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

Vlad Sebastian Velici
December 9, 2014

# Similar nodes in large graphs

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

A progress 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.

# 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 not computationally feasible to calculate the exact similarities between vertices of very large graphs. In this project, we investigate a method to compute an approximation of similarities between vertices and attempt to evaluate its performance (accuracy) on different datasets.

# Brute-force approach

The similarity algorithm is described regarding the undirected graph $G(v,\epsilon)$ with adjacency matrix $M$. The number of nodes is $N = |v|$ and the number of edges is $|\epsilon|$. In order to derive an algorithm to find similarities between vertices in a graph, a measure of similarity between two vertices must be defined. $M$ is the adjacency matrix of the graph and $M_{ij} = [(i,j) \in \epsilon]$, that is $M_{ij} = 1$ if there is an edge between vertices $i$ and $j$, otherwise $M_{ij} = 0$. Let $D$ be a normalised adjacency matrix such that each row sums up to 1:

$$D_{ij} = \frac{1}{\sum_{j=1}^{N} M_{ij}} \tag{1}$$

Let $\delta_\mathbf{i}$ be the $i^{th}$ column of the $N \times N$ identity matrix, and $\mathbf{C_i}$ a vector that represents how vertex $i$ is connected to the graph. In a brute-force approach, $\mathbf{C_i}$ can be computed as:

$$\mathbf{C_i} = \sum_{t=0}^{\infty} \mu^t D^t \delta_\mathbf{i} \tag{2}$$

Where $\mu \in (0,1)$ is a penalising factor. The similarity between two vertices $i$ and $j$ is the (squared) norm of the difference between $\mathbf{C_i}$ and $\mathbf{C_j}$:

$$\|\mathbf{C_i} - \mathbf{C_j}\|^2 \tag{3}$$

The brute-force approach was run on a small graph designed by hand with 50 iterations and it gave meaningful results (vertices connected in a similar way have a small $\|\mathbf{C_i} - \mathbf{C_j}\|^2$) but it does not scale for large datasets. In the next section an algorithm that scales up to large datasets is described.

## Approximative approach

Geometrically, if vectors $\mathbf{C_i}$ and $\mathbf{C_j}$ start from the origin, then $\mathbf{C_i}$, $\mathbf{C_j}$ and $\mathbf{C_i} - \mathbf{C_j}$ describe a triangle. Let $\theta$ be the angle between $\mathbf{C_i}$ and $\mathbf{C_j}$. From the Law of Cosines:

$$\|\mathbf{C_i} - \mathbf{C_j}\|^2 = \|\mathbf{C_i}\|^2 + \|\mathbf{C_j}\|^2 - 2\|\mathbf{C_i}\|\|\mathbf{C_j}\| \cos\theta \tag{4}$$

Take the dot product $\mathbf{C_i^T C_j} = \|\mathbf{C_i}\|\|\mathbf{C_j}\| \cos\theta$. The angle between $\mathbf{C_i}$ and itself is 0 and $\cos 0 = 1$, thus $\|\mathbf{C_i}\|^2 = \mathbf{C_i^T C_i}$. Now *Equation 4* becomes:

$$\|\mathbf{C_i} - \mathbf{C_j}\|^2 = \mathbf{C_i^T C_i} + \mathbf{C_j^T C_j} - 2\mathbf{C_i^T C_j} \tag{5}$$

Therefore, it is sufficient to be able to compute all dot products $\mathbf{C_i^T C_j}$ (for all $i, j \in v$) to compute the similarities between all vertices of the graph. Computing and storing all $\mathbf{C_i^T C_j}; i, j \in v$ 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 $\mathbf{C_i^T C_j}$ will be described. Let $W$ be a diagonal matrix:

$$W_{ij} = \frac{[i = j]}{\sum_{j=1}^{N} M_{ij}} \tag{6}$$

Now *Equation 1* can be written as $D = WM$, thus

$$D^t = (WM)^t \tag{7}$$

Define a new matrix $A$:

$$A = W^{1/2} M W^{1/2} \tag{8}$$

From *Equation 7* and *Equation 8*:

$$D^t = W^{1/2} A^t W^{-1/2} \tag{9}$$

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

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

$$\mathsf{v} = \begin{bmatrix} | & | & & | \\ \mathsf{v}^{(1)} & \mathsf{v}^{(2)} & \dots & \mathsf{v}^{(N)} \\ | & | & & | \end{bmatrix} \tag{10}$$

$$\Lambda = \begin{bmatrix} \lambda_1 & 0 & \dots & 0 \\ 0 & \lambda_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & & \lambda_N \end{bmatrix} \tag{11}$$

The diagonalisation of the matrix A is $A = \mathsf{v}\Lambda\mathsf{v}^{-1}$. The graph $G$ is an undirected graph which makes $A$ a real symmetric matrix, thus $\mathsf{v}^{-1} = \mathsf{v}^T$ [1]. The diagonalisation of the matrix $A$ is therefore:

$$A = \mathsf{v}\Lambda\mathsf{v}^T \tag{12}$$

$$\Rightarrow A^t = \left(\mathsf{v}\,\Lambda\,\mathsf{v}^T\right)^t = \mathsf{v}\Lambda^t\mathsf{v}^T \tag{13}$$

$$\Leftrightarrow A^t = \sum_{a=1}^{N} \lambda_a^t\, \mathsf{v}^{(a)}\, \mathsf{v}^{(a)^T} \tag{14}$$

From *Equation 9* and *Equation 14*:

$$D^t = W^{1/2} \sum_{a=1}^{N} \lambda_a^t\, \mathsf{v}^{(a)}\, \mathsf{v}^{(a)^T} W^{-1/2} \tag{15}$$

Plug $D^t$ from *Equation 15* into *Equation 2*:

$$\mathbf{C_i} = \sum_{t=0}^{\infty} \mu^t W^{1/2} \sum_{a=1}^{N} \lambda_a^t\, \mathsf{v}^{(a)}\, \mathsf{v}^{(a)^T} W^{-1/2} \delta_{\mathbf{i}} \tag{16}$$

Rearrange the equation and obtain:

$$\mathbf{C_i} = W^{1/2} \sum_{a=1}^{N} \sum_{t=0}^{\infty} \mu^t \lambda_a^t \mathrm{v}^{(a)} \mathrm{v}^{(a)^T} W^{-1/2} \delta_{\mathbf{i}} \tag{17}$$

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:

$$\sum_{t=0}^{\infty} \mu^t \lambda_a^t = \frac{1}{1-\mu\lambda_a} \tag{18}$$

Observe $\mathrm{v}^{(a)^T} W^{-1/2} \delta_{\mathbf{i}}$ is a scalar:

$$\mathrm{v}^{(a)^T} W^{-1/2} \delta_{\mathbf{i}} = \mathrm{v}_i^{(a)} W_{ii}^{-1/2} \tag{19}$$

Substitute *Equation 19* and *Equation 18* back into *Equation 17* and obtain:

$$\mathbf{C_i} = W^{1/2} \sum_{a=1}^{N} \frac{1}{1-\mu\lambda_a} \mathrm{v}^{(a)} \mathrm{v}_i^{(a)} W_{ii}^{-1/2} \tag{20}$$

An approximation of the vectors $\mathbf{C_i} (\forall i \in v)$ can be obtained by only using the $m$ leading eigenvectors and eigenvalues of $A$ instead of all of them. Compute the largest $m$ eigenvalues and eigenvectors of the matrix $A$ using the Lanczos Method for real symmetric matrices[2]. For directed graphs the Arnoldi Iteration[3] will be used instead. These algorithms perform well on large sparse matrices. In real datasets it is very often the case that the number of edges is orders of magnitude smaller than the number of vertices squared, resulting in sparse adjacency matrices and a sparse $A$. Define $\hat{\mathbf{C}}_{\mathbf{i}}$ to be an approximation of $\mathbf{C_i}$:

$$\hat{\mathbf{C}}_{\mathbf{i}} = W^{1/2} \sum_{a=1}^{m} \frac{1}{1-\mu\lambda_a} \mathrm{v}^{(a)} \mathrm{v}_i^{(a)} W_{ii}^{-1/2} \tag{21}$$

To simplify the equation, let $Z$ be a $N \times m$ matrix such that:

$$Z_{ij} = \frac{\mathrm{v}_i^{(j)} W_{ii}^{-1/2}}{1-\mu\lambda_j} \tag{22}$$

Substitute in *Equation 21* and get:

$$\hat{\mathbf{C}}_{\mathbf{i}} = W^{1/2} \sum_{a=1}^{N} Z_{ia} \mathrm{v}^{(a)} \tag{23}$$

The dot product between $\mathbf{\hat{C}_i}$ and $\mathbf{\hat{C}_j}$ becomes:

$$\mathbf{\hat{C}_i^T\hat{C}_j} = \left(W^{1/2}\sum_{a=1}^{m} Z_{ia}\vee^{(a)}\right)^T\left(W^{1/2}\sum_{a'=1}^{m} Z_{ia'}\vee^{(a')}\right) \tag{24}$$

$$\Leftrightarrow \mathbf{\hat{C}_i^T\hat{C}_j} = \sum_{a=1}^{m} Z_{ia}\vee^{(a)^T} W \sum_{a'=1}^{m} Z_{ia'}\vee^{(a')} \tag{25}$$

$$\Leftrightarrow \mathbf{\hat{C}_i^T\hat{C}_j} = \sum_{a=1}^{m}\sum_{a'=1}^{m} Z_{ia}Z_{ja'}\vee^{(a)^T} W\vee^{(a')} \tag{26}$$

Define a new $m \times m$ matrix $Q$ such that:

$$Q_{ij} = \vee^{(i)^T} W\vee^{(j)} \tag{27}$$

$\mathbf{Z_i}$ is the $i^{th}$ row of the matrix $Z$. Substitute $Q$ into *Equation 26* and obtain:

$$\mathbf{\hat{C}_i^T\hat{C}_j} = \sum_{a=1}^{m}\sum_{a'=1}^{m} Z_{ia}Z_{ja'}Q_{aa'} \tag{28}$$

$$\Leftrightarrow \mathbf{\hat{C}_i^T\hat{C}_j} = \mathbf{Z_i^T}Q\mathbf{Z_j} \tag{29}$$

From *Equation 29* it is deduced that it is only required to store the matrices $Q$ and $Z$, which only takes $O(m^2 + Nm)$ memory, instead of $O(N^2)$ to store all $\mathbf{\hat{C}_i}$ vectors.

To compute the similarity between two vertices, $i$ and $j$, use the norm formula from *Equation 29* and obtain:

$$\|\mathbf{\hat{C}_i^T} - \mathbf{\hat{C}_j}\|^2 = \mathbf{Z_i^T}Q\mathbf{Z_i} + \mathbf{Z_j^T}Q\mathbf{Z_j} - 2\mathbf{Z_i^T}Q\mathbf{Z_j} \tag{30}$$

After computing the matrices $Z$ and $Q$, the time complexity of computing the similarity between two nodes is the same as multiplying three small matrices (of sizes $1 \times m$, $m \times m$ and $m \times 1$).

## Vectorisation

It is usually more efficient to have vectorised implementations (to use matrix operations in favour of iterative solutions) in programming languages like Matlab, Octave or R. In this subsection is presented a way of computing the matrices $Z$ and $Q$ using matrix operations. For this purpose, in this subsection, $\vee$ only contains the leading $m$ eigenvectors (is of size $N \times m$), and $\Lambda$ only contains the leading $m$ eigenvalues (is of size $m \times m$).

To vectorise the computation of $Z$ from *Equation 22*, define a new diagonal matrix $R$:

$$R = \begin{bmatrix} \frac{1}{1-\mu\lambda_1} & 0 & \dots & 0 \\ 0 & \frac{1}{1-\mu\lambda_2} & \dots & 0 \\ \vdots & \vdots & \ddots & \\ 0 & 0 & \dots & \frac{1}{1-\mu\lambda_m} \end{bmatrix} \tag{31}$$

And then define $Z$ as:

$$Z = W^{-1/2} \vee R \tag{32}$$

A vectorised version of *Equation 27* (to compute $Q$) is:

$$Q = \vee^T W \vee \tag{33}$$

## Implementation

A choice had to be made on what tools and languages to use for the implementation of this algorithm. It involves linear algebra so programming languages that support such operations or have good libraries for this purpose were candidates. Matlab was chosen for this task because it has very strong linear algebra features built-in, it is easy and fast to try different experiments (can also be used interactively) and programs are usually smaller in size (lines of code) than their variants in C++ or other languages. At a later stage when the algorithm is closer to its final version, porting it to a language like Go or C++ will be considered.

Matlab has the function *eigs(A,m)* to compute the leading $m$ eigenvectors and eigenvalues[4]. Internally, it uses the software package ARPACK which implements implicitly restarted Arnoldi methods.[5]

Sparse matrices were used for the implementation of both the approximative and the brute-force approaches. A sparse matrix in Matlab only stores the non-zero values of a matrix, therefore a sparse matrix only uses memory for the positions where it holds values.[6] For instance, to store the adjacency matrix of a very large graph $G(v,\epsilon)$, using a dense matrix would take $O(|v|^2)$ memory whereas using a sparse matrix only takes $O(|\epsilon|)$ memory. Sparse matrices are faster at linear algebra operations.

The approximative approach was implemented in Matlab as a couple of Matlab functions. The function signatures along with descriptions is presented in *Table 1.* Some helper functions are not included in the table.

| | |
|---|---|
| similarity(adj, mu, m) | Computes and returns the 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(nodes, vertices, directed) | Generates a random connected graph. |

**Table 1:** Implementation interface. Parameters: q represents the matrix $Q$, z represents the matrix $Z$, adj represents the adjacency matrix, and i and j represent vertex numbers.

The brute-force algorithm was implemented as a separate set of Matlab functions with a very similar interface.

## Evaluation

### Comparison of approximative and brute-force approaches

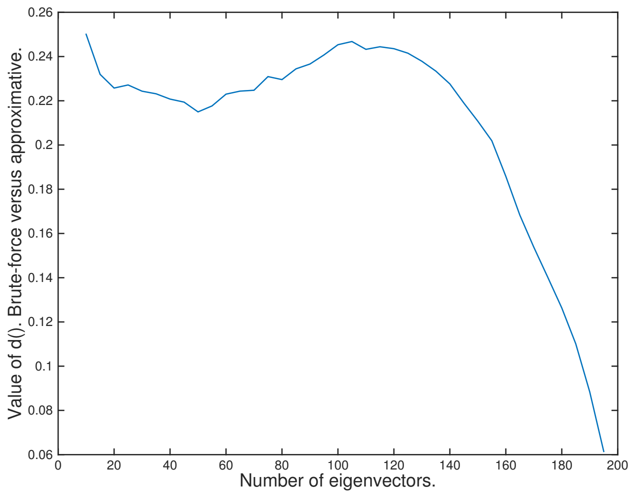
A simple evaluation metric of how well the approximative approach retains the most important information from the graph is to evaluate how different are the top similar vertices for the brute-force approach and the approximative approach with different number of eigenvalues and eigenvectors used.

For a dataset of $N$ vertices, let $B_{ij}^{m}$ be the position of vertex $j$ in the top similar vertices of vertex $i$ for the brute-force approach with $m$ iterations. Similarly let $T_{ij}^{m}$ be the position of vertex $j$ in the top similar vertices of vertex $i$ for the approximative approach with $m$ eigenvectors. Define the difference between two such tops to be:

$$d(B^{m}, T^{m'}) = \frac{1}{N^3} \sum_{i=1}^{N} \sum_{j=1}^{N} \mathrm{abs}(B_{ij}^{m} - T_{ij}^{m'}) \tag{34}$$

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 connected graph with 200 vertices



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

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

**(b)** Table of some of the values.

**Figure 1:** Difference of results between brute-force and approximative approaches.

and 350 edges, between the brute-force approach with $m = 20$ iterations and the approximative approach with $m' = 10$ to 199 eigenvectors. The choice of the penalising factor $\mu = 0.5$ was used for all the runs. The results can be observed in *Figure 1*. We can observe 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 $m'$ in the range of 20 to 40, where 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 connected 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.

Although the evaluation metric described above might be useful to compare the two approaches directly, it does not tell anything about how well the algorithm achieves its goal of finding similar entities in a real dataset. We will now examine ways to evaluate the performance of the algorithm on real problems.

## Visualisation

A subjective method of evaluation on real data 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.

## Discussion of further evaluation

None of the evaluation methods described above give an objective, efficient and automatic evaluation metric. In this subsection we discuss a few possible evaluation methods of the algorithm.

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.

Given some datasets that fit the above requirements, we can run our algorithm and compare the results with the reference (natural) similarities. In reality, datasets have many features and natural *similarities* between entities depend on many features, thus this type of evaluation is not practical.

For specific tasks we can evaluate the performance of our algorithm by adding or removing some edges from the graph. For instance, if we want to find duplicates we can create a dummy vertex by duplicating an existing one and randomly removing some of its edges. We then evaluate whether the dummy vertex is very similar to the original vertex.

Implementing the evaluation method described above is included in the future plans of this project, along with investigating more methods of evaluation.

## Limitations of current implementation

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

Directed graphs are not currently supported. In practice, datasets have meaningful unidirectional relationships (e.g. in a social network person A follows person B, but B does not follow A), and often datasets are represented as directed graphs rather than undirected graphs. The algorithm can be adapted to support both directed and undirected graphs but it will have the disadvantage of requiring to compute $V^{-1}$ (for undirected graphs, $V^{-1} = V^T$).

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. The current implementation of the algorithm can compute similarities if it is given a weighted (symmetric) 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.

## Future plans

The highest priority improvements are: develop a good evaluation method, add support for directed graphs, and use different edge weights for different types of relationships. A detailed plan can be observed in the Gantt Chart in *Figure 2*.

An evaluation method is not trivial to design and build for this algorithm, but it is crucial to the validation, further development and fine-tuning of the algorithm. Researching an evaluation model has the highest priority in the list of tasks.

Plenty of datasets on which this algorithm can be applied describe directed graphs with multiple types of relationships. The relevant tasks are the next tasks in the Gantt Chart.

A demo on one or more datasets will be built. The first step into getting feedback and evaluating how the algorithm performs on real data is to build an interface where users can interact with the results.

A small research on how to make the algorithm run on multiple machines will be done. Also, if the time allows, the code will be ported to a language like Go or C++ and released as an open-source project.

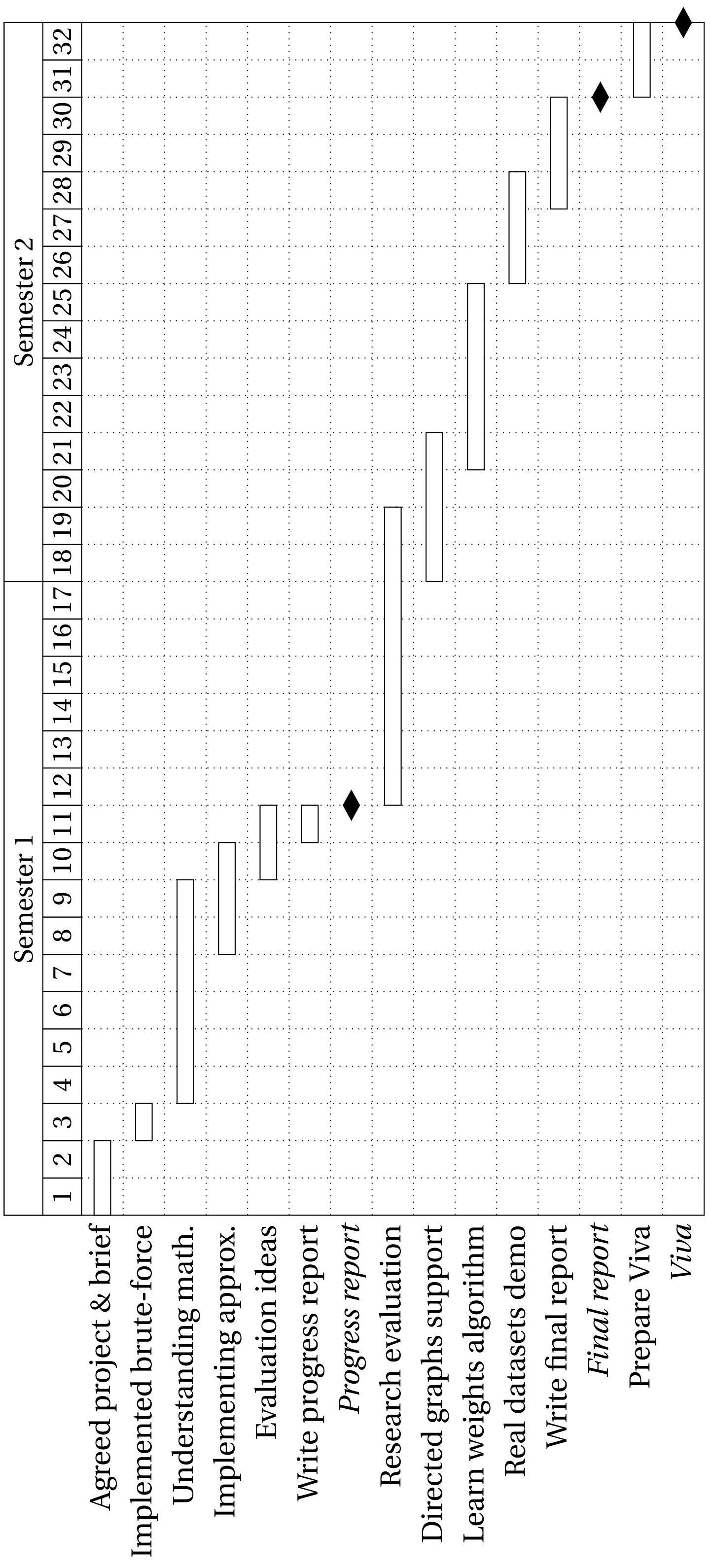


**Figure 2:** Gantt Chart of done and planned tasks over weeks.

## References

[1]: Strang, Gilbert. Eigenvalues and Eigenvectors (p 258-338). "Introduction to linear algebra." Cambridge Publication (2003).

[2]: Calvetti, Daniela, L. Reichel, and Danny Chris Sorensen. "An implicitly restarted Lanczos method for large symmetric eigenvalue problems." Electronic Transactions on Numerical Analysis 2.1 (1994): 21.

[3]: Lehoucq, Richard B., and Danny C. Sorensen. "Deflation techniques for an implicitly restarted Arnoldi iteration." SIAM Journal on Matrix Analysis and Applications 17.4 (1996): 789-821.

[4]: "Matlab 2014b Documentation." Largest Eigenvalues and Eigenvectors of Matrix. The MathWorks, Inc., 3 Oct. 2014. Web. Accessed 07 Dec. 2014. <http://uk.mathworks.com/help/matlab/ref/eigs.html>.

[5]: Lehoucq, Richard B., Danny C. Sorensen, and Chao Yang. ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods. Vol. 6. Siam, 1998.

[6]: Gilbert, John R., Cleve Moler, and Robert Schreiber. "Sparse matrices in MATLAB: design and implementation." SIAM Journal on Matrix Analysis and Applications 13.1 (1992): 333-356.