DEPARTMENT OF COMPUTER SCIENCE

Election Manipulation Data Graphical Analysis

Andreas Hadjiantoni

A dissertation submitted to the University of Bristol in accordance with the requirements of the degree of Bachelor of Science in the Faculty of Engineering.

Monday 13th May, 2019

Declaration

This dissertation is submitted to the University of Bristol in accordance with the requirements of the degree of BSc in the Faculty of Engineering. It has not been submitted for any other degree or diploma of any examining body. Except where specifically acknowledged, it is all the work of the Author.

Andreas Hadjiantoni, Monday 13th May, 2019

Acknowledgements

I would like to thank Miranda and Luis for their guidance. I would also like to thank my parents for their continuous support.

Abstract

It is a commonly known fact that fake political news are non-desirable by social media users. In this project, we developed a software product that analyzes twitter fake news data, and used it to analyze a particular data set. The data that was studied was released by twitter, and it involves tweets and accounts associated to them. These tweets originate from Iran. It involves fake news from both left and right wing users, and their purpose is to drift the result of the 2016 US presidential elections towards their political views. The particular dataset is of great interest, since not much research has been done on it, unlike other datasets released by twitter, that belong to the same context of Fake news.

The developed software product was used to perform a graph theoretic analysis on a graph that can be built from the data set. Indeed, some interesting results have been identified for the particular dataset.

Also some graph-theoretic algorithms had to be adapted, so that they fit our use case. These adapted algorithms are also presented in this project.

Contents

Chapter 1

Introduction

1.1 The Problem

One of the important advancements of the 19th century was the relatively cheap newsprint process [5]. Even though this advancement allowed the cheap distribution of news on a large scale, it has also been exploited by some "malicious" members of society to distribute fake news to the masses, as it provided no filtering mechanism to check the validity of what was to be published. On top of this, exploits of the same nature, but with different underlying technology have risen in the 20th century, namely the use of radio to promote Nazi propaganda and the use of television by JFK to build his election campaign [5].

In the 21st century, news are also distributed by social media. This has of course given rise to another way of distribution of fake news as well. As verified by *Vosoughi et al.* the use of social media to distribute fake news has the characteristic that the victims are not only passive consumers of the news, but they also have the potential to help the propagation of it by sharing the news to their connections [19].

The propagation of fake news in social media when seen from a more abstract and mathematical perspective becomes a graph. Even though this abstraction is one of the most obvious, not much research has been done in this context. This is counter-intuitive when the previous work on (algorithmic) graph theory is considered, as it offers a diverse range of algorithms and principles that can be applied in the context of fake news.

1.2 Aims, Objectives and Value Added

There are three main aims and objectives for this project. First, we will build a software product that can use fake news data released by Twitter, interpret it as a graph, and derive graphical properties from it. Secondly, we will adapt some existing graph-theoretic algorithms and definitions, effectively suggesting new ones that can fit better some use cases, including the needs of our software product. Thirdly, we will use the developed software product to test a number of hypothesis on the dataset released, that contains users and accounts from Iran. The hypotheses are as follows:

1. There exist "fake news clusters" in the dataset. People are most likely to retweet from the same account(s), and hence social circles are likely to form in the network.

2. There exist accounts of high importance in the dataset. The clusters mentioned in hypothesis one are likely to form around those nodes. Also, high importance accounts are more likely to tweet first rather than retweet from another account.

3. When fake news are propagated within a cluster, there exists a small subset of accounts that is responsible for their propagation to other clusters. This is based on the intuition that social circles usually have members that act as "connecting links" between the two circles. i.e. members that belong to both clusters and may transfer information from one circle to the other.

Producing software to verify the above for the given data set will assume the identification of entities that witness the validity of the hypotheses. In a practical setting, after a data set is partitioned to fake and non-fake news, it would then be sensible to consider interpreting the fake news partition which will initially consist of raw data. For instance, it could be desirable to find the accounts that are most likely to publish fake news in the future. Also, the influence of certain accounts on the social media graph can be predicted, by considering as a prior the success of previous campaigns and past fake news they have released.

Chapter 2

Literature Survey

2.1 What is Fake News?

Given that the data set that will be studied consists entirely of fake news, we will not be concerned about differentiating fake from non fake news. However, it is still worth defining the term for completeness. Even though the definitions vary, this is not to say that they contradict each other.

For example, *Lazer et al.* defines fake news as "fabricated information that mimics news media content in form but not in organizational process or intent." Then, it is mentioned that fake news can be classified as misinformation, or disinformation. The former indicates false information which is spread by someone who is not aware of its falsity, whilst the latter is spread by someone who knows it is factually incorrect, but they purposely spread it to deceive people[8]. In the specific domain of this project, and as mentioned by Campan et al. [5] the accounts described in the second hypothesis in section 1.2, are those who aim to disinform people, whilst other accounts who propagate fake news are those who misinform people.

2.2 Existing fake news analyses

2.2.1 Social Media Users/Messages Classification

The applications of techniques studied in this project partly rely on the social media users/messages being already classified to fake and non-fake news. Hence, it is sensible to ask if this classification is something achievable, based on previous work in the matter. Indeed, a lot of literature exists that presents classification techniques.

Wu et al. presents TraceMiner, a generic classification technique for social media messages, based on the network structure, rather than on the content information [14]. In particular, embeddings of users are learned by considering the social status of each user (e.g. friendships and community memberships). Using the embeddings of users, the messages are represented as a sequence of its spreaders, and they are modeled using Recurrent Neural Networks (RNNs). The classification is then achieved using a softmax function. In terms of performance, TraceMiner achieves a 91.24% accuracy in fake news detection. In comparison to mainstream classification techniques, like SVMs, TraceMiner significantly outperforms them when little training data is available, and the margin is reduced when larger training sets are used.

2.2.2 Fake News Propagation and Spread

In a work by *S. Vosoughi et al.* it was verified by experimenting on a dataset that falsehood diffuses and propagates significantly further, faster, and deeper than truth, in all categories of information [19]. It was also discovered that the structural elements of the network (e.g. number of followers of the user who first releases a fake news tweet) is not related to the potential of that tweet becoming viral, even though no explanation was given for why this was the case. Furthermore, the high diffusion of falsity was justified by its novelty. In particular it was suggested that novelty is more likely to induce stronger emotions, and thus increases the probability of an individual to retweet.

In an attempt to quantify the probability of further propagating a piece of information in a social network, *J. Sun et al.* presents a number of Diffusion Influence Models [13]. The maximisation of said probability depends on the set of nodes that an attacker will choose to first publish the information they wish. We present one out of five models here, namely *Linear Threshold Model*. First, each node in the social network is either active or inactive (representing that they were affected or not affected by some campaign respectively). Initially, all nodes are inactive. Activation of some node v depends on a function f_v which maps a set of neighbours of v which are all active in time $t-1$ to some value in $[0,1]$. v is then activated if $f_v(S) > \theta_v$, where θ_v is a threshold value and $S.$ is a subset of all active neighbours of v. It is worth mentioning that these influence maximization models use the structural properties of the network, which is contradictory to the claim of *S. Vosoughi et al.* [19].

2.3 Techniques and Algorithms for hypothesis testing

We now define the mathematical structure constructed from the data set: Graph $G = (V, E)$ is constructed such that V is the set of nodes, where each twitter account is represented by some $v \in V$. E is the set of edges, where $(u, v) \in E$, iff v has retweeted some tweet published by u. The weight of each edge increases with the number of retweets occurred between the respective pair of accounts. In particular, it will be based on a simplified version of the action model defined in [16]. The action model is based on the intuition that each user has some "action potential" which is distributed to accounts which has retweeted from, replied to some of their tweets, or mentioned them somewhere. The number of retweets, replies and mentions, determines the weight of the edge. The version we will be using only concerns the "retweets" factor. Formally, the weight function $\omega : E \to \mathbb{R}$ is defined by:

$$\forall(u,\ v) \in E :\ \omega((u,v)) = \frac{\text{Number of retweets of } u \text{ from } v}{\text{Total retweets of } u} \tag{2.1}$$

The reason for dividing with the total retweets of u is to ensure that the sum of weights of all outgoing edges of some node equals to one.

A number of algorithms from existing algorithmic graph theory literature will be used to test the mentioned hypotheses. Before presenting those algorithms, we formalize the hypotheses mentioned in a graph-theoretic context. Each of the following statements corresponds to one of the hypotheses in the respective order mentioned in chapter 1.

1. There exist clusters of nodes in G. Two subsets of nodes within a cluster have high connectivity compared to one of two subsets not belonging to the same cluster.

2. There exist nodes of high graphical importance in the set of nodes V. These accounts are the ones that most likely to have clusters formed around them. They are also likely to have a higher disinformation/misinformation ratio.

3. Per two clusters, there exists a small subset of nodes that is responsible for the two clusters being connected.

A discussion of possible algorithms that can be used for the testing of the above hypotheses follows:

2.3.1 Graphical Importance of Nodes

PageRank. PageRank is one of the algorithms that assigns an importance value to every node in a graph. It encapsulates the idea that heavily linked nodes and nodes that are linked by heavily linked nodes receive a high PageRank value[4]. Formally:

$$R(u) = c \sum_{v \in B_u} \frac{R(v)}{N_v} + \frac{1-c}{N}, c < 1 \tag{2.2}$$

where $R(u)$ is the PageRank value of node u, B_u is the set of nodes that point to u, N_v is the number of nodes to which v points, N is the total number of nodes and c is a scalar. The idea in equation 2.2 is that the PageRank value is recursively computed such that each node v in each iteration of the algorithm evenly distributes its current PageRank value to all of the nodes that it points to. Similarly, the PageRank value of some node u is computed by summing all of the values received from all the nodes that point to it. Furthermore, when the total PageRank of the graph sums to one, the PageRank value of each node can be interpreted as the probability that a surfer who traverses the graph randomly will end up on that node, after enough steps. It is also possible that some nodes do not have any outgoing edges. These nodes will act as sinks and they will absorb all of the PageRank of the graph. To avoid this, the damping factor c is introduced, which decreases the "PageRank accumulation rate" of sinks. Additionally, notice that the computation of the PageRanks for all the nodes forms a Markov Chain, which will converge to the sinks sharing the PageRank. To avoid this, the second term of the RHS of equation (2.1) is introduced, which places a "prior probability" that the random surfer will jump to an arbitrary node.

2.3.2 Graph Clustering

An algorithm that partitions the graph to clusters is also needed, so that the first hypothesis is tested. As *E. Schaeffer* suggests, there is a number of ways to define closely related nodes [18]. All of the ways fall in one of two categories, namely Vertex Similarity and Cluster Fitness Measures. In the former case, a distance function is defined, which takes two nodes and returns a value representing how similar the two nodes are. In the latter case, functions that rate the quality of a cluster are used.

The algorithm that will be used is proposed by *G. W. Flake et al.* and it assumes the maximum-flow value of the graph can be computed [11]. Maximum-flow is discussed in the next subsection. The fitness of a given cluster is measured using a quantity defined

in the mentioned work, namely its expansion. Expansion falls in the latter of the two categories mentioned above, and it intuitively returns the per node weight of a cut of a given (sub)graph.

Flexibility of the algorithm. In contrast to other graph-clustering algorithms, this algorithm especially fits the needs of this project. In particular, it can be tuned by adjusting the α value, to control the trade-off between cluster intraconnectivity and cluster interconnectivity. This is useful in our case, as we have no prior-belief regarding how dense the clusters will be between them and within them.

Computational Complexity. Most parts of the algorithm run in $O(n)$ time, where $n = |V|$. The only exception is the computation of the minimum-cut tree for the input graph. Even though it could be naively computed using $O(n^2)$ calls to a maximum-flow subroutine, *Gomory et al.* proves that it can be computed in $O(n)$ such calls by using a Gomory-Hu tree[17]. Hence, the overall time complexity is $O(n \times$Time Complexity of Max-Flow subrout Furthermore, computation of such a Gomory-Hu tree is parallelizable.

2.3.3 Betweenness Centrality

The third hypothesis claims that there exist nodes that act as "bridges" between clusters. Mathematically, this can be formalized by stating that those nodes have a high betweenness centrality. Betweenness centrality was defined by *L. C. Freeman*[10] and it quantifies the intuition that different nodes in a graph have different importance when it comes to connecting different parts of the graph together. For a particular node, it can be thought as the "impact" the removal of that node will have to the connectivity of the graph. Figure 2.1 gives a graphical intuition for an unweighted graph. Freeman defined betweenness centrality for a node v as:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2.3}$$

where σ_{st} and $\sigma_{st}(v)$ is the number of shortest paths from node s to node t and the number of shortest paths from s to t that pass through v respectively. In order to keep the above result in the range $[0,\ 1]$, it is normalized by the number of node pairs in the graph. This is $(n-1)(n-2)$ and $(n-1)(n-2)/2$ for directed and for undirected graphs respectively.

Computing Betweenness Centrality In a work by *U. Brandes*[7], an algorithm for fast computation of betweenness centrality is proposed. In particular, the "pair dependency" of a pair of vertices $s,\ t$ on some vertex v is defined as:

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{2.4}$$

In order to remove the need for the second vertex t, but to take into account the whole graph, the dependency of a vertex s on a single vertex V is defined as:

$$\delta_{s\bullet}(v) = \sum_{t \in V} \delta_{st}(v) \tag{2.5}$$

The Betweenness centrality therefore becomes:

$$\begin{aligned} C_B(v) &= \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \\ &= \sum_{s \neq v \in V} \sum t \neq v \in V \delta_{st}(v) \\ &= \sum_{s \neq v \in V} \delta_{s\bullet}(v) \\ &= \sum_{w : v \in P_s(w)} \frac{\sigma_{st}(v)}{\sigma_{st}} (1 + \delta_{s\bullet}(w)) \end{aligned} \tag{2.6}$$

where $P_s(w)$ is the set of ancestors of w in the shortest path from s to w. In equation 2.6 the last equality holds by Theorem 6 of the same paper. The above result can be exploited for computing the betweenness centrality of every vertex in the graph by solving one single-source shortest-paths problem for each $s \in V$. In each iteration, the betweenness centrality of each vertex is increased by adding the centrality that is due to s.

Computational Complexity Given that Breadth first search or Dijkstra's algorithm are used depending on whether the graph is weighted or unweighted respectively, the time complexities are $O(nm)$ and $O(nm + n^2 log\ n)$. Furthermore, since single-source shortest paths algorithms are executed for each vertex independently, and results from previous executions of the single-source shortest paths algorithm are not required, the algorithm can be parallelized.

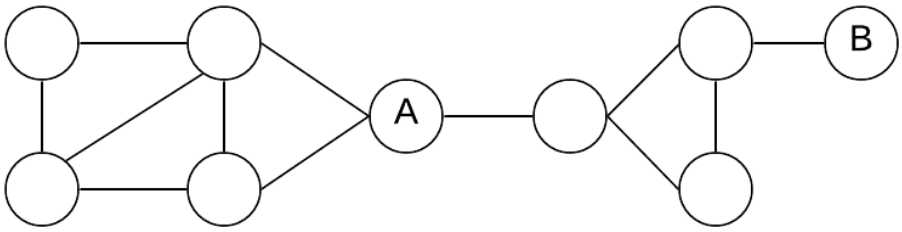

Figure 2.1: In the above graph, A has the highest betweenness centrality, as removing it renders the graph disconnected. B has the lowest.

2.3.4 Maximum-Flow

As discussed above and in the previous subsection, a maximum-flow algorithm needs to be defined. *Edmonds and Karp*[12] defines a concrete maximum-flow algorithm based on the *Ford and Fulkerson* method[15] which runs in $O(VE^2)$. Furthermore, *Dinic* defines an algorithm which runs in $O(V^2E)$[9]. Discussing which of those algorithms parallelizes better is not sensible in this case, since the max-flow algorithm will be used as a subroutine by other algorithms further up the call hierarchy. Hence, and all of the processing units will be used by those algorithms.

Chapter 3

Requirements and design

We now start designing the structure of the software that will be used for testing the hypotheses mentioned. The software will need to satisfy a number of technical requirements. It will not be used solely for the evaluation of the above hypotheses on the particular dataset, but it should also be in a state in which it can be used on other datasets of the same structure. Hence, those requirements will be defined as if the project was being developed using the waterfall model. We will therefore assume that there exists an artificial client that wants to make use of such a software on a regular basis.

3.1 User Requirements

Since we are assuming an "artificial client" for who the software is being developed, it makes sense to define the technical background of this client. Based on this, the Software Requirements will be defined in the next subsection.

It is assummed that the user is familiar with command line interfaces, and they can adapt quickly to a new command line interface of limited scope. Furthermore, it is assumed that the user is familiar with the algorithms that will be used for the evaluation of the hypotheses with respect to what each algorithm computes in the graph and how it can be parametrized. This implies that the user is also be familiar with basic graph-theoretic concepts.

3.1.1 Software Requirements

Functional requirements

Data Input The software will need to be able to locate and load two different datasets that will be stored locally as csv files. The two files will contain account information, and the tweets of those accounts respectively. The data in each file will follow the RFC4180 file format. The columns of each file are shown in table 3.1

Requiring the input data to have the exact same format as described above may seem limiting with respect to the scope in which the produced software can be used. However, this is not the case. In particular, similar data sets released by twitter before and after the one that will be studied follow the same format with no exception. Even though this reasoning is not a guarantee, it is clearly a strong indication that datasets released in the future will be eligible to be analyzed by the software.

ACCOUNTS:		TWEETS:	
Field Name:	**Type:**	**Field Name:**	**Type:**
userID	Integer	tweetID	Integer
userDisplayName	String	tweetLanguage	String
userScreenName	String	tweetText	String
userReportedLocation	String	tweetTime	Time
userProfileDescription	String	tweetClientName	String
userProfileUrl	String	inReplyToTweetID	Integer
followerCount	Integer	inReplyToUserID	Integer
followingCount	Integer	quotedTweetTweetid	Integer
accountCreationDate	Date	isRetweet	Boolean
accountLanguage	String	retweetUserID	Integer
		retweetTweetID	Integer
		latitude	Float
		Longitude	Float
		quoteCount	Integer
		ReplyCount	Integer
		LikeCount	Integer
		RetweetCount	Integer
		Hashtags	List<String>
		urls	List<String>
		userMentions	List<Integer>
		pollChoices	List<String>
		userID	Integer

Table 3.1: The fields of the "Accounts" and "tweets" files with their respective data types. The last field of the tweets dataset(userID) contains the ID of the user that tweeted it, and it will be used to link the two datasets.

Hypotheses evaluation The software will need to 'link' the tweets with their respective tweeters and evaluate the hypotheses described in chapter 2. This will be done by deriving results that prove or disprove that each hypothesis holds true.

Parametrization The algorithms that will be used to compute results that evaluate the hypotheses can be parametrized as discussed in chapter 2. For example, the rate of convergence of the PageRank algorithm[4] can be tweaked by changing the damping factor. Those parameters will be a secondary input to the program and the user will be able to change them.

Robustness The software will need to assume as little as possible for the correctness of the input given to it. This includes input data that is not in the expected format, unexpected command input order, erroneous commands and erroneous parameters. In particular, each of the above needs to be handled in a sensible manner so that crashing and/or wrong computations are avoided, and return an explanatory error message to the user.

Scalability The software will need to be as efficient as possible with respect to loading of the data and the hypotheses evaluation. In particular, it needs to support graphs with

node set size up to 2000 nodes, and node to edges ratio up to 2/5. The density will of course increase as the node set decreases in size.

This can be controlled by the choice and design of algorithms, the programming language, and proper implementation techniques that will allow the compiler to eagerly perform optimizations on the code.

User interface requirements

Visualization The software will need to provide a graphical representation of the constructed graph that is used for the evaluation of the hypotheses.

User Interface The software will take input and give output from/to the user using a command line interface. The visualization described in the previous requirement will be the only exception, where a graphical user interface will be used.

Maintainability requirements

Expandability. The software needs to be easily expandable with respect to adding extra functionality, and changing existing functionality.

3.2 Algorithm Design

In this section we will discuss some of the algorithms that will be used in the implementation. All of the algorithms are derived from the ones presented in section 2.3. Given the scalability requirement specified in the previous section, we need to make a decision for each algorithm to use. We use the computational complexity of each algorithm as the most important metric for assessing the efficiency of the algorithm. Hence, constant factors imposed by implementations of algorithms will be used as a "second class" metric. The same holds for parallelizability of algorithms, which eventually does not provide any asymptotic speedup, but boils down to an improvement within a constant factor.

3.2.1 Graphical Importance of Nodes

Adaptation of PageRank

As discussed in the respective sub-section of section 2.3 the pageRank algorithm will be used. However, in our case the graph will be weighted, as defined in section 2.3. Hence, in an arbitrary iteration of the algorithm each vertex must not distribute its current pageRank value equally to each vertex pointed to by its outgoing edges. Instead the amount of pageRank to be distributed to each vertex will be proportional to the weight of the edge pointing to it. We still need to ensure that the total pageRank value distributed by some vertex to the target vertices of its outgoing edges equals to the pageRank value that vertex had in the previous iteration, just like in the vanilla version of the algorithm. This invariant allows us to infer a normalization coefficient for each vertex, which needs to scale the weight of its outgoing edges to infer the amount of pageRank value "transferred" to each neighbouring vertex. If we consider an arbitrary vertex u, with outgoing edge set O_u, pageRank value in the i^{th} iteration $Pr^i(u)$, normalization coefficient λ_u and pageRank value transferred through the edge $(u,\ v)$ in the i^{th} iteration $Pr^i(u,\ v)$, we have:

$$\begin{aligned} Pr^i(u,\ v) &\propto\ \omega(u,\ v)\ \times\ Pr^i(u) \\ Pr^i(u,\ v) &= \lambda_u\ \times\ \omega(u,\ v)\ \times\ Pr^i(u) \end{aligned} \tag{3.1}$$

Summing over all outgoing edges of u:

$$\sum_{e \in O_u} Pr^i(e) = \lambda_u \times \sum_{e \in O_u} \omega(e) \times Pr^i(u) \tag{3.2}$$

From the invariant we have:

$$\sum_{e \in O_u} Pr^i(e) = Pr^i(u) \tag{3.3}$$

Substituting 3.3 into 3.2:

$$\begin{aligned} \cancel{Pr^i(u)} &= \lambda_u \times \sum_{e \in O_u} \omega(e) \times \cancel{Pr^i(u)} \\ \lambda_u &= \frac{1}{\sum_{e \in O_u} \omega(e)} \end{aligned} \tag{3.4}$$

By specializing the above for our graph defined in section 2.3, and by equation 2.1, the sums of weights of all outgoing edges of some vertex u will be:

$$\begin{aligned} \sum_{(u,\ v) \in O_u} \omega((u,\ v)) &= \sum_{(u,\ v) \in O_u} \frac{\text{Number of retweets of } u \text{ from } v}{\text{Total retweets of } u} \\ &= \frac{\text{Total retweets of } u}{\text{Total retweets of } u} \\ &= 1 \end{aligned} \tag{3.5}$$

Therefore $\lambda_u = 1/1 = 1$ for every vertex in the graph. We now adapt the definition of pageRank based on the above result and equation 2.2 as follows:

$$Pr^i(v) = c \sum_{u \in B_v} [Pr^{i-1}(v) \times \lambda_u \times \omega((u,\ v))] + \frac{1-c}{N}, c < 1 \tag{3.6}$$

Which in our case reduces to:

$$Pr^i(v) = c \sum_{u \in B_v} [Pr^{i-1}(v) \times \omega((u,\ v))] + \frac{1-c}{N}, c < 1 \tag{3.7}$$

The above equation will therefore be used in every iteration of the algorithm, for the non-uniform distribution of the pageRank value of every vertex to the destination vertices of its outgoing edges.

Correlation based metrics

The second hypothesis states that "high importance accounts are more likely to tweet first rather than retweet from another account." To develop an algorithm that evaluates this part of the hypothesis, we first define the metric disinfo-per-misinfo for a particular account u:

Definition 1

$$\textit{disinfo-per-misinfo}(u) = \frac{\textit{Disinformation}(u)}{1 + \textit{Misinformation}(u)} \tag{3.8}$$

Given that Disinformation(u) is the number of fake news tweets u has originally generated, and Misinformation(u) is the number of fake news tweets u has retweeted from some other account, the above definition quantifies how many times more likely is it for some account to generate a new tweet rather than retweet, based solely on its tweet-to-retweet ratio so far. The 1+ term in the denominator is used to avoid division by zero. This part of the hypothesis can then be evaluated by evaluating the Pearson's correlation coefficient of two different distributions, each having a size of $|V|$. The i^{th} element of each distribution will correspond to the same account. We define the two distributions as follows:

1. **Distribution 1:** Contains the pageRank values of the accounts in increasing order.

2. **Distribution 2:** Contains the disinfo-per-misinfo metric for every user account, sorted in increasing order of the respective pageRank value of the account.

We can therefore quantify whether and how much the likelihood of tweeting original Fake news to retweeting Fake news increases as the graphical importance of the account increases by computing the correlation coefficient of the above distributions.

Even though it is not required by our hypotheses, it is interesting to see whether and how much the importance of nodes increases as they get more followers. Since some papers use the number of followers of the accounts as the metric to quantify their importance, answering this question will quantify "how close" their metric is to the weighted pageRank metric. We can compute this by using the same first distribution as above, and using the number of followers of each user as the second distribution.

3.2.2 Graph Clustering

Clustering on undirected graphs

To compute clusters in the graph, the clustering algorithm discussed in section 2.3 will be used. Since we are interested in clusters of user accounts that have interacted a lot with each other irrespective of whether they have tweeted from and have been retweeted by other accounts, we will remove the direction of the edges of the graph. Doing so will require some precomputation as described in [11]. In particular, to convert a directed graph to undirected so that the clustering algorithm described still applies, outgoing edge normalization (so that their total weight sums to one), removal of edge direction and combination of parallel edges by summing their weights are required.

Graphical Importance and formation of clusters.

The last part of the second hypothesis states: "The accounts of high graphical importance are the ones that most likely to have clusters formed around them". Therefore we need to compute whether and how much those accounts are responsible for the formation clusters in the network. This assumes that the previous part of hypothesis two and hypothesis one are indeed true, i.e. there exist clusters of high connectivity in the graph, and the distribution of pageRank values is not close to uniform for the node set. Given these assumptions, we can evaluate this part of the hypothesis by computing the correlation coefficient between two distributions which contain both contain elements from the following set, but in different order:

$S = \{\ v | v$ is the highest pageRank of all the pageRanks of the nodes in a cluster $\}$

The distributions are defined as follows:

1. **Distribution 1:** Contains the elements of S sorted with increasing pageRank value.
2. **Distribution 2:** Contains the elements of S sorted with increasing cluster size. The cluster size of each $v \in S$ is the size of the cluster from which the node whose pageRank value is v, belongs to.

The correlation coefficient of the above distributions quantifies how much the importance of the "leading node" of each cluster increases with the size of the cluster that it belongs to. The higher this correlation is, the stronger the evidence that clusters are formed around graphically important nodes will be, and they are not simple social circles.

3.2.3 Betweenness Centrality

For the purpose of computing the betweenness centrality of every node in the graph, the algorithm given by *U. Brandes* [7] will be used. However, in order to test the third hypothesis, (i.e. testing if some nodes lie between clusters and are responsible for the clusters being connected like node **A** in Figure 2.1), we need to modify the definition of betweenness centrality. In particular, for every pair of clusters in the graph, for every vertex v in the vertex set that is not included in the clusters, we would like to compute its betweenness centrality with respect to the pair of clusters, rather than the whole graph. The idea is that we are interested in the ratio of shortest paths that begin and end in either of the two clusters that pass through v, rather than the ratio of shortest paths that start and end on an arbitrary vertex in the graph that pass through v. An illustration of this idea is presented in Figure 3.1.

3.2.4 Maximum Flow

In order to compute the Gomory Hu tree, as needed by the clustering algorithm, we need to choose a maximum flow algorithm. In section 2.3 it was mentioned that the algorithms given by *Dinic*[9] and *Edmonds and Karp* run in $O(V^2E)$ and $O(VE^2)$. Assuming that a graph is sparse such that $E = \Theta(V)$, both algorithms run in $O(V^3)$. For graphs that are denser than that, i.e. $E = \omega(V)$, we have that $O(V^2E) \subset O(VE^2)$ and hence *Dinic's*

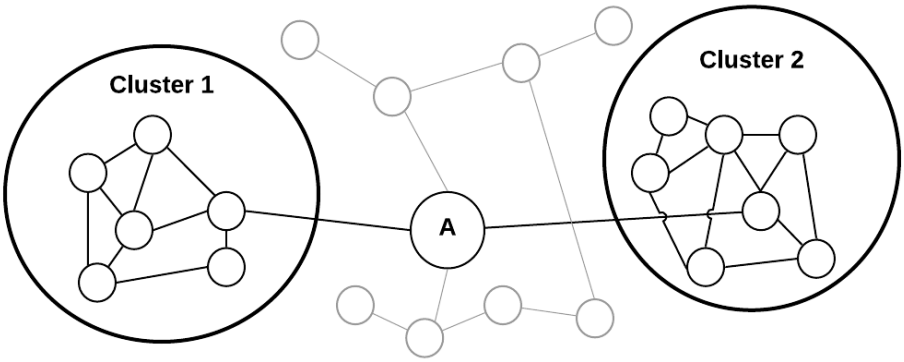

Figure 3.1: Computing the betweenness centrality of vertex **A** with respect to the two clusters rather than the whole graph implies that only paths that start and end in the two clusters will be used. Hence the greyed out vertices and edges will not be considered as start or end nodes in any of the paths used. However they will be considered as intermediate nodes in the paths.

algorithm is asymptotically faster. In the case where $E = o(V)$ and hence the graph is very sparse, we have that $O(V^2E) \supset O(VE^2)$, and hence *Edmond's and Karp's* is asymptotically faster. Even though there is some work that compares the performance of the two algorithms[20], it does not consider the sparsity of the graph as a variable. Hence, the algorithm for computing the Gomory-Hu tree will be chosen dynamically, based on the sparsity of the graph.

3.3 Structure of the solution

We now design the solution from an Object Oriented perspective. The solution will be designed in such a way to facilitate expandability in the future.

3.3.1 Functional Hierarchy of Classes.

To build a class structure on which the solution will be based, we adopt a hierarchical approach with respect to how close to the user is each class. In specific, there will be three hierarchy levels as seen in Figure 3.2. Each level makes use of the level(s) below it, and is responsible to provide a suitable abstraction to the level(s) above it.

Top level. The top level relates to the direct communication of the user with the software and the storage of raw data and computed information. In particular, the input/output from/to the user will be handled by the class `Terminal`, as described in the next section. This level receives the information as computed by the levels below it and presents it to the user. It is responsible for the robustness of the whole solution, as it will propagate any user requests to the level below it only if they are valid.

Middle level. This level implements the necessary graph theoretic algorithms for the computation of the various results in the form of data structures as required by the needs of the solution. It makes use of the raw data supplied by the level below it, and supplies the results to the level above it to be displayed to the user. This level is expected to create potential computational bottlenecks.

Bottom level. The bottom level deals with the organizational structure of raw data. It loads the datasets and stores them in a format that can be used by the level above it. Any anomalies in the datasets are handled by this level.

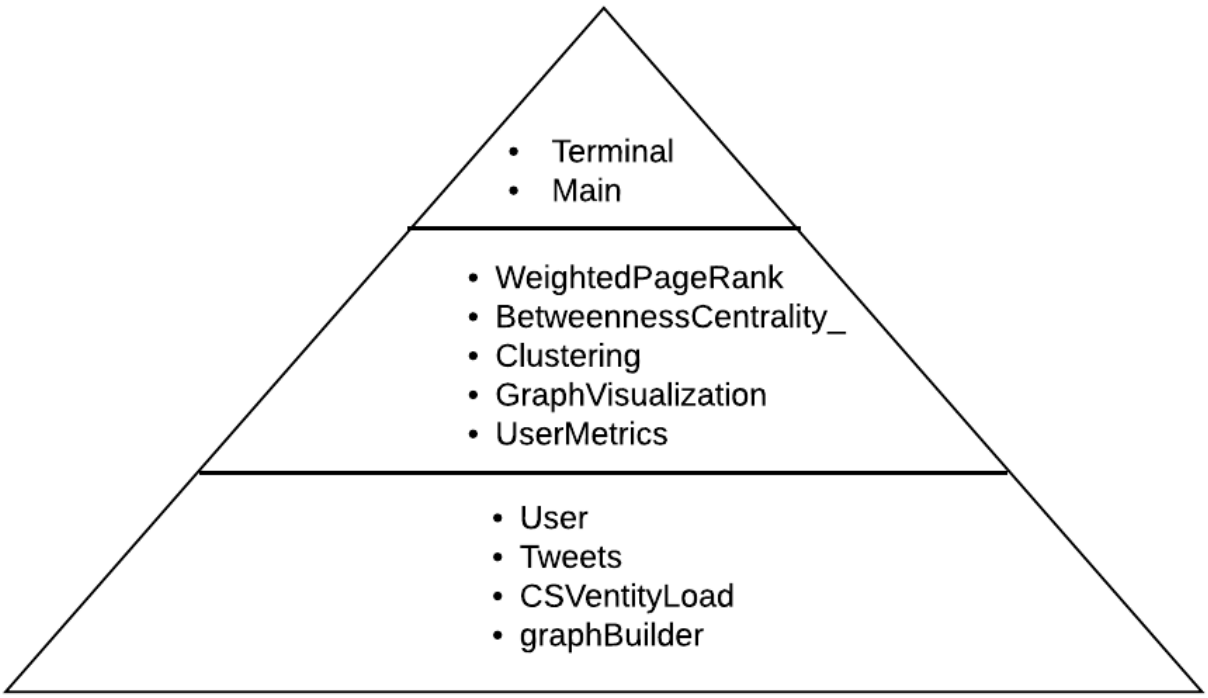

Figure 3.2: Visual representation of the hierarchy of the classes with respect to the context they operate.

3.3.2 Class Relationships and Functionality

In this subsection, each class is defined in terms of the functionality and abstraction it will provide to other classes, as well as how it uses these traits as provided by other classes. A UML diagram of the solution is shown in Figure 3.3. For the purpose of keeping the UML diagram concise, only the methods that will be used by other classes are shown in the diagram. Also, the scope of every attribute of each class will be "Private", and methods to get and set it will be defined for every class so that each object remains consistent. An explanation of the functionality of each class, beginning from the bottom level, follows:

Classes "User" and "Tweet". Each object of each of those classes will represent one User or Tweet, as loaded from the dataset. They will store the fields defined in table 3.1 for each object. Instances of those classes will be created by supplying an array of strings, which will correspond to one line of the respective CSV file, and each of its elements will correspond to one field of that line. Their constructors will therefore need to parse each element of that array, and set each attribute based on the result of the parsing.

Interface "Entity" and class "CSVentityLoad". The class "CSVentityLoad" is responsible for reading lines from CSV files, parsing each of them to an array of Strings, and then constructing a User or a Tweet object, based on the elements of the array. Since it is required that User or Tweet objects are constructed, both of these classes are

abstracted by defining the "Entity" interface. CSVentityLoad objects are only concerned with loading "Entities", rather than Tweets or Users. When the whole CSV file is loaded, a map with the key set being the set of IDs of the loaded entities, and the value set being the set of entity objects to which the keys correspond to, is returned. The functionality of the CSVentityLoad, User and Tweets class will fulfill the Data Input requirement

Class "GraphBuilder". This class is used for the construction of the "Influence graph", by using the two map data structures which were constructed by class "CSVentityLoad". The influence graph will be constructed as defined in section 2.3. Once the graph is constructed, the reference to it will be given to the top level. Only one instance of this class is needed for the construction of the graph, and hence the singleton pattern will be implemented. Also, as is evident from the name and functionality of this graph, it implements the Builder pattern.

Class "GraphVisualization". Even though the user interface will be primarily command line based, this class will present a visual representation of the graph in a separate window. The graph that will be used will be the one constructed by "graphBuilder", but the reference to it will be received from the level above it rather than from the graphBuilder object. The method *init()* will initialise the applet as defined by its superclass and the method *visualize()* will launch the visualization. The exact interface will be defined in the next section. This class will be implemented as an extension of Japplet. The functionality of this class will fulfill the Visualization requirement.

Class "BetweennessCentrality" . Instances of this class will make use of the influence graph and they are responsible for computing a range of betweenness centrality measures for different nodes. Firstly, method *allCentralities()* will compute the betweenness centrality of every vertex in the graph and return it in the form of a mapping from vertex to double. Secondly, the method *averageCentrality* will compute the average centrality of every vertex in the vertex set of the graph. Those two methods will make use the algorithm given by *Brandes* [7]. The method *clusterBetweennessCentrality* will compute the betweenness centrality of some given vertex by considering only paths from two clusters given to it as defined in section 3.2. The functionality of this class is essential for the testing of the third hypothesis on our dataset.

Class "Clustering". The purpose of this class is to compute clusters of vertices of high connectivity in the influence graph. Clusters will be returned as a *List<Set< V>>* by the method *getClusters()*, such that *V* is the abstract type of each vertex, *Set< V>* is a cluster of vertices stored as a set data structure, and *List<Set< V>>* is a list containing all clusters. Since some clusters are expected to be very small, and possibly uninteresting, *getBiggestClusters(minSize : Int)* will provide functionality to remove all such clusters from the list. The functionality of this class is essential for the testing of the first hypothesis on our dataset.

Class "WeightedPageRank". This class will provide functionality for computation of the pageRank values for all the vertices in the given graph, as defined in the section 3.2. They will be returned as a mapping of the abstract vertex type *V* to a *Double*. Construction of objects of this class will require a reference to the influence graph, the

damping factor value, and the tolerance value. The functionality of this class is essential for the testing of the second hypothesis on our dataset.

class "UserMetrics". The purpose of this class is to provide various statistical metrics regarding the user accounts. Methods *getMean(distr : List<Double>)* and *stdDev(distr : List<Double>)* will compute the mean and standard deviation of the given distribution of numbers respectively. The two overloaded *getCorrelation(...)* methods will compute Pearson's correlation coefficient given two distributions, as ordered lists or as a mapping of users to doubles.

class "Terminal". This class will provide a command line interface to the user, and it will act as the coordinator for the whole system. It will make use of the functionality provided by the lowest level classes for loading the datasets and constructing the influence graph. After the graph is constructed and returned to it as a reference, it will be able to pass it to the mid-level classes which will perform computations and return results to it, as described in the previous paragraphs. The robustness of the whole system with respect to the user input is delegated to this class. Additionally, it makes sense to have only one instance of this class, and hence it implements the singleton pattern.

class "Main". This class will instantiate the *Terminal* class and invoke its *start()* method. Further computations during the running time of the software are delegated to that method.

Software scalability

The above design is very scalable with respect to both context of use and functionality. We discuss each aspect in the following paragraphs.

Context of use. The software can very easily be modified/expanded so that this type of analyses are performed on datasets of different structure, and/or that originate from a different social network. In particular, the classes tweet and user can be replaced with other classes that implement the "Entity" interface (for example FacebookUser and FacebookPost), and since most other classes only care about "Entities", and not about Users and Tweets in specific, they can remain intact. This however may require some modifications in the Terminal and userMetrics classes, as some functionality implemented there is specific to the particular datasets.

Functionality. The software is expandable with respect to the functionality it can offer as well. In graph-theoretic terms, by the term "Functionality" we mean algorithms applied on the influence graph. One could keep adding "Functionality Classes" in the middle level of the hierarchy illustrated in Figure 3.2 without affecting most of the other classes, with the exception of class Terminal, which would need modifications, so that the new functionality is used.

3.4 User Interface

We now define the user interface, which includes command line syntax and some graphical syntax. The set of allowable commands will be a strict subset of all the instances generated

by the following syntax:

$$
\begin{aligned}
Exp &:= Instruction\ Parameter^* \\
Instruction &:= \textbf{load} \mid \textbf{link} \mid \textbf{visualise} \mid \textbf{pageRank} \mid \textbf{cluster} \\
&\quad \mid \textbf{betweennessCentrality} \mid \textbf{quit} \\
Parameter &:= ParameterName\ ParameterValue \\
ParameterName &:= String \\
ParameterValue &:= \mathbb{R} \mid \mathbb{N} \mid String \mid \epsilon
\end{aligned}
$$

Where the variables in **bold** are terminal variables. Not every possible expression generated by the above abstract syntax is a valid command, since there are only finitely many ParameterNames for each command, which will be discussed next.

Instruction "load". Loads a tweet CSV file or a user CSV file in memory. The files must follow the RFC4180 CSV standard and must contain the fields defined in table 3.1 respectively. "load" takes one parameter, with parameter name **"-tweets"** or **"-users"** that specifies whether the file contains users or tweets and a String parameter value, specifying the location of the file to be loaded. An instance of the above command is: `load -tweets /dir1/dir2/.../myTweets.csv`

Instruction "link". Specifies that the tweets and users that were previously loaded must be combined to construct the influence graph. Tweets are internally assigned to the user that tweeted them based on their **userID** field. It reports the number of tweets whose user could not be found in the user dataset, and hence were discarded. This instruction takes no parameters.

Instruction "visualise". Assumes that the users and tweets datasets were loaded and the influence graph was constructed. A graphical representation of the influence graph will be produced as shown in Figure 3.4. This instruction takes no parameters.

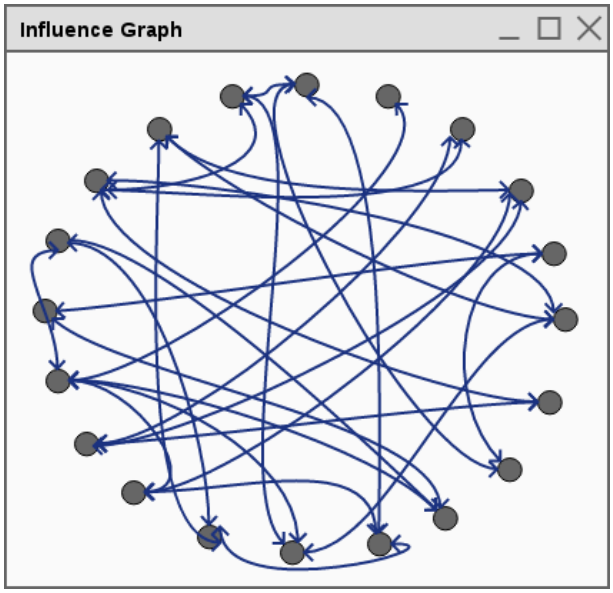

Figure 3.4: A sketch of how the constructed graph will look like is presented in a circular layout. The real graph in the implementation is expected to have a lot more vertices.

Instruction "pageRank". This command will be used for every computation related to pageRank. Its functionality can be defined by its parameters.

In order to compute the pageRank values of every node in the graph, it requires the `-compute`, `-tolerance` and `-damp-factor` parameters. `-compute` takes no value, and `-tolerance` and `-damp-factor` take a real number in [0, 1]. Those two parameters define the tolerance and damping factor of the algorithm as defined in section

2.3. An example of the `pageRank` instruction is: `pageRank -compute -tolerance 0.07 -damp-factor 0.15`.

`pageRank` can also be used for getting the pageRank value of a particular user, assuming it has already been computed. to do so, it must be given the `-of-user` option, which takes no values, the `-id` option which takes an integer, representing the ID of the user, and the `-base` option which is either `10` or `16` and represents the base in which the id is represented. An example of such usage is: `pageRank -of-user -id 15452 -base 10`.

Furthermore, `pageRank` can take the options `-disinfo-to-misinfo-pagerank-correlation` or `-followers-pagerank-correlation`, none of which takes any values, and they are used for computing the Pearson's correlation coefficients as defined in section 3.2.

Instruction "cluster" . This instruction is responsible for handling every computation related to clustering. We define each of the possible arguments it takes in turn.

The argument for internally clustering the graph is `-alpha`. It takes a real number as a value, and it represents the *alpha* value defined in section 2.3, in the *Clustering* sub-section. Execution of this command results in internally computing the clusters, but no output is given. An example of using `cluster` with the `-alpha` option is: `cluster -alpha 0.2`.

In order to get the sizes of those clusters that were computed using the `-alpha` argument, the `-get-cluster-distribution` argument needs to be used. This argument takes no values.

Furthermore, functionality for computing the Pearson's correlation of the pageRank values of the leading vertices of the clusters, sorted with ascending cluster size and sorted with ascending pageRank value, as defined in section 3.2. The argument to compute and output the above is `-cluster-pagerank-correlation`, and it takes no parameter values.

Finally, some clusters may be small and uninteresting. To truncate the small clusters, the argument `-truncate-small-clusters` needs to be passed.

Instruction "betweennessCentrality". Under this instruction lies all the functionality for computations related to the betweenness centrality of the graph. We will define each argument in turn.

Parameter `-average` returns the average centrality of the vertices of graph. This parameter takes no values.

Parameter `-largest` returns the average largest of the vertices of graph. This parameter takes no values.

Parameter `-inter-cluster-bet-centrality` computes the modified betweenness centrality measure defined in section 3.2 for computing the maximum betweenness centrality of the vertices with respect to some pair of clusters, for all pairs of clusters. It returns the largest such betweenness centrality throughout all vertices and all pairs of clusters. This parameter takes no values.

Instruction "quit". `quit` is used to terminate the program. It takes no parameters.

3.5 Language and Technologies Used

3.5.1 The Language

The project will be implemented in Java, as it offers a reasonable running time performance, an adequate abstraction for the implementation of the algorithms outlined in chapter 2, and it is strict with respect to most modern computer programming principles like object orientation, strong and static typing. In particular, Java 8 will be used since it offers a basic functional framework that can be used in order to reduce the algorithmic complexity and allow for a higher level specification of each algorithm. Also, the running time abstraction offered by the Java virtual machine (JVM) allows for the software to be platform independent, even though this may have some penalty on performance.

3.5.2 Technologies

Maven. Maven will be used as the build automation tool to manage build dependencies. It simplifies the building process, and allows the programmer to specify the libraries needed for the compilation process. If those libraries are not found locally, they are downloaded in each compilation. This relaxes the requirement of the user having installed those libraries on their machine, before the software is run.

JGraphT. JGraphT is the main library that will be used, as it provides some functionality that will allow the implementation process to be lifted in a more abstract level. In particular, it offers a graph API for the construction of graphs, and provides some of the algorithms that will be needed.

3.6 Potential problems and their solutions.

Computational Intractability. Graph-theoretic algorithms do not generally scale very well. The algorithms outlined in chapter 2 are not an exception, as none of them has a linear worst case complexity. Also, the number of nodes in the graph is quite large. The conjunction of those two facts will potentially render the problem computationally intractable. Hence, not only the required results will not be derived from the dataset, but it will also imply that the software produced will not be applicable to be used by other researchers for -arbitrarily large- datasets.

Potential Solutions. To address the above, a number of techniques can be adopted. First, memory usage - running time tradeoffs can be exploited. Furthermore, the language could be changed to C++, which is faster, and on which various other hardware related optimizations can be done. Additionally, specialised hardware can be used, like a GPGPU. As a last resort, a dataset of smaller size can be used.

Meaningless Results. It can be the case that the hypotheses result in all being negative. This will imply that the graph is similar to a random graph, given the specific node number and sparsity.

Potential Solution. In this unlikely case, the hypotheses can be tested on other similar data sets released by twitter. If the same result is derived, then a conclusion that there is little graphical meaning in fake news datasets can be made.

Similar projects are published. Even though no similar research has been carried out so far, it can be the case that a project is published before this work is finished, which tests similar hypotheses on the same dataset, and gives out a similar -or potentially better- software to the public.

Potential Solution. The software developed can be ran on the other datasets published by twitter. If this is not possible, a result comparison can be made.

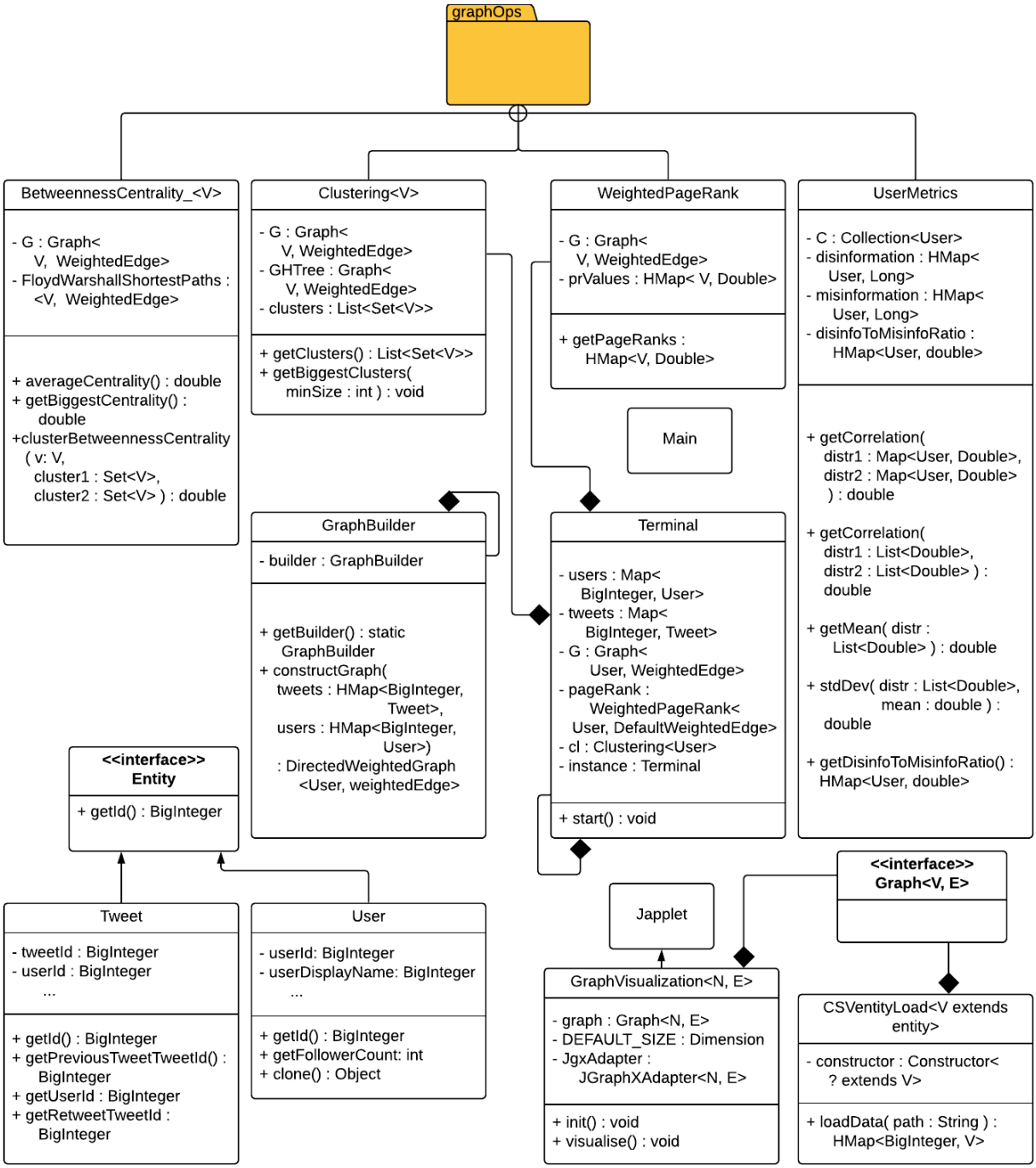

Figure 3.3: UML diagram of the solution.

Chapter 4

Implementation

4.1 Execution Flow

To describe the object creations and interactions, we follow an approach that orders the commands according to their expected execution order. When the application is launched, the single object of the `Terminal` class is created, and its `start()` method is invoked by `main(...)`. The execution flow of the program then remains in the `start()` method at the highest point in the method call hierarchy. This will launch the interactive and stateful console, and real-time input/output will be initiated. The software then assumes that some logical sequence of commands will be entered by the user. For example, it does not make sense to ask to compute the pageRank values of the nodes of the graph before constructing the graph. The expected command sequence is described diagrammatically, as a topologically sorted directed acyclic graph in Figure 4.1. We describe the object actions based on each command, starting from the commands that occupy a topologically lower position in the graph. Furthermore, we ignore some commands that simply display information that was computed earlier, as they contain very little computational intuition.

4.1.1 Algorithm independent Commands

Command "load". `load` will construct a `CSVentityLoad` object. `CSVentityLoad` objects are used for loading users or tweets in memory, as they are both entities. In particular, when they are being constructed they are provided with the class name of the entity that the particular object is tasked to load, (`User.class` or `Tweet.class`). This allows `CSVentityLoad` to extract the constructor of the entity implementation that was provided, and delegate the construction of individual entities to it. The instantiation of Tweets or Users from a `CSVentityLoad` object is therefore achieved using what is known as dynamic dispatch. When the construction of every entity from the file is finished, The entities will be returned as a map to the `Terminal` object, whose keyset will be the set of IDs of the entities, and the stored values will be the references to the entity objects. Once both the users and their tweets are loaded, the state of the `Terminal` object will be updated, and the next command set will be available for execution, which in this case only contains the `link` command.

Command "link". After the `Terminal` object assigns each tweet to its respective user (the account that it was tweeted from), it uses the single `GraphBuilder` object to construct the influence graph by passing it the tweet and user datasets which were previously loaded.

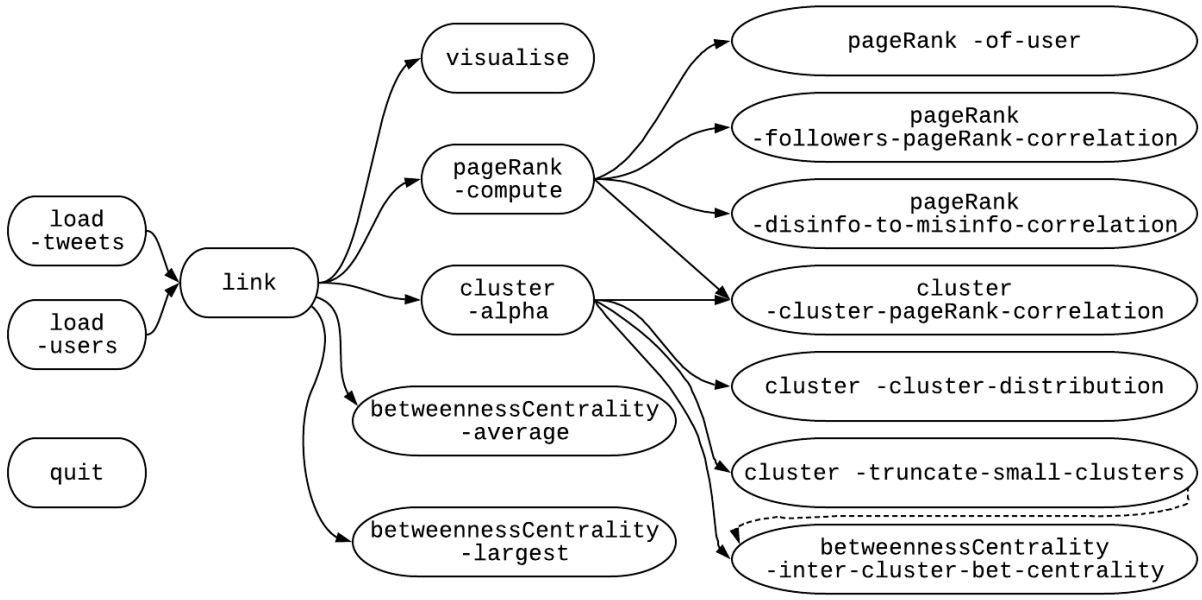

Figure 4.1: The command set as a topologically sorted graph. The software assumes that when a command is entered by the user, all of its ancestor commands have been entered and executed successfully. It is first required that the datasets are loaded in memory (load command) and linked(link command). Then, the different algorithms can be executed on the graph. One can easily see that the four commands that pageRank -compute points to, require the pageRank values to have already been computed. The same holds for the four commands that cluster -alpha points to. Executing a command results in resetting any state that was mutated by its descendants. For example, if pageRank -compute is executed on a dataset but then a different dataset is loaded, the pageRank computations of the previous dataset will be discarded. The dotted edge in the bottom right part of the Figure suggests that its source command is not strictly required, but highly recommended to improve the efficiency of its target command. We discuss this in a subsequent subsection. quit can be executed at any time, and so it has no prerequisite commands.

The reference to the graph is returned to the `Terminal` object and the available instruction set is now expanded, based on Figure 4.1.

Command "visualise". Constructs a `GraphVisualisation` object and presents the constructed influence graph in a new window, based on the sketch presented in Figure 3.4. This does not update the state of the `Terminal` object.

Commands "pageRank -compute, cluster -alpha, and betweennessCentrality -*". A common implementation pattern is followed for these three commands and we therefore describe it in this paragraph for all three of them. Each of these commands will invoke the constructor of their respective classes. These constructors will eagerly compute the pageRank values, clusters and betweenness centralities of every vertex respectively. i.e. The algorithms that compute those metrics are executed upon object creation, and not upon invocation of a particular method in the object. Once execution of the respective algorithm is finished, the results are stored in a data-structure in the object that executed them, and the object is then stored as a reference by the `Terminal` instance, and its state is updated. The user will therefore be able to execute the above commands with the

arguments specified on the right side of Figure 4.1.

4.1.2 Correlation Based Commands

This subsection concerns some of the commands on the rightmost part of Figure 4.1 that involve the computation of some correlation coefficient. Again, a common implementation pattern is followed for the three correlation-based metrics and we therefore describe it in this paragraph. Those commands are used for the computation of the Pearson's correlation coefficients of some particular distributions as defined in section 3.2. All three of them work by computing the two distributions which are needed for computing their correlation and by defining a `userMetrics` object. These actions are done in the `Terminal` object. The `getCorrelation(..)` method of the `userMetrics` object is invoked, and the two distributions are passed to it. This method then uses private methods to split the task of computing the correlation coefficient of the two distributions. The coefficient is then returned to the `Terminal` object. The `userMetrics` object, the two distributions, and the returned coefficients are then discarded, and will not mutate the state of the `Terminal` object.

4.1.3 Betweenness Centrality Based Commands

The `betweennessCentrality_` object of the `Terminal` object is created when any of the applicable parameters are given to the `betweennessCentrality` command.

Parameters -average and -largest. Invokes the `averageCentrality()` or `getBiggestCentrality()` method of the `BetweennessCentrality_` object of the `Terminal` object. These methods compute the appropriate metric and return it to their caller, namely `Terminal`.

Parameter -inter-cluster-bet-centrality. Utilizes the `clusterBetweennessCentrality(...)` method to compute the betweenness centrality of all the vertices with respect to all pairs of clusters. As defined in section 3.3.2, this method computes the betweenness centrality of some specific vertex, with respect to some specific pair of clusters. To perform this computation for all vertices and for all pairs of clusters, we invoke it for all vertices, and for all pairs of clusters in the `Terminal` object, and return the biggest for all invocations. The `clusterBetweennessCentrality(...)` does not implement the algorithm suggested by *Brandes*, as it is only applicable for the general case of betweenness centrality rather than the specific one we defined in section 3.2. Instead it naively computes it by counting the shortest paths from every vertex of `cluster1` to every vertex of `cluster2` (and vice versa), and taking the sum of the ratio of the number of those paths that pass through the vertex `v`. To speed up the counting of shortest paths part of the algorithm, it pre-computes the shortest paths of all pairs of vertices by using the *Floyd-Warshall* all pairs shortest paths algorithm [21]. The shortest paths are computed when the `BetweennessCentrality_` object is created.

4.1.4 Clustering Based Commands

Parameter -truncate-small-cluster. Invokes the `getBiggestClusters(...)` of the `Clustering` object to remove the small and potentially uninteresting clusters. Computing

the betweenness centrality with respect to the clusters instead of the whole graph is done in a naive and inefficient way, as described in the previous paragraph. Truncating the small and uninteresting clusters will significantly speed up its computation. Even though this is not strictly required, it is highly recommended.

4.2 Robustness

Given that the program is operated with a command line, it highly relies on the accuracy of the input supplied to it by its operator. Making the assumption that the operator will be fully accurate is unreasonable, and hence functionality that handles unexpected input needs to be implemented. There are three types of improper inputs that the software needs to deal with:

1. **Unexpected/misspelled commands:**

   This type of error is about the first word of the input, namely the command. It occurs iff the first word is not one of the seven Instructions defined in the abstract syntax, in section 3.4.

2. **Unexpected command ordering:**

   As mentioned in the previous subsection, each command has some "prerequisite commands", as defined in the graph in figure 4.1. This type of error occurs when the proper command ordering is not followed.

3. **Unexpected/wrong parametrization of commands:**

   Based on the User Interface definition of section 3.4, each command has a unique list of parameters that can take. Failure to pass the correct parameter names to some command results in this error.

4. **Unreasonable parameter values:**

   Some of the parameters take one input value. If that value is unreasonable, e.g. a path that does not exist is given to the `-tweet` parameter of the `load` command, or a `-tolerance` value of `0.0` is given to the `pageRank` command, this type of error will be triggered.

5. **Wrong parameter value types:**

   All of the parameters that take a value are strict with respect to the data type of said value. If some value has an unexpected type, this type of error it triggered.

To deal with the above, a combination of conditional tests with Java's Exception framework are used. Conditional tests are executed prior to the execution of the main code, and if the test condition fails, an error message is printed, and the main code is not executed. Exceptions on the other hand take the more "optimistic" approach by executing the code without checking correctness, and some exception is thrown only if execution fails. In Figures 4.2 and 4.3 an instance of the second type of error is presented from the perspective of the user and the underneath code that handles it respectively. A similar pattern for handling incorrect inputs that fall in the same or different category is followed.

```
>>>> cluster -cluster-distribution
Please compute clustering first and try again!
>>>>
```

Figure 4.2: The command ordering error handler from the users' perspective

```
case "-cluster-distribution":
    if ( ! checkLength( arguments, length: 2 ) ) return;

    if ( cl != null ) {
        int[] clusterSizes = cl.getClusters().stream().mapToInt(set -> set.size() ).toArray();
        Arrays.stream(clusterSizes).sorted().forEach( size -> System.out.println( size ) );
        System.out.println( "num of clusters = " + Arrays.stream(clusterSizes).count() );
    } else
        System.out.println("Please compute clustering first and try again!");
    break;
```

Figure 4.3: This is the code that ensures that the user can only execute the cluster -cluster-distribution command only if the prerequisite commands have been executed successfully. In particular, the (cl != null) will hold *true* iff the clustering has already been computed.

4.3 Functional Streams

Throughout the implementation of the software, Java streams were heavily used. Their advantage with respect to easier reasoning of the correctness of the program was exploited, and even though the following is a hardly quantifiable metric, the confidence of the implementation process has been significantly increased. Even though it was attempted to keep the intermediate operations as "pure" as possible and avoid side-effects, it was unavoidable to do so to a full extend, due to the lack of support of monadic operations in the particular Java version. For example, operations that would have been wrapped around an `IO()` monad, (e.g. `IO( globalVariable += 1; )`) and evaluated "at the end of the world", are forced to produce a side-effect by interacting with the outside world. This was however kept in mind throughout the development, and it was known that such practise does not align with the general principles of functional programming. Further information about Java streams can be found in the official Java documentation[2]. An example of using streams is presented in Figure 4.4.

4.4 Other Design Choices / Techniques used

4.4.1 Dynamic Choice of Maximum Flow Algorithm

In section 3.2 it was discussed that *Dinic's* algorithm performs asymptotically better in sparse graphs, whereas the *Edmonds and Karp's* algorithm performs asymptotically better in denser graphs. In the implementation, the algorithm that is used is determined by the density of the input graph. In particular, if the graph has more edges than vertices, *Dinic's* algorithm is used, whilst *Edmonds and Karp's* algorithm is used if the vertices are more. Hence, the algorithm that better fits the particular graph is used.

```
// Sorts the clusters according to their size, in ascending order
List<Set<User>> clustersSorted = cl.getClusters().stream()
        .filter( users1 -> users1.size() > 1 )
        .sorted(new Comparator<Set<User>>() {
            @Override
            public int compare(Set<User> users, Set<User> t1) {
                int s1 = users.size();
                int s2 = t1.size();
                if (s1>s2) return 1;
                if (s1<s2) return -1;
                return 0;
            }
        }).collect(Collectors.toList());
```

Figure 4.4: An example where streams are used is illustrated. cl.getClusters() returns a list of set of vertexes, with each set representing a cluster. After filtering the clusters that are made up by a single vertex, a Comparator object is provided, that is required by sort(...) to sort the incoming stream based on the size of its elements. It is worth noting that sort(...) is a stateful operation since it may need to perform multiple passes on the stream to sort it, and it may retain some state while running. collect(...) is the effectful operation that constructs a List by gathering the elements from the stream.

4.4.2 Concurrency

One last advantage that is inherent to streams, is the easy parallelization of their evaluation. In particular, the method `parallelStream()` of every class of the `Collections` framework, automatically spawns many threads that will evaluate elements from the respective collection in parallel. This must be used with care, as effectful and stateful intermediate operations in the stream evaluation pipeline remove any guarantees about inter-Thread safety.

Chapter 5

Testing

A crucial part of the development process is the verification that the developed product behaves as it should (according to its specification) for all inputs. Describing the specification of the program and formally proving that the program meets the specification in some proof system is outside the scope of this project. Instead, we will follow the more straightforward approach of testing. Ensuring that the program passes a finite set of tests does not guarantee that it will output the correct results, or even terminate for all inputs. However, it increases our confidence about it.

To verify that the program is functionally correct, we must employ two different testing techniques, each operating at a different level of abstraction: Unit testing and End-to-End testing. Unit Testing isolates and tests components of the system individually, and end-to-end testing tests the system as a whole, as if it was operated by the user. We have used both techniques, and we describe the testing process in the following sections.

For the testing process, the implementation of the tests must not only satisfy the obvious purpose of testing, namely the evaluation of the correctness of the program, but it also needs to fulfill two more requirements:

- **Readability:** The purpose of each test must be easily understandable by any developer, and the identification of the origin of any unexpected behaviour must be easily derived from the failed test.
- **Extensibility:** The testing framework must be easy to use, and any developer must be able to easily add more tests for the existing or new functionality of the system.

5.1 Unit Testing

The strategy followed for implementing the tests is to test every public method of every concrete class which does not rely on other classes. In specific, it does not make sense to implement unit tests for the class `Tesminal` since it relies on the functionality of the rest of the program. We have chosen to only test public methods as a result of one of our implementation choices, namely, every method that would be needed only internally in the class was kept private. Hence, Private methods are eventually called by at least one public method in the class. Therefore by testing only the constructor and the other public methods, we implicitly test private methods as well. Furthermore, we do not test classes that contain very little computational intuition, like `Tweet` or `User`.

To facilitate unit testing, `JUnit` was used, as it is considered best practise for implementing unit tests for java programs. `JUnit` allows for independent development of

unit tests in the form of java methods. Its main idea it that the tester develops a mock state and/or mock data for a particular component of the program. The component is then run, and the results returned are compared with the expected results with a set of assertions. If all of the assertions defined in the test succeeded, the test is successful. An example of such a test is presented in figure 5.1. For the other components, the unit tests follow a similar layout.

```
@Test
public void testPageRankCompute4() {
    Graph<Character, DefaultWeightedEdge> g =
            new DefaultDirectedWeightedGraph<>( DefaultWeightedEdge.class );
```



```
    HashMap<Character, Double> computedPageRank;
    HashMap<Character, Double> expectedPageRank =
            new HashMap<>();

    WeightedPageRank<Character, DefaultWeightedEdge> weightedPR =
            new WeightedPageRank<>( g,  tol: 0.18,  damp: 0.15 );

    computedPageRank = weightedPR.getPageRanks();

    expectedPageRank.put( k: 'a',  v: 0.253);
    expectedPageRank.put( k: 'b',  v: 0.226);
    expectedPageRank.put( k: 'c',  v: 0.193);
    expectedPageRank.put( k: 'd',  v: 0.186);
    expectedPageRank.put( k: 'e',  v: 0.142);

    computedPageRank.entrySet().stream().
            forEach( entry ->
                    assertEquals(
                            entry.getValue(),
                            expectedPageRank.get(entry.getKey()),  delta: 0.001
                    )
            );
}
```

Figure 5.1: A unit Test for the pageRank algorithm is shown. We firsts construct a small scale mock graph, which is then passed to the algorithm. The three vertical dots represent the part of the test in which vertices and edges are added in the graph. The true pageRank values of the vertices are computed manually and they are added in the "expectedPageRank" Map. In the "computedPageRank" Map the values computed by the tested component are stored. The elements of the two Maps are asserted to be equal, within a tolerance of ± 0.001.

5.2 End-To-End Testing

End-To-End testing is about what the user sees, after they perform a particular action on the interface. It may involve interactions of many components of the software, and since we are not concerned about the execution of a particular component, we can say that it is a form of black-box testing, and compared to unit testing it lies at a higher point in the testing techniques hierarchy.

Since End-To-End tests are about sending the input that the user would send, and checking the output that the user would receive, they are less straight forward to im-

plement. We need to emulate the execution of the software, pass it the commands that the user would pass, receive the output, and check it against some expected output. To achieve this, we first develop a small protocol for writing tests in a text file, and a script that will execute the testing.

5.2.1 Framework

The script

We now describe the script that will run the tests briefly. It will be written in Python 2, and it has three main functionalities:

1. **Read the test file:** The tests will be specified in a separate file, using a protocol that will be defined in the next subsection. The script will read the contents of the file, parse them, and store the parsed tests in a datastructure. Each test will have two components: The command line instruction, and the expected output.

2. **Emulate the software:** The script will run the software as a child process as defined in Appendix A3 of POSIX [3]. This will allow the script to send input to the software's `stdin` file and receive output from its `stdout` and `stderr` files, which in our terms translates to sending it command line instructions and receiving its output.

3. **Run tests.** This functionality is a combination of the above two. In fact, when we have the tests in and a framework that will allow us to send and receive text to and from the software, we can send the instructions of each test to it, receive the output, and compare it with the expected output. This is repeated for every test.

Since the tests run by the script are made up of instructions that could have been executed by some fictional user, and output that the fictional user would see, we can say that this testing strategy is end-to-end.

The end-to-end tests protocol.

As it was mentioned in the previous subsection, the tests will be stored in a .txt file and will be read and parsed by the script. Each test will be made up of two parts: The command line instruction that is expected to be the input to the software, and zero or more lines that are expected to be the output. Their format in the file will be as follows:

- The start of some test will be indicated by a line which contains the word "**test:**"
- After each such line, the command line instruction is specified in the next line.
- After the line that contains the instruction follows a line that contains the text "**Expected Output:**"
- Then a number of lines specifying the expected output that should result from the execution of the instruction line follow. A line with the text **Empty** indicates that no output is expected.

A small portion of an example test file that follows the above protocol follows:

```
test:
load -users /project/src/main/resources/users.csv
expectedOutput:
770 users loaded successfully
test:
load -tweets /project/src/main/resources/tweets.csv
expectedOutput:
1122936 tweets loaded successfully
test:
link
expectedOutput:
Users are now linked with their tweets.
0 tweets could not be assigned to a user.
Influence graph is constructed.
```

It is worth mentioning that the tests -unlike the unit tests- in this case are not independent of each other. As it was specified in section 4.1, some instructions mutate the state of the running software, and since these tests are dummy instructions, such state mutations are also expected to happen.

5.3 Regression Testing

Regression testing is a type of testing, which is identical to end-to-end testing in structure terms but differs slightly in functionality. It is the act of running end-to-end tests each time a component of the software is finished. This is done to ensure that new features added during the development process do not break any functionality that was previously working as expected. To integrate regression testing in the development process, we simply run the end-to-end testing script in the run configuration of the IDE. Therefore, all of the end-to-end tests for the completed components are executed each time the software is run.

5.4 Testing Evaluation

To conclude this chapter, we evaluate the testing strategy we have followed with regards to its effectiveness and the fulfillment of the specified requirements. It is worth emphasizing that it is the strategy and the framework that will be assessed and not the tests themselves. This is because the tests can be freely changed and/or increased by future developers, whilst the testing strategy used is more "bound" to the project.

Even though very limited in scope with regards to the diversity of modern software engineering testing techniques[1], the testing strategy we have followed has the potential to identify most implementation bugs, mainly due to two abstraction levels at which tests were implemented.

It is evident that both secondary requirements were fulfilled, since both test types (end-to-end and unit tests) follow a common pattern with regards to tests of the same type, and hence understanding the idea behind one test from each type makes it easier to understand every other test, therefore satisfying the readability requirement. A future developer can easily add/remove tests of both types, and the underlying framework(s) (the python script and JUnit) will ensure that the tests are executed and any errors are

reported. The only complication is the dependence on the state produced by other tests in the end-to-end tests, but this is inherent to the software and has nothing to do with the testing. Hence the second requirement is fulfilled as well.

Chapter 6

Experiment Results

In this chapter, we present the results obtained from running the produced software on the dataset. We also evaluate our original hypotheses, and we give an explanation of each result with reference to the real world and human interactions in social networks. The results will be presented in a graphical form, plotted with auxiliary code written in python as the software does not offer such functionality.

6.1 Clusters of Vertices

The question of whether a graph contains clusters of vertices of high connectivity does not have a binary (yes/no) answer, simply because the "high connectivity" part of the question is of course not well defined and very subjective. What we can say for sure however is that we want those clusters to have high intra-connectivity compared to their inter-connectivity. This tradeoff is quantified by the clustering algorithm we have already discussed in sections 3.2 and 2.3 by its α parameter.

In the case of this dataset, the α value had to be adjusted several times until a satisfactory number and size of clusters was achieved. In specific, a high α value sets a high upper bound in intra-cluster connectivity, resulting in many clusters that only contain a single node. A low α value sets a low lower bound in inter-cluster connectivity, resulting in clusters whose size is of the order of the vertex set. Loosely speaking, we would like "not to many" clusters, whose size is "not too big" nor "too small".

This is achieved by employing a manual binary search of the $\mathbb{R}$ space to find the value of α that satisfies the above criterion. In particular, the algorithm is initially run with a very high and a very low values of α. If the results from both runs imply that α is between the two bounds, the algorithm is run again, and α is set to be the midpoint of the two previous bounds. The results of that run will allow us to determine whether the desired value is between the upper bound and the midpoint or the lower bound and the midpoint, effectively halving the space of values that α can take. This process is repeated until the space is narrowed enough, and the results of the clustering algorithm are displayed and explained in Figure 6.1.

The first hypothesis is based on the intuition that highly connected vertices will represent social circles that are more likely to interact with vertices within that circle. This is indeed the case, and it was verified by printing the account descriptions of the accounts in the clusters. It was observed that the descriptions of the accounts in a specific cluster are all in English or in Farsi, or all the accounts tend to lean towards a common political ideology. Such case is seen in Figure 6.2.

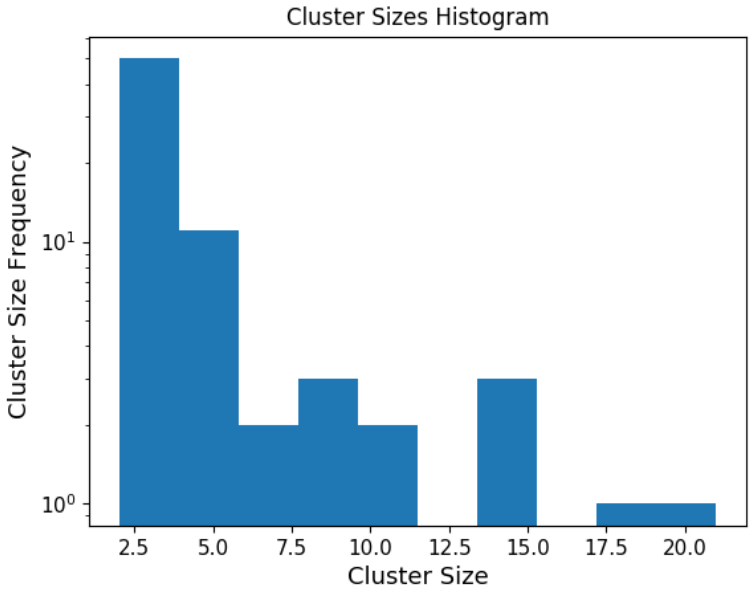

Figure 6.1: The figure shows a visualization of the cluster distribution obtained using $\alpha = 0.2$. Notice that the **Cluster Size Frequency** axis is in logarithmic scale. It is evident that clusters of small size are more frequent compared to bigger clusters. This is reasonable and expected, as it is more probable for a random set of nodes to form a cluster that satisfies the criteria imposed by α, given that the cluster is small, compared to if it was bigger. The bigger (and fewer) clusters however are those that are more significant(since high connectivity in small clusters may be due to randomness) and are expected to contain important vertices around which they were formed.

It is important to realize that getting clusters of vertices whose account descriptions are related in any of the ways described in the previous paragraph (common language or common political ideology), serves as a confirmation that the computed clusters indeed define real world social circles, and therefore the clusters are plausible.

Hypothesis Evaluation. We are now ready to evaluate the first hypothesis, which claims the existence of clusters. Since there exist clusters of accounts that are highly connected, and those clusters indeed form social circles, we can conclude that the hypothesis is evaluated to true.

6.2 Graphical Importance of Vertices

This section regards the evaluation of vertices with respect to their importance in the network. As it was already discussed in section 3.2, we will use a modified version of the pageRank algorithm for this evaluation. One key difference between this project and other analyses done in the context of social networks is that the importance of nodes is usually evaluated using a greedy metric, namely the indegree of each node. This of course does not consider the indegree of nodes that are two or more edges away from the node in question, and so on. PageRank solves this problem as explained in section 2.3. In this work we have taken the algorithm a step further by modifying the algorithm, and distributing the pageRank value of each vertex in some iteration unequally to the

```
>>>> cluster -user-descriptions 72
> As an independent news media organization we aim to inspire action on the likes
  of social justice & human rights & advocate transparency in politics.
> gamer / game tester don't forget your inner child or you won't survive this mad
  world
> i am an civil engineer and love politics and human rights
> sport book politic
> a mother and a cook
> art student writer photographer
> I am intrested in peace and politics
> programmers unite :))
> As an independent news media organization we aim to inspire action on the likes
  of social justice & human rights & advocate transparency in politics.
>>>>
```

Figure 6.2: The user descriptions of all the user accounts that belong in a common cluster. The highlighted portion of the descriptions shows that these accounts support left wing politics.

destination vertices of its outgoing vertices, based on the weight of the edge that connects them, as described in section 3.2.

In order to run this experiment, we will need to adjust the tolerance and damping factor parameters. The damping factor had to be adjusted so that the distribution of pageRank values is not too uniform, but also it had to prevent nodes with no outgoing edges to continuously absorb the pageRank from the other nodes. To achieve this, an binary search of the space $[0, 1)$ was carried out. The experiment was similar in structure with the one used in the previous section for the α value. The tolerance was set as low as possible, to achieve convergence. Setting it to a very low value will create an infinite loop that alters between two "pageRank value assignment states", due to the finite floating point accuracy allowed in modern computers. We present the distribution of pageRank values in Figure 6.3

Hypothesis Evaluation. We are now ready to evaluate the second hypothesis, which has three components:

- There exist accounts of high importance in the dataset.
- Clusters are likely to have been generated around those nodes of high importance.
- The nodes of high importance are more likely to tweet at a higher rate than they retweet.

We saw that it was the case that some accounts played a leading role in the given Fake News dataset. In particular, some accounts claimed more than 1% of the whole pageRank, which is a significantly high value in a dataset with 770 accounts. Hence, the first part of the hypothesis is positive.

To evaluate the second part of the hypothesis, we use the algorithm defined in section 3.2 that evaluates the correlation between the two distributions. Executing the algorithm results in a correlation of those two distributions of 0.79, suggesting that clusters are likely to be formed around highly ranked nodes, and hence the second part of the hypothesis is true.

To evaluate the third part of the same hypothesis, the correlation of two distributions had to be computed as per section 3.2. Computing it, results in a correlation of $7.85 \times$

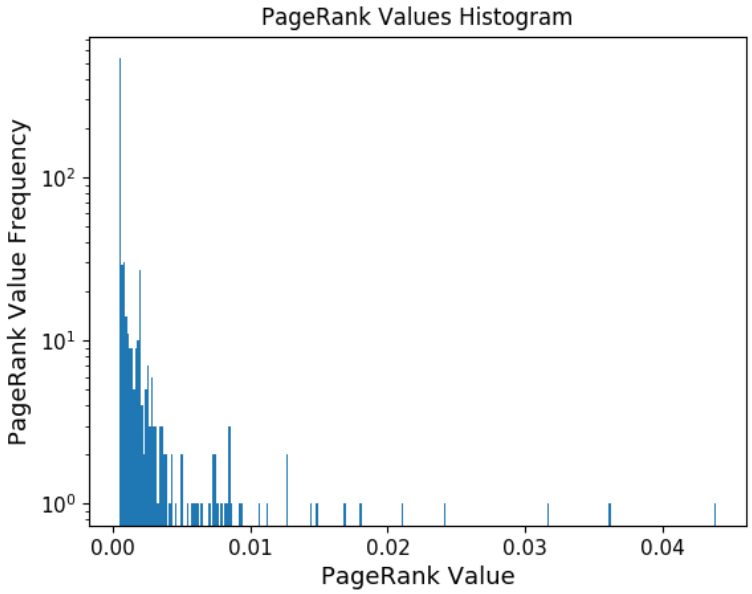

Figure 6.3: The figure shows the distribution of pageRank values in the graph, with tolerance value = 0.12 and damping factor = 0.3. The **PageRank Value Frequency** axis is in logarithmic scale. The **PageRank Values** axis represents the percentage of pageRank that is claimed by the number of vertices represented by each vertical bar. From the graph, we can conclude that some nodes have significantly higher importance (and hence influence) in the dataset than others. Also, the pageRank of the majority of accounts is very close to the average (namely 1/770, since there are 770 accounts and the total pageRank sums to 1), as expected.

10^{-4}, which is negligile. Hence the third part of this hypothesis is negative. Therefore, graphically important accounts are not more likely to have a higher tweet/retweet ratio.

Side Experiment 1. We again attempt to compute the correlation coefficient of two distributions. One of them contains the pageRank values assigned to users as before, and the other contains the number of followers of each cluster. Again, we ensure that the i^{th} element of each distribution corresponds to the same user, for all i. We obtain that there is a small but statistically significant correlation between the two distributions of 0.39

6.3 High Betweenness Centrality in Vetrices Between clusters

Hypothesis Evaluation. The motivation behind the final hypothesis is that there may exist users that act as "links" between social circles. In graph-theoretic terms, this translates to vertices having a significantly higher betweenness centrality with respect to a pair of clusters than the average betweenness centrality of the graph, as defined in section 3.2. The clusters that were used have size 14-21 vertices. We therefore compute the average betweenness centrality of the whole graph, and get the result of 4.03×10^{-4}. We also calculate the largest betweenness centrality with respect to the clusters, and get a result of 10.0×10^{-4}. Even though the latter is 2.5 times larger than the former, we cannot

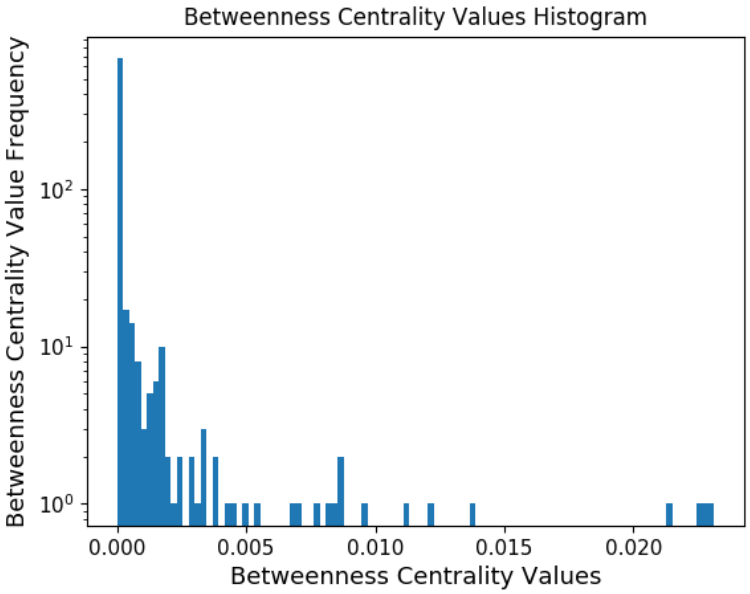

Figure 6.4: In this figure a histogram of the betweenness centrality values of the vertices are shown. It can be seen that some nodes have a very high betweenness centrality value, compared to the average vertex.

conclude that the hypothesis holds true, as their difference may be due to some random structure. The hypothesis is therefore false.

Side Experiment. The betweenness centrality value of every vertex is gathered and plotted in Figure 6.4 as a histogram. Even though nodes of high betweenness centrality that connect clusters could not be found, we can still see that there exist some nodes whose betweenness centrality is significantly high in the graph, even if they are not "cluster connectors". Those accounts play an important role when it comes to fake news propagation, as they can be seen as "retweet bridges". In specific, due to them, a significant amount of tweets can be transferred to other parts of the graph.

Chapter 7

Conclusion

7.1 Contributions of the project

7.1.1 The Iranian Dataset

We have discovered some very important properties about the particular dataset. The results become of even bigger importance considering that not much research has been done on the particular dataset, unlike other fake news datasets released by twitter.

First, we have seen that not all users are equal contributors to the spread of fake news. Some of them have a significantly high importance in the dataset, in terms of the attention they get from other less important users. Furthermore, we have observed some positive correlation between the importance of the nodes and the number of followers they have. Additionally, we have seen that the size of the formed clusters and the pageRank of the leading node in the cluster are positively correlated, which implies that the high-importance accounts are the ones that are likely to lead those fake news campaigns. Last but not least, we have observed that a minority of users act as "fake news bridges" in the graph, as their betweenness centrality measure is disproportionaly big compared to the average centrality of all the nodes. Even though this does not mean that these accounts are the ones that first publish fake news, they are certainly responsible for their propagation.

Real World Implications. It is important to realize that the findings for the particular dataset serve not only as an interesting result, but also they can be used to derive effective countermeasures against fake news. Starting with the accounts which were given a high pageRank value, some of which are the "leading" accounts in the formed clusters, one could naturally think that minimizing their influence (say by reducing the tweets other users get in their news feed from them) would have been an effective countermeasure against their campaigns. Furthermore, since the high betweenness centrality accounts serve as "bridges" when it comes to tweet propagation, minimizing either the amount of fake news tweets these users get on their news feed, or the amount of fake news tweets other users get from them, will significantly minimize the propagation of tweets through them. This kind of logical "fake news countermeasure inference" can be done on any dataset on which a similar analysis was performed by the developed software.

7.1.2 The Software Product

A significant part of this project was not only the results discovered for the particular data set, but also the robust software produced that performs this analysis. It can derive results of the same nature for other datasets, and due to the weakly coupled object relationships, expandability and/or modifiability are facilitated with respect to both functionality and the structure of the datasets it can receive as input. Therefore this software product can be seen as a starting point, or at least a motivation, for the utilization of algorithmic graph theory for Fake News Analysis.

7.1.3 Algorithmic Contributions

Apart from the results that are directly related to the context of fake news, in section 3.2 we have also derived two new metrics whose application is not bound to the particular context, but are purely graph theoretic definitions/algorithms.

PageRank. We have modified the original pageRank algorithm to apply to weighted graphs. The intuition is that the pageRank value of some node is distributed to its neighbors according to the weight of the edge connecting them, rather than uniformly dividing it by the number of neighbors.

Betweenness Centrality. The betweenness centrality definition was also modified, so that we only consider a certain subset of nodes that can act as the start/end nodes on the considered paths. The intermediate nodes in the paths can be any node in the cluster. When using this metric we say that we compute the betweenness centrality of a certain node "with respect to some subset of nodes".

7.2 Reflection

In this section, we outline some parts of the project that can be improved in the future, either by adding functionality, or by modifying existing functionality.

7.2.1 Aspects With Potential Improvements

Command line interface implementation. The command line interface was implemented using the string manipulation functions that the Java String class provides. Ideally, it should have been implemented as a proper tokenizer and a parser that would do the task.

Graph Visualization GUI. The visualization of the graph could have been more useful if functionality like zooming in/out of the graph, getting information on a specific user or a particular connection between users.

End-To-End Testing. Even though the framework that was developed for end-to-end testing (Python script and test protocol) is fully functional, it is also minimal. One could improve it by adding more features in the script, like reporting the instruction of the failed test, the expected output, and the actual output received. Furthermore, the script assumes a syntactically correct input file, and is not robust with respect to parsing the

data from the file. Hence, improving it with respect to robustness is another potential improvement to consider. Finally, it can be extended to receive more general-purpose data format, like a YAML file.

7.2.2 Future Directions

Connectivity. Another metric that could have been useful is computing the connectivity of the graph. In particular, one might be interested in finding the strongly/weakly connected components of the influence graph, which would define a partition of the set of users such that any user in any subset of the partition has no interactions with any user belonging to some other subset. An example where such a metric could have been used in practise, is to facilitate the prevention of propagation of fake news from one subset to another, by avoiding suggesting users in one subset to follow a user in some other subset, and hence creating a potential to connect the strongly/weakly disconnected components.

Better choice of maximum flow algorithm. In this project, the choice is made naively by simply comparing the number of vertices to the number of edges. In practice, it would be more sensible to derive two functions such that each of them stochastically predicts the running time of each algorithm. Solving this problem reduces to solving a linear regression task as described by *C. Bishop*[6], with input variables the density and size of the graph.

Client-server model. For the purpose of increasing the scale of datasets which the software can analyze, it would make sense to implement the functionality included in the middle and bottom layers described in section 3.3.1 in a different executable program, which could run on a more powerful machine as the server. The top layer described in the same section could be the client which would run on the local machine, and whose purpose would be to parse the commands given by the user, send them to the server as a request, and receive the results as the response. This of course opens up a big range of other design choices, to include the security of the communication channel, the agreement of the request-response protocol and usage of http request-responses vs webSockets.

References

[1] Introduction to software engineering/testing. https://en.wikibooks.org/wiki/Introduction_to_Software_Engineering/Testing#Regression_testing. Accessed: 2019-05-11.

[2] Java 8 functional streams. https://docs.oracle.com/javase/8/docs/api/java/util/stream/package-summary.html. Accessed: 2019-04-15.

[3] Portable operating system interface(posix). http://www.open-std.org/jtc1/sc22/open/n4217.pdf. Accessed: 2019-05-11.

[4] The pagerank citation ranking: Bringing order to the web. 01 1998.

[5] Traian Marius Truta Alina Campan, Alfredo Cuzzocrea. Fighting fake news spread in online social networks: Actual trends and future research directions. 2017.

[6] Christopher Bishop. *Pattern Recognition and machine learning.*

[7] Ulrik Brandes. A faster algorithm for betweenness centrality. 2001.

[8] Y. Benkler A. J. Berinsky K. M. Greenhill F. Menczer M. J. Metzger B. Nyhan G. Pennycook D. Rothschild M. Schudson S. A. Sloman C. R. Sunstein E. A. Thorson D. J. Watts J. L. Zittrain D. M. J. Lazer, M. A. Baum. The science of fake news. 03 2018.

[9] E. A. Dinic. Algorithm for solution of a problem of maximum flow in a network with power estimation. 1970.

[10] Linton C. Freeman. A set of measures of centrality based on betweenness. 03 1977.

[11] K. Tsioutsiouliklis G. W. Flake, R. E. Tarjan. Graph clustering and minimum cut trees.

[12] Richard M. Kapr Jack Edmonds. Theoretical improvements in algorithmic efficiency for network flow problems. 04 1972.

[13] Jie Tang Jimeng Sun. A survey of models and algorithms for social influence analysis.

[14] Huan Liu Liang Wu. Tracing fake-news footprints: Characterizing social media messages by how they propagate. 02 2018.

[15] D. R. Fulkerson L.R. Ford. Maximal flow through a network. 09 1955.

[16] Sunil Thulasidasan MS Srinivasan, Srinath Srinivasa.

[17] T. C. Hu R. E. Gomory. Multi-terminal network flows. 12 1961.

[18] S. E. Schaeffer. Graph clustering. 01 2007.

[19] Sinan Aral Soroush Vosoughi, Deb Roy. The spread of true and false news online. 03 2018.

[20] Anatoly Shalyto Viktor Arkhipov, Maxim Buzdalov. Worst-case execution time test generation for augmenting path maximum flow algorithms using genetic algorithms. 2013.

[21] Ronald L. Rivest y Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. page 1312, 2009.