Detecting Overlapping Communities in ISEBEL

Jonas Asogwa (219100245)

Universität Rostock

gegründet 1419

Traditio et Innovatio

May 2022

Supervisors
Dr.-Ing. Holger Meyer
apl. Prof. Dr.-Ing. habil. Meike Klettke

Abstract

Real-world complex networks are evident in the social networks, the internet, and biological networks. The complexities of most real-world networks make them have community structures that can be divided into subgroups based on some statistical features or how strongly the vertices are closely connected. A vertex may be a member of more than one community, resulting in overlapping communities in such a real-world network.

The ISEBEL story network is yet another complex network with varieties of folklore of werewolves, witches, and legends which form communities with overlapping vertices. Many algorithms for detecting overlapping communities in real-world networks exist. In this work, we propose a framework built on Apache spark using the BigClam algorithm that is able to detect overlapping communities in ISEBEL dataset.

Keywords: Graph mining , Community detection, Overlapping community detection, BigCLam

Acknowledgements

I would like to use this opportunity to thank my supervisor, Dr.-Ing. Holger Meyer for giving me the opportunity to work on this topic. I have really learnt a lot working on this topic. I would also like to thank him for his outstanding supervision and fast response time to my questions, even during some weekends.

My gratitude also to Zahra Khorsand, her previous work on detecting communities in ISEBEL gave me an insight, and also she was open to answer my questions.

I remain grateful to the following people: Fr. Augustine Chikezie, Fr. Thaddeus Ejiofor, Frau Georgette and my Family for making my studies in Deutschland possible and helping me succeed.

Declaration

I confirm that the work in this MSc project report has been composed solely by me and has not been used in any previous application for a degree. However, all sources of information have explicitly been acknowledged, and quotation marks distinguish all verbatim extracts.

Signed .... Date ....

Jonas Asogwa (219100245)

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Stories matter, who told them, where they are told and what it is told about. Good stories of an event or people are a source of promotion. The people's cultural heritage is also being known to the outside world through stories. Because of the importance of stories, it is important to preserve the original content of a story as it is transferred from generation to generation.

The ISEBEL project (Intelligent Search Engine for Belief Legends) is an important story search engine that aims to build an intelligent search platform to make stories from three existing databases namely: Dutch Folktale Database, the Danish Folktale Database and the Mechlenburgian Folklore Database readily accessible. ISEBEL initiators also aim for extensibility, whereby the system is also able to accommodate collections of stories from other countries in different languages and dialects.

The search for information in a modern search engine has taken a new dimension since the discovery of better data mining techniques. Modern information retrieval systems such as Google and Yahoo have leveraged big data processing techniques to retrieve content related to each other. In addition, the use of some methods in graph mining, such as motif finding, has proven to be effective in retrieving related content and predicting content that is likely to be related in the future, just like in the Facebook friend's suggestion system.

Stories in the ISEBEL search engine, just like the Google contents or Facebook friend's suggestions, are related not just in the content of the stories but also to the author's co-authorship relationship. The reality of storytelling, how authentic the story is, the richness of the content drills down to the narrator's viewpoint and the volume of information available to the narrator during the content creation. Thus, the content of a story is never complete when judged by one narrator's write-up. As a result, we need to compile stories across different slits in the ISEBEL search platform to get an entire account, and that is the need for an effective way of mining stories in the system.

Various techniques of mining data have proven to be effective in learning-related content. The community detection technique is used for massive amounts of data mining that represent this data in a graphical format, with vertices defined as the main subject and connections between topics represented as edges. Many community detection techniques exist, some of which will be discussed and made use of in implementation in this thesis.

As this platform accumulates data from various sources, existing now and future, more information overlaps will occur. Therefore, in mining a massive volume of data with overlaps, it is crucial not just to use only an overlapping community detection algorithm but also to employ a system that can work along with an overlapping community detection algorithm to facilitate data processing. And that is why we will be looking at combining a big data processing engine with an overlapping community detection method to achieve a sound system in this thesis.

1.1 Thesis Objective

This work is aimed at making the search of the users of the ISEBEL archive more relevant to their query. To achieve this objective, one needs to mine the data to find stories and their motifs that are related.

Also, Stories may belong to more than one group, that is an overlapping Story which may occur due to some network matrices being similar and Story narrators contributing to more than one Story, and unavailing these overlaps in the dataset gives a more robust search result when incorporated into the overall ISEBEL system.

1.2 Thesis outline

The remaining part of this work is structured as follows:

Chapter 2 Background. In this chapter, we discuss basic concepts required for an easier understanding of the rest of the work. The chapter first introduced the basics of graphs, application areas of graphs, techniques used in graph mining and community detection.

Chapter 3 Literature Review. To understand what is new in graph mining, we have taken this chapter to review the state of the arts in graph mining. First, the chapter reviewed some algorithms for non-overlapping community detection, then a detailed discussion of three well-known algorithms for detecting overlapping communities; the chapter ends with a review of tools used in graph mining.

Chapter 4 Analysis. This chapter presents the analysis of various datasets from the ISEBEL archive used in this thesis. This analysis has been carried out mainly using the Gephi visualization tool, which gives a good representation of the dataset in a graphical format and some statistical features of the network.

Chapter 5 Proposed Framework. This chapter explains the design and implementation of the framework for detecting overlapping communities in ISEBEL.

Chapter 6 Result and evaluation. This chapter discusses the results obtained from implementing and testing the framework.

Chapter 7 Conclusion and future work. The conclusions of the thesis are explained

Chapter 2

Background

2.1 Graph mining concepts

2.1.1 What is a Graph?

A graph provides a convenient way to represent the relation between entities and respective data. A vertex represents each entity, and their relationship is defined by an edge, where the vertex and/or edges can have an arbitrary label [Diane J Cook, 2006].Mathematically, a graph 'G' composed of a set of vertices 'V' and a set of edges ξ can be represented as $G = (V, \xi)$.

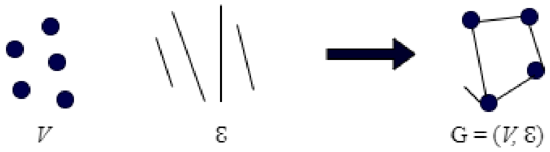


Figure 2.1: Forming a Graph

A graph can be undirected or directed.

Undirected graphs are graphs having edges with no direction associated with them. Thus, undirected graphs can be traversed in either direction.

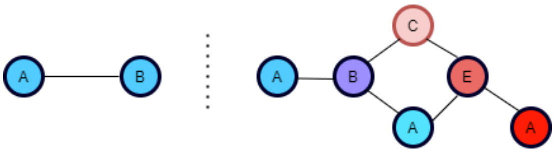


Figure 2.2: Undirected graph example

A directed graph is a graph with a set of vertices connected by edges, with each vertex having a direction associated with it.

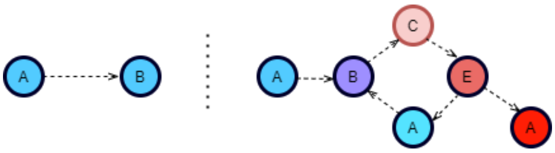


Figure 2.3: Directed graph example

Graph Examples

- Web pages
- Social Networks
- Computer Networks
- Transportation Networks
- Power Networks

2.1.2 Graph Mining

For a graph dataset, we need to exploit relational information to extract patterns and gain new knowledge in the dataset; this process is called graph mining. New knowledge can be a pattern such as a subgraph, an undiscovered relation to other elements, or on a more abstract level expression of trends in data [Diane J Cook, 2006]. The purpose of graph mining is to discover valuable insights from graph data. For example, in figure (2.4), new knowledge has been gained in the part of the graph with people having similar behaviour using graph clustering technique. In like manner, the prediction technique has been used in the graph to identify people with a probability of connecting in the near future.

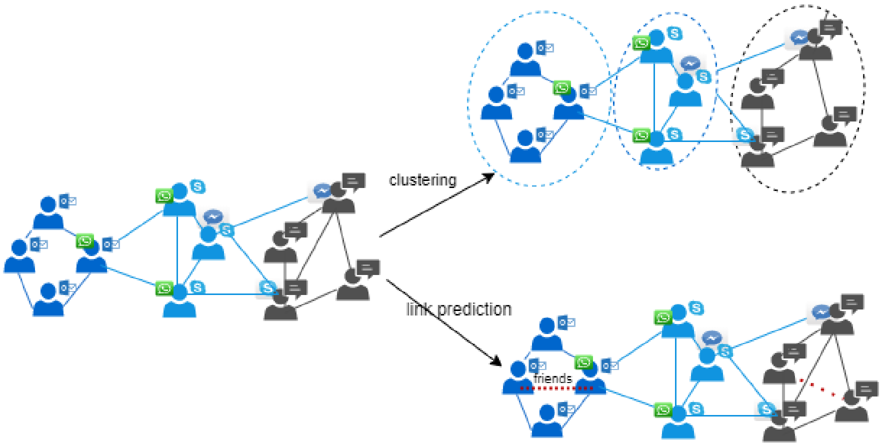


Figure 2.4: Graph mining

2.1.3 Applications of graph mining

Web graphs

A web graph is a collection of web pages connected by hyperlinks. In a web graph, static HTML pages represent the graph's nodes, while hyperlinks represent the edges [Kumar et al., 2000].

Web graphs find application in:

- Providing more accurate search services
- Identifying hubs and authorities
- Computing PageRank of www-pages
- Computing personalized PageRank

Social Network graphs

A social network is a network where individuals collaborate. Such networks are naturally modelled as a graph called a social network graph, where individuals represent nodes, and the connection between them represents the graph edges [Leskovec et al., 2020]. Social network graphs can be undirected, such as the Facebook friends' graph, where individuals connected are both friends. It can also be directed; an example is the Twitter followers' graph, where person A can follow person B without person B following person A. Social network graphs find applications in:

- Identifying the most influential people
- Recommending friends

- Companies targeting the right consumers of a product

Cybersecurity graphs
Cybersecurity graphs are graphs where computers represent the graph nodes and message traffic means the graph edges. Such a graph provides insight of:

- Knowledge of computer viruses' propagation
- Identifying intruders' machines
- Predict computers without proper authorization.

Entertainment graphs
Entertainment graphs are graphs where actors/movies represent the graph nodes, and the movies' attribute represent the graph edges. Such a graph finds applications in:

- Predicting upcoming movies' popularity
- Most likely movies to win award
- Determining most popular movies

2.1.4 Graph mining techniques

Frequency subgraph mining technique

Frequency subgraph mining (FSM) is a graph mining technique which discovers subgraphs that often occur in a graph. The discovery of frequent subgraphs usually consists of two steps:

- Candidate generation
- Support calculation

The process of discovering the frequent subgraph from the figure 2.5 is as follows: The input to the FSM algorithm is a graph dataset G, and user-defined minimum support (min-sup), and the output is the set of frequent subgraphs S. The support can be calculated by evaluating the number of isomorphs in a given graph [Dhiman and Jain, 2016].

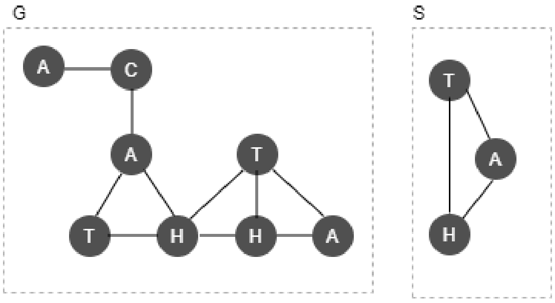


Figure 2.5: The Input Graph G and Subgraphs S of G. [Dhiman and Jain, 2016]

The candidate generation techniques can be classified into two:

- Apriori based approach
- Pattern-growth approach

Apriori based approach

The apriori-based approach algorithm works by joining two subgraphs which are frequently occurring. The apriori-based approach is similar to frequent item-set mining, which is recursive. The AGM [Inokuchi et al., 2000] algorithm is one of the algorithms for generating a candidate graph that increases the substructure size by one vertex at each iteration of the algorithm. Two frequent graphs of size k can be merged together only if they have the same $(k-1)$ subgraph. The new candidate subgraph has $(k-1)$ sized component, and the additional two vertices [Diane J Cook, 2006]. A possible problem of this algorithm is the generation of multiple candidates.

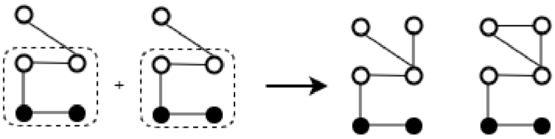


Figure 2.6: AGM [Diane J Cook, 2006]

Pattern based growth approach

The pattern growth algorithm extends the frequent graph by adding a new edge in each possible position. Using this edge-extension method prevents the overhead caused by merging two subgraphs in Apriori approach-based algorithms. However, a potential problem with this technique is that the same subgraph can be discovered many times [Diane J Cook, 2006].

Link prediction

The relationship between two individuals in a network could be identified by studying the network and predicting a possible existence of a link between them. The sole aim of link prediction is to identify individuals in the network that have the chance of forming a connection or not in the future.

The statement of the link prediction problem is as follows:

Given the connections in a social network at time t or during a time interval I, we wish to predict the connections that will be added to the network during the later time interval from time t' to some given time in the future [Al Hasan and Zaki, 2010].

Examples of application of link prediction in a real-world network:

- Friend recommendation in social networks
- Product recommendation in ecommerce
- Interaction prediction in biological networks
- Knowledge graph completion
- Suggesting collaborations between researchers based on co-authorship

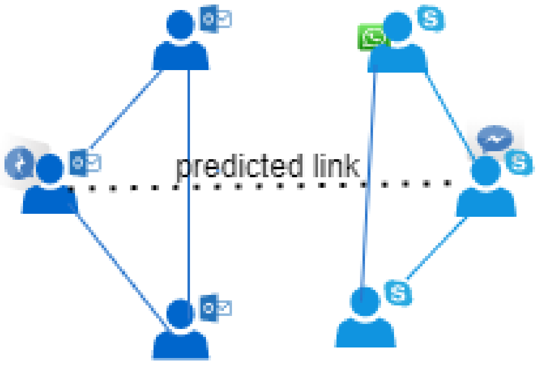


Figure 2.7: Link prediction

Methods of link prediction

- Node neighborhood based
- Ensemble of all path-based

Classification technique

Graph classification is a supervised learning problem that focuses on calculating certain graph statistics (graph features) that help in differentiating between graphs of different classes. The statistical features of the graph that the classification is based upon could be the total count of different subgraphs or graphlets [Lee et al., 2018]. For a graph classification task, a set of graphs, say, $g_1, g_2, \ldots, g_n$, is given, in which only the labels of a subset of graphs is seen, the goal is to predict the label of unseen graphs. Various methods of graph classification have been developed, some of which are:

- Graph classification using neural network
- Kernel graph
- Classification using structural attention

Kernel Graph

The kernel method is a machine learning method that measures the similarity of data points used for classification. Different kernel methods exist; the difference between these methods is usually based on how they extract features from graphs. The common kernel methods are: Radom-walk graph kernels [Borgwardt et al., 2005], shortest-path graph kernel [Borgwardt and Kriegel, 2005], graphlet graph kernels [Shervashidze et al., 2009]. In the random-walk kernel, two graphs g_1 and g_2 are compared by measuring the random walks of the graphs. Given a graph g_i with vertex $v \in V$ and edges $l^v \in l_1, l_2, l_k$. A vector $q(g_i)$ with edges l^i and l^j is formed by counting the number of tuples (l_s^i, l_j^d, m) in the source vertex s and destination vertex d in a walk with m as the length of the walk. In the shortest-path kernel method, the tuple (l_s^i, l_j^d, m) is formed from the shortest path instead of random walk. The graphlet kernels count the number of substructures, graphlets, in the graph.

Graph classification using neural network

Two methods used for graph classification using neural networks are Convolutional neural networks (CNNs) and deep learning.

The model developed by Niepert et al. [Niepert et al., 2016] for CNNs graph classification technique is a case in which a collection of graphs is given, "the CNNs is required to learn a function that can be used for classification and regression problem of unseen graphs or a large graph is given, and the CNNs is required to learn a representation(s) that can be used to infer unseen graph properties such as vertices and edges". The CNNs solve this problem by extracting features from local regions (receptive field) of the input by moving a filter over different areas of the graph.

One of the approaches of graph classification using deep learning developed by Li et al. [Li et al., 2016] called the DeepGraphs uses heat kernels to capture the features from the input graphs needed for the network training. They defined heat kernels over graphs using the eigenvalues and eigenvectors of the normalized graph Laplacian: $L_N = D^{\frac{-1}{2}} A D^{\frac{-1}{2}}$.

Clustering technique

Graph clustering (also called graph partitioning) divides data in the form of a graph. This could be vertex clustering which tends to group nodes of the graph into a group of densely related regions based on either edge weights or edge distances, or the graph is treated as an object to be clustered and cluster these objects based on similarity[Aggarwal, 2010].

Many algorithms used for graph clustering have been developed. Based on the input parameter, these algorithms are broadly divided into global and local techniques. In the global clustering technique, the whole graph is used as input for the clustering process, while in local clustering, only a certain seed vertex is used. Figure (2.8) shows a grouping of the algorithms into local or global clustering technique [Salem et al., 2019].

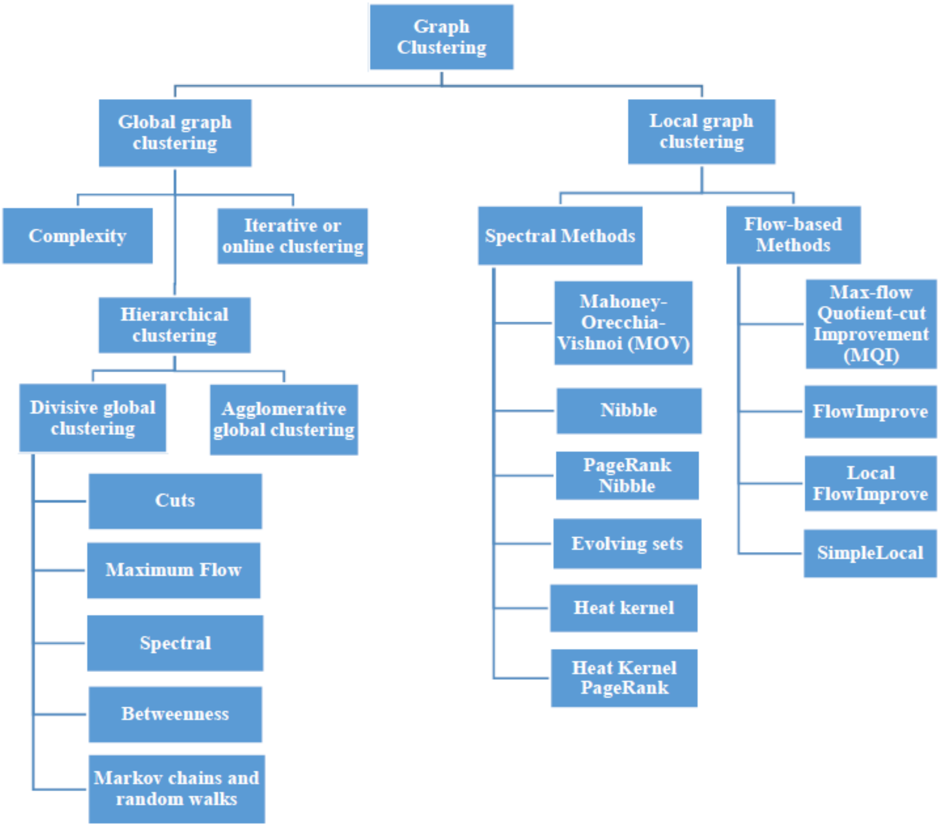


Figure 2.8: Different techniques of Graph Clustering [Salem et al., 2019]

2.2 Community detection

Communities in a network are a group of nodes that are distinguishable from the rest of the nodes in the network [Yang et al., 2010]. In a social network, communities represent a group of individuals in the network that have a dense connection to each other than to the rest of the group in the network.

Community detection is the task of discovering groups of nodes/individuals in a network based on their structural attributes. This is often difficult in a large-scale network with millions of nodes and edges, such as in a social or biological network (for example, a protein-protein interaction network). In such a huge network, one often needs a community detection algorithm to unravel the structure and dynamic characteristics of the network.

Various community detection algorithms exist which can detect non-overlapping, overlapping and nested communities in a network. In this work, the major focus will be on the study and application of the algorithms used to detect non-overlapping and overlapping communities. Some of these algorithms will be reviewed in chapter three.

Chapter 3

Literature review

To know the best method for detecting overlapping communities in the ISEBEL dataset, we need to review the state-of-the-art known to have achieved good results. In this chapter, we present some established related work on community detection, with a major focus on methods of detecting overlapping communities. At the end of this chapter, the reader should have a good understanding of suitable performing methods in non-overlapping and overlapping community detection and also some tools used in graph mining in general.

3.1 Non-overlapping community detection algorithms

3.1.1 D-walks

The D-walks (discriminative random walks) is a technique introduced by Callut et al. in [Callut et al., 2008] that can tackle semi-supervised classification problems in large graphs. The input graph is interpreted as a Markov chain (MC) in which random walks (D-walks) are performed. This technique is based on betweenness measures, in which the betweenness of a node 'x' with respect to class 'y' is measured as the average number of times 'x' is visited during D-walks. The D-walks can classify the unlabelled nodes of both directed or undirected graphs with a linear time complexity $\theta(|y|.m.L)$ and memory requirement of $\theta(m + L.n)$ where $|y|$ is the number of classes, 'm' is the number of edges in the input graph and 'L' is the maximum walk length considered. With such low complexities, this technique can deal with huge graphs containing several millions of nodes and edges. Moreover, a different experiment performed with the implementation of this technique in the CORA database shows that this technique can efficiently and accurately classify the unlabelled nodes of the graphs.

3.1.2 Multi-level kernel k-means

This is a general algorithm for graph clustering presented by Dhillon et al. in [Dhillon et al., 2005] that is based on multilevel methods using weighted kernel K-means objective function as refinement algorithms. This technique can handle a graph having many nodes and very large number of edges. The technique is divided into three phases: coarsening phase, initial clustering phase and refinement phase. In the coarsening phase, a graph G_0 is passed through a transformation function which transforms G_0 into a smaller set $G_1, G_2, \ldots, G_m$. In the initial clustering phase, a parameter which indicates how small the coarsest graph should be is specified using the spectral algorithm of Yu and Shi [Yu and Shi, 2003]. The refinement phase is an improvement. The technique has been implemented on the IMDB movie dataset. The dataset has 1.2 million nodes and 7.6 million edges. The proposed techniques compute 5000 cluster and 5000 eigenvectors.

3.1.3 Bi-partite graph co-clustering

Chen et al., in [Fonseca, 2003] proposed a graph model used to handle the many-to-many correspondences problem among concepts in ontologies. Their proposed technique made use of a weighted bipartite graph to model ontologies. Weights of graph edges are calculated through the use of similarity measure techniques; the similarity degree is then assigned to the weights of the edges in the graph. Graph partition techniques are applied to co-cluster the vertex sets of the bipartite graph, and mappings are established between two ontologies based on the resulting cluster pairs. The proposed technique used a threshold, in which the edges in the graph having a weight greater than the threshold are maintained while others are purged out. The main success of this technique is that many-to-many mapping can be established among ontologies.

3.2 Overlapping community detection algorithms

3.2.1 Community-Affiliation Graph Model (AGM)

In [Yang and Leskovec, 2012] Yang et al. motivated by the "*findings that community overlaps are more densely connected than the non-overlapping parts*" introduced the Community-Affiliation Graph Model (AGM) for overlapping network community detection.

The AGM is a generative type of model that:

- Generates a network from a model

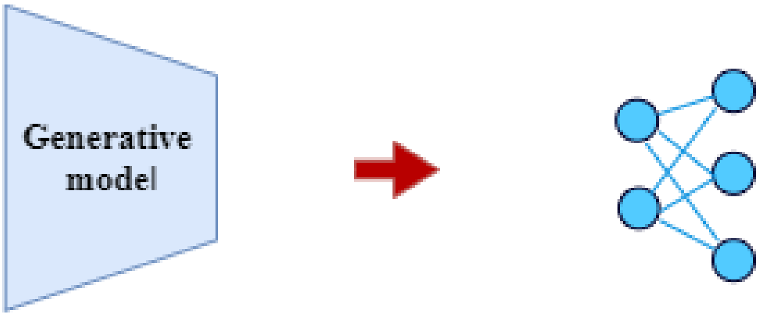


Figure 3.1: Generate graph from model

- For a network, defines a good generative model for the network.

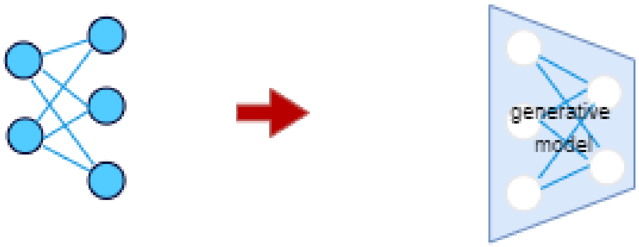


Figure 3.2: How to define good generative model from graph

The question of which AGM provides a solution is how communities generate the network's edges given a set of nodes.

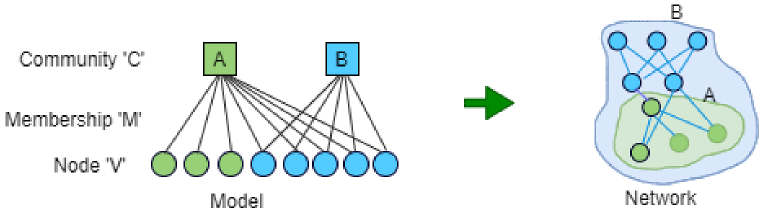


Figure 3.3: Model to Network of AGM

They defined the underlying model of AGM as a bipartite community affiliation, denoted as $B(V, C, M)$ where V: represents individuals in the community which correspond to a set of vertices in the network's graph representation, C: represents the communities within the network, M: represents membership/affiliation of the community.

To generate a network from the model, the model assigns a parameter p_c to each community $c \in C$, which denotes the probability for two individuals in the same community 'c' to connect with each other. The intra-community connectivity probabilities

p_c can be computed as a non-negative vector $p = (p_c)_{c \in C}$

In an AGM, given a bipartite community affiliation model $B(V, C, M)$, with p as the probability of connection between two members of the community, the AGM generates a graph $G(V, E)$ by creating an edge (u, v) between a pair of nodes u, $v \in V$ with probability given by:

$$p(u, v) = 1 - \prod_{k \in C_{uv}} (1 - P_k)[\text{Yang and Leskovec, 2012}]. \tag{3.1}$$

where $c_{uv} \subset C$ is a set of communities that u and v share. Thus, this ensures that individuals belonging to multiple common communities have a higher probability to connect.

The probability equation (3.1) does not permit individuals with no mutual community affiliation to link with each other. To create this possibility of link between these individuals, AGM uses an assumed additional community called ε-community with a very small probability of connection, given by:

$$\varepsilon = \frac{2|E|}{|V|(|V| - 1)}[\text{Yang and Leskovec, 2012}]. \tag{3.2}$$

Given a graph $G(V, E)$, how does AGM detect the network communities?
AGM does community detection by "fitting". In this process, the AGM tries to find affiliation graph B and parameters $\{p_c\}$ to G using maximum likelihood estimation $L(B, \{p_c\}) = P(G|B, \{p_c\})$ of G:

$$\arg \max_{B, \{p_c\}} L(B, \{p_c\}) = \prod_{(u,v) \in E} p(u, v) \prod_{(u,v) \notin E} (1 - p(u, v))[\text{Yang and Leskovec, 2012}]. \tag{3.3}$$

where 'B' is the same as $B(V, C, M)$, and E is the set of edges in the given network. 'P' can be updated by reformulating the log-likelihood as a convex function, thus enabling the use of gradient ascent or Newton's method to obtain a global optimum for 'p'. 'B' is updated using Metropolis-Hasting [Janke, 2008] algorithm.

The flexibility of AGM is in its ability to detect non-overlapping, overlapping and nested communities, see figure (3.4). However, it is unable to handle very large dataset.

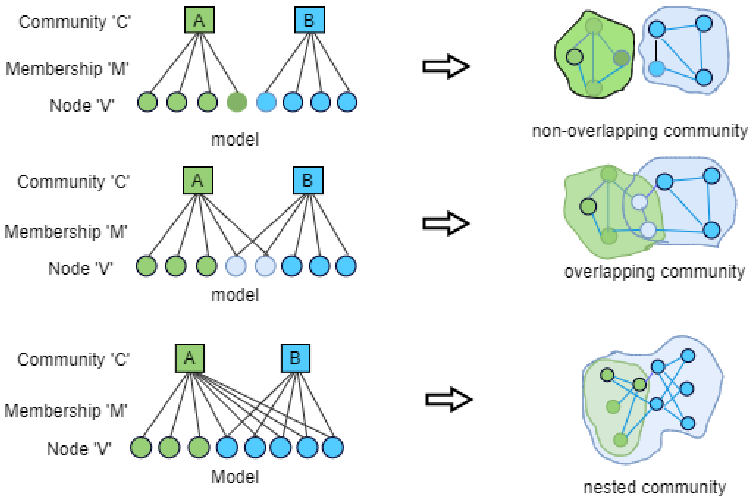


Figure 3.4: AGM flexible community representation

3.2.2 Cluster Affiliation Graph Model for Big Networks (BigClam)

In [Yang and Leskovec, 2013] Yang and Leskovec present the BigClam "an overlapping community detection technique that scales to large networks of millions of nodes and edges" The BigClam is built on AGM [Yang and Leskovec, 2012], but instead of modelling an intra-community connection probability where communities are detected through "fitting" (which is so hard), the BigClam models a community with weights associated with each community member. This model assigns a non-negative weight to each edge of a bipartite graph; the strength of this weight defines the degree of affiliation.

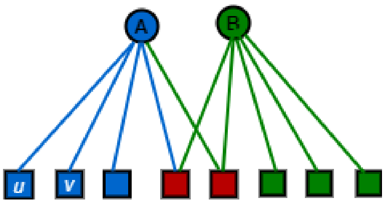


Figure 3.5: Community Affiliation Network

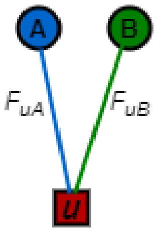


Figure 3.6: Weight for affiliation
[Yang and Leskovec, 2013]

Given the graphs as shown in figure (3.5, 3.6), here a membership vector F is assigned to each node, and this membership vector tells us how strongly a node belongs to the communities. F_{uA} is the membership strength of node u to community A. If, $F_{uA} = 0$ then u and A has no membership relationship between them.

For example, each community in the network, “A”, has a probability of having an edge with the nodes (u, v) independently. This independent edge probability of community A with node (u, v) is proportional to the product of strengths and is given by:

$$P_A(u, v) = 1 - \exp(-F_{uA} \cdot F_{vA})\text{[Yang and Leskovec, 2013]}. \tag{3.4}$$

There also exists a probability of at least one common community C having an edge with the nodes (u, v) given by:

$$P(u, v) = 1 - \prod_C (1 - P_C(u, v)), \tag{3.5a}$$

$$= 1 - \exp(\sum_C F_{uc} \cdot F_{vc}), \tag{3.5b}$$

$$= 1 - \exp(-F_u \cdot F_v^T)\text{[Yang and Leskovec, 2013]}. \tag{3.5c}$$

Equation (3.5c) holds on the assumption that when two nodes share the same community, there is an independent, non-zero chance of connection between the two nodes. Consequently, the more the number of common communities between two nodes, the higher their chances of connecting. Also, the higher the edge weight, the higher the chances of connecting.

BigClam, just like the AGM, also assumes a ε-community with a very small chance of connection ε for nodes that does not share a community in common.

Given a network $G(V, E)$, how does BigClam detect network communities?

The BigClam does community detection by finding the most likely affiliation factor matrix $\hat{F}$ to the network G by maximizing the likelihood $l(F) = logP(G|F)$ of G:

$$\hat{F} = \arg\max_{F \geq 0} L(F), \tag{3.6}$$

where

$$l(F) = \sum_{(u,v) \in E} log(1 - exp(-F_u \cdot F_v^T)) - \sum_{(u,v) \notin E} (F_u \cdot F_v^T)\text{[Yang and Leskovec, 2013]}. \tag{3.7}$$

To obtain the gradient for a single row F_u of F:

$$\bigtriangledown l(F_u) = \sum_{v \in N(u)} F_v(\frac{\exp(-F_u F_v^T)}{1-\exp(-F_u F_v^T)}) - \sum_{v \notin N(u)} (F_v)[\text{Yang and Leskovec, 2013}]. \tag{3.8}$$

where $N(u)$ is a set of neighbours of u under G.

Computing $l(F)$ and $\bigtriangledown l(F_u)$ will take a linear time (every node in the network needs to be iterated through) of $O(N)$ which is not scalable for a network with node greater than a million. In order to reduce the complexity to, $O(|N(u)|)$ we compute as in equation(3.9) which will significantly speed up the computation speed for large networks.

$$\sum_{v \notin N(u)} F_v = (\sum_v F_v - F_u - \sum_{v \in N(u)} F_v)[\text{Yang and Leskovec, 2013}]. \tag{3.9}$$

3.2.3 Ego-Splitting Framework

In [Epasto et al., 2017] Alessandro et al. present the Ego-splitting framework, which is based on the concept of Ego-net. Ego-networks are subnetworks that are centred on a certain node. The Ego-net of node E (see figure (3.7)) is defined as the subgraph of E (i.e., the Ego-net minus ego E)

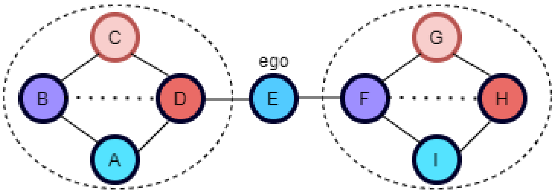


Figure 3.7: Example of an Ego-net of a node

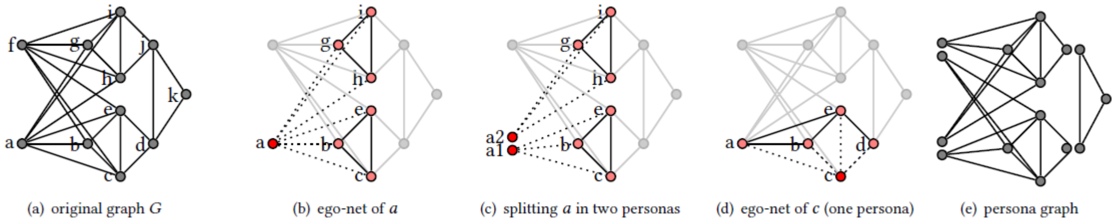
The Ego-Splitting framework works as follows:

- Create the ego-net of each node
- Split up each ego-net with a non-overlapping clustering algorithm A1
- Create the persona Graph
- Split up the persona Graph with a non-overlapping clustering algorithm A2
- Get the overlapping clusters of the original graph

In a more formal definition, the Ego-net framework requires two clustering algorithms A^l (a local clustering algorithm) and A^g (a global clustering algorithm). For a node u

with an ego-net G_u. The following five steps are required for the ego-splitting algorithm to create an overlapping cluster of a graph:

- Step 1: Given a node u, partition the ego-net of u using a local clustering algorithm. such that $A^l(G_u) = N_u^1, N_u^2, \ldots, N_u^{t_u}$
- Step 2: Create a set $V^{'}$ of personas. Each node corresponds to t_u personas denoted as $u_i, i = 1, \ldots, t_u$
- Step 3: Add edges between personas. For each edge $(u, v) \in V$, find \$(u$_i, v_j$ such that $u \in N_u^i$ and $v \in N_v^j$. Add (ui, vj) to persona edge $E^{'}$.
- Step 4: Using the global clustering algorithm on persona graph $G^{'}$ to obtain clusters $C^{'} \cdot A^l(G_u) = C^{'}$.
- Step 5: Creation of communities in G. For each cluster $c^{'} \in C^{'}$, create the associated community $c \subseteq V$ formed by the corresponding nodes of $V : c(c)^{'}) = u \in V | \exists i s.t. u_i \in c^{'}$. Output is $C = \{c(c^{'}) | c^{'} \in C^{'}\}$.



(a) original graph G (b) ego-net of a (c) splitting a in two personas (d) ego-net of c (one persona) (e) persona graph

Figure 3.8: Clustering the ego-nets, splitting the ego and building the persona graph [Epasto et al., 2017]

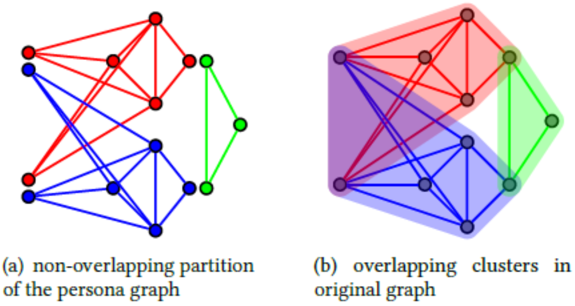

(a) non-overlapping partition of the persona graph (b) overlapping clusters in original graph

Figure 3.9: Clustering the persona graph [Epasto et al., 2017]

Figure (3.8) illustrates the execution steps of the framework using connected components as clustering algorithm. In figure (3.8b), the ego-net of ‘a’ is obtained and partitioned into two clusters. In figure (3.8c), for each of the two clusters detected in figure (3.8b), a persona is created. For example, a_1 is associated with the nodes b, c, e. The same is done for node c in figure (3.8d) but this time only one cluster is detected, and one persona c_1 associated to nodes a, b, e, d is created. After this is done for all the nodes (in parallel), then a persona graph as shown in figure (3.8e) is created with edges as in the original graph. In figure (3.9a), a global clustering algorithm is applied on the persona graph. Finally, in figure (3.9b), the clusters defined on the personas are mapped to the overlapping clusters on the original nodes.

Calculating the computational complexity of the ego-nets could be done in the order of, $O(mn)$ which is computationally expensive. An optimal algorithm can create all ego-nets in time, $O(m^{\frac{3}{2}})$ with the upper bound depending on the number of triangles in the graph.

The beauty of the ego-splitting framework is that it does not rely on any specific clustering algorithms. Any appropriate algorithm could be used depending on the requirements. The framework can also handle weighted and/or directed graphs, provided that the local and global algorithms also support it.

3.2.4 Summary

Table 3.1: Summary of Overlapping community detection methods reviewed

Algorithms	Idea	Advantages	Disadvantages
Community-Affiliation Graph Model (AGM)	Detects communities through fitting method	It can be used to express a variety of community structures: Non-overlapping, Overlapping, and Nested communities	It is unable to handle very large dataset
Cluster-Affiliation Graph Model for Big Networks (BigClam)	Detects communities by assigning a non-negative weight to each edge of a bipartite graph, the strength of this weight defines the degree of affiliation.	Can detect communities in large networks	Computing $l(F)$ and $\bigtriangledown l(F_u)$ will take a linear time of $O(N)$ which is not scalable for a network with node greater than a million
Ego-Splitting Framework	Detects communities based on the concept of ego-net analysis	Can make use of any clustering algorithm. Can handle weighted and/or directed graphs. Can handle graphs with tens of billions of edges.	Computational cost usually depends on the clustering algorithm used.

[Yang and Leskovec, 2012] [Yang and Leskovec, 2013] [Epasto et al., 2017]

3.3 Graph mining tools

There are several tools available for graph mining. This section discusses a review of some tools currently used for graph mining.

3.3.1 Gephi

Gephi is a modular and extensible open-source Java visualisation application built on the NetBeans platform. It is suitable for analysing complex networks and is mainly used for social network analysis. Gephi uses OpenGL, a cross-language, cross-platform API for the 3D rendering of large networks in real-time and speeds up the exploration. The features of Gephi which make it a good tool for graph visualisation are:

- Real-time visualization: Gephi is powered by OpenGL, which makes it a rich tool for real-time visualization of graphs.
- Layout: It has a good layout palette which allows users to change layout settings to increase user experience.

- Metrics: Gephi has a good statistics and metrics framework suitable for social network analysis and scale-free networks.

- Networks over time: In Gephi, users can manipulate the embedded timeline, allowing them to visualize how the network evolves.

- Dynamic filtering: With Gephi interactive user interface, users can filter the network in real-time and gives them the ability to select nodes and/or edges based on the network structure.

- Input/Output: Different graph file formats such as CSV and relational databases import are supported by Gephi.

- Extensibility: One of Gephi's menu items is "Plugins", which makes it possible to import a wide range of community-built plugins to extend its functionalities [Bastian et al., 2009].

3.3.2 NetworkX

NetworkX is a python language package for exploring and analysing networks and network algorithms. Data structures for representation of many types of networks or graphs. Its flexibility makes it ideal for representing networks and large real-world graphs found in many fields. Its good features are:

- Ability to convert graphs to several formats

- Ability to find subgraphs, cliques, k-cores

- Draw networks in 2D and 3D

- Ability to convert graphs to several formats

- Explore adjacency, degree, diameter, radius, centre, betweenness, etc [NetworkX,].

3.3.3 Neo4j

Neo4j is the world's leading graph database implemented in Java and accessible from software written in other languages using the Cypher query language through a transactional HTTP endpoint or Bolt protocol. It scales billions of nodes and connections in a system. Its first version was released in 2007 and is available in a GPLv3-licensed open-source "community edition", with advanced and enterprise versions accessible under AGPLv3.

Neo4j uses edges, nodes, or attributes as its data structure. In version 2.0, indexing was added to Cypher with the introduction of schemas [Neo4j,].

3.3.4 Cytoscape

Cytoscape is an open-source visualization data mining tool initially developed at the Institute of Systems Biology in Seattle in 2002 to visualize molecular interaction networks and integrate them with gene expression profiles and other state data. The biomedical research community started using this first, and it is helpful to understand the gene and protein interaction in biology. Its core features are:

- Support for many standard network and annotation file formats like SIF, GML, XGMML, BioPAX, GraphML etc
- Ability to connect to external public databases and import network and annotation data
- Supports RESTful API for programmatic access
- Support for different image export, including PDF, PS, SVG, PNG, JPEG, and BMP files.
- Filter used to select subsets of nodes and/or interactions based on current data [cytoscape,].

3.3.5 Apache Spark

Apache Spark is a unified analytics engine first released in 2014 with Apache 2.0 licence. It is suitable for a large-scale data process, ETL functions, machine learning, and cluster computing. The main advantages of Apache Spark are:

- Speed: It has great performance for both streaming and batch data
- Multiple language support (e.g., Python, Scala, Java, etc)
- Easy to use
- Good cluster management
- Supports for RDDs
- Fault tolerance [Apache Spark, a]

Figure 3.10 is a representation of the Apache Spark ecosystem. It shows that Apache spark supports languages such as Scala, Python, and R. Also, it has support for libraries as listed in the figure. Of interest to this study is the GraphFrames, a graph processing library for Spark that succeeded GraphX. GraphX uses GraphFrame, which provides DataFrame-based graphs[Needham and Hodler, 2019].

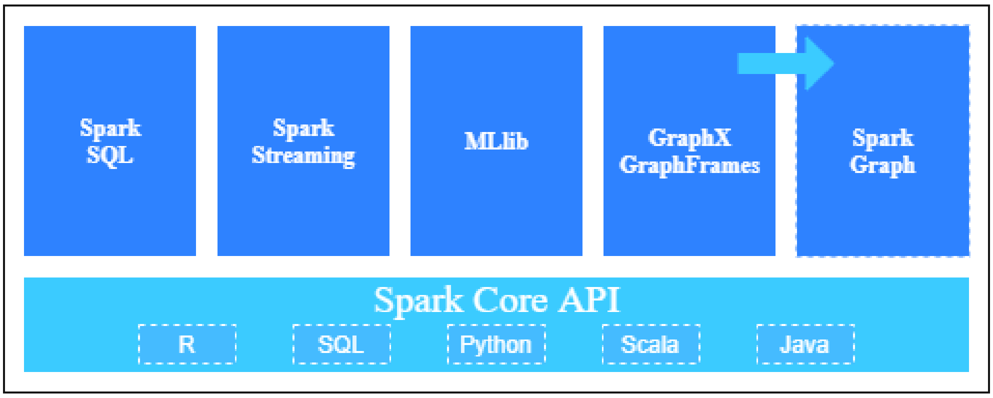


Figure 3.10: Apache Spark Ecosystem
[Needham and Hodler, 2019]

3.3.6 Summary

Table 3.2: Summary of reviewed tools

Tools	Uses	Advantages	Disadvantages
Gephi	Analysis and visualization of large networks graphs	Extremely fast Simple Modular Easy data import	Few visual glitches Navigation of the graph can be improved
Neo4j	For visualization of data, Data analysis and pattern detection	Fast retrieval and ease of representation of connected graphs, It uses simple and powerful data model Ease of representation of relationships without joins or indexes	It does not support Sharding.
Cystoscope	Visualize and analyze large networks in life sciences	Support of community-provided applications that enhance it and offer additional functionality, It is free, open source, works on all operating systems with Java	Requires high memory consumption while working with big network, Analysis on other network that are non-life science may require other tools
Apache spark	Large-scale data processing, ETL functions, Machine learning, Cluster computing	Fast speed for streaming and batch data processing, Good cluster management, Supports for RDDs, Support for multiple languages	It does not have a file management system of its own, No support for real-time processing, Fewer algorithms, Small files issue

[Bastian et al., 2009] [NetworkX,] [Neo4j,] [cytoscape,] [Apache Spark, a] [Needham and Hodler, 2019]

Chapter 4

Analysis

4.1 Problem Definition

We are presented with the ISEBEL project, a digital archive of stories from belief legends found in three well known digital collections by Evald Tang Kristensen from Denmark (Etkspace), Richard Wossidlo from Mecklenburg (Wossidia) and several collectors and narrators from the Netherlands (Verhaalenbank). These databases are made up of stories originating from different sources. Stories are composed by various authors and spread across many slips of papers in the database; stories in these papers have facts and contents which are related, and also, an author may contribute related ideas to the content of different papers. Thus, these databases are stocked with various stories, with some stories so interrelated that comprehensive information cannot be found in a single slit; stories have to be compiled from slits all across the archive. Furthermore, the facts and contents of the stories may be related either by the stories themselves or by the author and co-authorship.

Therefore, we are faced with the challenge of modelling and implementing a framework to present the users of ISEBEL search system with a more relevant search result that effectively gives results of stories, related stories, possible stories that will be interconnected in the feature and a way to visualize stories interrelatedness through the use of data mining techniques.

4.2 Data Extraction Process

One of the critical steps toward successful data mining is data availability. In this research, the XML story data is harvested from the three databases (Wossidia, Verhaalenbank and Etkspace) using the Open Archives Initiative Protocol for Metadata

Harvesting (OAI-PMH). The OAI-PMH defines an open interface for the exchange of metadata. It has an architectural model that allows data providers to make metadata available through a well-defined protocol. The metadata exposed by the data provider enables the service providers to harvest it and then aggregate it, post-process it, and refine it to develop services that add value[oaipmh, b], see figure (4.1).

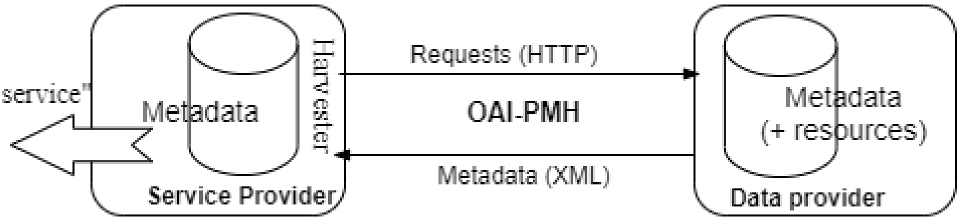


Figure 4.1: OAI-PMH Architecture
[oaipmh, b]

For this project, communication occurs between the following databases (Wossidia, Verhaalenbank, Etkspace, Icelandic and some other Scandinavian) as the data provider and the ISEBEL archive as the service provider. See the pictorial representation of the communication topology in the figure (4.2).

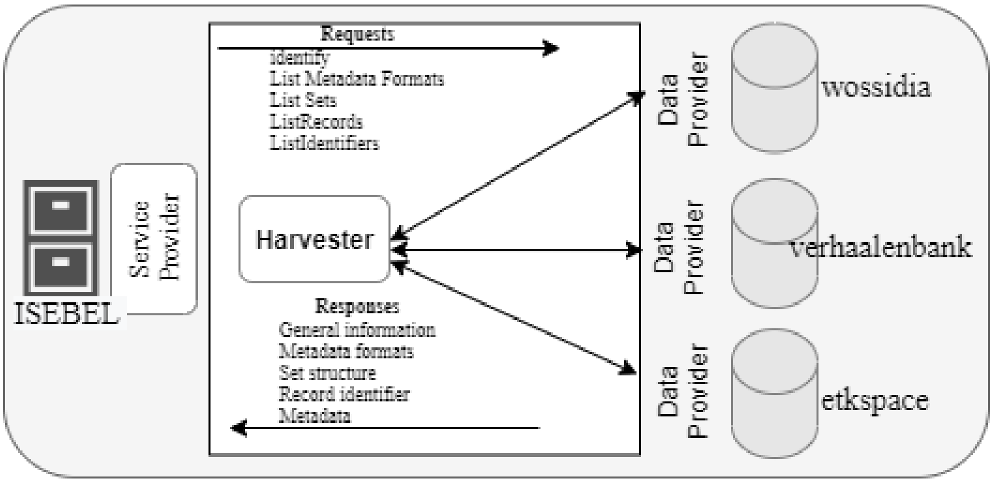


Figure 4.2: OAI-PMH Architecture for ISEBEL
[oaipmh, a]

In simple terms, the OAI-PMH has been implemented in our system, allowing access to metadata in XML format made available from different sources and used for further analysis in this project.

4.3 Dataset Analysis

To make all the sources of data that are harvested and made available in the ISEBEL archive to be realizable as graph data, a general schema has been defined, which sets the minimum requirements for the structure of the ISEBEL XML story document. Hence, all sources must use this schema to provide their stories. The schema defines all the valid elements and attributes in the XML documents. It also specifies tags that are allowed within another tag. The code snippet shows a portion of the XML document for the person roles element 4.1.

```
    <!-- Person roles. -->
    <xs:simpleType name="typePersonRole">
        <xs:restriction base="xs:string">
            <!-- Content roles. -->
            <xs:enumeration value="actor"/>
            <!-- Provenience roles. -->
            <xs:enumeration value="narrator"/>
            <xs:enumeration value="contributor"/>
            <xs:enumeration value="scholar"/>
            <xs:enumeration value="collector"/>
            <xs:enumeration value="observer"/>
            <xs:enumeration value="informant"/>
        </xs:restriction>
    </xs:simpleType>
```

Listing 4.1: Person roles code snippet

This element describes an enumeration of possible role values that a person authoring a story can have. This means that, outside the possible enumerated roles of actor, narrator, contributor, scholar, collector, observer, and informant, the element cannot accept any other value from a source for the person role. Likewise, other elements in the document, though there are still some elements that do not have restricted values, such as the element for story texts in code 4.2. Code 4.3 shows an extract of an XML story document from Wossidia where all the elements defined within the story root element have been represented to form a complete XML story document.

```
<!-- Container type for all story texts. -->
<xs:complexType name="typeContents">
    <xs:sequence>
        <xs:element name="content" type="typeContent" maxOccurs="unbounded"/>
    </xs:sequence>
</xs:complexType>
```

Listing 4.2: Story texts code snippet

```xml
<?xml version='1.0' encoding='UTF-8'?>
<isebel:story xmlns:isebel="http://www.isebel.eu/ns/isebel" xmlns:dc="http://purl.org/dc/elements/1.1/" xml:id="xmd-s001-000-002-749" xml:lang="deu">
  <dc:identifier>de.wossidia.xmd-s001-000-002-749</dc:identifier>
  <isebel:purl>https://apps.wossidia.de/webapp/run/node/XMD-S001-000-002-749</isebel:purl>
  <dc:title>[2749] grausame Herrin</dc:title>
  <dc:title xml:lang="deu"/>
  <dc:title xml:lang="nds"/>
  <isebel:contents>
    <isebel:content xml:lang="deu">Katelbogen</isebel:content>
    <isebel:content xml:lang="nds">Katelbogen</isebel:content>
  </isebel:contents>
  <isebel:places>
    <isebel:place id="1980">
      <dc:title>Neukloster</dc:title>
      <isebel:point>
        <datacite:pointLatitude xmlns:datacite="http://datacite.org/schema/kernel-4">53.86765</datacite:pointLatitude>
        <datacite:pointLongitude xmlns:datacite="http://datacite.org/schema/kernel-4">11.68002</datacite:pointLongitude>
      </isebel:point>
      <isebel:role>narration</isebel:role>
    </isebel:place>
    <isebel:place id="1331">
      <dc:title>Katelbogen</dc:title>
      <isebel:point>
        <datacite:pointLatitude xmlns:datacite="http://datacite.org/schema/kernel-4">53.84266</datacite:pointLatitude>
        <datacite:pointLongitude xmlns:datacite="http://datacite.org/schema/kernel-4">11.86807</datacite:pointLongitude>
      </isebel:point>
      <isebel:role>action</isebel:role>
    </isebel:place>
  </isebel:places>
  <isebel:persons>
    <isebel:person id="10005511">
      <isebel:name>Roggentin</isebel:name>
      <isebel:gender>Frau</isebel:gender>
      <isebel:role>narrator</isebel:role>
      <isebel:livingPlace id="1331">
        <dc:title>Katelbogen</dc:title>
        <isebel:point>
          <datacite:pointLatitude xmlns:datacite="http://datacite.org/schema/kernel-4">53.84266</datacite:pointLatitude>
          <datacite:pointLongitude xmlns:datacite="http://datacite.org/schema/kernel-4">11.86807</datacite:pointLongitude>
        </isebel:point>
      </isebel:livingPlace>
    </isebel:person>
  </isebel:persons>
  <isebel:events>
    <isebel:event id="990011391">
      <isebel:date>1915-12-27</isebel:date>
      <isebel:role>narration</isebel:role>
    </isebel:event>
  </isebel:events>
  <isebel:keywords>
    <isebel:keyword id="XMD-M030-000-001-628">eisern</isebel:keyword>
    <isebel:keyword id="XMD-M030-000-001-685">Ofen</isebel:keyword>
  </isebel:keywords>
</isebel:story>
```

Listing 4.3: Wossidia XML code snippet

Of more importance to this thesis is how elements in a schema can be related across the document to form an interconnection, which helps to relate stories across documents that eventually form the nodes and edges of graph data. A close look at the Wossidia XML document shows that some tags such as a person, event and place have an ”id”, which can be used to identify them. It means that for two stores authored by the same person or in the same place, or during the same event but represented in a different document, there exists a possibility of the two stories having the same person ID or event ID or place ID or a combination of two or maybe even the three. Thus, an edge has been created between the two-story documents, and that is how a graph is formed using different XML story documents, see illustration in figure 4.3.

Also, for elements without an identifier, such as the content tag, edges can still be formed using such elements by mining-related keywords of texts placed between the tags using a natural language processing method. However, employing the natural language processing method to obtain these fine-grain details may be above the scope of this study; thus, we will not discuss it further.

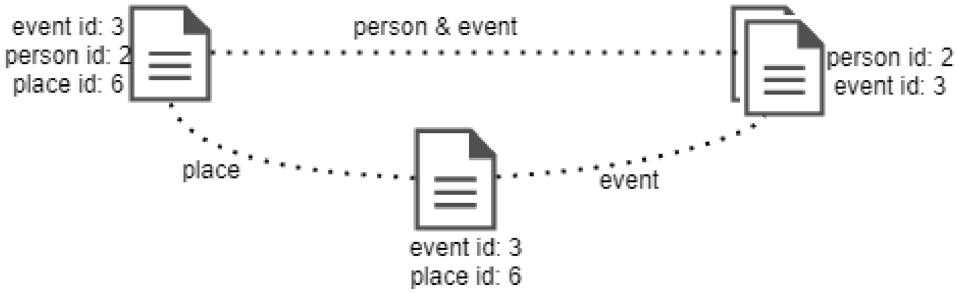


Figure 4.3: XML Story document relationship

From visual analysis of this XML document, two inferences can be drawn:

- XML story documents/story schemas are likely to represent graph nodes
- Elements attributes like person ID, event ID, place ID and content keywords are likely candidates for graph edges.

To understand and make a more informed decision about the XML story data harvested from the ISEBEL archive, these data have been further converted to a comma-separated value (CSV) file, which can be easier to process as a graph data.

4.3.1 WossiDiA data

The WossiDiA dataset is a dataset extracted from WossiDiA digital archive information system. The archive contains more than 2.5 million digital presentations of Richard Wossidlo's folklore, an ethnologist and ethnographer who, in his study, gathered stories related to the ancient customs originating from Mecklenburg between 1883 and 1939 [Schering et al., 2007]. The archive is rich with stories that narrate people's cultural heritage from different authors in different places and times.

The dataset has been extracted from the archive in XML format and converted to CSV files for this thesis. Two CSV files are presented, one containing the node details and the other containing the edge list. Table 4.1 and 4.2 shows the first ten rows of the node and edge dataset, respectively.To understand the dataset better, a comparison was made between the two tables and a Gephi file extract got by a combination of the two tables, and it can be observed that the node table has an ID, label and type which represents the nodes' ID, name and type of interconnection relationship with other nodes respectively. Matching this observation with the edge table, we notice that the edge table has a source and target, which shows the connected nodes and a label that is the attribute that connects the two nodes. In addition to understanding this node-edge relationship better, a view of code 4.4, shows that this logic follows seeing that nodes that are related in the edge tables share the same attribute value in common in the XML snippet.

Table 4.1: wossidia node list

id	label	type
1	xmd-s001-000-001-510	story
2	[1510] Werwolf	story
3	Man macht einen...	content
4	100001647	place
5	Laupin	place
6	53.25098	place
7	narration	place
8	Ernst Pegel	person male contributor
12	Lehrer	person male contributor

Table 4.2: wossidia edge list

id	source	target	label
1	1	2	title
2	1	3	content
3	1	4	place
4	4	5	title
5	4	6	pointLatitude
6	4	7	pointLongitude
7	4	8	role
8	1	9	person
11	9	12	profession
12	1	13	keyword

```
  <nodes>
    <node id="1" label="xmd-s001-000-001-510">
      <attvalues>
        <attvalue for="0" value="story" />
        <attvalue for="1" value="deu" />
      </attvalues>
    </node>
    <node id="2" label="[1510] Werwolf">
      <attvalues>
        <attvalue for="0" value="story" />
      </attvalues>
    </node>
    <node id="3" label="Man macht einen Werwolf.">
      <attvalues>
        <attvalue for="0" value="content" />
        <attvalue for="1" value="deu" />
      </attvalues>
    </node>
    <node id="4" label="100001647">
      <attvalues>
```

```
          <attvalue for="0" value="place" />
        </attvalues>
      </node>
      <node id="5" label="Laupin">
        <attvalues>
          <attvalue for="0" value="place" />
        </attvalues>
      </node>
    </nodes>
```

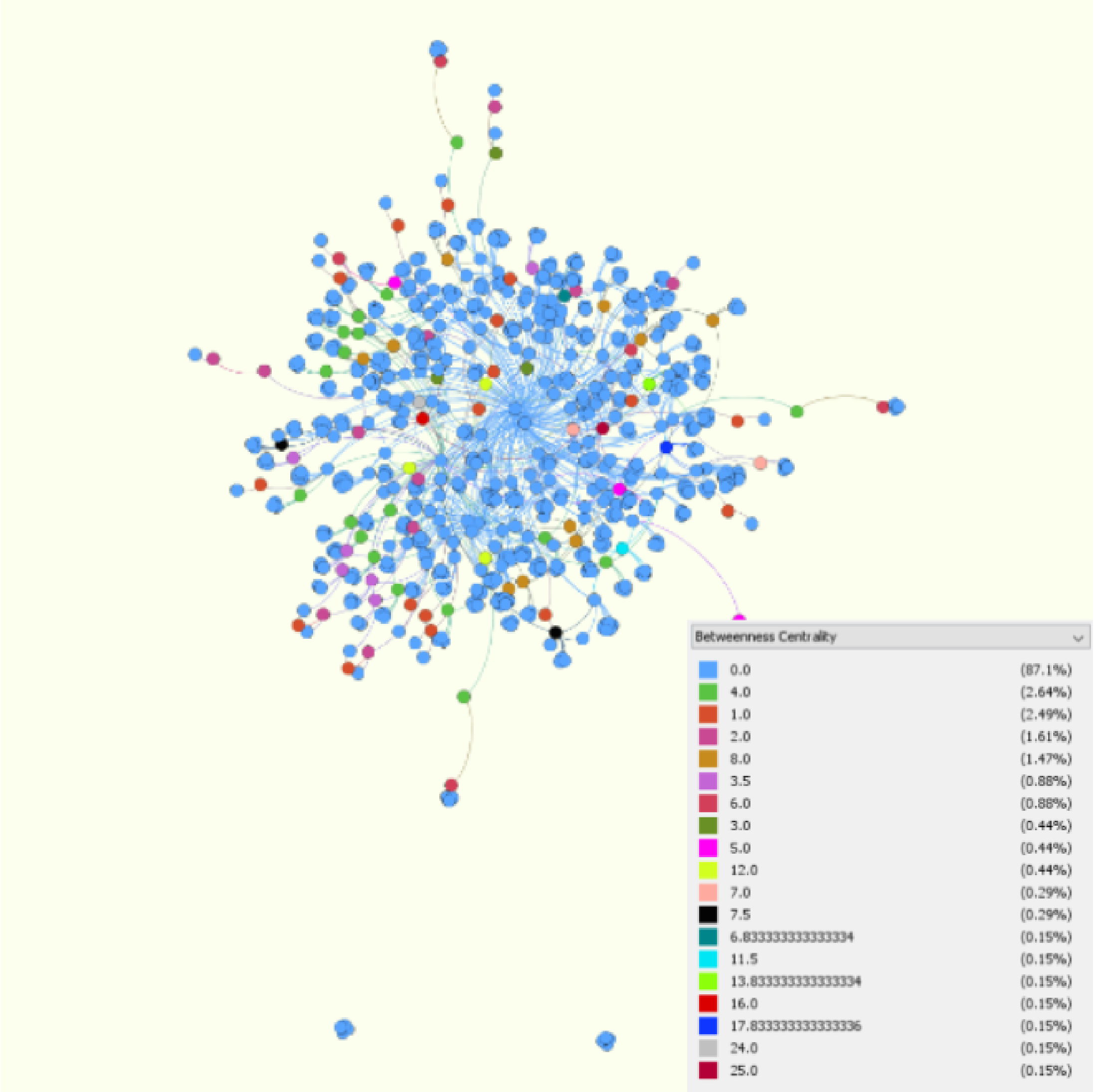
Listing 4.4: Node-Edge listing

Since the Wossidia dataset is graph data, to get an in-depth analysis of it, it is important to visualize it using a data visualization tool like Gephi. Figure 4.7 represents a network obtained when the dataset is imported into the Gephi tool.

The partitioning of this network has been done based on the betweenness centrality measure, and colour code has also been applied. Statistics obtained based on this show that most of the nodes have an interconnection; as can be seen in the attached table beside the network, 87.1 percentage of the nodes are somehow linked together; this is also evident in the visualization of the network. Other statistics obtained from the network have also been tabulated in the table 4.3

Table 4.3: WossiDia graph statistics

Graph type	Number of nodes	Number of edges	Average degree	Network diameter	Graph density	Average Path length
Directed	682	1133	1.661	3	0.002	1.288



Betweenness Centrality

■	0.0	(87.1%)
■	4.0	(2.64%)
■	1.0	(2.49%)
■	2.0	(1.61%)
■	8.0	(1.47%)
■	3.5	(0.88%)
■	6.0	(0.88%)
■	3.0	(0.44%)
■	5.0	(0.44%)
■	12.0	(0.44%)
■	7.0	(0.29%)
■	7.5	(0.29%)
■	6.833333333333334	(0.15%)
■	11.5	(0.15%)
■	13.833333333333334	(0.15%)
■	16.0	(0.15%)
■	17.833333333333336	(0.15%)
■	24.0	(0.15%)
■	25.0	(0.15%)

Figure 4.4: Gephi View of Wossidia dataset

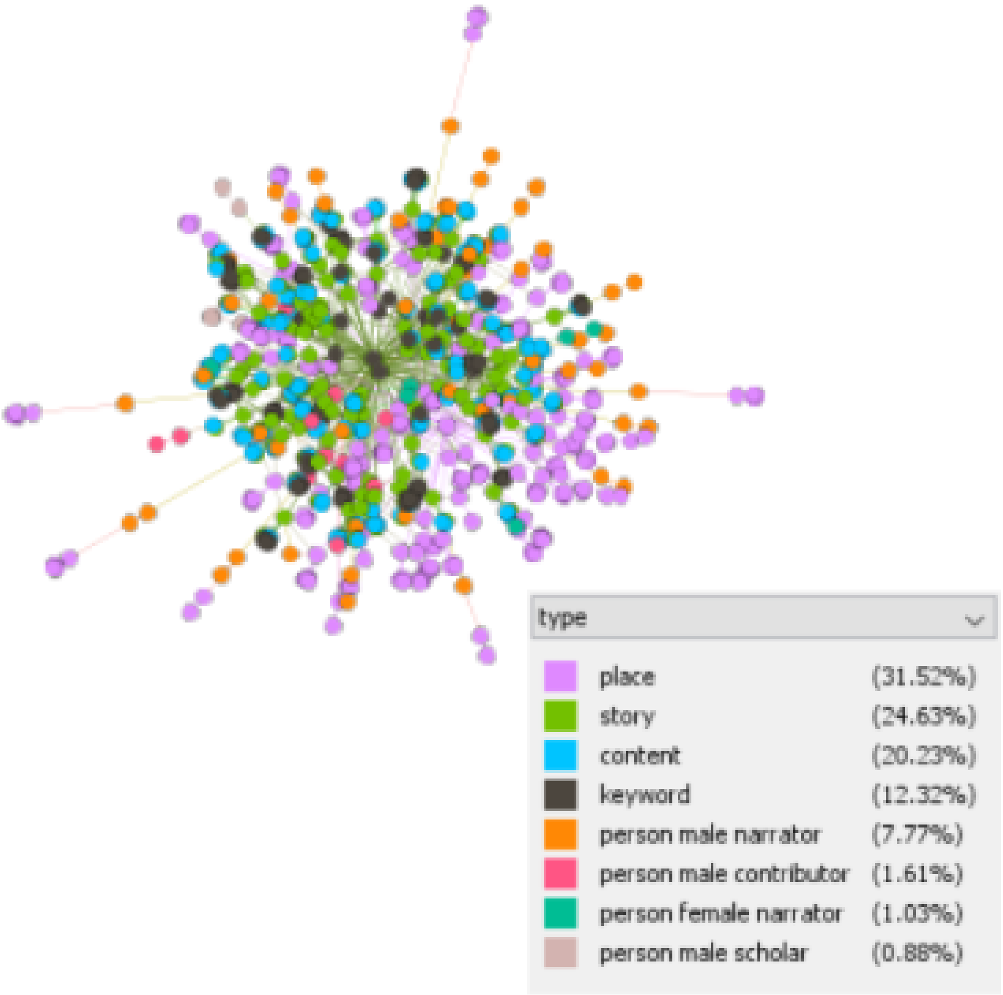

type
🟪 place (31.52%)
🟩 story (24.63%)
🟦 content (20.23%)
⬛ keyword (12.32%)
🟧 person male narrator (7.77%)
🟥 person male contributor (1.61%)
🟩 person female narrator (1.03%)
🟫 person male scholar (0.88%)

Figure 4.5: Gephi View of Wossidia dataset based on keyword

4.3.2 Verhalenbank

Table 4.4: Verhalenbank graph statistics

Graph type	Number of nodes	Number of edges	Average degree	Network diameter	Graph density	Average Path length
Directed	1653	2944	1.781	2	0.001	1.172

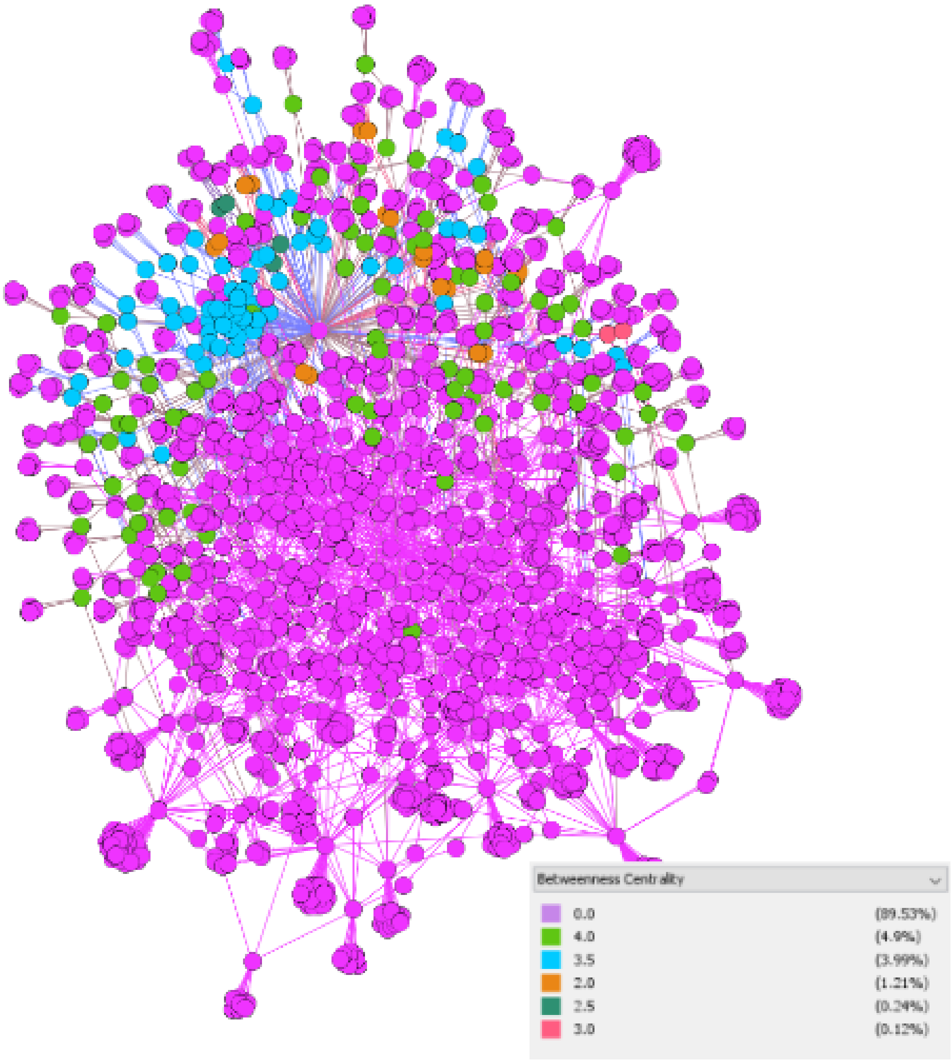


Figure 4.6: Gephi View of Verhalenbank dataset

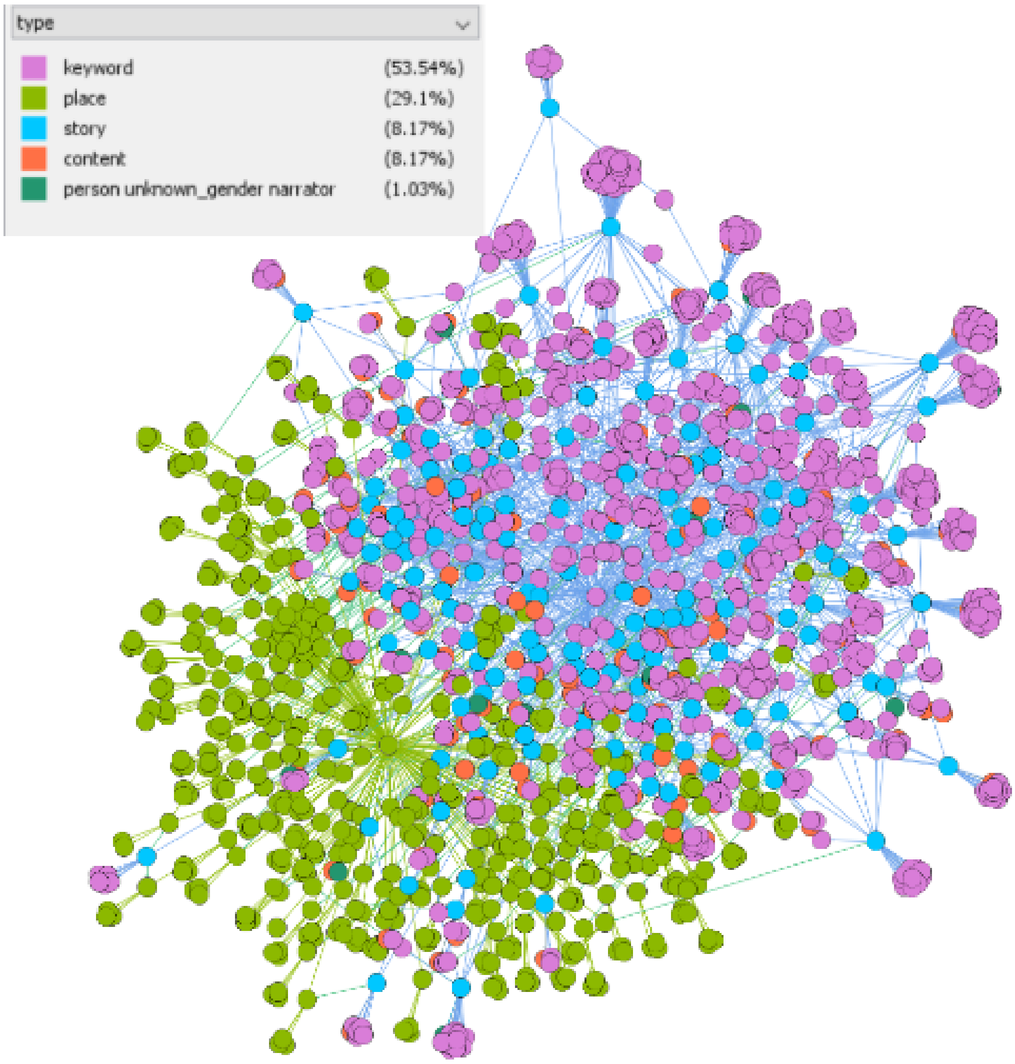


Figure 4.7: Gephi View of Verhalenbank dataset based on keyword

4.3.3 Summary of Data-set

Table 4.5 shows the summary of the statistics gathered from all available story data-set

Table 4.5: Graph statistics

Data-set	Graph type	No. of nodes	No. of edges	Avg. degree	Network diameter	Graph density	Avg. Path length
Verhalenbank	Directed	1653	2944	1.781	2	0.001	1.172
Wossidia	Directed	682	1133	1.661	3	0.002	1.288
wossidia-werewolf	Directed	682	1000	1.466	3	0.002	1.281
wossidia-witches	Directed	639	1000	1.565	3	0.002	1.249

Chapter 5

Proposed Framework

This research aims to design and implement a framework capable of detecting overlapping communities in the ISEBEL dataset. Considering that a well-designed system will give a better performance, we have taken the time to provide a detailed design of our proposed framework. At the end of this chapter, the reader should understand the architecture of our proposed framework and the road map for the implementation of the framework.

5.1 Framework Design

The proposed framework presented in this section has been built based on the already existing overlapping community detection algorithm discussed in the chapter 3 while leveraging on the good quality of the GraphX module in the Spark parallel processing engine. This novel framework for detecting overlapping communities in the ISEBEL dataset consists of four major execution phases. As shown in the framework architectural diagram in figure 5.1, the four execution steps are:

- File processing
- Spark processing
- Community detection
- Performance evaluation

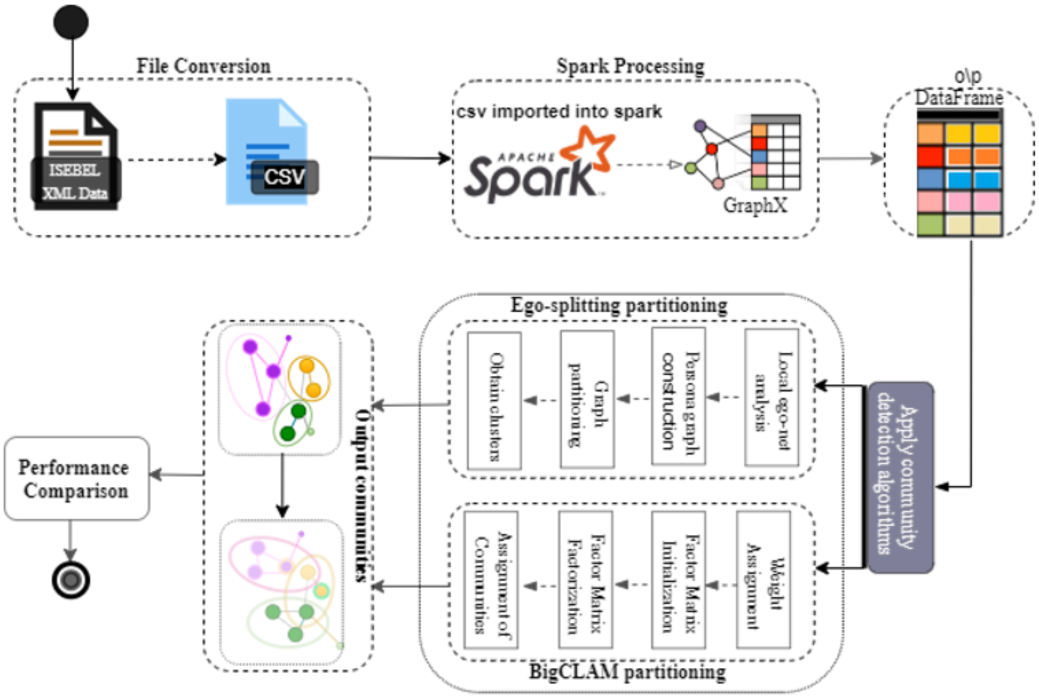


Figure 5.1: Framework Architecture

As mentioned earlier in chapter 4, the XML story data for the ISEBEL archive is harvested from different databases using OAI-PMH, but mining graph data is easier done when the data is in CSV or JSON format; that is why the first step of our proposed algorithm is to convert the XML data to a more suitable format (CSV), the output of this phase is fed to the Spark processing step.

The output of the file processing step is imported into the Spark processing step, where the CSV file is converted to graph data using GraphFrames, a graph processing library for spark. GraphFrame: DataFrame-based graphs is a distributed graph processing framework on top of the Spark DataFrames. This step can output a graph network suitable for clustering using a graph clustering algorithm.

The community detection step applies a community detection algorithm to detect the non-overlapping and overlapping communities. In this step, we have used the BigCLAM algorithm introduced in chapter 3. First, the algorithms generate the non-overlapping clusters and then find the overlapping candidate node. In the final step of the algorithm, we try to measure the algorithm's performance on our dataset. This performance measure will be based on the computation time of the algorithm.

5.2 Implementation

The BIGCLAM algorithm for detecting communities in this framework requires four execution steps, as shown in the figure 5.2. The Factor Matrix Initialization step is an essential step in which a matrix called the Factor matrix with $N \times k$ dimension is initialized. N— represents the number of nodes in the community, and k-represents the number of communities. The k value used for this implementation has been generated using three different methods:

- Random generation by defining $k = 8, 50, 100 \ldots$
- Using some node property (for example, node type)
- Parallel community seed set generation algorithm (example Girven-Newman algorithm on Spark)

Thus, the initially proposed algorithm in figure 5.1 has been modified to figure 5.2. Other execution steps of the BIGCLAM algorithm remains as discussed in the literature review.

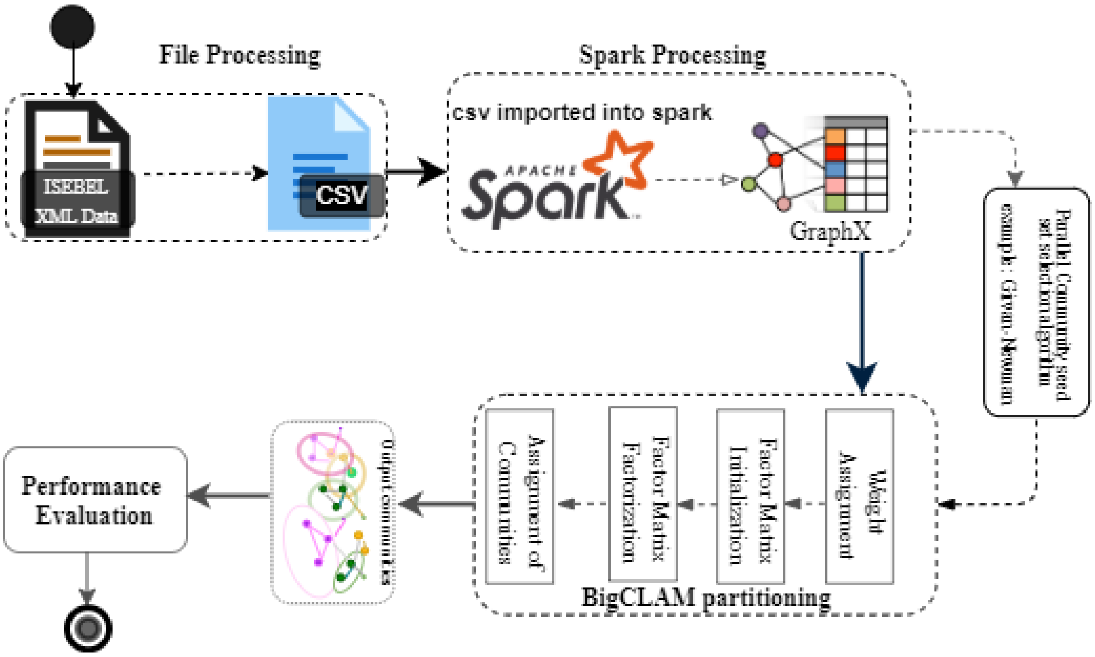
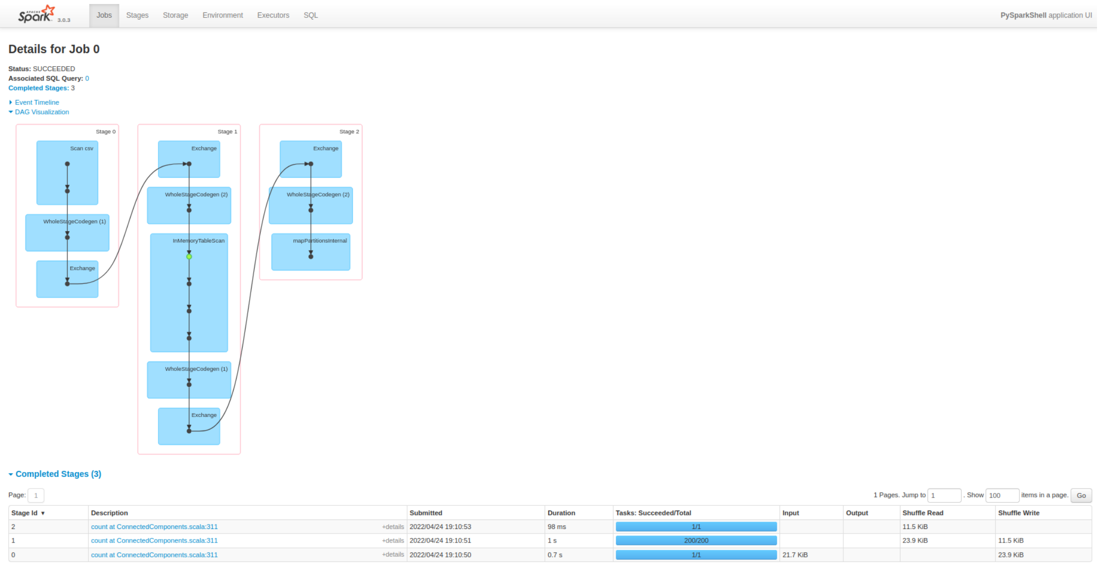


Figure 5.2: Framework Architecture with parallel community seed selection

This experiment has been implemented on an Intel Core i7-2600K CPU and 32 GB of RAM, with Apache Spark as the analytical engine.

The main benefit of using Apache Spark as the analytical engine is its capacity to write jobs in multiple steps and its ability to execute jobs in parallel. Multiple threads can use a single SparkContenxt instance to submit multiple Spark jobs, which may or may not be executed in parallel, depending on the available CPUs to handle the task. With enough CPUs, multiple spark jobs will be running concurrently.

Two types of job scheduling are possible in Spark: FIFO and FAIR scheduling mode. In a FIFO scheduling mode, jobs are divided into “stages” (for example. map and reduce phases). The first job has higher precedence in the available resources over the second job. The resources are distributed equally to all running jobs for a FAIR scheduling mode [Apache Spark, b].



Details for Job 0

Status: SUCCEEDED
Associated SQL Query: 0
Completed Stages: 3

▸ Event Timeline
▾ DAG Visualization

▾ Completed Stages (3)

Page: 1

1 Pages. Jump to 1 . Show 100 items in a page. Go

Stage Id ▾	Description	Submitted	Duration	Tasks: Succeeded/Total	Input	Output	Shuffle Read	Shuffle Write
2	count at ConnectedComponents.scala:311 +details	2022/04/24 19:10:53	98 ms	1/1			11.5 KiB	
1	count at ConnectedComponents.scala:311 +details	2022/04/24 19:10:51	1 s	200/200			23.9 KiB	11.5 KiB
0	count at ConnectedComponents.scala:311 +details	2022/04/24 19:10:50	0.7 s	1/1	21.7 KiB			23.9 KiB

Figure 5.3: Spark UI showing how jobs are executed

Figure 5.3 shows a presentation of one of the executions of a block of code implemented in this work in Spark UI (a web interface used to monitor Spark applications during execution). The figure explains various physical execution units of Spark application: Job, Stage, and Task.

Chapter 6

Result and evaluation

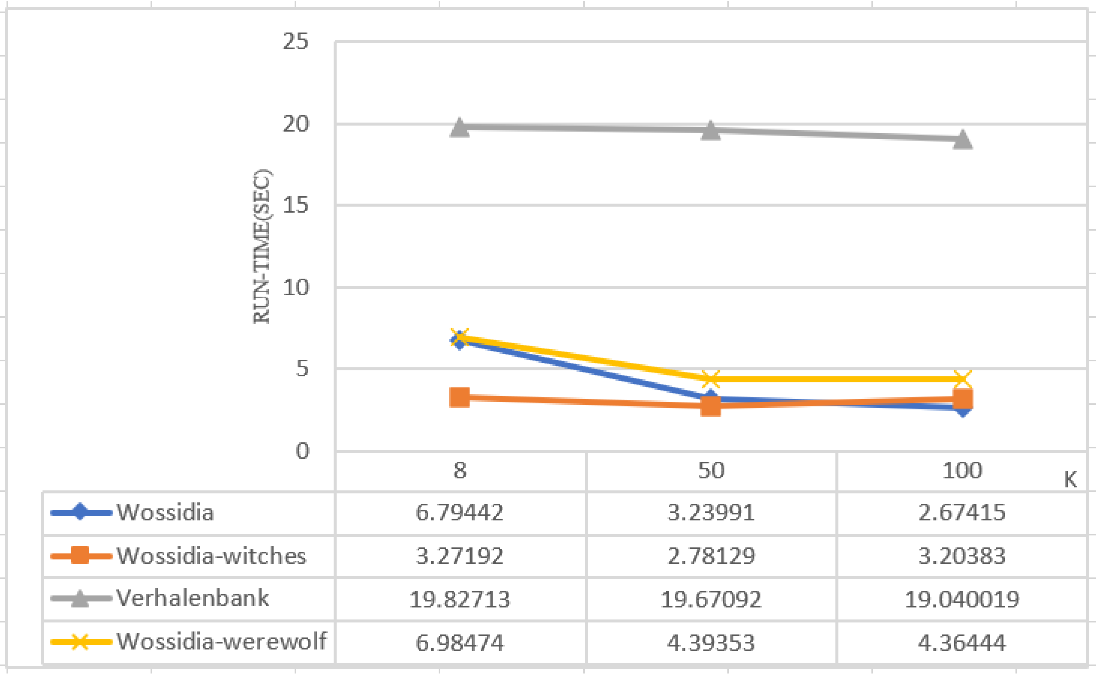
6.1 Evaluation of BigClam result

The framework developed based on the BigClam algorithm on Apache spark using Python programming language has been tested on four ISEBEL real-world datasets, with the summary as shown in table 4.5.

To evaluate the performance of a community detection algorithm, it is always common to evaluate its ability to unravel so-called "ground truth" communities. Ground truth communities are easier to obtain in synthetic networks where such communities are explicitly defined. However, there are no explicitly defined communities for real-world networks such as the ISEBEL network; instead, the ground-truth communities are obtained using some discrete-valued node properties, arbitrarily chosen random values and using a parallel community seed set generation algorithm.

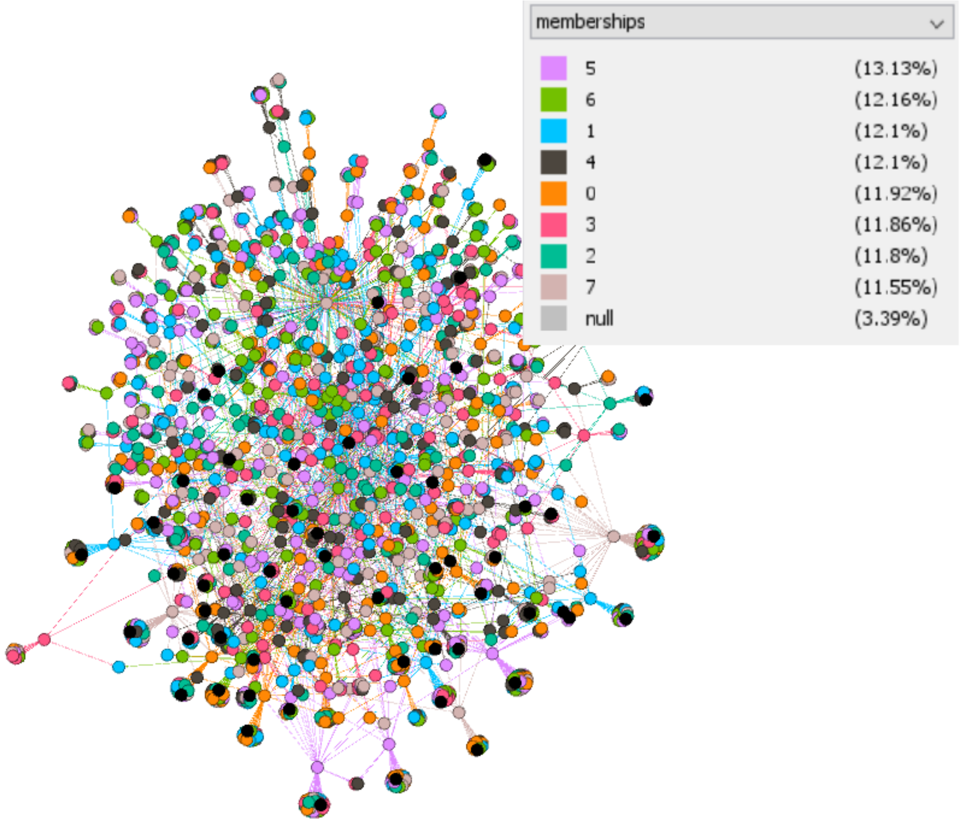
6.2 Result Analysis

The performance of the algorithm has been tested for different numbers of k (number of communities) on the various datasets presented in this research; the plot in figure 6.1 shows the result for k = 8, 50, 100 while figure 6.2 is a Gephi view of communities obtained using Verhalenbank data-set for k=8. Furthermore, the graph shows the decrease in run-time of the algorithm as the number of k increases.



	8	50	100
Wossidia	6.79442	3.23991	2.67415
Wossidia-witches	3.27192	2.78129	3.20383
Verhalenbank	19.82713	19.67092	19.040019
Wossidia-werewolf	6.98474	4.39353	4.36444

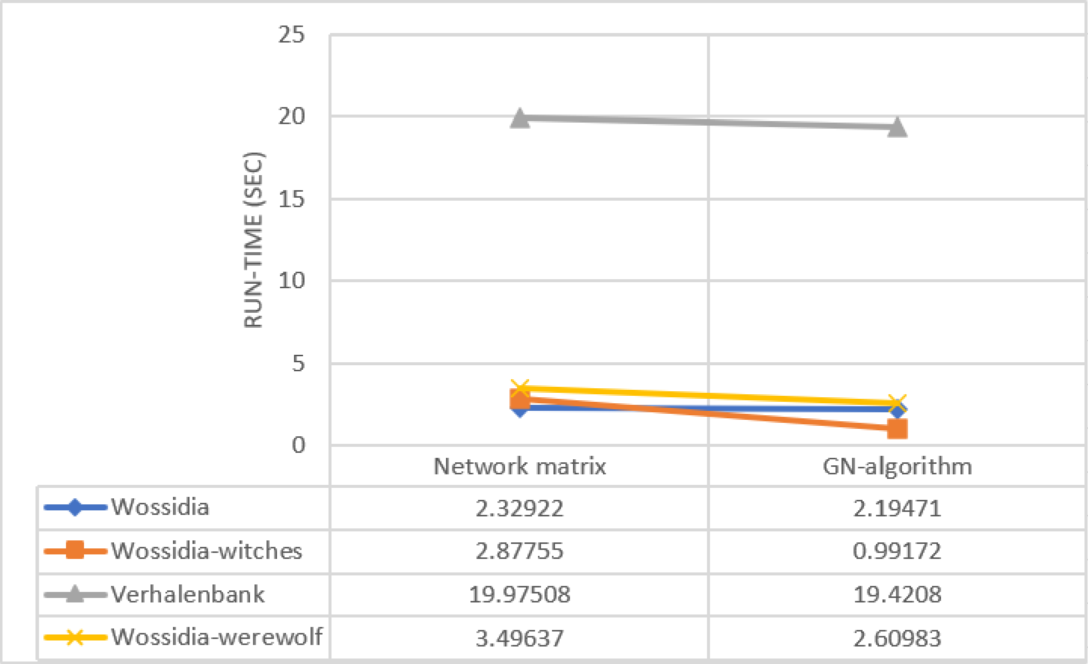
Figure 6.1: Plot of run-time using different k values


memberships

■	5	(13.13%)
■	6	(12.16%)
■	1	(12.1%)
■	4	(12.1%)
■	0	(11.92%)
■	3	(11.86%)
■	2	(11.8%)
■	7	(11.55%)
■	null	(3.39%)

Figure 6.2: Gephi View of Verhalenbank communities for K=8

To verify the statement in figure 6.1 more, a plot of the run-time of the algorithm against k with k-values equal to the Network matrix (node properties) and the number of communities obtained from the Girvan-Newman algorithm for each of the datasets is presented in figure 6.3. The following values were obtained for k given that $k =$ the network node matrix: Wossida-29, Wossidia-witches-27, Verhalenbank-118, Wossidia-werewolf-8 and the following values were obtained for k given that $k =$ number of communities obtained by the Girvan-Newman algorithm; Wossidia-144, Wossidia-witches-126, Verhalenbank-308,Wossidia-werewolf-128. The plot of figure 6.3 still shows a decrease in the run-time of the algorithm as the value of k increases.



	Network matrix	GN-algorithm
◆ Wossidia	2.32922	2.19471
■ Wossidia-witches	2.87755	0.99172
▲ Verhalenbank	19.97508	19.4208
× Wossidia-werewolf	3.49637	2.60983

Figure 6.3: Plot of run-time using Network matrix and girvan newman algorithm as ground-truth

Chapter 7

Conclusion

In this thesis, overlapping community detection in ISEBEL, three overlapping community detection methods have been studied, and we have developed a framework based on BigClam on Apache Spark for detecting overlapping communities in ISEBEL. Furthermore, the framework's performance has been evaluated based on its run-time for different community sizes.

Based on the evaluation of the implemented framework, we can say an inverse relationship exists between the assumed number of communities and the run-time of the BigClam algorithm on Apache Spark.

7.1 Contribution

Having thoroughly defined the problem of detecting overlapping communities in ISEBEL, we have, through this thesis, made the following contributions:

- Review of the state of the arts for detecting non-overlapping and overlapping communities.
- Review of state of the arts in graph mining techniques and tools.
- Analysis of ISEBEL data-set using graph mining tools.
- Designed a framework for detecting overlapping communities in ISEBEL.
- Python Implementation of BigClam framework on Apache spark for detecting overlapping communities in ISEBEL.
- This framework has been evaluated based on some pre-defined values of communities derived as follows:

- Arbitrarily choosing some community values (for example, K=8,50,100…).
- Selecting the number of community values based on some network node matrix.
- Determining the number of communities based on the number of communities defined by the Girven-Newmann algorithm.

7.2 Challanges

While implementing this framework, some challenges were encountered, especially in working with Spark and Hadoop; here, we discuss some problems and how to resolve them.

Small File Problem in HDFS

Hadoop faces many performance issues if the files being processed are extremely smaller than the block size. By default, the block size of HDFS is 128 MB. When files are processed in HDFS, each file will have to occupy one block even if its size is less than the default HDFS block size. As a result, if you have many small files of sizes like 150 bytes, 120 bytes etc.; you are not utilizing your full block size capacity as each of these small files will have to occupy 128 MB each, and this will make the HDFS memory to fill up and cause lots of performance issue. Performance issues will also arise while processing these files. For example, applying a map task to them, each map task will have to process a block, so if there are one million small files, there will also be one million map tasks processing very little inputs which impose booking overhead [Aggarwal et al., 2021].

In this thesis, the small file issue was encountered while trying to execute the connected components' algorithm. This algorithm was used to get the neighbours of each node in a graph, and for each node, it has to store its neighbours in a separate small file. Processing these small files slowed down the speed of computing to up to 2 hours for graphs with nodes up to 1000. This problem was solved by eliminating the process which generated the small file issue, in this case, replacing the connected component algorithm with another neighbourhood search algorithm.

Error Handling in Spark

The datasets contain some null value rows which cause errors when loaded to the Spark system for processing. Spark provides different options to handle such errors. The method used to handle the errors caused by null rows during implementation is the **DROPMALFORMED mode**. In this mode, Spark drops any null records in the file when it is imported. Listing B.3 is the code block that shows how these null values were dropped in Spark while importing the records.

7.3 Future work

A way to determine the number of communities k has always been an issue in implementing the BigClam algorithm for a real-world network. In this thesis, we have used three methods to determine k, of which the technique using the Girven-Newman algorithm seems to be closer to the actual values of the communities in the network.

I propose a future work that evaluates other algorithms' performance, such as the Louvain algorithm, to determine the value of k.

Bibliography

[Aggarwal, 2010] Aggarwal, C. C. (2010). *Graph Clustering*, pages 459–467. Springer US, Boston, MA.

[Aggarwal et al., 2021] Aggarwal, R., Verma, J., and Siwach, M. (2021). Small files' problem in hadoop: A systematic literature review. *Journal of King Saud University - Computer and Information Sciences*.

[Al Hasan and Zaki, 2010] Al Hasan, M. and Zaki, M. J. (2010). Link prediction in social networks. *Department of Computer Science, Rensselaer Polytechnic Institute, Troy, USA*.

[Apache Spark, a] Apache Spark. https://spark.apache.org/.

[Apache Spark, b] Apache Spark. Job Scheduling, https://spark.apache.org/docs/latest/job-scheduling.html.

[Bastian et al., 2009] Bastian, M., Heymann, S., and Jacomy, M. (2009). Gephi:an open source software for exploring and manipulating networks.

[Borgwardt and Kriegel, 2005] Borgwardt, K. and Kriegel, H. (2005). Shortest-path kernels on graphs. In *Fifth IEEE International Conference on Data Mining (ICDM'05)*, pages 8 pp.–.

[Borgwardt et al., 2005] Borgwardt, K. M., Ong, C. S., Schönauer, S., Vishwanathan, S. V. N., Smola, A. J., and Kriegel, H. (2005). Protein function prediction via graph kernels. In *Proceedings Thirteenth International Conference on Intelligent Systems for Molecular Biology 2005, Detroit, MI, USA, 25-29 June 2005*, pages 47–56.

[Callut et al., 2008] Callut, J., Françoisse, K., Saerens, M., and Dupont, P. (2008). Semi-supervised classification from discriminative random walks. In Daelemans, W., Goethals, B., and Morik, K., editors, *Machine Learning and Knowledge Discovery in Databases*, pages 162–177, Berlin, Heidelberg. Springer Berlin Heidelberg.

[cytoscape,] cytoscape. https://cytoscape.org/.

[Dhillon et al., 2005] Dhillon, I., Guan, Y., and Kulis, B. (2005). A fast kernel-based multilevel algorithm for graph clustering. In *Proceedings of the Eleventh ACM SIGKDD International Conference on Knowledge Discovery in Data Mining*, KDD '05, page 629–634. Association for Computing Machinery.

[Dhiman and Jain, 2016] Dhiman, A. and Jain, S. (2016). Optimizing frequent subgraph mining for single large graph. *Procedia Computer Science*, 89:378–385.

[Diane J Cook, 2006] Diane J Cook, L. B. H. (2006). Mining graph data.

[Epasto et al., 2017] Epasto, A., Lattanzi, S., and Paes Leme, R. (2017). Ego-splitting framework: From non-overlapping to overlapping clusters. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 145–154, New York, NY, USA. Association for Computing Machinery.

[Fonseca, 2003] Fonseca, Y. C. F. (2003). A bipartite graph co-clustering approach to ontology mapping. In *Proceedings of the Workshop on Semantic Web Technologies for Searching and Retrieving Scientific Data. Colocated with the Second International Semantic Web Conference (ISWC-03), CEUR-WS. org*, page .

[Inokuchi et al., 2000] Inokuchi, A., Washio, T., and Motoda, H. (2000). An apriori-based algorithm for mining frequent substructures from graph data. In Zighed, D. A., Komorowski, J., and Żytkow, J., editors, *Principles of Data Mining and Knowledge Discovery*, pages 13–23, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Janke, 2008] Janke, W. (2008). Monte carlo methods in classical statistical physics. In Fehske, H., Schneider, R., and Weiße, A., editors, *Computational Many-Particle Physics*, pages 79–140, Berlin, Heidelberg. Springer Berlin Heidelberg.

[Kumar et al., 2000] Kumar, R., Raghavan, P., Rajagopalan, S., Sivakumar, D., Tompkins, A., and Upfal, E. (2000). The web as a graph. In *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, page 1–10, New York, NY, USA. Association for Computing Machinery.

[Lee et al., 2018] Lee, J. B., Rossi, R., and Kong, X. (2018). Graph classification using structural attention. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery amp; Data Mining*, KDD '18, page 1666–1674, New York, NY, USA. Association for Computing Machinery.

[Leskovec et al., 2020] Leskovec, J., Rajaraman, A., and Ullman, J. D. (2020). *Mining of Massive Datasets*. Cambridge University Press, 3 edition.

[Li et al., 2016] Li, C., Guo, X., and Mei, Q. (2016). Deepgraph: Graph structure predicts network growth. *CoRR*, abs/1610.06251.

[Needham and Hodler, 2019] Needham, M. and Hodler, A. (2019). *Graph Algorithms: Practical Examples in Apache Spark and Neo4j*. O'Reilly Media.

[Neo4j,] Neo4j. https://neo4j.com/product/.

[NetworkX,] NetworkX. https://networkx.org.

[Niepert et al., 2016] Niepert, M., Ahmed, M., and Kutzkov, K. (2016). Learning convolutional neural networks for graphs. *CoRR*, abs/1605.05273.

[oaipmh, a] oaipmh. https://intechweb.wordpress.com/2010/11/03/intech-supports-the-open-archives-initiative-protocol/.

[oaipmh, b] oaipmh. http://www.openarchives.org/pmh/.

[Salem et al., 2019] Salem, R. K., Moneim, W. T. A., and Hassan, M. (2019). Graph mining techniques for graph clustering: starting point.

[Schering et al., 2007] Schering, A. C., Bruder, I., Schmitt, C., Meyer, H., and Heuer, A. (2007). Towards a digital archive for handwritten paper slips with ethnological contents. In *Proceedings of the 10th International Conference on Asian Digital Libraries: Looking Back 10 Years and Forging New Frontiers*, ICADL'07, page 61–64, Berlin, Heidelberg. Springer-Verlag.

[Shervashidze et al., 2009] Shervashidze, N., Vishwanathan, S., Petri, T., Mehlhorn, K., and Borgwardt, K. (2009). Efficient graphlet kernels for large graph comparison. In *Artificial intelligence and statistics*, pages 488–495. PMLR.

[Yang et al., 2010] Yang, B., you Liu, D., and Liu, J. (2010). Discovering communities from social networks: Methodologies and applications. In *Handbook of Social Network Technologies*.

[Yang and Leskovec, 2012] Yang, J. and Leskovec, J. (2012). Community-affiliation graph model for overlapping network community detection. In *2012 IEEE 12th International Conference on Data Mining*, pages 1170–1175.

[Yang and Leskovec, 2013] Yang, J. and Leskovec, J. (2013). Overlapping community detection at scale: A nonnegative matrix factorization approach. In ,, WSDM '13, page 587–596. Association for Computing Machinery.

[Yu and Shi, 2003] Yu and Shi (2003). Proceedings ninth ieee international conference on computer vision. In *Computer Vision, IEEE International Conference on*, volume 3, page i, Los Alamitos, CA, USA. IEEE Computer Society.

Appendix A

Abbreviations

ISEBEL - Intelligent Search Engine for Belief Legends

FSM – Frequency Subgraph Mining

AGM - Apriori-based approach algorithm

CNNs – Convolutional Neural Networks

D-walks – Discriminative Random Walks

MC – Markov Chain

AGM – Community-Affiliation Graph Model

BIGCLAM - Cluster Affiliation Graph Model for Big Networks

ETL – Extraction, Transformation, and Load

RDD - Resilient Distributed Dataset

XML – Extensible Markup Language

OAI-PMH – Open Archives Initiative Protocol for Metadata Harvesting

ID – Identity

CSV – Comma-Separated Value

JSON – JavaScript Object Notation

CPU – Central Processing Unit

FIFO – First In First Out

UI – User Interface

HDFS - Hadoop Distributed File System

k - Number of Communities

Appendix B

Code Documentation

This code documentation only highlights the important parts, the *k* value has been left out for the user of this documentation to decide the process to use and generate *k* for the network under examination.

```
import pyspark
from pyspark.sql import *
from pyspark import SparkContext, SparkConf, StorageLevel
from pyspark.sql.types import *
from graphframes import *
from pyspark.sql.functions import collect_set, expr
from tqdm import tqdm
import random
import community
import numpy as np
from typing import Dict
```

Listing B.1: Import packages

```
sc = pyspark.SparkContext("local[*]")
spark = SparkSession \
    .builder\
    .appName(’BigClam-ISEBEL’)\
    .getOrCreate()
SparkContext.setSystemProperty(’spark.executor.memory’, ’8g’)
SparkContext.setSystemProperty(’spark.driver.memory’, ’8g’)
SparkContext.setSystemProperty(’spark.sql.shuffle.partitions’, ’4’)
ss = SparkSession(sc)
```

Listing B.2: Set-up spark session

```
def create_graph_frames():
    """
    Import nodes and edge csv files and create graphframes with them
    """
    vertices_fields = [
        StructField("id",IntegerType(), True),
        StructField("label",StringType(), True),
        StructField("type",StringType(), True)
    ]
    vertices = (spark.read.option("mode","DROPMALFORMED")
                .option("columnNameOfCorruptRecord","corrupt_record")
                .csv("nodes.csv",header=True,schema=StructType(vertices_fields)))

    edge_fields = [
        StructField("id",IntegerType(), True),
        StructField("src",StringType(), True),
        StructField("dst",StringType(), True),
        StructField("label",StringType(), True)
    ]
    edges = (spark.read.option("mode","DROPMALFORMED")
             .option("columnNameOfCorruptRecord","corrupt_record")
             .csv("edges.csv",header=True,schema=StructType(edge_fields)))

    return GraphFrame(vertices, edges)
```

Listing B.3: Import nodes and edge csv files and create graphframes with them

```
    def process(entry):
        revisedEntries= entry[0].split(’,’)
        return (revisedEntries[0], revisedEntries[1])

    def generate_community_users(user_business_map, filter_threshold):
        nearby_users_map = {}
        users = user_business_map.keys()

        for u1 in users:
            related_users = set()
            for u2 in users:
                if u1 != u2:
                    u1_businesses = set(user_business_map.get(u1))
                    u2_businesses = set(user_business_map.get(u2))
                    common_businesses = u1_businesses.intersection(u2_businesses)

                    if len(common_businesses) >= filter_threshold:
                        related_users.add(u2)
            if len(related_users) > 0:
                nearby_users_map.update({u1:related_users})

        return nearby_users_map
```

Listing B.4: Generate community users

```
    def get_neighbors():
        input_file_path = "edges.csv"
        filter_threshold = 1
        user_businessRdd = sc.textFile(input_file_path)\
                        .map(lambda entry: entry.split('\n'))\
                        .map(lambda entry: process(entry))
        headers = user_businessRdd.take(1)
        finalRdd = user_businessRdd.filter(lambda entry: entry[0] != headers ↩
             [0][0]).persist()
        user_business_map = finalRdd\
                .groupByKey()\
                .mapValues(lambda entry: list(set(entry)))\
                .collectAsMap()
        nearby_users_map = generate_community_users(user_business_map,  ↩
             filter_threshold)
        return nearby_users_map

    neighbors = get_neighbors()
```

Listing B.5: Get node neighbors

```
    def bfs(graph, node):
        visited = []
        queue = [node]
        while queue:
            node = queue.pop(0)
            if node not in visited:
                visited.append(node)
                try:
                    neighbours = graph[node]
                except KeyError as e:
                    continue
                for neighbour in neighbours:
                    queue.append(neighbour)
        return visited
```

Listing B.6: Get each node neighbors with bfs

```
class BigClamISEBEL(object):
    def __init__(
        self,
        dimensions: int = k,
        iterations: int = 50,
        learning_rate: int = 0.005,
        seed: int = 42,
    ):
        self.dimensions = dimensions
        self.iterations = iterations
        self.learning_rate = learning_rate
        self.seed = seed

    def _initialize_features(self, number_of_nodes):
        self._embedding = np.random.uniform(0, 1, (number_of_nodes, self.dimensions))
        self._global_features = np.sum(self._embedding, axis=0)

    def _calculate_gradient(self, node_feature, neb_features):
        raw_scores = node_feature.dot(neb_features.T)
        raw_scores = np.clip(raw_scores, -15, 15)
        scores = np.exp(-raw_scores) / (1 - np.exp(-raw_scores))
        scores = scores.reshape(-1, 1)
        neb_grad = np.sum(scores * neb_features, axis=0)
        without_grad = (
            self._global_features - node_feature - np.sum(neb_features, axis=0)
        )
        grad = neb_grad - without_grad
        return grad

    def _do_updates(self, node, gradient, node_feature):
        self._embedding[node] = self._embedding[node] + self.learning_rate * gradient
        self._embedding[node] = np.clip(self._embedding[node], 0.00001, 10)
        self._global_features = (
            self._global_features - node_feature + self._embedding[node]
        )

    def get_memberships(self) -> Dict[int, int]:
        indices = np.argmax(self._embedding, axis=1)
        memberships = {i: membership for i, membership in enumerate(indices)}
        return memberships

    def get_embedding(self) -> np.array:
        embedding = self._embedding
        return embedding

    def get_graph_nodes(self,graph):
        nodes = graph.vertices.rdd.map(lambda x: x.id).collect()
        self.graph_nodes = nodes

    def neighbours(self,graph):
        connected = graph.connectedComponents()
        connected.persist(StorageLevel.DISK_ONLY)
        group = connected.select("*").groupby("component").agg(
                collect_set(’id’).alias(’nodes’)).sort("component")
        self.group = group

    def get_neighbours(self,nodeid):
        group = self.group
        node_1 = group.select(’nodes’).where("component="’’+nodeid+’’)
        node_2 = node_1.rdd.map(lambda x: x.nodes).collect()
        return node_2[0]

    def fit(self, graph):
        self.graph = graph
        self.get_graph_nodes(graph)
        number_of_nodes = graph.vertices.count()
        self._initialize_features(number_of_nodes)
        nodes = [node for node in self.graph_nodes]
        for i in range(self.iterations):
            random.shuffle(nodes)
            for node in nodes:
                try:
                    nebs = [neb for neb in (bfs(neighbors,str(node)))]
                    neb_features = self._embedding[nebs, :]
                    node_feature = self._embedding[node, :]
                except IndexError as e:
                    continue
                gradient = self._calculate_gradient(node_feature, neb_features)
                self._do_updates(node, gradient, node_feature)
```

Listing B.7: Class to compute BigClam algorithm

```
big = BigClamISEBEL()
graph = create_graph_frames()
big.fit(graph)

membership = big.get_memberships()

print(membership)
```

Listing B.8: Output result

```
import numpy as np
import csv

nodes = graph.vertices.rdd.map(lambda x: x.id).collect()
```

```
edges = graph.edges.select("id").rdd.map(lambda x: x.id).collect()

memberships = [v for k, v in membership.items()]
nodes = [node for node in nodes]
edges = [edge for edge in edges]

dictA = dict(zip(nodes,memberships))

with open(’result.csv’, ’w’, newline=’’) as csvfile:
    header_key = [’id’, ’memberships’]
    new_val = csv.DictWriter(csvfile, fieldnames=header_key)

    new_val.writeheader()
    for new_k in dictA:
        new_val.writerow({’id’: new_k, ’memberships’: dictA[new_k]})
```

Listing B.9: Output result to CSV

```
sc.stop()
```

Listing B.10: Stop spark session