \boldsymbol{x} \neq \mathbf{0},\ \boldsymbol{x}^T \mathbf{Q} \boldsymbol{x} = (\hat{\mathbf{v}} \boldsymbol{x})^T \mathbf{W} (\hat{\mathbf{v}} \boldsymbol{x}) \tag{35}$$

$$= \sum_i \mathbf{W}_{ii} (\hat{\mathbf{v}} \boldsymbol{x})_i^2 > 0, \tag{36}$$

because $(\hat{\mathbf{v}} \boldsymbol{x}) = \mathbf{0}$ if and only if $\boldsymbol{x} = \mathbf{0}$ since $\hat{\mathbf{v}}$ is orthogonal. Therefore, by definition, $\mathbf{Q}$ is

positive definite.

Using Cholesky decomposition, $\mathbf{Q}$ can be written as $\mathbf{Q} = \mathbf{L}\mathbf{L}^T$, which can be used to simplify the dot product even further as follows

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \mathbf{Z}_i^T \mathbf{L}\mathbf{L}^T \mathbf{Z}_j \tag{37}$$

$$= (\mathbf{L}^T \mathbf{Z}_i)^T (\mathbf{L}^T \mathbf{Z}_j). \tag{38}$$

Define a new $k \times N$ matrix $\boldsymbol{\omega}$ with columns $\boldsymbol{\omega}_i = \mathbf{L}^T \mathbf{Z}_i$, which can be vectorised as

$$\boldsymbol{\omega} = \mathbf{L}^T \mathbf{Z}^T. \tag{39}$$

The final form of the approximative dot product between cloud vectors is

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \boldsymbol{\omega}_i^T \boldsymbol{\omega}_j, \tag{40}$$

therefore the only matrix that needs to be computed and stored is $\boldsymbol{\omega}$, which takes $O(kN)$ space. Computing an approximative dot product has the time complexity $O(k)$, because the only operation is a dot product of two vectors of size k.

In the rest of this report, we will refer to a computed ω as an *undirected index*, because it is the only matrix required to compute all the relevant similarity measures for an undirected graph.

5.2 Directed graphs

In this subsection, assume that the graph $\mathcal{G}$ is directed and, as a consequence, the adjacency matrix $\mathbf{M}$ is a general real matrix. As $\mathbf{M}$ is not symmetric, there is no property to take advantage of, thus the normalised adjacency $\mathbf{D}$ will be used directly.

Let $\boldsymbol{\Lambda}$ be a diagonal matrix with the eigenvalues of $\mathbf{D}$, where λ_i is the i^{th} eigenvalue. $\mathbf{v}_{\boldsymbol{R}}^{(i)}$ is the i^{th} right eigenvector and $\mathbf{v}_{\boldsymbol{R}}$ is a matrix of the right eigenvectors. $\mathbf{v}_{\boldsymbol{L}}^{(i)}$ is the i^{th} left

eigenvector and $\mathbf{v}_L$ is a matrix of the left eigenvectors.

$$\mathbf{v}_R = \begin{bmatrix} | & | & & | \\ \mathbf{v}_R^{(1)} & \mathbf{v}_R^{(2)} & \dots & \mathbf{v}_R^{(N)} \\ | & | & & | \end{bmatrix}, \quad \mathbf{v}_L = \begin{bmatrix} — & \mathbf{v}_L^{(1)} & — \\ — & \mathbf{v}_L^{(2)} & — \\ & \vdots & \\ — & \mathbf{v}_L^{(N)} & — \end{bmatrix}, \quad \mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & & \lambda_N \end{bmatrix}. \tag{41}$$

Some of the eigenvalues and some elements of the eigenvectors might be complex. However, because $\mathbf{D}$ is real, if an eigenvalue λ_i is complex, there exists an eigenvalue λ_j that is its complex conjugate, that is $\lambda_i = \bar{\lambda_j}$. The same applies to eigenvectors, if an eigenvector $\mathbf{v}^{(i)}$ (left or right) has complex elements, there exists another eigenvector $\mathbf{v}^{(j)}$ (left or right, respectively) that is its complex conjugate, $\mathbf{v}^{(i)} = \bar{\mathbf{v}^{(j)}}$.

Knowing that $\mathbf{v}_R^{-1} = \mathbf{v}_L$, the eigendecomposition of $\mathbf{D}$ is

$$\mathbf{D} = \mathbf{v}_R \mathbf{\Lambda} \mathbf{v}_R^{-1} = \mathbf{v}_R \mathbf{\Lambda}\, \mathbf{v}_L. \tag{42}$$

Also

$$\mathbf{D}^t = \mathbf{v}_R \mathbf{\Lambda}^t \mathbf{v}_R^{-1} = \mathbf{v}_R \mathbf{\Lambda}^t\, \mathbf{v}_L. \tag{43}$$

We can now write $\boldsymbol{c}_i$ as

$$\boldsymbol{c}_i = \sum_{t=0}^{\infty} \mu^t \mathbf{D}^t \boldsymbol{\delta}_i \tag{44}$$

$$= \sum_{t=0}^{\infty} \mu^t\, \mathbf{v}_R\, \mathbf{\Lambda}^t\, \mathbf{v}_L \boldsymbol{\delta}_i \tag{45}$$

$$= \sum_{t=0}^{\infty} \mu^t \sum_{a=1}^{N} \lambda_a^t\, \mathbf{v}_R^{(a)}\, \mathbf{v}_L^{(a)} \boldsymbol{\delta}_i. \tag{46}$$

Note that $\mathbf{v}_L^{(a)} \boldsymbol{\delta}_i$ is the i^{th} value of $\mathbf{v}_L^{(a)}$ and obtain

$$\boldsymbol{c}_i = \sum_{t=0}^{\infty} \mu^t \sum_{a=1}^{N} \lambda_a^t\, \mathbf{v}_R^{(a)}\, \mathbf{v}_{Li}^{(a)}. \tag{47}$$

Observe that $\sum_{t=0}^{\infty} \mu^t \lambda_a^t$ is a sum of a geometric series, rearrange and obtain

$$\boldsymbol{c}_i = \sum_{a=1}^{N} \frac{\mathsf{v}_{\boldsymbol{L}i}^{(a)}}{1-\mu\lambda_a} \mathsf{v}_{\boldsymbol{R}}^{(a)}. \tag{48}$$

Same as for undirected graphs, an approximation of the cloud vectors can be made using only a small number of the largest eigenvalues and eigenvectors. Because $\mathbf{D}$ is not symmetric, it might have complex eigenvalues, and in order to have real approximations of the cloud vectors, it is necessary to check that for every complex eigenvalue used its complex conjugate is also used.

Let $\hat{\mathbf{\Lambda}}$ be a diagonal matrix containing the leading k eigenvalues, $\hat{\mathbf{v}}_{\boldsymbol{R}}$ be a matrix that contains the leading k right eigenvectors, and $\hat{\mathbf{v}}_{\boldsymbol{L}}$ a matrix of the leading k left eigenvectors. Arnoldi Iteration is an efficient iterative algorithm to compute the leading eigenvalues and eigenvectors of large sparse matrices [11]. Assume that in the k eigenpairs computed, if there is a complex eigenvalue λ_i, there also is its complex conjugate, $\lambda_j = \bar{\lambda}_i$.

Let $\hat{\boldsymbol{c}}_i$ be the approximation of the cloud vector $\boldsymbol{c}_i$,

$$\hat{\boldsymbol{c}}_i = \sum_{a=1}^{k} \frac{\mathsf{v}_{\boldsymbol{L}i}^{(a)}}{1-\mu\lambda_a} \mathsf{v}_{\boldsymbol{R}}^{(a)}. \tag{49}$$

For any complex number $x = a + bi$, the sum $x + \bar{x} = a + bi + a - bi = 2a$ is real. The approximative cloud vectors $\hat{\boldsymbol{c}}_i$ are real vectors because they are sums of real numbers and pairs of complex and complex conjugates, as

$$\text{for all } \lambda_a \in \mathbb{C} - \mathbb{R} \text{ there exists } \lambda_b = \bar{\lambda}_a \text{ such that } b \leq k, \tag{50}$$

$$\text{for all } \mathsf{v}_{\boldsymbol{L}i}^{(a)} \in \mathbb{C} - \mathbb{R} \text{ there exists } \mathsf{v}_{\boldsymbol{L}i}^{(b)} = \bar{\mathsf{v}}_{\boldsymbol{L}i}^{(a)} \text{ such that } b \leq k, \tag{51}$$

$$\text{for all } \mathsf{v}_{\boldsymbol{R}}^{(a)} \in \mathbb{C} - \mathbb{R} \text{ there exists } \mathsf{v}_{\boldsymbol{R}}^{(b)} = \bar{\mathsf{v}}_{\boldsymbol{R}}^{(a)} \text{ such that } b \leq k. \tag{52}$$

The assumption about k only covers the first rule (*Equation 50*) but it is sufficient because an eigenvector is complex only if its corresponding eigenvalue is complex and if two eigenvalues are conjugates $\lambda_a = \bar{\lambda}_b$ then their corresponding eigenvectors are also conjugates $\mathsf{v}^{(a)} = \bar{\mathsf{v}}^{(b)}$.

Equation 49 can be rewritten as

$$\hat{\boldsymbol{c}}_i = \sum_{a=1}^{k} \mathbf{Z}_{ia} \mathbf{v}_{\boldsymbol{R}}^{(a)}, \tag{53}$$

where $\mathbf{Z}$ is a $N \times k$ matrix such that

$$\mathbf{Z}_{ij} = \frac{\mathbf{v}_{\boldsymbol{L}i}^{(j)}}{1-\mu\lambda_j}, \tag{54}$$

or vectorised using a new $k \times k$ diagonal matrix $\mathbf{R}$,

$$\mathbf{Z} = \hat{\mathbf{v}}_{\boldsymbol{L}}^T \mathbf{R}, \qquad \mathbf{R}_{ii} = \frac{1}{1-\mu\lambda_i}. \tag{55}$$

The dot product between $\hat{\boldsymbol{c}}_i$ and $\hat{\boldsymbol{c}}_j$ becomes

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \Big(\sum_{a=1}^{k} \mathbf{Z}_{ia} \mathbf{v}_{\boldsymbol{R}}^{(a)} \Big)^T \Big(\sum_{a'=1}^{k} \mathbf{Z}_{ja'} \mathbf{v}_{\boldsymbol{R}}^{(a')} \Big) \tag{56}$$

$$= \sum_{a=1}^{k} \sum_{a'=1}^{k} \mathbf{Z}_{ia} \mathbf{Z}_{ja'} \mathbf{v}_{\boldsymbol{R}}^{(a)T} \mathbf{v}_{\boldsymbol{R}}^{(a')} \tag{57}$$

Define a $k \times k$ matrix $\mathbf{Q}$ such that

$$\mathbf{Q}_{ij} = \mathbf{v}_{\boldsymbol{R}}^{(i)T} \mathbf{v}_{\boldsymbol{R}}^{(j)}, \tag{58}$$

or vectorised

$$\mathbf{Q} = \hat{\mathbf{v}}_{\boldsymbol{R}}^T \hat{\mathbf{v}}_{\boldsymbol{R}}, \tag{59}$$

and substitute it into the dot product (*Equation 57*) to obtain

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \sum_{a=1}^{k} \sum_{a'=1}^{k} \mathbf{Z}_{ia} \mathbf{Z}_{ja'} \mathbf{Q}_{aa'}. \tag{60}$$

$\mathbf{Z}_i$ is the i^{th} row of the matrix $\mathbf{Z}$, as a column vector. The final approximative dot product between two cloud vectors in a directed graph is then

$$\hat{\boldsymbol{c}}_i^T \hat{\boldsymbol{c}}_j = \mathbf{Z}_i^T \mathbf{Q} \mathbf{Z}_j. \tag{61}$$

In this case, Cholesky decomposition cannot be applied because $\hat{\mathbf{v}}$ is not orthogonal and, as

a consequence, $\mathbf{Q}$ is not positive-definite.

It is only required to compute and store $\mathbf{Q}$ and $\mathbf{Z}$, which together take $O(k^2 + kN)$ space. The time complexity of computing the approximative dot product between two cloud vectors is the same as multiplying two vectors of length k and a $k \times k$ matrix.

In the rest of this report, we will refer to a computed pair of matrices $\mathbf{Q}$ and $\mathbf{Z}$ as a *directed index*.

5.3 Comparison with naive approach

While the naive approach takes significantly more time and space ($O(N^3)$ space compared to $O(k^2 + kN)$), it also represents the graph more accurately because it uses the whole adjacency matrix and not only an approximation of it.

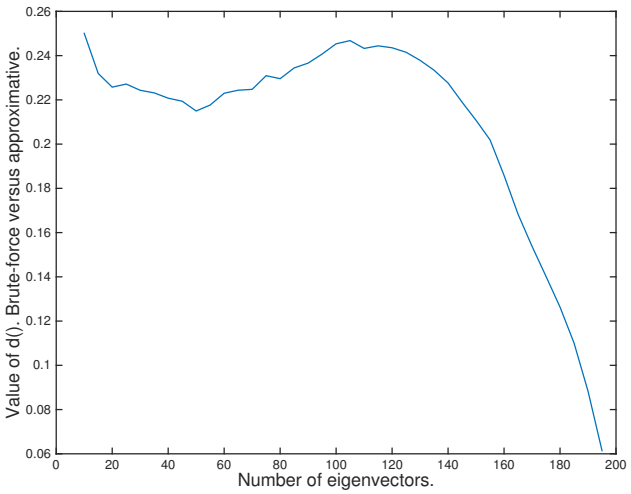
A simple evaluation metric of how well the approximative approach retains the most important information from the graph is to evaluate how different are the top similar vertices for the naive approach and the approximative approach with different choices of k.

For a dataset of N vertices, let $B(i, j)$ be the position of vertex j in the top similar vertices of vertex i for the naive approach. Similarly, let $T_k(i, j)$ be the position of vertex j in the top similar vertices of vertex i for the approximative approach with k eigenvalues. Define the difference between two such tops to be

$$d(B, T_k) = \frac{1}{N^3} \sum_{i=1}^{N} \sum_{j=1}^{N} \text{abs}(B(i, j) - T_k(i, j)). \tag{62}$$

The equation is normalised by N^3 because the maximum value we can get lies in the interval (N^2, N^3).

The value of $d()$ was evaluated on a randomly generated undirected graph with 200 vertices and 350 edges, between the naive approach and the approximative approach with $k = 10, 11, \ldots, 199$ eigenvectors. The choice of the penalising factor $\mu = 0.5$ was used for all the runs. The results can be observed in *Figure 2*. We can see that if we use the majority of eigenvalues, the value of $d()$ is very small. It is computationally infeasible to do that on a large dataset thus we will aim for a number of eigenvectors k in the range of 20 to 40, where



(a) A plot of the differences on a range of eigenvalues.

Parameters	Value of d()
$d(B, T_{10})$	0.2501
$d(B, T_{20})$	0.2257
$d(B, T_{30})$	0.2243
$d(B, T_{40})$	0.2208
$d(B, T_{50})$	0.2150
$d(B, T_{100})$	0.2453
$d(B, T_{199})$	0.0325

(b) Table of some of the values.

Figure 2: Difference of results between naive and approximative approaches.

the difference seems to be at a local minimum (for this graph). The local minimum differs from dataset to dataset depending mainly on the size of the dataset. For completeness, the experiment was run on a randomly generated undirected graph with 500 vertices and 700 edges. The first local minimum of $d()$ was at around 100 eigenvectors with a value of approx. 0.2. At 40 and 20 eigenvectors, the values of $d()$ obtained are 0.2297 and 0.2557, respectively.

This is just an evaluation of how well the approximation performs compared to the naive implementation. However, on real datasets it might be the case that ignoring some information is better than using it and the approximation approach might perform better. For large enough datasets it is impossible to compute the exact similarities.

6 Heuristics and the N^2 comparisons

After having a directed or undirected index, thus a fast way of computing the dot products between cloud vectors and the relevant similarity measures, it is still required to compute the similarities between all the vertices in the graph to obtain the pairs of top similar vertices, which is a task of complexity $O(kN^2 + 2N^2 \log N)$ (computing all similarities and sorting). For some applications like recommender systems, evaluating the top similarities on demand for one vertex at a time and having a good cache will save some computing power, however it is

still a $O(kN + N\log N)$ task for every cache miss, which might be too slow for large systems that require a constant low latency.

Assume that if two vertices are too far away from each other in the graph, they are unlikely to be similar. The computation of similarities can then be limited to only those pairs of vertices that are not too far away. Formally, let $\text{dist}(i, j)$ be the distance between vertices i and j (the length of the shortest path). Only compute similarities for those pairs of vertices such that $\text{dist}(i, j) \leq d$ for d being the maximum distance allowed, and consider all the other pairs of vertices not similar. This filter is implemented using a trivial breadth first search stopping at the desired maximum depth.

We will call the filter applied on the graph to drop pairs of edges that are unlikely to be similar a *heuristic*, and the maximum distance filter described above will be called the *maximum depth heuristic*.

Using a heuristic can reduce the number of similarities computed and thus speed up the computation while not having a big impact on the overall accuracy. As an example, on a friends dataset of 4039 vertices, 83,822 edges and a diameter of 8 (longest shortest path), the maximum depth heuristic with depth 3 has reduced the number of comparisons by approx. 36.3%, and the overall change in accuracy is negligible. The evaluation of the algorithm on the SNAP Facebook dataset (on a range of k and μ) with no heuristic takes about 37 minutes, and with the maximum depth heuristic of depth 3, it only takes about 18 minutes (approx. $2x$ speed-up).

7 Implementation

7.1 Early versions

At the beginning of the project, a choice had to be made on what tools and languages to use for the implementation of this algorithm. At a very early stage the requirement was a language that provides easy interfaces to common and not very common linear algebra functions and algorithms. Matlab was picked because it has all the common matrix operations, good sparse matrix support and all the eigenvalue algorithms required built-in. Matlab is also easy to

prototype in.

The approximative approach for undirected graphs was implemented in Matlab as a couple of functions. The function signatures along with descriptions are presented in *Table 1*. Some helper functions are not included in the table. The (iterative) naive algorithm was implemented as a separate set of Matlab functions with a very similar interface.

`similarity(adj, mu, k)`	Computes and returns matrices **Q** and **Z**.
`sim2(q, z, i, j)`	Computes the similarity between vertices i and j.
`simlist(q, z, i)`	Computes the similarity between vertex i and all other vertices.
`top(q, z, i)`	Uses simlist(q,z,i) and sorts the results.
`randomGraph(v, e, dir)`	Generates a random connected graph with v vertices and e edges. If dir==true, the graph is directed, otherwise it is undirected.

Table 1: Matlab implementation interface.

Matlab is, however, not very good at handling different types of files, loops, and general data wrangling. It also cannot be easily used from the command line or from other languages which is very useful for data processing and for incorporating the algorithm in larger systems.

7.2 Data processing tools

In an ideal world, the data comes in the exact format that we want. Unfortunately, this is never the case in the real world where data comes in all shapes and sizes and thus there is a need for tools to process it, and reshape it to fit into the system.

In the Matlab model, a graph file could be a CSV file where each row represents an edge, and each vertex is an integer starting from 1 and incrementing. Nearly no real world dataset comes in this format and in order to make the conversion some tools must be implemented.

The tools should be quick and easy to use and simple to change and adapt if the requirements change, thus it is best if they are small command line programs that are good at doing one task each. A good language for writing data wrangling tools is simple, easy to compile and run, easy to prototype in and has a good standard library for I/O and parsing common file formats. Go was picked due to my fluency in the language and because it fits all the requirements for writing such tools.

A way to parse an input dataset to CSV edge lists was required. The tools are different from dataset to dataset, and the implementations can vary from small Go programs to bash scripts, depending on the input formats.

After the dataset is in CSV format, a tool is needed to make the vertex IDs integers from 1 to N (the number of vertices). A command line tool called *autoincr* was implemented in Go. It takes one or more CSV edge lists and assigns each vertex an integer ID starting from one and incrementing. It also stores a JSON file with the mappings created so the original vertex names can be looked up. The help message of the command line tool explains how it works in more detail.

Given a CSV edge list file, it was necessary to be able to easily find basic information about the graph it describes, like how many connected components it has, whether it is directed or undirected, the number of vertices, the number of edges. It was also required to be able to manipulate these graphs easily with actions like forcing a directed graph to be undirected (by adding the missing reverse edges), splitting a disconnected graph into more graphs, one for each connected component, and removing random edges from a connected graph while keeping it connected. The command line tool *conncomp* was written in Go for this purpose. The tool is used like this (snippet from the help message):

```
conncomp -action <action> [flags] <list of edge lists>

Possible actions:

details           Output details about the graphs.
components        Splits the graph(s) in connected components.
remove -n P       Removes at most P% random edges from graph(s).
force-undirected  Force the graph(s) into undirected graph(s).
```

Unit tests were written for both *conncomp* and *autoincr*.

All the tools described in this section are publicly available on GitHub under the MIT licence. The URL of the repository is `https://github.com/vladvelici/graph-dataset-tools`.

7.3 Python, SciPy and NumPy

After starting to process a real-world dataset for the evaluation of this algorithm, it turns out that Matlab was not the best choice for a couple of reasons:

1. It is very hard to write command line tools and they are slow to run because they need to open Matlab before running the scripts. This could be solved by using Octave to run the existing Matlab codebase, but it would not solve the other limitations.

2. The language is designed for linear algebra tasks, and it is inconvenient and non-intuitive to implement even simple graph algorithms in it.

3. It is hard to integrate the Matlab scripts into another system (e.g. a web service for building an interactive visualisation for the algorithm).

Other languages were investigated to port the algorithm in. The first one to look at was Go because some tools were already written in Go. There are some libraries for linear algebra in Go (e.g. bigo.matrix [10] which uses BLAS, or skelterjohn/go.matrix [1] which has parallel matrix multiplication implemented), but they are not mature enough and do not have (or have very limited) support for sparse matrices and eigenvalue/eigenvector computations.

Linear algebra libraries for C++ were also investigated. SLEPc [7] is a mature library for large sparse eigenvalue problems implemented as an extension of PETSc [2]. It includes implementations of the Lanczos Method and Arnoldi Iteration, however the interfaces are very low level and prototyping using it would be too slow.

Armadillo [14] is another C++ linear algebra library. It has a high level API backed by efficient implementations of all the basic matrix operations. It also has support for sparse matrices and computation of leading eigenvalues and eigenvectors. The API Armadillo provides is very similar to Matlab, making the algorithm easier to port.

The last language and set of libraries analysed is Python with NumPy and SciPy [9]. Python is a scripting language that is easy to read and write and can be used interactively. SciPy is a mature scientific computing library for Python which contains all the basic linear algebra functions, has good support for sparse matrices and uses ARPACK (same as Matlab and Armadillo) to access efficient implementations of the Lanczos Method and the Arnoldi

Iteration. It also has a very good and accessible documentation, a relatively large user base and active community.

A choice had to be made on which of these tools to use. C++ with Armadillo and Python with NumPy and SciPy were the best options in terms of linear algebra support and ease of use. Simple code was written in C++ with Armadillo and it turned out that routine parts of the code like parsing CSV files, dealing with I/O and command line flags were taking too long to write. In Python these tasks are trivial and the Python Standard Library has parsers for plenty of file formats and command line flag parsers. The interactive shell was also very handy to try things out before implementing them in a larger codebase.

As a result, Python with NumPy and SciPy were chosen to port the algorithm and continue the development.

7.4 Final implementation

The final implementation is based on two simple concepts. A similarity index, called a `Sim` object and a provider (`Provider`).

In the package `sim`, the class `Sim` represents a directed or undirected index. A list of methods is in *Table 2*. Internally, a `Sim` object has either a $\boldsymbol{\omega}$ matrix or a pair of $\mathbf{Q}$ and $\mathbf{Z}$, which it uses to compute the dot products and the similarities. The implementation of `Sim.nib(a)` is simply returning `a`. This is because the `Sim` class represents an index directly trained from an adjacency matrix but, in practice, vertices usually come with different IDs, strings or integers that are not necessarily consecutive.

`_dotprod(a, b)`	Computes the dot product between vertices `a` and `b`.
`score(a, b)`	Computes the Euclidean distance squared.
`save(f)`	Writes the index into file `f`.
`nodelist()`	Returns a list of all vertices.
`nid(a)`	Returns the index of vertex `a` in the index.
`__len__()`	Returns the number of vertices.

Table 2: Methods of class `Sim`.

The class `Simp` extends the class `Sim`. A `Provider` is required to promote an object of type `Sim` to `Simp`. Any object that can create an adjacency matrix and a mapping from any (relevant)

type of vertex ID to an integer index in the adjacency matrix is a `Provider`. The list of methods an object is required to implement to be a `Provider` is in *Table 3*.

`__getitem__(a)`	Get the index of `a` in the adjacency matrix.
`__len__()`	Return the number of vertices.
`adj()`	Get the adjacency matrix.
`nodelist()`	Return a list of all vertices.
`save(f)`	Saves itself in the file `f`.

Table 3: List of methods required for an object to have to be a valid `Provider`.

The class `Simp` (also defined in package `sim`) overwrites the `nib(a)`, `__len__()` and `nodelist()` methods of `Sim` by simply delegating them to the `Provider` object it has. The `save(f)` method is overwritten to persist both the `Provider` object and the `Sim` object into a single tar file.

The main algorithm is implemented in package `sim`, where there are some other helper functions defined. A partial list of functions is in *Table 4*.

`train(adj, mu, k)`	If `adj` is symmetric, it calls `train_undirected()`, otherwise it calls `train_directed()`. Returns a `Sim` object.
`train_undirected(adj, mu, k)`	Implementation of the undirected approximative algorithm. It uses the `k` leading eigenvalues and the penalising factor `mu`. Returns a `Sim` object that contains the matrix $\boldsymbol{\omega}$.
`train_directed(adj, mu, k)`	Implementation of the directed approximative algorithm. It uses the `k` leading eigenvalues and the penalising factor `mu`. If some of the eigenvalues are complex and their complex conjugates are not included, they are dropped without any warning. Returns a `Sim` object that contains the matrices $\mathbf{Q}$ and $\mathbf{Z}$.
`prov(p, sim)`	Combines the `Sim` object `sim` with the `Provider p` to return a `Simp` object.
`loadp(path)`	Load an index (`Sim` or `Simp`) from `path`.

Table 4: Partial list of functions from package `sim`.

There are two providers implemented in the `provider` package. These are

1. `EdgeList` takes a list of edges (as a Python array of tuples), it generates a unique integer ID starting at 0 for each vertex and creates an adjacency matrix. The mapping is kept as a Python dict;

2. `Offset` takes a list of edges that has integer vertex IDs and an offset. The mapping from the original vertex ID and the index in the adjacency matrix is made by adding the offset to the vertex ID. For the CSV files made for Matlab, the offset should be set to -1.

The maximum depth heuristic is implemented as the class `Maxdepth` from package `heuristics`. It takes an edge list and a depth and has a method `top(node=None)` that returns a dict of pairs if `node` is `None` or a list of nodes, or it returns a list of nodes if `node` is just one node. For example, `top(3)` returns something like `[2, 4, 9]`, but `top([3, 4])` returns something like `{3: [2, 4, 9], 4: [5, 9]}`. The maximum depth heuristic is implemented using NetworkX [6] for the graph algorithms.

The dot products and similarity measures can be cached. Three caching methods were implemented in the package `simcache`. A cache function takes any type of `Sim` object (can be `Sim`, `Simp` or any object that has the required methods, `_dotprod(a,b)` and `score(a,b)`) and extends it to add the relevant cache functionality. The caching functions are

1. `precompute` pre-computes the dot products and scores using matrix operations. It is unsuitable for large graphs as it uses $O(2N^2)$ memory for storage from the start.
2. `score` uses two hash maps to cache scores after they are computed the first time. It always uses the smaller vertex ID as the key for the first map and the larger ID for the second hash map, because scores are symmetric.
3. `dotprod` is similar to `score` but caches dot products instead.

A command line tool was built on top of the implementation. It is the main tool used to compute evaluations and visualisations of this algorithm. It has the following capabilities

train Takes an edge list and parameters like μ and k to create an index file that can be used for computing similarities and visualisations. It can be configured to use the `Edgelist` or `Offset` providers. Example usage:

```
$ ./cmd.py train snap.csv -k 40 -mu 0.5 --direct -o snap.index
```

info Outputs information about an index file, like the number of nodes and the type of index. Example usage:

```
$ ./cmd.py info snap.index

tar index file with 4039 nodes.
Index is in short format (for Q=L^T*L, stores L^T * Z^T).
```

sim Takes an index file and a list of vertices for which to compute similarities and outputs the results. Example usage:

```
$ ./cmd.py sim snap.index 2 3 --to 5 6
2    5    7.902533e-02
2    6    6.617750e-02
3    5    1.143215e-02
3    6    2.437439e-02
```

top Computes the top similarities. Takes an index file, and optional parameters to control the heuristic, the blacklist, a list of nodes to use and a limit for the total results. Example usage:

```
$ ./cmd.py top snap.index 2 3 --limit 4 -b snap.csv
3    55     7.642811e-05
3    276    8.820870e-05
3    212    1.432812e-04
3    103    1.813990e-04
```

dot Creates a dot file using a graph and an index file. Takes parameters to control heuristics and limit the total number of similarity edges created. The dot file produced is used with Graphviz [5] to create visualisations like the ones in *Figure 4* and *Figure 5*. Graphviz is used for visualisation because it produces high quality visualisations very fast and the *dot* file format is easy to generate. Sigma.js [8] was also tried but it did not scale to graphs with thousands of vertices. Example usage:

```
$ ./cmd.py dot snap.index snap.csv --limit=5 -d 3 -o snap.dot
```

And to create a visualisation with Graphviz

```
$ dot snap.dot -Tpng -o snap.png
```

or a faster way for large graphs

```
$ sfdp -Gsize=30!  -Goverlap=prism -Tsvg -o snap.svg snap.dot
```

eval Perform an evaluation of an index file. Takes an index file and a test set file and computes the following evaluation metrics

1. average position in top of the test pairs,
2. percentage of test pairs in top 5 similarities,
3. average score of test pairs,
4. average relative score of test pairs (squared difference of the test pair score and the best score of the first vertex),
5. percentage better than random,
6. random average top position,
7. random average score, and
8. and random average relative score.

Example usage:

```
$ ./cmd.py eval snap_short.index snap_removed.csv
```

trev Train and evaluate. It takes a graph, a test set and a range of k and μ and evaluates the same similarity metrics as **eval** but on all the combinations of k and μ in the ranges given. Example usage:

```
$ ./cmd.py trev snap_short.index snap_removed.csv -t u --direct -k 20
30 40 -mu 0.3 0.5 0.7
```

The command line tool has useful help messages when the `-t` flag is used, like

```
$ ./cmd.py -h
$ ./cmd.py trev -h.
```

The project was developed using git as a version control system with a remote repository on GitHub (`https://github.com/vladvelici/3yp`).

8 Evaluation

To objectively evaluate how well the outputs of the algorithm reflect the reality of the dataset it is being used on, we need to carefully define the goals of the algorithm in respect of the

dataset. We want labelled datasets where the natural *similarities* between entities is known and depends exclusively on the graph structure. We do not want to evaluate how well the weights of different relationship types are defined, thus the evaluation dataset must have only one relationship type.

In reality, datasets have multiple types of relationships and the similarity between two entities is not trivial to define. However, an evaluation metric can be formed by randomly removing a number of relationships and then running the algorithm to see how well it finds the removed relationships. It is subjective to some noise due to naturally similar vertices that are not linked in the original dataset (e.g. a social network might not contain all the real friendships and the algorithm might pick some of these relationships to be more similar then the ones removed).

8.1 Undirected evaluation on a friends dataset

One of the datasets used for the evaluation of the algorithm is a small subset of the friends relationships from the social network Facebook. The data is completely anonymised and it was downloaded from SNAP [12].

The friends graph was provided as an edge list in TXT format and it was converted to CSV using a `sed` command. Vertices are consecutive integers starting from 0, so there is no need for remapping. The dataset is a connected graph that contains 4039 vertices and 88,234 undirected edges. A vertex represents a person and an edge represents a friendship relationship.

The evaluation is based on the assumption that the likelihood of two people being friends is higher if the two people have more common friends, or friends of friends, or friends of friends of friends and so on. For this algorithm the similarity between two vertices in a graph is how similar their neighbourhoods are, which is the likelihood of people being friends as defined above.

To evaluate the algorithm on this dataset, 5% of the edges were randomly removed and two files were obtained, `05fb.csv` is the dataset with the edges removed, the training set (or $\mathscr{E}$), and `05removed_fb.csv` is a list containing the removed edges, the test set (or R). The `conncomp` tool command used is:

```
conncomp -action remove -p 0.05 -o "05" fb.csv
```

To evaluate how well the Euclidean distance between cloud vectors is predicting the removed edges, we run the algorithm on the training set, and compute the similarity for all pairs in the test set.

For any vertex i we can define the top similar vertices to be an ordered list of all the other vertices in the graph, sorted in ascending order by the score (Euclidean distance squared). For example, if $\text{top}(i) = [a, b, c, d]$, then $\|\boldsymbol{c}_i - \boldsymbol{c}_a\|^2 \leq \|\boldsymbol{c}_i - \boldsymbol{c}_b\|^2 \leq \|\boldsymbol{c}_i - \boldsymbol{c}_c\|^2 \leq \|\boldsymbol{c}_i - \boldsymbol{c}_d\|^2$. The position of vertex a in the top of vertex i is denoted by $\text{pos}(a, \text{top}(i))$, and in the example given it is 1.

If $\text{top}(i) = [a, b, c, d]$ and there are two removed edges (i, b) and (i, c), we cannot say which of those pairs should be more similar, therefore we want the top to ignore the ordering of the pairs in the test set (in other words, we want $\text{top}(i) = [a, b, c, d]$ to be equivalent to $\text{top}(i) = [a, c, b, d]$ in terms of the evaluation of the algorithm). In fact, ignoring the pairs in both the test set and the training set is better as we do not want to compare the similarity between vertices that are already connected. To do this we introduce a set, Blacklist (or B), that is the union of the test set, training set and all edges of type (i, i), then we define the $\text{bpos}(a, \text{top}(i))$ to be the position of a in the top of i without counting the pairs that are in the blacklist. Mathematically,

$$\text{Blacklist} = B = R \cup \mathscr{E} \cup \{(i, i) | \forall\, i \in \mathscr{V}\}, \tag{63}$$

$$\text{bpos}(a, \text{top}(i)) = \text{pos}(a, \text{top}(i)) - \sum_{(i,k) \in B} g(a, i, k), \tag{64}$$

$$g(a, i, k) = \begin{cases} 1, & \text{pos}(k, \text{top}(i)) < \text{pos}(a, \text{top}(i)) \\ 0, & \text{otherwise.} \end{cases} \tag{65}$$

One evaluation metric is the percentage of the removed edges that have their end vertex in the the top 5 similar vertices of the start vertex using the bpos top position. A chart of this measure versus the number of eigenvalues used, k, can be seen in *Figure 3a*. According to this metric it can be concluded that the accuracy increases with the number of eigenvalues used, and that, regardless of the choice of μ, it increases quicker up to 30 eigenvalues and more slowly for higher values.

The measure described above ignores the exact position of pairs in the test set in the top,

which might prove to be useful, especially because if the dataset is large the top 5 vertices would be a too small threshold and, in general, picking a sensible threshold is not trivial – it is dependent on the use case, but a choice can be a percentage of the size of the test set (e.g. 1%). The average of the positions in top of all the pairs in the test set,

$$\frac{1}{|R|}\sum_{(i,j)\in R}\text{bpos}(j,\text{top}(i)), \tag{66}$$

is a similarity metric that contains more information. A plot of this average versus the number of eigenvalues used is in *Figure 3d*, from which we can observe that for all choices of μ, the average top position decreases until $k = 40$ eigenvalues, after which the difference between the means for different μ increases and for large μ the average top position still decreases, but it increases for small μ. Despite the fact that the percentage of removed edges in top 5 increases with k, the average top position does not always decrease with the increase of k, fact explained by the similarities outside of top 5 being very low (thus being placed far in the top).

Another evaluation metric is comparing the scores obtained for the removed edges with the scores of random pairs of vertices. The random pairs were chosen such that they are not in Blacklist. The average percentage (from three runs) of the edges that have smaller scores (higher similarities) than random vs the number of eigenvalues used, k, is plotted in *Figure 3b*. It can be observed that it increases until $k = 40$ and $k = 50$ eigenvalues for all choices of mu, where it gets to values between 88% and 88.5%, after which the values and slopes differ with μ (for large μ it increases and for small μ it decreases).

The peaks and changes at $k = 40$ are related to the local minimum observed in *Figure 2*, where the naive and approximative approaches are compared, and it is deduced that there is a $k \ll N$ for which the approximation best represents the graph. We can argue that for this size of datasets the best choice is at around $k = 40$ eigenvalues. For this particular dataset, a penalising factor of approx. $\mu = 0.5$ is a good choice.

The average score is plotted in *Figure 3c*, along with the relative score (squared difference between best score and the removed edge score),

$$\frac{1}{|R|}\sum_{(i,j)\in R}\left(\|\boldsymbol{C}_i - \boldsymbol{C}_j\|^2 - \text{best}(i)\right)^2, \tag{67}$$

where $\text{best}(i) = \min_j(\|\boldsymbol{C}_i - \boldsymbol{C}_j\|^2), j \neq i$. It can be observed that the scores increase with the number of eigenvalues. This is explained by the fact that the dot products between cloud vectors are sums of k terms.

For a choice of $k = 40$ and $\mu = 0.5$, the algorithm performs between 88% and 88.5% better than random at finding the randomly removed friendships. However, the percentage of the removed edges that have a position in top 5 is arguably low at around 4.6%; the percentage increases to around 15% for top 20, and to around 26% for top 40 (1% of the test set). It might be the case that there are natural friendships in the dataset which are missing from the data. As a conclusion, it can be argued that the algorithm performs reasonably well under the assumption that the likelihood of two people being friends increases with their network of common friends, but in reality the friendship relationship is more complicated and depends on many features that are not modelled in this dataset and not considered by this algorithm.

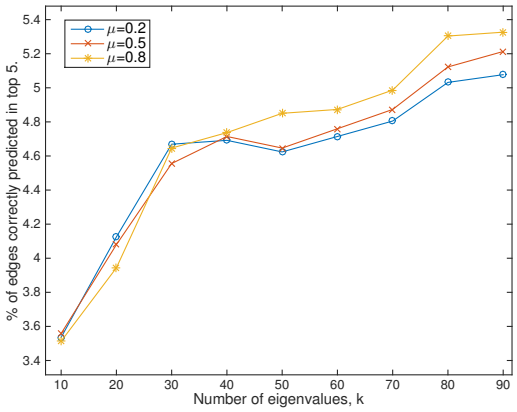
8.2 Visualisation

A subjective method of evaluation is attempting to visualise the algorithm output on real data and deduce whether it gives sensitive results. This method has the disadvantage of being subjective and it cannot scale to very large graphs, but it might give a broad idea of what the results are and help fine-tune the algorithm on specific problems.

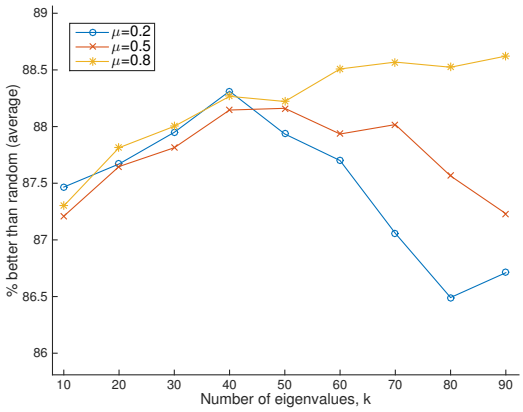
The undirected graph from *Figure 1* was rendered using Graphviz [5] in *Figure 4*, where the grey edges represent real edges in the graph and the red edges represent similar vertices. Only the top two similar vertices (that are not in Blacklist) are rendered. The parameters used for the algorithm are $k = 6$ and $\mu = 0.5$. The graph has 31 vertices and 41 undirected edges.

The SNAP Facebook dataset with the top 5 similar vertices and parameters $k = 40$ and $\mu = 0.5$ is rendered in a similar way in *Figure 5*.

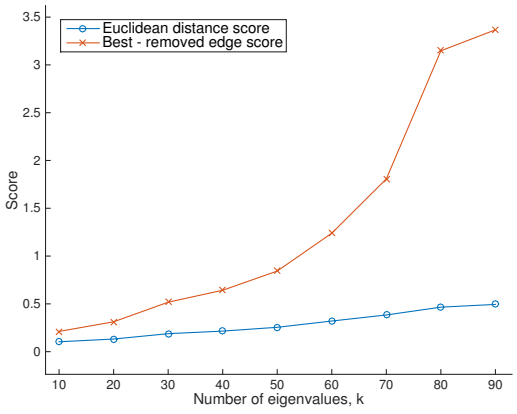
It is difficult to come to any conclusion based on the visualisation of the results, especially on the larger graphs. However, having a way to visualise the results can be helpful in some cases, especially for small graphs where natural similarities are trivial to infer for the human eye.



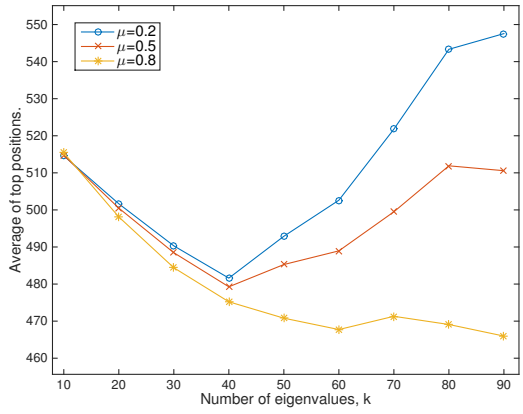
(a) Percentage of removed edges that have their end vertex in the top 5 similar vertices of their start vertex, using bpos().



(b) Percentage of test pairs that have a higher similarity than a random pair of vertices (that is not blacklisted).



(c) Test pair and relative scores. $\mu = 0.5$.



(d) Average sum of top positions.

Figure 3: Different evaluation metrics on the friends dataset using the squared Euclidean distance between cloud vectors as the similarity measure (score).

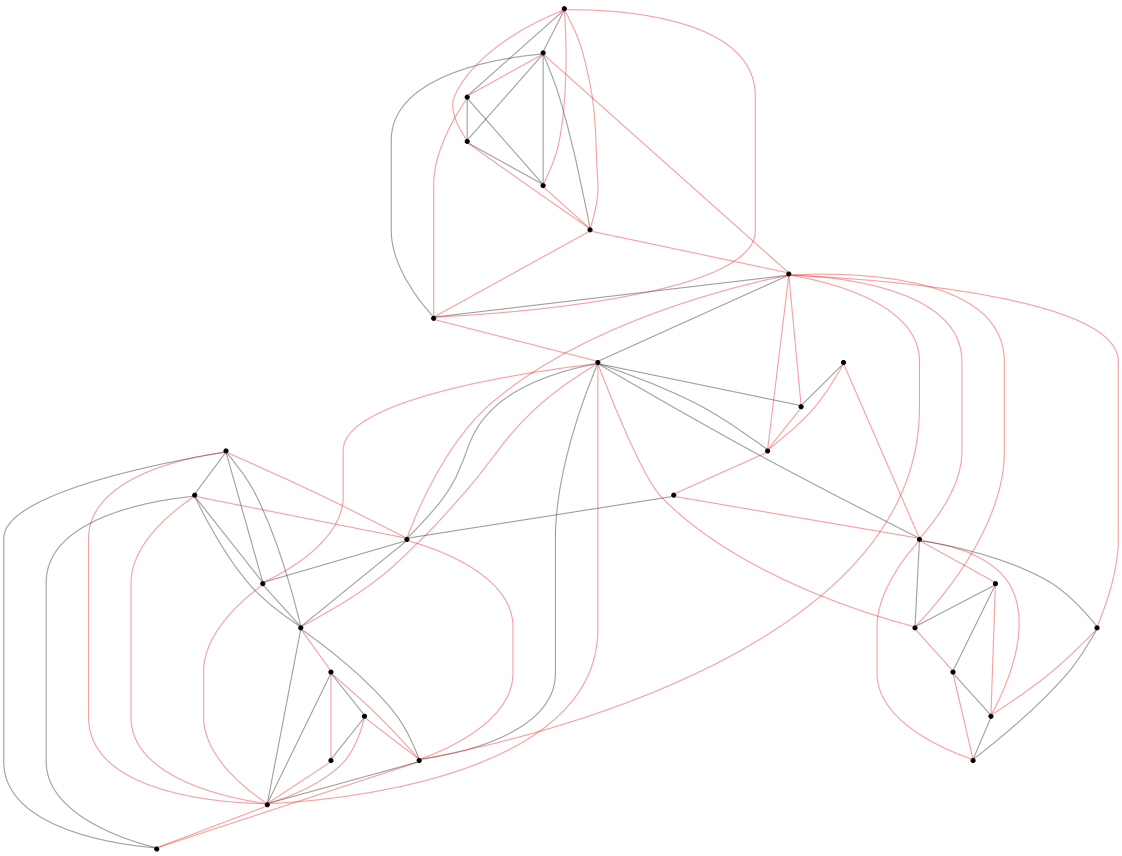


Figure 4: Top 2 similar vertices on a small undirected graph. Grey edges are real edges in the graph and red edges represent similarities (top two of one of the vertices).

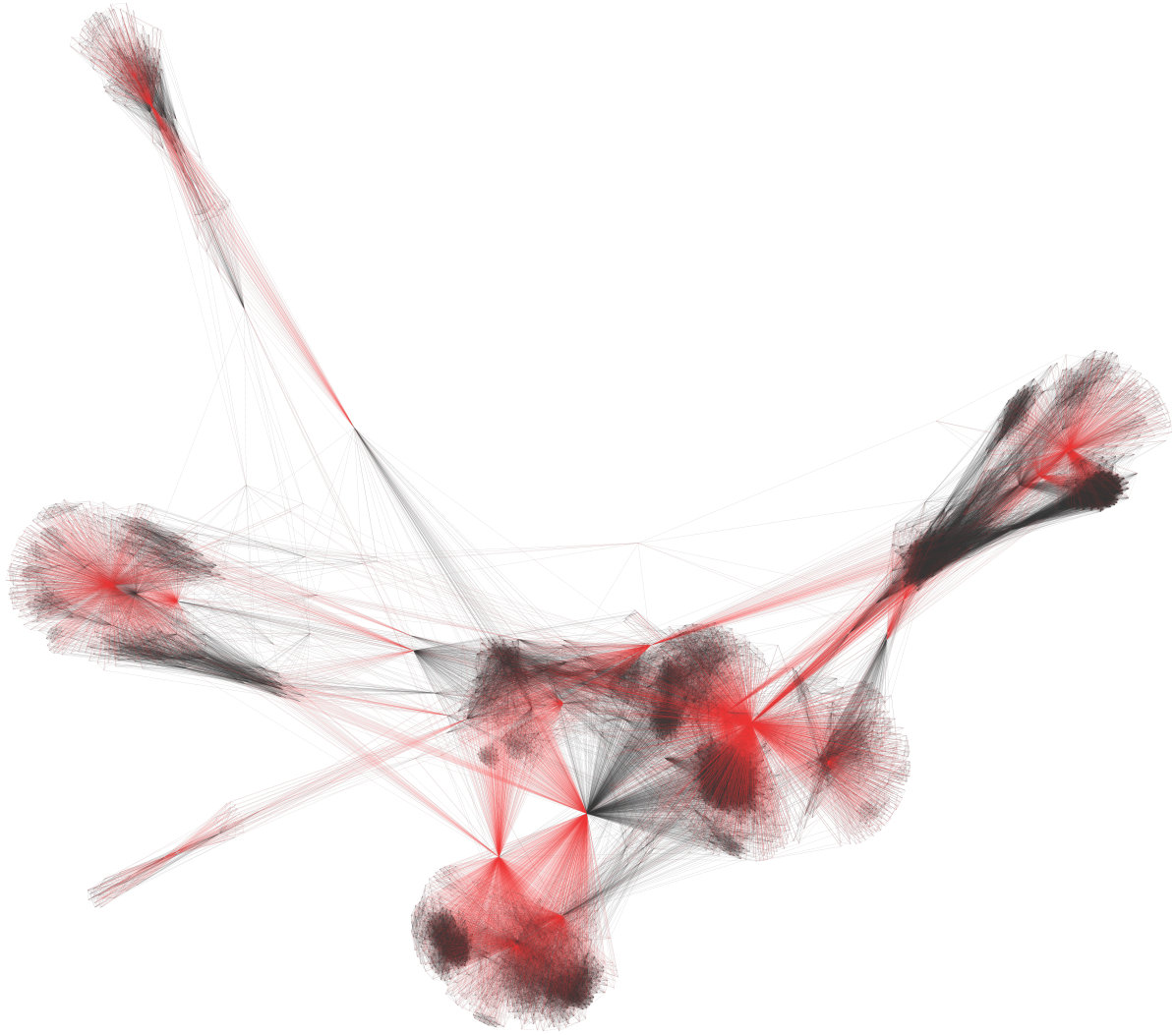


Figure 5: SNAP Facebook dataset where the grey edges represent real friendships in the graph and the red edges represent similar vertices (in top 5 for each vertex).

9 Limitations and future plans

The current implementation of this algorithm has various limitations which are discussed along with possible improvements.

The algorithm has only been evaluated on an undirected dataset of (arguably) medium size. The next immediate step is to find directed graphs on which the algorithm can be evaluated, and perform a similar evaluation. For larger graphs, the evaluation process used so far would not scale and different, more efficient methods will be required.

A comparison with similar algorithms, like the one described in [4], could be performed to see if this algorithm has advantages compared to other work.

Cosine similarity and Euclidean distance similarity were presented in this report, however only euclidean distance is well developed and evaluated. It is a trivial task to add support for the cosine similarity, and then perform a similar evaluation. After this, a comparison of the two metrics can be done.

In the real world datasets might have different types of relationships between entities. Some relationship types might be more relevant than others in finding a specific result. For instance, in a social network two people being friends might be more relevant for recommending new friends than two people following the same topic, but also using both features might be better than only one of them. The current implementation of the algorithm can compute similarities if it is given a weighted adjacency matrix but it does not have a way to automatically obtain (learn) these weights from the dataset.

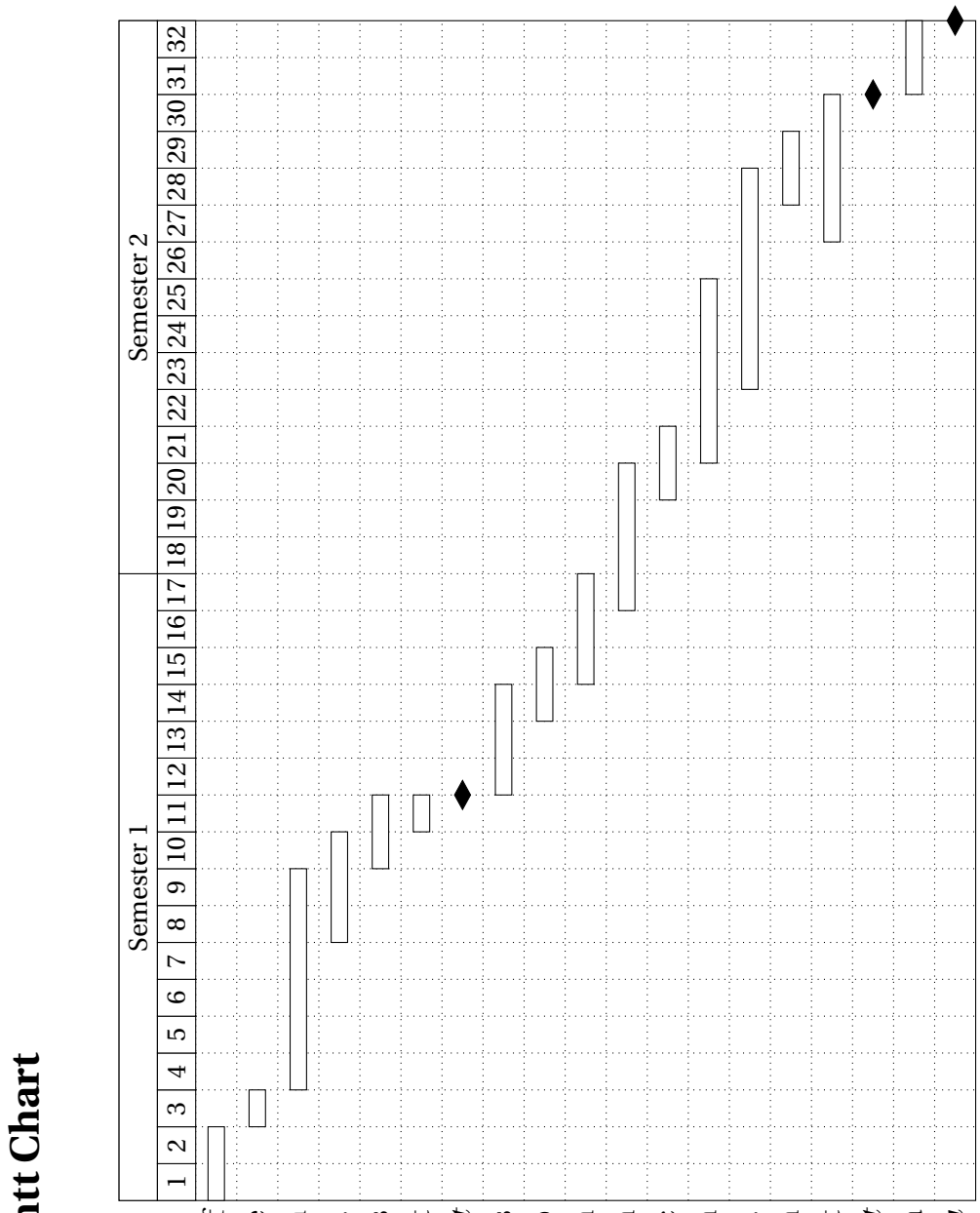
What if the dataset is too large to fit into main memory? The algorithm is currently designed to run only on one machine, but finding ways to distribute it over multiple machines will help run it on even larger datasets. One extension of the algorithm is to make it suitable for distributed computing. An immediate step could be distributing the computations of similarities and not the training, because it reduces to simply distributing the storage of a matrix or two and computing the dot product between two vectors or two vectors and a matrix.

Finding communities in graphs is not in the immediate goals of this work but it can be extended to perform such tasks. Instead of computing similarities between vertices directly, it

might be useful to try to apply different methods of clustering on the results to find a structure of the graph.

The implementation for directed graphs has the limitation of needing to check whether the eigenvalues are real or, if they are complex it needs to check if the complex conjugate has also been computed. If these conditions are not met for an eigenpair, the whole eigenpair is dropped. A (small) optimisation could be to find a way to compute the correct number of eigenpairs taking into consideration the possibility of values being complex.

Appendix A - Project Gantt Chart



Bibliography

[1] John Asmuth. go.matrix. `https://github.com/skelterjohn/go.matrix`. [Online; accessed 23 Apr. 2015].

[2] Satish Balay, Shrirang Abhyankar, Mark F. Adams, Jed Brown, Peter Brune, Kris Buschelman, Victor Eijkhout, William D. Gropp, Dinesh Kaushik, Matthew G. Knepley, Lois Curfman McInnes, Karl Rupp, Barry F. Smith, and Hong Zhang. PETSc Web page. `http://www.mcs.anl.gov/petsc`, 2014.

[3] Daniela Calvetti, L Reichel, and Danny Chris Sorensen. An implicitly restarted lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2(1):21, 1994.

[4] Francois Fouss, Alain Pirotte, J-M Renders, and Marco Saerens. Random-walk computation of similarities between nodes of a graph with application to collaborative recommendation. *Knowledge and data engineering, ieee transactions on*, 19(3):355–369, 2007.

[5] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[6] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference (SciPy2008)*, pages 11–15, Pasadena, CA USA, August 2008.

[7] Vicente Hernandez, Jose E. Roman, and Vicente Vidal. SLEPc: A scalable and flexible toolkit for the solution of eigenvalue problems. *ACM Trans. Math. Software*, 31(3):351–362, 2005. Available at `http://slepc.upv.es`.

[8] Alexis Jacomy and Guillaume Plique. Sigmajs. `http://sigmajs.org`, August 2014. [Online; accessed 23 Apr. 2015].

[9] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2015-04-23].

[10] R Daniel Kortschak and David L Adelson. bíogo: a simple high-performance bioinformatics toolkit for the go language. *bioRxiv*, 2014.

[11] Richard B Lehoucq and Danny C Sorensen. Deflation techniques for an implicitly restarted arnoldi iteration. *SIAM Journal on Matrix Analysis and Applications*, 17(4):789–821, 1996.

[12] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[13] Pascal Pons and Matthieu Latapy. Computing communities in large networks using random walks. In *Computer and Information Sciences-ISCIS 2005*, pages 284–293. Springer, 2005.

[14] Conrad Sanderson. Armadillo: An open source c++ linear algebra library for fast prototyping and computationally intensive experiments. 2010.

[15] Gilbert Strang. Eigenvalues and eigenvectors. In *Introduction to Linear Algebra*, pages 258–338. Wesley-Cambridge Press, 3rd edition, 2003.

[16] Inc. The MathWorks. Matlab 2014b documentation, largest eigenvalues and eigenvectors of matrix, Oct 2014. [Online; accessed 07 Dec. 2014].