Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Matrix-Based Analysis of Multiplex Graphs

Scientific Students' Association Report

Author:

Petra Várhegyi

Advisor:

Gábor Szárnyas

2018

Contents

Kivonat

A gráfok és gráfalgoritmusok az informatika több területén is használatosak. Gráfalgoritmusokat alkalmaznak például a routerek hálózati forgalomirányításhoz, az operációs rendszerek folyamatok ütemezéséhez, és a logisztikában is jelentős szerepük van. Ezen túl gráfalgoritmusok segítségével számos olyan probléma megoldható, amely a tudomány több területén (pl. biológia, társadalomtudományok) is felmerülnek.

Viszonylag újabb kutatási terület a hálózatkutatás, amely a gráfok topológiai sajátosságaival foglalkozik. Gráfok jellemzésére számos metrika létezik, amelyek segítségével jobban megérthetjük a gráfreprezentáció alapjául szolgáló rendszereket is. Hagyományosan a hálózatkutatás homogén, típus nélküli gráfokkal foglalkozik, nem különbözteti meg egyes csomópontok és élek típusát. Kevésbé kutatott terület az ún. multiplex gráfok (más néven heterogén hálózatok) vizsgálata. Ezek többféle típusú élt különböztetnek meg lehetőséget adva komplexebb szemantikai kapcsolatrendszerek modellezésére és a gráfok mélyebb elemzésére.

A metrikák egy csoportja a gráf klaszterezettségét írja le, amelyhez háromszögek keresésére van szükség. Ez típus nélküli gráfok esetén is számításigényes művelet, amely nem skálázódik jól, típusok bevezetése pedig még komplexebbé teszi a számításokat. Ennek egy lehetséges megoldása az lehet, ha a gráfok szomszédossági mátrixát használjuk és ezeken definiáljuk a metrikákat.

Dolgozatomban célom, hogy a szomszédossági mátrixokat felhasználva hatékonyabbá tegyem néhány számításigényes multiplex metrika számítását. Ehhez implementáltam őket a gráfok szomszédossági mátrixán mátrixműveleteket végezve, majd oknyomozói újságírók által közzétett adathalmazokból (pl. Panama-iratok) származó gráfokon kiszámíttattam őket.

Abstract

Graphs and graph algorithms are used in many areas of computer science. Applications of graph algorithms include routing in computer networks, scheduling processes in operating systems, and they also play a major role in logistics planning. Furthermore, they solve several problems that appear in other fields such as biology and social sciences.

Network science, which is a relatively recent area of research, targets the in-depth topological analysis of graphs. Several metrics are defined to characterize graphs and to provide a better understanding of the underlying systems represented by these graphs. Traditionally, network science studies exclusively homogeneous, untyped graphs and does not distinguish between different types of nodes and edges. There is significantly less research that involves multiplex graphs (also known as edge-labelled graphs, heterogeneous networks). These graphs can contain different types of edges, which makes it possible to represent more complex systems of semantic connections and to conduct a deeper analysis of the given graph.

A class of metrics aim to capture the clusteredness of graphs, which necessitates counting the triangles in the graph. This is a resource-intensive operation and does not scale well even in case of homogeneous graphs, and introducing types brings along additional complexity. One possible solution to this problem is to use the adjacency matrix representation of the graphs and define metrics using matrix operations.

In this report, my goal is to make the computation of some multiplex metrics more efficient. For this, I implemented them using matrix operations on the adjacency matrices of these graphs and evaluated them on graph datasets published by investigative journalists (e.g. the Panama papers).

Chapter 1

Introduction

Graphs are ubiquitous in many aspects of our lives today. For an example, we should look no further than the internet, which uses graph algorithms in order to connect users and transmit data between any two of its endpoints. Our social connections, the public transportation system, the power grid, and even our brains can all be represented with graphs. Therefore, understanding these systems and having techniques for analyzing them has huge potential benefits. By being able to describe real-life networks accurately, we can understand how they emerge and use the techniques to conduct predictive analysis on them, e.g. forecast how communities form in social networks.

In the traditional approach of *network science*, systems are modelled as homogeneous networks, consisting of a single node and edge type. However, real-life networks are often more complex than this, as they tend to have different types of nodes and edges. Therefore, a new body of research models systems as *multiplex graphs* and attempts to develop techniques to analyze them. As this field is relatively new, there are no out-of-the-box tools to carry out such analyses. Additionally, implementing such a tool poses a non-trivial challenge as some of the metrics defined in multiplex networks have a high computational complexity and are difficult to scale for large graphs.

During our work, we selected some promising multiplex graph metrics [32]. Then, we translated them to the language of *linear algebra*. We experimented with different *matrix* libraries and used them to implement the selected metrics. We then parallelized some of these implementations and integrated them with the Graph Analyzer framework. To conclude the work, we evaluated our implementations on a data set taken from the field of investigative journalism and analyzed the scalability of the different approaches. The structure of this report is as follows:

- In Chapter 2, we introduce the concepts that underpin our framework and present some advanced graph data models, as well as the selected data set.
- In Chapter 3, we study the typed metrics selected for analysis and translate them to the language of linear algebra.
- In Chapter 4, we present the technologies used for the implementation and how they were used to compute the previously defined metrics.
- In Chapter 5, we analyze the results of the performance experiments.
- In Chapter 6, we present scientific studies related to our work.
- Chapter 7 concludes the report and provides plans for future work.

Chapter 2

Preliminaries

2.1 Network science

In this section we introduce the basic concepts of graph theory as well as the foundations of network science. Finally, we enumerate some of the fields of science in which network science is commonly used.

2.1.1 Graphs

A graph or network, denoted G is a catalog of a system's components which are called *nodes* or *vertices*, denoted V, and the connections between them, named *links* or *edges*, denoted E. These edges can be *directed* or *undirected*. A graph is directed if all of its edges are directed and undirected if all of its edges are undirected. In both cases, an edge is defined as $(v, w) \in E$ where $v, w \in V$. In directed graphs v is the source node and w is the target node of the edge.

Each node can be characterized by the number of nodes that it is connected with, which is called its *degree*. In directed graphs, each node has an *in-degree* and *out-degree*, meaning the number of incoming and outgoing edges of the given node.

We can also calculate the average degree of the nodes of each graph, which is defined as:

$$\langle k \rangle = \frac{2e}{n}$$

where $\langle ... \rangle$ denotes the average.

For directed graphs, the average in-degree and the average out-degree are the following:

$$\langle k_{in} \rangle = \langle k_{out} \rangle = \frac{e}{n}$$

The *degree* distribution p_k is defined as the probability that a randomly selected node in the graph has degree k. As p_k is a probability, it must be normalized:

$$\sum_{k=1}^{\infty} p_k = 1$$

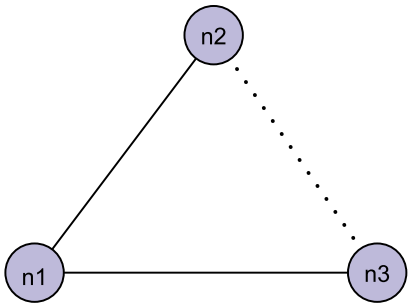

Figure 2.1: An example of a wedge

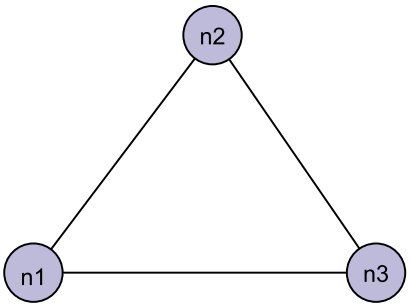

Figure 2.2: An example of triangle

Network science is a field of research which focuses on understanding the graphs or networks in real-life systems and aims to model them through various structural properties. In this report we will use the terms *network* and *graph* interchangeably.

2.1.2 Clustering coefficient

One of the important properties of graphs is how much their nodes tend to cluster together. One way to measure this in simple graphs is through their *clustering coefficient*. In order to define this metric, we first introduce two concepts: *triangles* and *wedges*.

A *wedge* is a set of three nodes of which two are connected to the third, and they may or may not be connected to each other. If they are, they also form a *triangle*. Examples are shown in Figure 2.1 and Figure 2.2, respectively.

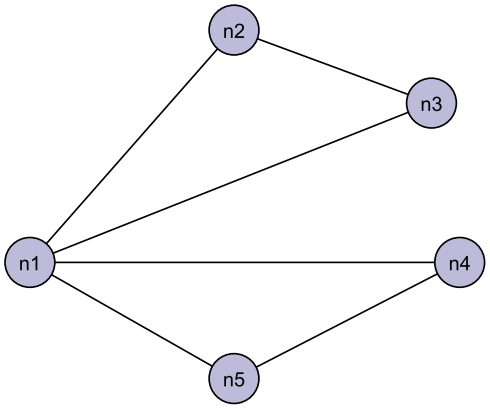

Figure 2.3: An example graph for illustrating the calculation of the clustering coefficient

In a simple untyped graph, the *global clustering coefficient* is defined in the following way:

$$\mathrm{CC} = \frac{3 \times \text{Number of triangles in the graph}}{\text{Number of wedges in the graph}},$$

where a wedge is a set of two edges that have a node in common. For the graph shown in Figure 2.3,

$$\mathrm{CC} = \frac{6}{9} = \frac{2}{3}$$

In a simple graph, the *local clustering coefficient* $\mathrm{LCC}(v)$ measures the probability for a given node in the graph, that the neighbours of a node $v \in V$ are also connected to each other. It is defined as the ratio of triangles containing a given node and the number of wedges that are centered around the node. In the untyped case it can be calculated using the following formula:

$$\mathrm{LCC}(v) = \frac{2e_v}{k_v(k_v - 1)},$$

where e_v denotes the number of connected pairs among the neighbours of v (each pair counts once, i.e. (a, b) and (b, a) count as a single pair), and k_v denotes the degree of v. For the graph in Figure 2.3,

$$\mathrm{LCC}(\mathsf{n1}) = \frac{2 \cdot 2}{4 \cdot 3} = \frac{1}{3}, \quad \mathrm{LCC}(\mathsf{n2}) = \mathrm{LCC}(\mathsf{n3}) = \mathrm{LCC}(\mathsf{n4}) = \mathrm{LCC}(\mathsf{n5}) = 1$$

2.1.3 Applications of network science

One of the most well-known examples of a network is the internet. It is an undirected graph, with machines (such as routers, switches or servers) represented as nodes and internet connections represented as links. Another common example is the WWW [14] with webpages as vertices and links from one website to another as directed edges.

Networks can also be used to model the interactions and acquaintances between people. With people represented as nodes, the links may be professional connections, friendships, relationships or family connections. Social networks can be used to map these connections and help us understand our social behaviour better.

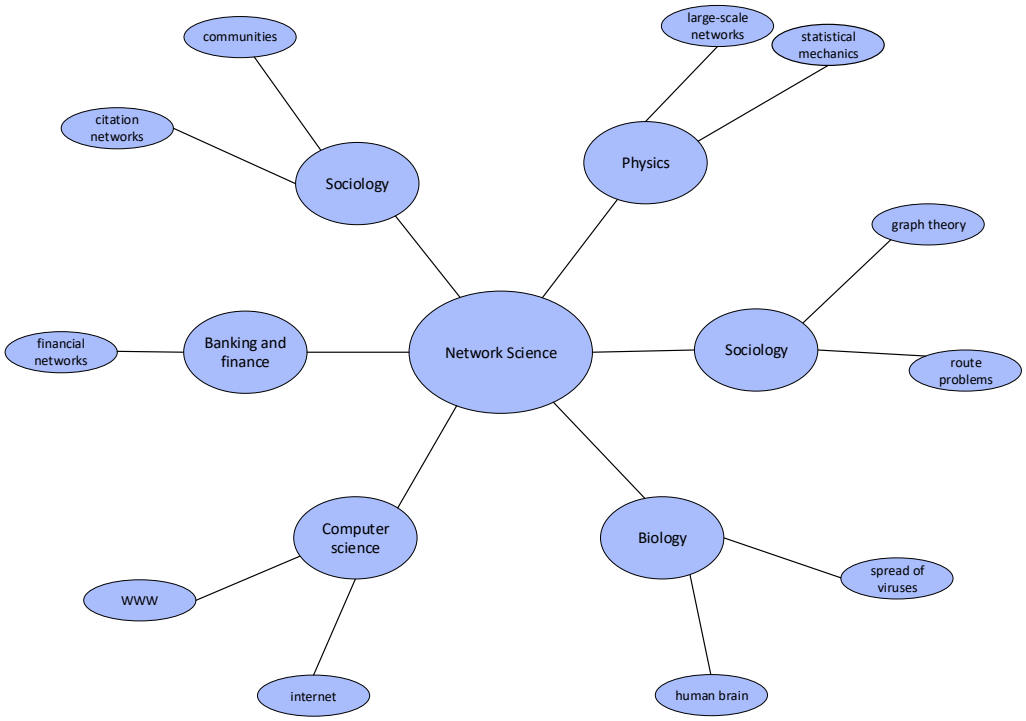

Figure 2.4: Some applications of network science

Networks are very common in biology, too. An example of this is the human brain, which consists of hundreds of billions of neurons which are connected to each other, but the

links between them have not yet been mapped. For this reason, the human brain is one of the least understood networks. Detailed maps of the human brain could allow us to understand and cure numerous brain diseases, which would be a major scientific advance.

Network science plays a significant role in fighting terrorism [6, 29], one of the biggest problems of the 21st century. Network thinking is used by various law-enforcement agencies whose goal is to act against terrorist activities. Network representations are used to disrupt the financial chains of such organizations as well as to understand the connections between members and uncover their roles and capabilities.

In organizations, management tends to traditionally rely on the official chain of command when making decisions. However, there is a growing recognition that informal networks within an organization are worth taking into consideration as well. By mapping these informal networks, and using them in management decisions, the organizations are able to work more efficiently in several cases.

2.1.4 Network topologies

In this, we introduce some of the most well-known network topologies [6] that attempt to model some of the networks that occur in real life, as well as describe the degree distribution that they follow.

Random networks

A random network is a network in which the probability of any given pair of nodes being connected is characterized by p, a probability that is uniform throughout the whole network. One of the ways such a network can be obtained is by first starting with n isolated nodes and iterating through each pair of nodes. For each pair, a random number between 0 and 1 is generated, and if this number is smaller than p, a link is created between them. This model was introduced by Gilbert [17] in 1959. Another definition, used by Erdős and Rényi [15] a year later, is that there is a given N number of nodes and a given L number of links, which are placed between the nodes randomly.

Since in an actual real-life network the number of links rarely remains the same, and most of the characteristics are easier to grasp using fixed probability, we chose to present the former model.

The degree distribution p_k of a random network follows the binomial distribution:

$$p_k = \binom{N-1}{k} p^k (1-p)^{N-1-k}.$$

Since real networks are sparse, meaning that $\langle k \rangle \ll N$ (where $\langle k \rangle$ is the average degree of the network, as defined previously), the degree distribution is well-approximated by the Poisson distribution:

$$p_k = e^{-\langle k \rangle} \frac{\langle k \rangle^k}{k!}.$$

The two distributions have a similar shape. One of the biggest advantages of using the Poisson distribution as an approximation is that in this case, the degree distribution does not depend on the size of the network, but only on the average degree of the network.

In real-life, networks often exhibit the so-called *small-world phenomenon*. This is a concept that if we choose any two people on Earth, they can be connected by a path of six

acquaintances at most. In networks, this property means that if we pick two nodes from the network, the path between them is always relatively short. This property tends to be true for random networks as if we consider one node from the network, we can say that there are $\langle k \rangle^d$ nodes at most at a distance d away. This means that for $\langle k \rangle = 1000$, which is a good estimation for the number of direct acquaintances an individual has, at a distance of 3, there are already about a billion people.

To calculate the clustering coefficient of one node in such a network, we need to estimate the expected number of links between the node's k_i neighbors. There are $k_i(k_i - 1)/2$ possible links between the neighbors of node i, so that the expected value of L_i is the following:

$$\langle L_i \rangle = p\frac{k_i(k_i - 1)}{2}.$$

where p is the characterization probability.

Using this formula and the definition of clustering coefficient, the expected clustering coefficient is

$$C_i = p = \frac{\langle k \rangle}{N}.$$

This means that for a random network, the clustering coefficient is independent of the given node's degree, and also that the bigger the network is, the smaller the local clustering coefficients of the nodes are.

By observation, the clustering coefficients are generally higher in real networks than this formula predicts. This is one of the factors that motivates the Watts-Strogatz Model.

Watts-Strogatz Model

The model proposed by Watts and Strogatz [36] is an extension of the random network model. In this model, we initially start with a ring of nodes, each node connected with their immediate and next neighbor. With a probability p each link is rewired to a randomly chosen node. The greater this probability is, the closer the network is to a random network. At the start of the generation with a $p = 0$, $\langle C \rangle = 3/4$ where p denotes the probability that a given edge is rewired compared to the starting construction and C denotes the average clustering coefficient in the graph. The clustering coefficient generally decreases with a greater p, and the values seem to be more realistic than the random network model.

Scale-Free Networks

The scale-free network model [7, 3], introduced by Barabási and Albert in 1999, addresses the observation that in real networks (unlike in random networks) it is a common phenomenon that some of the nodes have a significantly larger degree than the average in the network. These nodes are often referred to as *hubs*.

Scale-free networks follow a power-law distribution, which means the following:

$$p_k \sim k^{-\gamma}$$

where γ is the *degree exponent*. If we take the logarithm of this formula, we get

$$\ln p_k \sim \gamma \ln k.$$

From this, the formula for the degree distribution of a scale-free network is the following:

$$p_k = Ck^{-\gamma},$$

where the constant C is determined so that the normalization condition

$$\sum_{k=1}^{\infty} p_k = 1$$

holds.

2.2 Representing graphs as matrices

Graphs can be represented as matrices which are used by several graph algorithms and can be used to perform various calculations. In this section, we present the basic concepts of linear algebra and how graphs can be represented with matrices.

2.2.1 Basics of linear algebra

A matrix [26] S that has n rows and m columns and stores real numbers is denoted as:

$$S \in \mathbb{R}^{n \times m}$$

A *square matrix* is a matrix that has the same number of rows as columns i.e. $n = m$.

A *vector* is a matrix which has one column, i.e. $m = 1$.

In order to express graph operations in the language of linear algebra, we first summarize the definitions of basic matrix operations. Addition of two matrices Q and R of the same size $n \times m$, denoted with $S = Q + R$ is defined as the following:

$$S_{ij} = Q_{ij} + R_{ij}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

We define subtraction, denoted with $S = Q - R$, similarly:

$$S_{ij} = Q_{ij} - R_{ij}$$

Multiplying a matrix Q of size $n \times m$ by a scalar c, denoted with $P = c \cdot Q$ is defined as:

$$P_{ij} = c \cdot Q_{ij}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

For a matrix Q of size $n \times m$, its *transpose* is denoted with $Q^\intercal$ and is defined as a matrix of size $m \times n$ in which:

$$Q^\intercal_{ji} = Q_{ij}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$.

For matrices Q of $n \times m$ elements and R of $m \times o$ elements, the *matrix multiplication* operation (also known as the *matrix multiplication*) is denoted with $P = Q \cdot R$ and is

defined as:

$$P_{ij} = \sum_{k=1}^{m} Q_{ik} \cdot R_{kj},$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, o$. The resulting matrix P has a dimension of $n \times m$. A special case of matrix product is multiplying a matrix by a column vector of ones, which results in a vector containing the sum of each row.

The *matrix power* operation denoted A^m where $m \in \mathbb{N}$, is defined as A multiplied by itself m times. Because of the nature of matrix multiplication, for $m \geq 2$, this operation is only applicable to square matrices.

The *element-wise multiplication* (also known as the *Hadamard-product*) of two matrices Q and R both of size $n \times m$, is denoted with $H = Q \odot R$ and defined as:

$$H_{ij} = Q_{ij} \cdot R_{ij}$$

for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. We define *element-wise division*, denoted with $Q \oslash R$ similarly to element-wise multiplication.

The diagonal vector of a matrix is defined the following way. Given a square matrix M of $n \times n$ elements, function $\text{diag}^{-1}(M)$ returns a vector of n elements, containing the values of the main diagonal of M.

2.2.2 Adjacency matrices

Let G be a graph with a set of nodes V and set of edges E of sizes $|E| = e$ and $|V| = n$ respectively, that does not contain any multiple edges. If G is undirected, its adjacency matrix A is defined as a square matrix of size n where the rows and columns correspond to the nodes of the graph and:

$$a_{ij} = 1$$

if there is an edge between nodes i and j in the graph, $a_{ij} = 0$ otherwise. In this case the matrix is symmetrical, meaning that $a_{ij} = a_{ji}$ for $\forall i, j \in V$ and $A^\intercal = A$.

In case of directed graphs, the rows of A represent the source nodes and the columns correspond to the target nodes. In this case,

$$a_{ij} = 1$$

if there is a directed edge with a source node i and target node j in the graph, $a_{ij} = 0$ otherwise.

As we are generally working with simple graphs, meaning that they are undirected and do not contain any loop or multiple edges, their adjacency matrices are symmetrical and $\text{diag}^{-1}(A) = \vec{0}$ generally holds for them. They are also generally *sparse matrices* as the number of edges in these graphs are comparable to the number of nodes.

2.3 Advanced graph data models

As we saw in the previous sections, graph theory and network science with the traditional graph model approach can successfully model concepts and real-life systems. In real-life networks, however, there are concepts that these approaches are unable to grasp. As an

attempt to model these concept, we introduce two more advanced graph data models: multiplex graphs and property graphs.

2.3.1 Multiplex graphs

Multiplex graphs are graphs in which edges have types. Multiplex graphs are defined as $G = (V, E, T)$ where V denotes the set of nodes, E denotes the set of edges and T denotes the set of types. An edge can be defined with $(v, w, \alpha) \in E$, where $v, w \in V$, $\alpha \in T$.

This construction can be thought of as a graph having different layers, as shown in Figure 2.5.

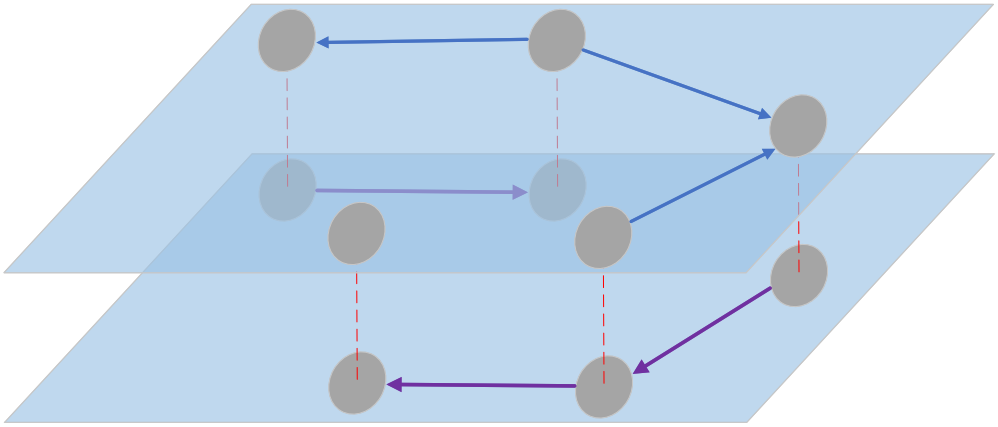

Figure 2.5: A multiplex graph with two edge types

Similarly to the traditional graph model, multiplex graphs can be directed or undirected. One of the most essential ones is the typed degree, which is defined as the number of neighbors of a given node with respect to a given type. Formally:

$$Degree\,(v, \alpha) = \left|\{w \in V \mid Cd(v, w, \alpha)\}\right|.$$

where $Cd(v, w, \alpha)$ means that v and w are connected through an edge typed α, i.e. $(v, w, \alpha) \in E$.

We can represent multiplex graphs with a set of adjacency matrices, each matrix corresponding to a type of the graph. Each of these matrices A^α is defined similarly to the untyped case:

$$a_{ij}^\alpha = 1$$

if there is an edge of type α between nodes i and j in the graph, $a_{ij}^\alpha = 0$ otherwise, where $\alpha \in T$.

2.3.2 Property graphs

Property graphs are a generalized variant of the multiplex graph model. In property graphs, similarly to the traditional graph concept, there are nodes and edges connecting them. Both edges and nodes may have properties, i.e. a set of key-value pairs to characterize them in more detail. In this model, edges are generally directed.

Property graphs can be stored in graph databases. As of 2018, Neo4j[1] is arguably the most widely used of such databases[2].

2.4 Investigative journalism

Investigative journalism has been gaining increasing attention in the recent years with the leak of millions of offshore documents, first in 2015, called the Panama Papers. This leak consisted of 11.5 million documents containing detailed financial and client-attorney information about over 200 000 offshore entities. Some of these documents date back to the 1970s and are taken from the Panamanian law firm and corporate service provider Mossack Fonseca. The records were obtained from an anonymous source by the German newspaper Süddeutsche Zeitung, which shared them with the International Consortium of Investigative Journalists (ICIJ) [21]. The ICIJ then shared them with a large network of international partners, including the Guardian [18] and the BBC [1]. They then carried out investigations that revealed information about many world leaders and their connection to offshore entities which were mostly used for tax avoidance. New stories about the content of these papers are continuously being released ever since.

Another notable data leak which also involves a large amount of data, happened two years later, in 2017. This set of documents are called the Paradise Papers. They originate from legal firm Appleby, the corporate services providers Estera and Asiaciti Trust, and business registries in 19 tax jurisdictions and is composed of 13.4 million documents with a volume of approximately 1.4 terabytes. They were also acquired by Süddeutsche Zeitung and are also a source of many news stories about famous individuals and numerous large corporations and their connections to offshore entities.

Both the Panama and the Paradise Papers are available in a property graph representation on Neo4j's website[3], with a schema shown in Figure 2.6. The former consists of approx. 560k nodes and 670k edges, while the latter is a considerably larger data set consisting of approx. 1.9 million nodes and 3.2 million edges.

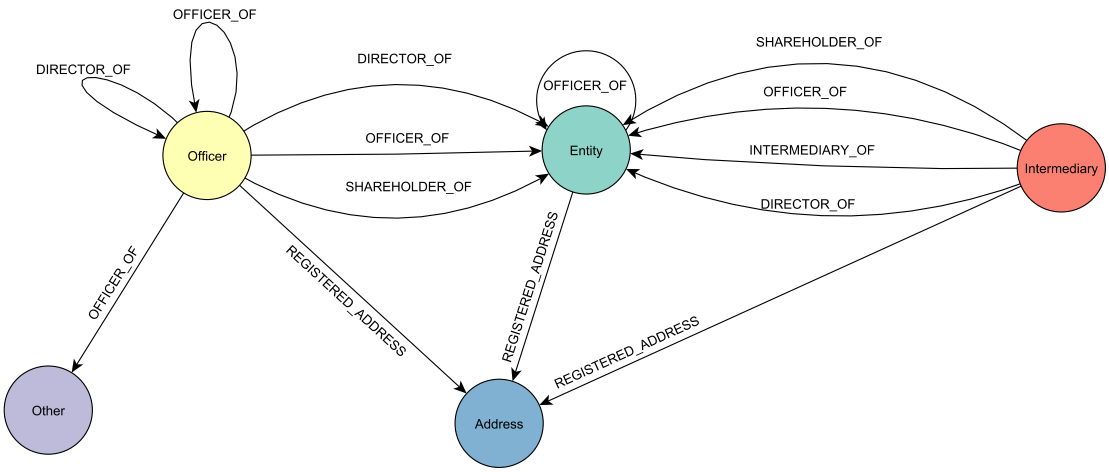

Figure 2.6: Common graph schema for the Panama and the Paradise Papers

[1] https://neo4j.com/
[2] https://db-engines.com/en/ranking_trend/graph+dbms
[3] https://neo4j.com/blog/icij-releases-neo4j-desktop-download-paradise-papers/

The nodes of the graph represent 5 different concepts:

- **Officer**: a person or company who plays a role in an offshore entity.
- **Intermediary**: go-between for someone seeking an offshore corporation and an offshore service provider — usually a law-firm or a middleman that asks an offshore service provider to create an offshore firm for a client.
- **Entity**: The offshore legal entity. This could be a company, trust, foundation, or other legal entity created in a low-tax jurisdiction.
- **Address**: postal address as it appears in the original databases.
- **Other**: additional information items.

The possible edges between these nodes can be found in the schema graph. All of these edges are directed. In Figure 2.7 there is an example for a graph that follows the schema

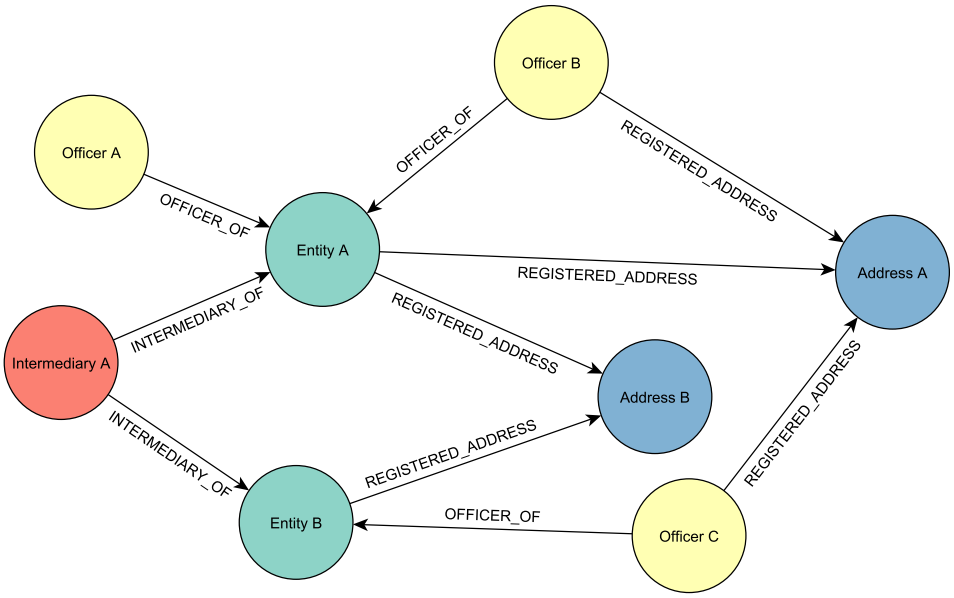

Figure 2.7: Example graph using the schema of the Panama and Paradise data sets

of the data set. This graph is not actually a subset of the data set, but is a simplification of parts of the original graph, for the sake of demonstration. It contains 8 nodes with 10 directed edges connecting them. In order to be able to carry out an analysis, we treat the graph as an undirected multiplex graph, i.e. omit the directions of the edges and the types of nodes, along with the properties of both nodes and edges.

Chapter 3

Multiplex Graph Metrics

In this chapter we introduce some metrics for multiplex graphs. We present the naïve approach to computing them and translate them to the language of linear algebra. Additionally, we discuss the related performance considerations.

3.1 Typed local clustering coefficient

Clustering coefficient, defined previously, when calculated for multiplex graphs does not fully capture the nature of emerging clusters in the graph, as there could be an added meaning in the types of the edges in a triangle and whether they are the same or not. For this reason, we study two variants of the typed clustering coefficient [8, 32].

Typed clustering coefficient 1

The first variant considers the triangles which have a wedge of two edges of one type centered around the given node and a third edge that has a different type. Formally:

$$\mathrm{TCC}_1(v) = \frac{\left|u, w, \alpha, \alpha' \mid Cd(v,u,\alpha) \wedge Cd(v,w,\alpha) \wedge Cd(u,w,\alpha') \wedge \alpha \neq \alpha'\right|}{\left|u, w, \alpha \mid Cd(v,u,\alpha) \wedge Cd(v,w,\alpha)\right|}$$

In the example graph, Figure 3.1 which consists of five nodes ($\mathsf{n1}, \ldots, \mathsf{n5}$), connected by edges of three types ($\mathsf{type1}, \ldots, \mathsf{type3}$):

$$\mathrm{TCC}_1(\mathsf{n1}) = 0.5,$$

As the only wedge consisting of edges of the same type, containing node $\mathsf{n1}$ is $(\mathsf{n1}, \mathsf{n2}, \mathsf{n3})$ which could be closed with an edge with a type of either $\mathsf{type2}$ or $\mathsf{type3}$, with one of which it is closed.

When calculating this metric for each node of a graph, the naïve approach is to iterate through the nodes and for each node, check for every pair of neighbours that are connected to the node with a given type of edge, if they are connected with an edge that suits the definition. For larger graphs, this has a large computational complexity.

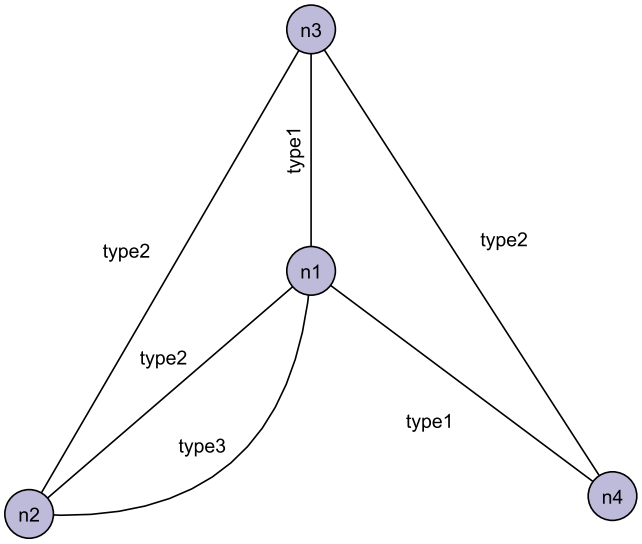

Figure 3.1: Example graph for demonstrating the calculation of the typed clustering coefficient

We can also define this metric using the adjacency matrix representation. The adjacency matrices of the graph in Figure 3.1 are the following:

$$A_{\text{type1}} = \begin{bmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}, \quad A_{\text{type2}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix} \quad A_{\text{type3}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

For node v_i, which corresponds to the i$^{\text{th}}$ row and column of the adjacency matrix, in terms of the elements of the adjacency matrices, the definition is as follows [8]:

$$\text{TCC}_1(v_i) = \frac{\sum_\alpha \sum_{\alpha' \neq \alpha} \sum_{i \neq j, m \neq i} \left(a_{ij}^{[\alpha]} a_{jm}^{[\alpha']} a_{mi}^{[\alpha]} \right)}{(M-1) \sum_\alpha \sum_{i \neq j, m \neq i} \left(a_{ij}^{[\alpha]} a_{mi}^{[\alpha]} \right)} = \frac{\sum_\alpha \sum_{\alpha' \neq \alpha} \sum_{i \neq j, m \neq i} \left(a_{ij}^{[\alpha]} a_{jm}^{[\alpha']} a_{mi}^{[\alpha]} \right)}{(M-1) \sum_\alpha k_i^{[\alpha]} \left(k_i^{[\alpha]} - 1 \right)}$$

where $k_i^{[\alpha]}$ denotes the typed degree of node i with respect to type α and M denotes the number of types in the graph. $k_i^{[\alpha]} \cdot \left(k_i^{[\alpha]} - 1 \right)$ is essentially the number of ways two edges of the given type (i.e. wedges), connected to a given node can be selected, each pair counted twice.

$$k_i^{[\alpha]} \cdot \left(k_i^{[\alpha]} - 1 \right) = 2 \cdot \binom{k_i^{[\alpha]}}{2}$$

For each of these variants, if the calculations involve division by zero, the corresponding node's clustering coefficient is regarded as 0.

From this, a vector containing the clustering coefficient for every node in the graph can be obtained with the following formula:

$$\text{TCC}_1 = \frac{\sum_{\alpha' \neq \alpha} \text{diag}^{-1} \left(A_\alpha \cdot A_{\alpha'} \cdot A_\alpha \right)}{(M-1) \cdot \sum_\alpha \left[\left(A_\alpha \cdot \vec{1} \right) \odot \left(\left(A_\alpha \cdot \vec{1} \right) - \vec{1} \right) \right]}$$

where $\left(A_\alpha \cdot \vec{1}\right)$ is essentially a vector containing the degree of each node with respect to type α.

Let's discuss the numerator first. For the example in Figure 3.1:

$$A_{\text{type1}} \cdot A_{\text{type2}} \cdot A_{\text{type1}} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{\text{type1}} \cdot A_{\text{type3}} \cdot A_{\text{type1}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_{\text{type3}} \cdot A_{\text{type1}} \cdot A_{\text{type3}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}, \quad A_{\text{type2}} \cdot A_{\text{type3}} \cdot A_{\text{type2}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$A_{\text{type2}} \cdot A_{\text{type1}} \cdot A_{\text{type2}} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 2 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \quad A_{\text{type3}} \cdot A_{\text{type2}} \cdot A_{\text{type3}} = \begin{bmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

The sum of their diagonal vectors is the following:

$$\sum_{\alpha' \neq \alpha} \text{diag}^{-1}\left(A_\alpha \cdot A_{\alpha'} \cdot A_\alpha\right) = \begin{bmatrix} 2 & 2 & 0 & 0 \end{bmatrix}^\intercal$$

Calculating the product of three adjacency matrices, however, is computationally complex. The reason for this is that although all of them are generally sparse matrices, the product of two of them can already get quite dense and multiplying that by a third one can involve many arithmetic multiplication operations. Intuitively, it also seems unnecessary to calculate the whole matrix when only its diagonal is needed. It can also be obtained the following way, using element-wise multiplication:

$$\text{diag}^{-1}\left(A_\alpha \cdot A_{\alpha'} \cdot A_\alpha\right) = A_\alpha \cdot A_{\alpha'} \odot A_\alpha \cdot \vec{1}$$

In this formula, the element-wise product is essentially a masking as all of the elements of adjacency matrices are binary. The multiplication with $\vec{1}$ sums the elements of each row. The interpretation of this formula is that we obtain the wedges for each node with the first multiplication, then filter out the ones that are closed by a third edge, by masking with the third adjacency matrix [5].

For the example graph:

$$A_{\text{type1}} \cdot A_{\text{type2}} \odot A_{\text{type1}} \cdot \vec{1} = \begin{bmatrix} 2 & 0 & 0 & 0 \end{bmatrix}^\intercal, \quad A_{\text{type1}} \cdot A_{\text{type3}} \odot A_{\text{type1}} \cdot \vec{1} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^\intercal$$

$$A_{\text{type3}} \cdot A_{\text{type1}} \odot A_{\text{type3}} \cdot \vec{1} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^\intercal, \quad A_{\text{type2}} \cdot A_{\text{type3}} \odot A_{\text{type2}} \cdot \vec{1} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^\intercal$$

$$A_{\text{type2}} \cdot A_{\text{type1}} \odot A_{\text{type2}} \cdot \vec{1} = \begin{bmatrix} 0 & 2 & 0 & 0 \end{bmatrix}^\intercal, \quad A_{\text{type3}} \cdot A_{\text{type2}} \odot A_{\text{type3}} \cdot \vec{1} = \begin{bmatrix} 0 & 0 & 0 & 0 \end{bmatrix}^\intercal$$

which are the same values as we obtained in the diagonals of the matrices computed with two matrix multiplications.

In the denominator:

$$A_{\text{type1}} \cdot \vec{1} = \begin{bmatrix} 2 & 0 & 1 & 1 \end{bmatrix}^\intercal \quad A_{\text{type2}} \cdot \vec{1} = \begin{bmatrix} 1 & 2 & 2 & 1 \end{bmatrix}^\intercal \quad A_{\text{type3}} \cdot \vec{1} = \begin{bmatrix} 1 & 1 & 0 & 0 \end{bmatrix}^\intercal$$

After performing the operations given in the formula, the final result is:

$$\mathrm{TCC}_1 = \begin{bmatrix} 0.5 & 0.5 & 0 & 0 \end{bmatrix}^\intercal$$

Typed clustering coefficient 2

The second variant of typed clustering coefficient considers triangles that have three different types of edges. Formally:

$$\mathrm{TCC}_2(v) = \frac{\left|u, w, \alpha, \alpha', \alpha'' \mid Cd(v,u,\alpha) \wedge Cd(v,w,\alpha') \wedge Cd(u,w,\alpha'') \wedge \alpha \neq \alpha' \wedge \alpha' \neq \alpha'' \wedge \alpha \neq \alpha''\right|}{\left|u, w, \alpha, \alpha' \mid Cd(v,u,\alpha) \wedge Cd(v,w,\alpha') \wedge \alpha \neq \alpha'\right|}$$

For the example graph in Figure 3.1:

$$\mathrm{TCC}_2(\texttt{n1}) = \frac{1}{4} = 0.25,$$

as there is only one triangle with three different edges, while the edges of `n1` could be completed into one in four possible ways.

The formula for each node v_i in terms of adjacency matrix elements is the following [8]:

$$\mathrm{TCC}_2(v_i) = \frac{\sum_\alpha \sum_{\alpha' \neq \alpha} \sum_{\alpha'' \neq \alpha, \alpha'} \sum_{i \neq j, m \neq i} (a_{ij}^{[\alpha]} a_{jm}^{[\alpha'']} a_{mi}^{[\alpha']})}{(M-2) \sum_\alpha \sum_{\alpha' \neq \alpha} \sum_{i \neq j, m \neq i} (a_{ij}^{[\alpha]} a_{mi}^{[\alpha']})}$$

Translating this to operations with the whole adjacency matrices, the numerator can easily be interpreted similarly to the first definition. However, the denominator cannot be simplified as in the first case. This is because we are not looking for wedges consisting of edges of the same type. This way, we cannot calculate the number of these wedges just using the typed degrees, as unlike edges of the same type, edges of different types may overlap. To work around this issue we can calculate the number of wedges in a similar way as the first definition, then subtract the ones where there is an overlap, determined with the expression $\left(A_\alpha \odot A_{\alpha'} \cdot \vec{1}\right)$. From this, the formula is the following:

$$\mathrm{TCC}_2 = \frac{\sum_\alpha \sum_{\alpha' \neq \alpha} \sum_{\alpha'' \neq \alpha, \alpha'} \mathrm{diag}^{-1}\left(A_\alpha A_{\alpha'} A_{\alpha''}\right)}{(M-2) \cdot \sum_\alpha \sum_{\alpha' \neq \alpha} \left[\left(\left(A_\alpha \cdot \vec{1}\right) \odot \left(A_{\alpha'} \cdot \vec{1}\right)\right) - \left(A_\alpha \odot A_{\alpha'} \cdot \vec{1}\right)\right]}$$

Similarly to the first definition, we can substitute the expression in numerator to a form that uses element-wise multiplication.

$$\mathrm{TCC}_2 = \frac{\sum_\alpha \sum_{\alpha' \neq \alpha} \sum_{\alpha'' \neq \alpha, \alpha'} A_\alpha \cdot A_{\alpha'} \odot A_{\alpha''} \cdot \vec{1}}{(M-2) \cdot \sum_\alpha \sum_{\alpha' \neq \alpha} \left[\left(\left(A_\alpha \cdot \vec{1}\right) \odot \left(A_{\alpha'} \cdot \vec{1}\right)\right) - \left(A_\alpha \odot A_{\alpha'} \cdot \vec{1}\right)\right]}$$

Using this formula for our example graph, we get the following results:

$$\mathrm{TCC}_2 = \begin{bmatrix} 0.25 & 1 & 0.5 & 0 \end{bmatrix}^\intercal$$

This result shows that while there is a correlation between TCC_1 and TCC_2 values, they uncover different types of structural features. For example, the TCC_1 value of node n3 is 0, but its TCC_2 value is 0.5.

Performance considerations

The calculation of clustering coefficients is quite a challenging task. Based on the classification of [19], our approach falls under the umbrella of *exact counting* of triangles, performing local counting for each node in the graph.

In order to grasp the complexity of calculating this metric, we need to look at the multitude of problems that it is related to, some of them already posing a challenge. The most basic problem in this group is globally counting triangles in a graph, without any kind of enumeration, which is already challenging [19], and is often calculated for large graphs using approximation. Local triangle counting brings added complexity, as it introduces the need for triangle enumeration. The calculation of the local clustering coefficient is even more complex as it requires local triangle count as well as counting the possible triangles (wedges) in a graph. Introducing types poses a different kind of complexity, as it necessitates to enumerate triangles with a specific property, therefore both of the typed clustering coefficient definitions are complex operations, the second one being slightly more problematic with the combination of 3 types in a triangle.

3.2 Edge overlap

Edge overlap is the conditional probability of the existence of an edge between nodes v_i and v_j of type α', given that there *is* and edge between these two nodes of type α. Formally, the definition is the following [8]:

$$P\left(\alpha' \middle| \alpha\right) = \frac{\left|a, b \mid Cd(a, b, \alpha) \wedge Cd(a, b, \alpha')\right|}{\left|a, b \mid Cd(a, b, \alpha)\right|}$$

This metric can directly be translated to operations with the elements of the adjacency matrices:

$$P\left(a_{ij}^{[\alpha']} \middle| a_{ij}^{[\alpha]}\right) = \frac{\sum_{ij} a_{ij}^{[\alpha']} a_{ij}^{[\alpha]}}{\sum_{ij} a_{ij}^{[\alpha]}})$$

where a_{ij}^{α} denotes the existence of an edge between a_i and a_j of type α. We refer to α as the *conditional type*. Clearly, edge overlap is only defined for multiplex graphs.

As we are working with undirected graphs, the adjacency matrices are symmetrical, meaning that $a_{ij}^{[\alpha]} = a_{ji}^{[\alpha]}$, therefore the formula can be expressed in the following way:

$$P\left(a_{ij}^{[\alpha']} \middle| a_{ij}^{[\alpha]}\right) = \frac{\left((A_\alpha \cdot A_{\alpha'}) \cdot \vec{1}\right)^\intercal \cdot \vec{1}}{\left(A_\alpha \cdot \vec{1}\right)^\intercal \cdot \vec{1}}$$

which is essentially the sum of all elements in the product of the two matrices, divided by the sum of all of the elements of A_α.

For the graph in Figure 3.1, the only overlap that ever occurs is between type2 and type3, between the nodes n1 and n2. For these two types:

$$P\left(v_{ij}^{[\mathsf{type3}]}\middle|v_{ij}^{[\mathsf{type2}]}\right)=\frac{1}{3}$$

and

$$P\left(v_{ij}^{[\mathsf{type2}]}\middle|v_{ij}^{[\mathsf{type3}]}\right)=1.$$

Between every other pair, the edge overlap values are 0.

Performance considerations

In terms of performance, as we will demonstrate in Chapter 5, the calculation of edge overlap with the naïve approach is relatively quick, therefore it might not be efficient to introduce matrix multiplications.

Chapter 4

Implementation

For the implementation of the different algorithms for the selected metrics, we used Graph Analyzer[1] [32], which was developed at the Department of Measurement and Information Systems and the MTA-BME Lendület Research Group on Cyber-Physical Systems.

For the naïve implementation the edges and nodes of the graph are stored in Java Collections. For each type, all of the edges are stored in an `ArrayListMultiMap` from the Google Guava library[2]. These data structures are stored in a `HashMap`, where the keys are the types of the graph.

4.1 Matrix libraries

	UJMP	ojAlgo	EJML
License	LGPL	MIT	Apache
Version	0.3.0	46.3.0	0.36
Latest release	2015	2018	2018
Stores sparse matrices in CRS/CCS	✗	✗	✓
Stores sparse matrices in LIL	✗	✓	✗
Stores sparse matrices in DOK	✓	✓	✗

Table 4.1: Comparison of matrix libraries for Java

In general, the adjacency matrix of graphs is quite sparse, hence, a crucial prerequisite for the implementation is to use a library that is able to efficiently store and perform operations on large sparse matrices.

Matrix density is defined in the following way:

$$\text{Density} = \frac{\text{Number of non-zero elements}}{n \cdot m}$$

where n and m denote the number of rows and columns of the matrix. In the Paradise Papers, the most dense adjacency matrix has a density of approximately 10^{-6}, while the sparsest type has an adjacency matrix with a 10^{-10} density.

[1] `https://github.com/ftsrg/graph-analyzer/`
[2] `https://github.com/google/guava`

For the algorithms using adjacency matrices, the following open-source libraries were used: Universal Java Matrix Package (UJMP)[3], Optimatika Java Algorithms (ojAlgo)[4] and Efficient Java Matrix Library (EJML)[5]. Table 4.1 shows a comparison of the selected libraries. It is worth pointing out that out of the three selected libraries, only EJML has support for advanced sparse matrix storage formats such as CRS (Compressed Row Storage) or CCS (Compressed Column Storage) [30], while the other libraries use the basic LIL (List of Lists) and DOK (Dictionary of Keys) representations. It is also worth noting that the UJMP library is no longer maintained and its latest version was released more than 3 years ago.

4.2 Clustering coefficient

In this section we present the details of the different implementations of typed clustering coefficient along with code snippets that demonstrate how the chosen libraries were used. The code snippets of the edge overlap implementations can be found in Section A.1. These examples are taken from the implementation of the first definition and they mainly focus on the way that the wedges and triangles in the graph are counted, as those are the main challenges. For the sake of simplicity, in these snippets some variable initializations are not detailed but are required in order to understand the algorithms. These variables are the following:

- `types`: List of types in the graph
- `adjacencyMatrices`: A map of types and adjacency matrices (the appropriate type for the implementation in each example)
- `n`: The number of nodes in the graph
- `types`: Collection of types in the graph
- `typePairs`: Collection of all of the possible pairs of the types in the graph (excluding pairs of identical types)

ojAlgo In the ojAlgo implementation of TCC_1, shown in Listing 2, we can see that some of the matrices are stored as dense matrices (in this example the vectors are stored this way), the adjacency matrices are stored as sparse matrices. It can also be noted that some of the operations provide the result as the return value of the method that it is performed by (e.g. in line 18), while some of them require an empty matrix to be initialized beforehand and when the operations are defined, they are explicitly passed on to the previously initialized empty matrix. This makes it possible to chain operations and evaluate them in a lazy way, and not to occupy space unnecessarily.

UJMP In the UJMP example in Listing 3 we can see that the only way to perform operations is by calling the corresponding method on the matrix object and the result is given by the return value. This way of computation makes the API more intuitive and the code more readable.

[3] `https://ujmp.org/`
[4] `http://ojalgo.org/`
[5] `http://ejml.org/`

```
long numberOfPossibleConnections = 0;
for (T type1 : getTypes(node)) {
    numberOfNeighbors = getNeighbors(node, type1).size();
    numberOfPossibleConnections += numberOfNeighbors * (numberOfNeighbors - 1);
    for (N neighbor1 : getNeighbors(node, type1)) {
        for (N neighbor2 : getNeighbors(node, type1)) {
            if (neighbor1 != neighbor2) {
                Set<T> types = getTypes(neighbor1);
                for (T type2 : types) {
                    if (type1 != type2) {
                        if (isAdjacentUndirected(neighbor1, neighbor2, type2)) {
                            interConnected++;
                        }
                    }

                }
            }
        }
    }
    double tcc1 = (double) interConnected / (double) numberOfPossibleConnections;
}
```

Listing 1: Naïve implementation of TCC_1 using the edge list representation

```
MatrixStore triangles      = PrimitiveDenseStore.FACTORY.makeZero(n, 1);
PrimitiveDenseStore wedges = PrimitiveDenseStore.FACTORY.makeZero(n, 1);

PrimitiveDenseStore ones    = PrimitiveDenseStore.FACTORY.makeZero(n, 1);
ones.operateOnAll(ADD, 1).supplyTo(ones);

for (T type1 : types) {
    SparseStore<Double> A = adjacencyMatrices.get(type1);
    for (T type2 : types) {
        if (type1 != type2) {
            SparseStore<Double> B = adjacencyMatrices.get(type2);
            SparseStore<Double> C = SparseStore.PRIMITIVE.make(n, n);

            SparseStore<Double> AB = SparseStore.PRIMITIVE.make(n, n);
            A.multiply(B).supplyTo(AB);

            AB.operateOnMatching(MULTIPLY, A).supplyTo(C);
            triangles = C.multiply(ones).add(triangles);
        }
    }
}
```

Listing 2: ojAlgo implementation of TCC_1

EJML Contrary to the other two matrix libraries, in the EJML example shown in Listing 4, a matrix with the appropriate size and type needs to be pre-initialized and there are interfaces dedicated for performing operations, where both the operands and the empty matrix to be filled with the results are provided. It is also interesting to note that sparse matrices are initially stored in a data structure that only makes it possible to build up the matrix from triples of the following format: $\langle row, column, value \rangle$. Once all of the elements have been added, it can be converted to a type which does not allow any further modifications. The reason for this is so that it can build up the data structure efficiently in a compressed format, to be used for performing operations. An example can be seen in Listing 5, where an identity matrix of size $n = 1000$ is created and is then converted from triplets to a sparse matrix and is multiplied by itself.

```
Matrix triangles = SparseMatrix.Factory.zeros(n,n);
Matrix wedges = SparseMatrix.Factory.zeros(n, 1);
for (T type1 : types) {
    Matrix A = adjacencyMatrices.get(type1);
    for (T type2 : types) {
        if (type1 != type2) {
            Matrix B = adjacencyMatrices.get(type2);
            triangles = triangles.plus(A.mtimes(B).mtimes(A));
        }
    }
    Matrix degreeVector = A.sum(Calculation.Ret.NEW, 1, false);
    wedges = wedges.plus(degreeVector.times(degreeVector.minus(1)));

}
```

Listing 3: UJMP implementation of TCC_1

```
SimpleMatrix triangles = new SimpleMatrix(n, 1);
SimpleMatrix wedges = new SimpleMatrix(n, 1);
for (T type1 : types) {
    DMatrixSparseTriplet tripletsA = adjacencyMatrices.get(type1);
    DMatrixSparseCSC A = ConvertDMatrixStruct.convert(tripletsA, (DMatrixSparseCSC) null);
    for (T type2 : types) {
        if (type1 != type2) {
            DMatrixSparseTriplet tripletsB = adjacencyMatrices.get(type2);
            DMatrixSparseCSC B = ConvertDMatrixStruct.convert(tripletsB, (DMatrixSparseCSC) null);
            DMatrixSparseCSC AB = new DMatrixSparseCSC(n, n, 0);
            DMatrixSparseCSC ABA = new DMatrixSparseCSC(n, n, 0);
            ImplSparseSparseMult_DSCC.mult(A, B, AB, null, null);
            CommonOps_DSCC.elementMult(AB, A, ABA, null, null);
            DMatrixRMaj rowSum = new DMatrixRMaj(n, 1);
            CommonOps_DSCC.sumRows(ABA, rowSum);
            triangles = triangles.plus(SimpleMatrix.wrap(rowSum));
        }
    }
    DMatrixRMaj degreeVector = new DMatrixRMaj(n, 1);
    CommonOps_DSCC.sumRows(A, degreeVector);
    SimpleMatrix simpleDegreeVector = SimpleMatrix.wrap(degreeVector);
    wedges = wedges.plus(simpleDegreeVector.elementMult(simpleDegreeVector.minus(1)));
}
```

Listing 4: EJML implementation of TCC_1

4.2.1 Parallelization

To parallelize the computation, we used the list of types and type pairs. The number of wedges are calculated for each type, the number of triangles are calculated for pairs of types. The fact that these calculations are done by types and type-pairs and they do not rely on each other or modify any of their common resources, made it possible to parallelize the computation. It was implemented using the Stream API of Java 8 [20].

For the second variant of the typed clustering coefficient, the parallelization is done along sets of three types for triangle calculation and type pairs for wedge calculation.

4.2.2 Testing

To test the different approaches, we implemented unit tests on 27 different graphs, which include various typed graph motifs. All implementations presented in this report successfully pass all tests.

```
int n = 1000;
DMatrixSparseTriplet triplets = new DMatrixSparseTriplet(n, n, n);
for (int i = 0; i < n; i++) {
    triplets.addItem(i, i, 1);
}
DMatrixSparseCSC M = ConvertDMatrixStruct.convert(triplets, (DMatrixSparseCSC) null);
DMatrixSparseCSC MM = new DMatrixSparseCSC(n, n, 0);
ImplSparseSparseMult_DSCC.mult(M, M, MM, null, null);
```

Listing 5: Building a matrix in EJML

```
SimpleMatrix wedges = typeList.stream().parallel().map(x -> countWedgesEjmlEw(x, adapter))
    .reduce(new SimpleMatrix(n,1),(a, b) -> a.plus(b));
SimpleMatrix triangles = typePairs
    .stream().parallel().map(x -> countTrianglesEjmlEw(x.get(0), x.get(1), adapter))
    .reduce(new SimpleMatrix(n,1), (a, b) -> a.plus(b)
    );
```

Listing 6: Parallelized EJML implementation using the Stream API

4.2.3 Visualization and loading the graph

To load the graphs stored in comma-separated values (CSV) into the Graph Analyzer, we used the Super CSV[6] library. After the execution of the implementations, we used the same library to output both the performance results and the metric values to CSV files. We then implemented a tool in R, using the ggplot2[7] library to visualize the results, as it is shown in Figure 4.1.

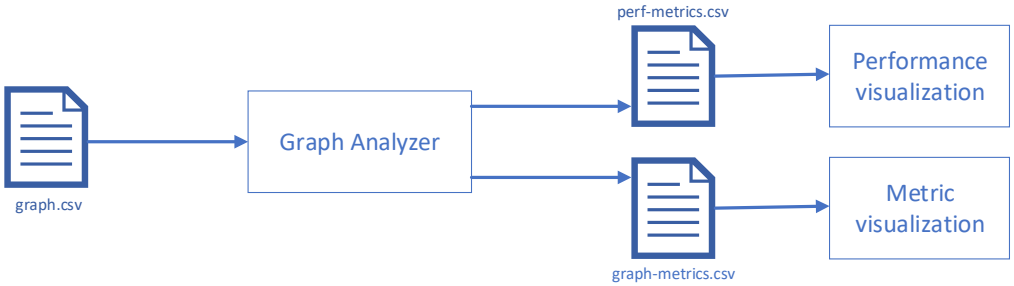

Figure 4.1: Workflow of the analysis

[6] http://super-csv.github.io/
[7] https://ggplot2.tidyverse.org/

Chapter 5

Evaluation

In this chapter we present a comparison of the different implementations in terms of performance and scalability, as well as analyze the data set using these calculated metrics.

5.1 Performance and scalability

First, we present the setup of performance experiments that we carried out to benchmark different approaches.

5.1.1 Performance measurement setup

For the evaluation of performance, the computations were run on 10 subsets of the data and the whole data set. The sampling was done in terms of the number of the edges. To obtain the subsets, the number of edges was divided by exponents of 2, the number obtained this way was the number of edges randomly selected to be a part of the subset. The set of nodes were obtained by including all the nodes that were parts of the selected edges. In total, this resulted in 11 graphs which we denote by the *scale factor*, $-10 \dots 0$, referring to the exponent of 2 that was used as the ratio of the subset and the original graph. For instance, the subset with a scale factor of -10 has approximately 5000 nodes and 3000 edges, the one of scale factor -5 has approx. 105 000 nodes and 100 000 edges, and the whole data set (with a scale factor of 0) has around 1.9 million nodes and 3.2 million edges.

Each configuration was executed 3 times, of which the median was used for the analysis. The execution times were measured using Java's `System.currentTimeMillis()` method. All of the computations were performed with 64 GB of heap memory and 24 CPU cores of an Intel Xeon Platinum 8167M CPU running at 2.00GHz. The system was running the Ubuntu 18.04.1 operating systems and Oracle JDK 8 (version 1.8.0u181).

5.1.2 Typed clustering coefficient 1

For this metric, in addition to the naïve edge-list approach, we implemented two matrix-based algorithms for each of the matrix libraries: one that uses two matrix multiplications to count triangles (denoted with *MMM*), and one that uses one matrix multiplication and one element-wise multiplication (denoted with *EW*).

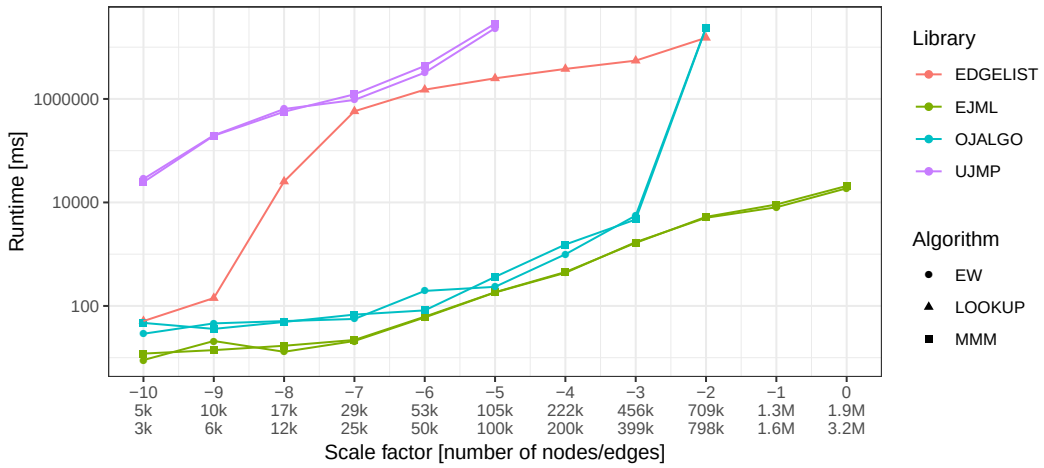

Figure 5.1: TCC_1 performance results

In Figure 5.1 it can be seen that the implementations using EJML had the best performance at every scale factor and they both also scaled well, with no significant difference between the element-wise version and the one with two matrix multiplications. They were also the only two single-threaded implementations where the calculations did not exceed the timeout.

Interestingly, the ojAlgo implementations seemed to scale well for smaller subsets but began to significantly degrade for scale factors larger than -3 and exceeded the timeout at scale factor -1.

The implementations using UJMP had the worst performance, the execution time for the smallest subset is comparable of that of the EJML implementations for the whole data set. It is worth pointing out that UJMP implementation is most out-of-date of the three Java libraries (see Section 4.1).

For the parallel implementations of the metric, we used the element-wise definition and the two better-performing matrix libraries from the single-threaded one. As it is shown in Figure 5.2, parallelization made an improvement in both of them, however, the ojAlgo implementation still became significantly worse in terms of scalability for scale factors larger than -3.

The improvement is not as significant as we might expect it to be when switching from one single thread to multiple threads. One of the reasons for that is that there are types which are more sparse in general and pairs of types which have less edges in common. For these, the threads finish quickly while for more dense and overlapping types, matrix multiplications are more time consuming. Therefore, these calculations are unable to utilize the CPU cores to their full capacity.

5.1.3 Typed clustering coefficient 2

For this metric, we again used the edge list implementation as a baseline and took the two better-performing matrix libraries and implemented a version that uses element-wise multiplication (EW) and one that uses two matrix multiplications (MMM) for each of them.

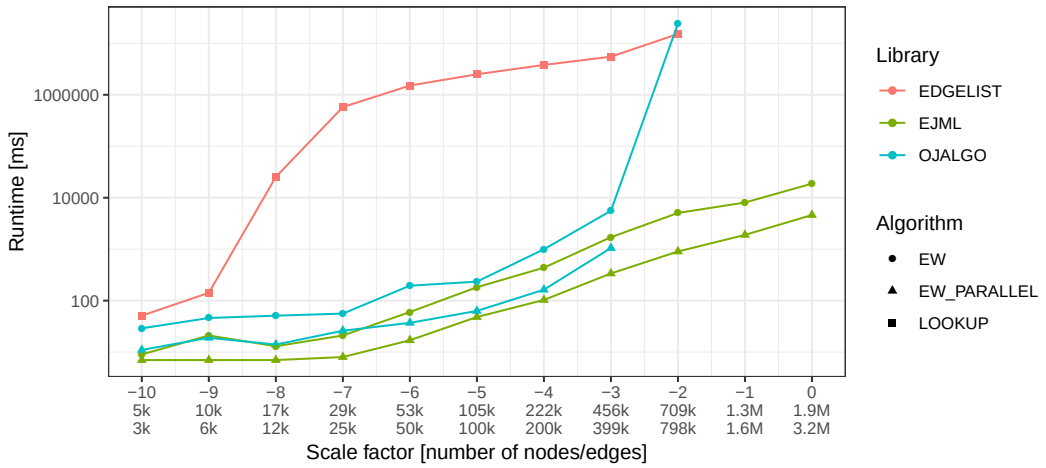

Figure 5.2: TCC_1 parallel implementation performance results

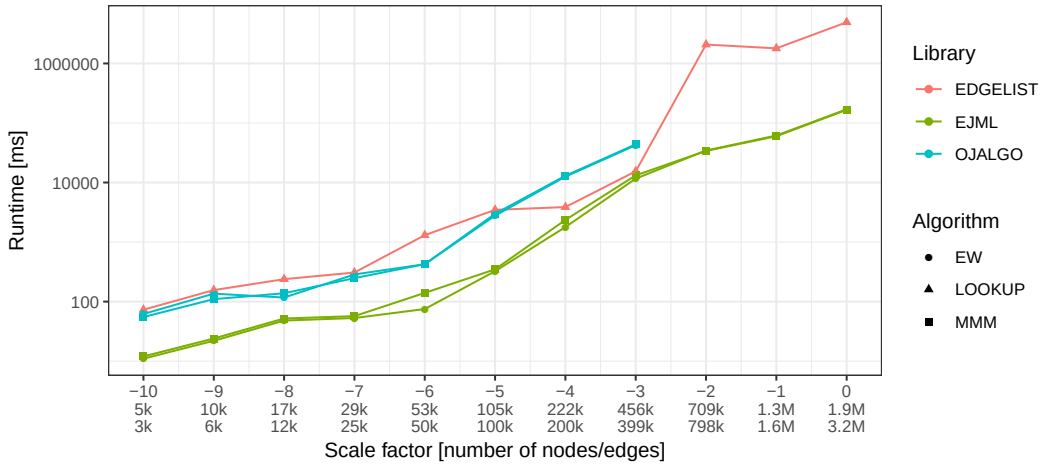

Figure 5.3: TCC_2 performance results

The results are shown in Figure 5.3. Similarly to TCC_1, there was no significant difference in terms of performance between the *MMM* and the *EW* implementations. Also similarly to TCC_1, the EJML implementations had the best performance by far.

Interestingly, in this case, the edge list implementation performed slightly better than for TCC_1. This might be because the wedges that the first definition considers are more common in this data set than the ones counted by the second definition.

The ojAlgo implementations are comparable to the UJMP implementations and seem to scale well for smaller subsets but similarly to the first definition, after the computations on the scale factor -3 data set, the execution time significantly increases.

5.1.4 Edge overlap

The results of the performance experiments of the edge overlap are shown in Figure 5.5. It is clear that neither of the implementations using matrix libraries was able to significantly outperform the edge-list implementation used as a baseline. Although the EJML implementation was slightly better in some cases, it might not be worth trading off the simplicity of the algorithm for a minor performance improvement.

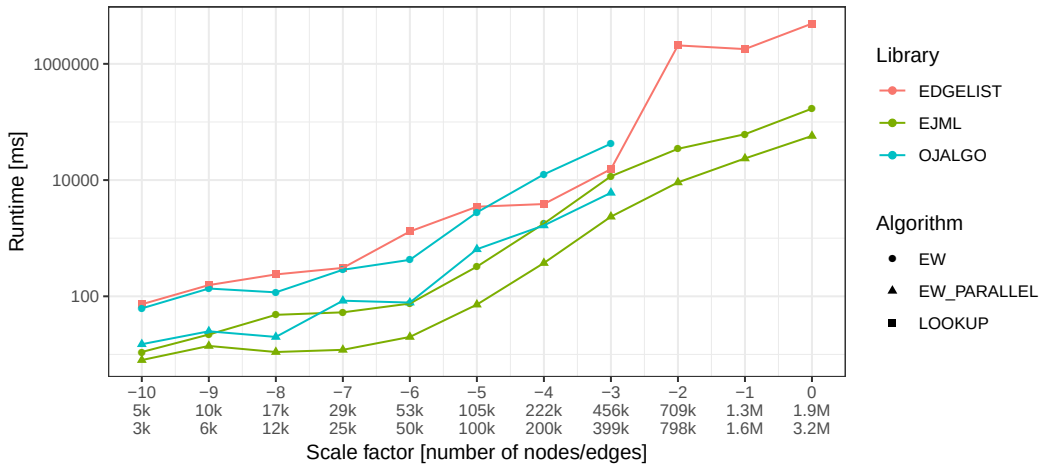

Figure 5.4: TCC_2 parallel implementation performance results

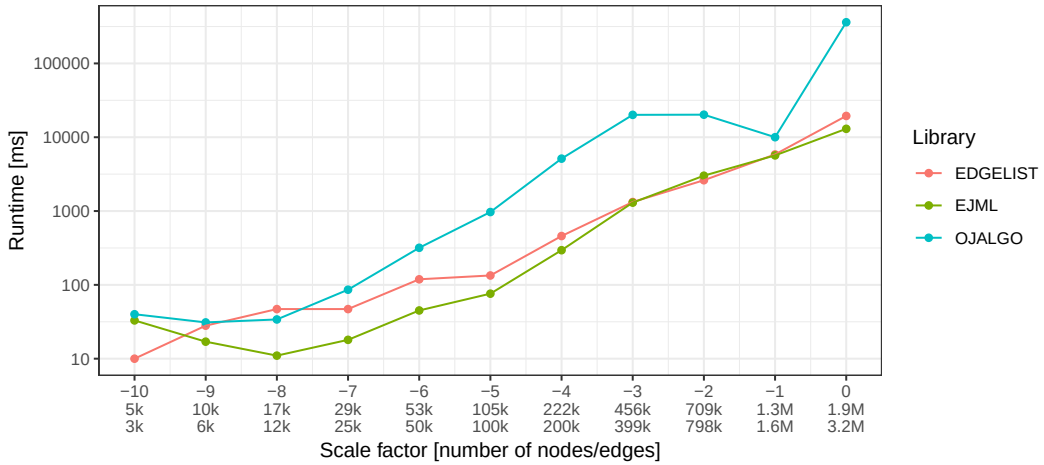

Figure 5.5: Edge overlap performance result

5.1.5 Loading the graphs

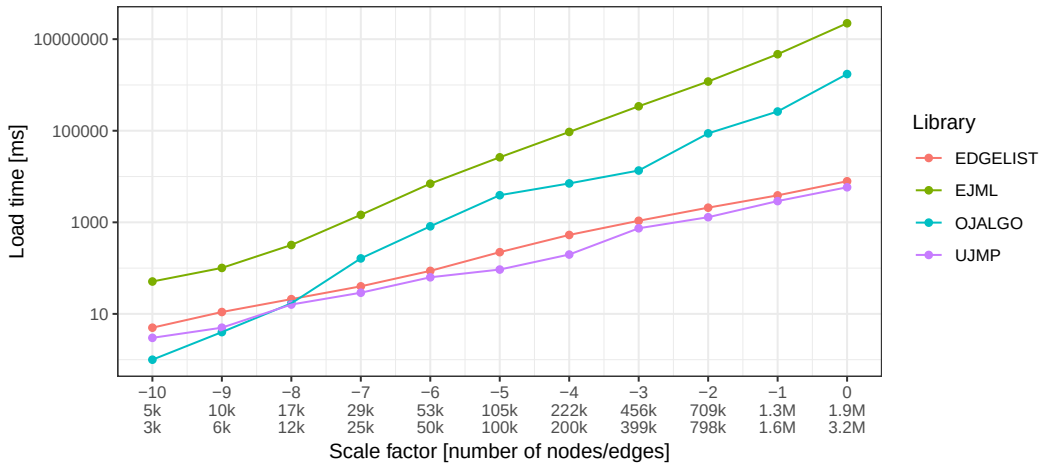

Figure 5.6: Load times of the graph with different libraries

We performed measurements on how much time it took for each of the graph representations to load each of the subsets of the graph separately. Similarly to the previous performance tests, this measurement has been carried out 3 times and the median time of each of the runs were used for assessment. The results are shown in Figure 5.6. For most of the cases it can be observed that there is a trade-off between the load times and the performance of the matrix operations, as the better-performing libraries take more time to store the matrices and vice versa. This is not surprising, as it adds complexity to building the matrices, to store them in a way that makes it possible to perform operations efficiently on them. For this reason, there is not a clear best among these libraries and depending on the properties, the size of the data set and the nature and number of operations to perform, the ideal choice can vary.

Highlights of observations

One of the main findings is that the matrix-based technique for calculating graph metrics is a feasible approach. However, there are certain trade-offs to take into consideration, such as the load time of the matrices. Another factor is the development effort, as the edge-list representation is trivial while the linear algebraic approach is not intuitive. The lack of all-around high-performing libraries for Java is also a key issue.

5.2 Multiplex Graph Analysis

In this section we present the results of the calculated metrics and carry out an analysis based on them.

5.2.1 Typed clustering coefficient

The typed clustering coefficient values of our data set are shown on a histogram in Figure 5.7. In this chart we can see that for both of the variants, for the significant majority of nodes the value is 0. This means that there are not many triangles in the graph that consist of two or three different types.

We can also observe that the values for typed clustering coefficient 2 tend to be lower than the values of the first variant, meaning that there are probably more triangles of two different types in the graph than triangles of three different types. This is especially true considering that a triangle of three different types increases the TCC_2 of all three nodes that it involves, unlike a triangle of two different types which only increases the TCC_1 value of one node, the node which the wedge of two identical types is centered around.

5.2.2 Edge overlap

The result of the edge overlap calculations is shown as a heat map in Figure 5.8. The *conditional types*, which were previously introduced in Section 3.2 can be found on the x axis. The colour of each square is determined by the probability that on condition that there is an edge between two nodes of the conditional type, there is also one of the given type.

The figure shows that many of the types have very few or no overlapping edges. In the diagonal, trivially all of the elements has a value 1, as every type perfectly overlaps with itself.

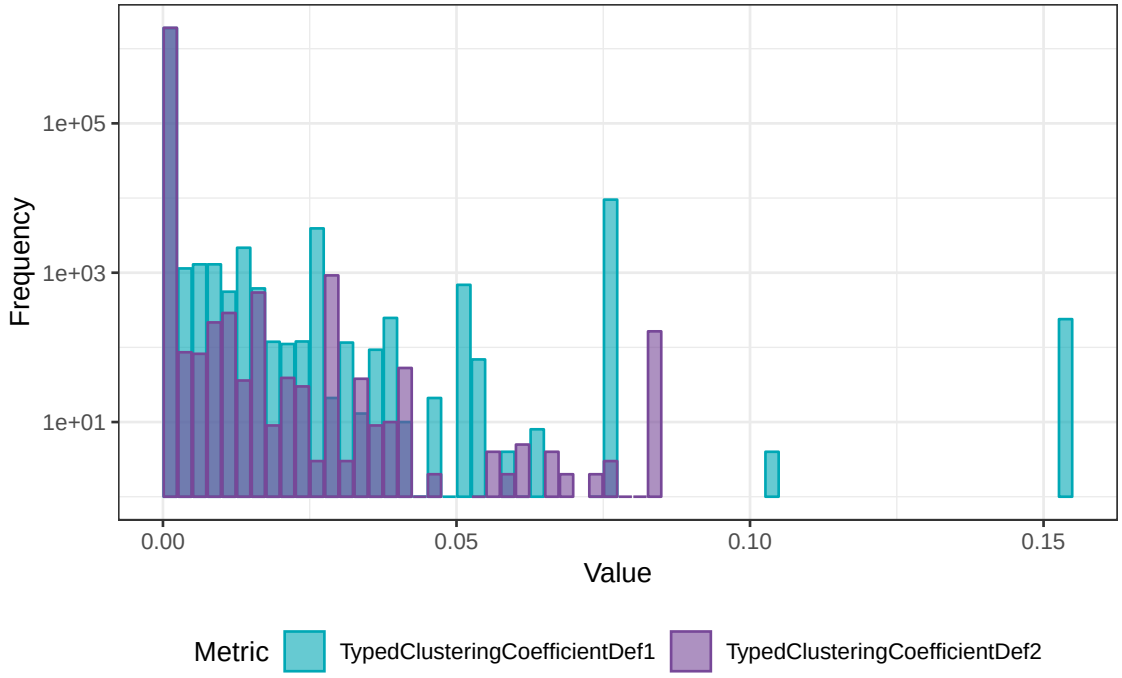

Figure 5.7: Metric values for typed clustering coefficients 1 and 2

One interesting result to be observed is the relationship between types SAME_NAME_AS and SAME_INTERMEDIARY_AS. At the point where the latter is the conditional type, the corresponding square indicates that the edge overlap is 1. If we swap their roles, however, we see that the edge overlap is very small. This means that wherever there is an edge of type SAME_INTERMEDIARY_AS, there is certainly an edge of type SAME_NAME_AS too, but not vice versa. This means that the type SAME_INTERMEDIARY_AS is a subset of the type SAME_NAME_AS. This is a functional redundancy in the data set. For types OFFICER_OF and INTERMEDIARY_OF, however, the connection seems to be more balanced, as these conditional probabilities are close to each other in value.

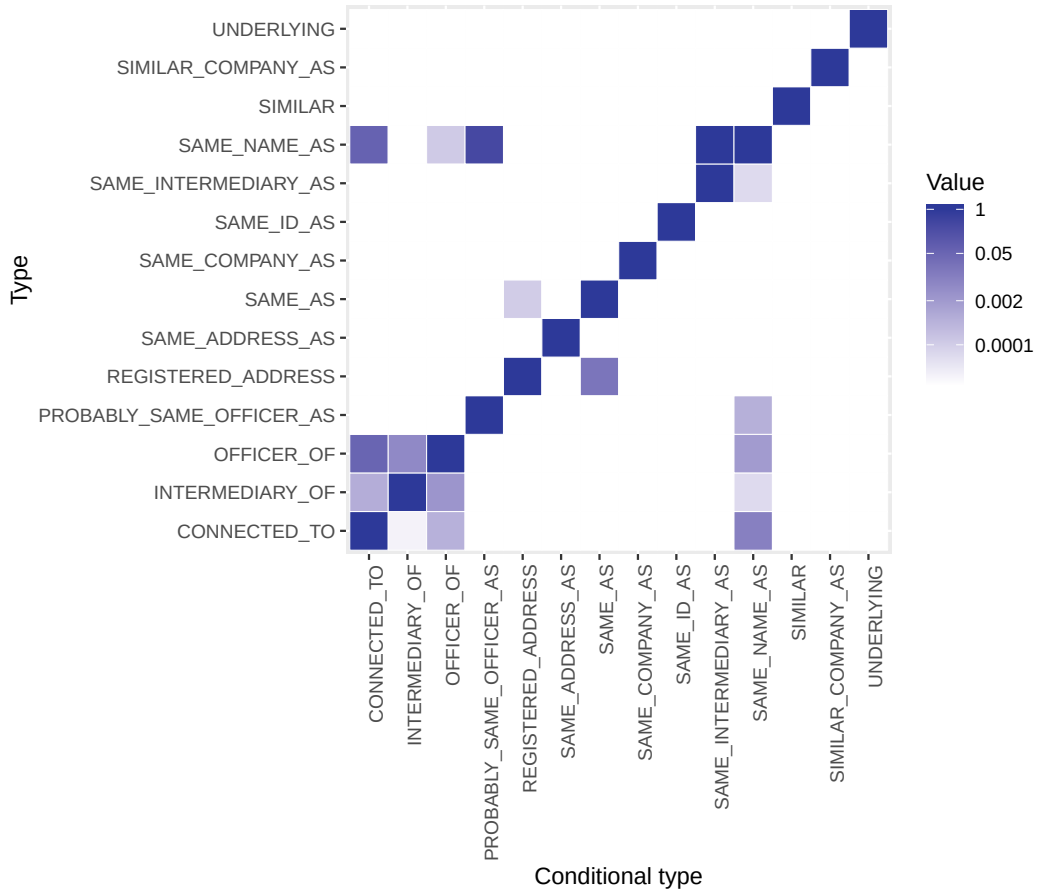

Figure 5.8: Edge overlap metric values

Chapter 6

Related Work

In this chapter we present some of the related scientific research in connection with network science, multiplex graphs, and related results in high-performance computing.

6.1 Multiplex networks

In [8] the authors defined a set of multiplex graph metrics, such as the typed clustering coefficient used in this report. Their metrics were validated on a real multiplex data set of Indonesian terrorists, a system with 78 nodes and four types.

In [10] the authors express the importance of multiplex network analysis and define a number of real-life multidimensional networks, such as Flickr and QueryLog. In [28] the authors investigated and characterized correlations between the different layers of a multi-dimensional network. They found instances in which two layers of the same network were positively correlated in terms of node degrees, while two others were negatively correlated.

A multiplex network is defined in [12], where the neighbourhood and centrality analysis of the Polish social network is presented. Another study involving multiplex networks is [24], which is a relevance and correlation analysis of different dimensions in Flickr. In [4], the authors present some of the theoretical background behind simple graphs and multiplex networks, and the transition between them. There are other studies that involve community detection in the network of YouTube [33], analysis of co-authorship in the DBLP network using shortest path [11] and the analysis of transportation networks such as the European Air Network in [13], and a network of cargo ship movements in [23].

In [22], the authors studied the effect of graph metrics on query performance. In particular, they calculated the correlation between graph metrics, result set size and the shape of the graph query, concluding that different engines affected by different input parameters.

6.2 Graph analytics

There are two key programming models used for graph analytics. The first one is the vertex-centric programming model which defines the computation as an iterative process between communicating vertices in the graph [27]. This model is used in the Apache Spark GraphX and the Apache Giraph frameworks. The second one is the linear algebra-based approach which defines the computation with matrix operations. This model is specified in the GraphBLAS standard [25], but is only available as a C library.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

In this report, I studied a set of multiplex graph metrics [8, 32] and presented performance optimizations, validated on a real-life data set. In particular, I made the following contributions:

- I formalized a set of multiplex graph metrics (TCC_1, TCC_2 and edge overlap) in the language of linear algebra.
- I optimized the TCC_1 and TCC_2 formalizations using element-wise matrix multiplication to decrease their computational complexity.
- I implemented the defined approaches using different libraries, studied their scalability and visualized the results. I wrote over 1400 lines of Java code to implement the analysis and over 150 lines of R code for the visualization.
- I identified the trade-offs that are involved in implementing linear algebra-based approaches.
- I evaluated the results on a real data set from the field of investigative journalism.
- I integrated my implementation to the open-source Graph Analyzer framework.

7.2 Future Work

First and foremost, I plan to study the original Paradise papers data set with respect to the derived multiplex graph metrics values. My goal is to understand the semantics and implications of a given clustering coefficient value, e.g. what does a high typed clustering coefficient value mean for a node in the graph and how can this be used for achieving greater insight to the significance of offshore entities?

As this work presents a step towards the efficient calculation of complex multiplex graph metrics, it hopefully opens up the possibility to use these metrics to characterize large graph instances with millions of nodes. Such analyses have numerous application areas, e.g. detecting *fraud triangles* in financial transaction graphs [35], studying the emergence of cooperating people in society [9] or spotting the collaboration between scientists [2]. Additionally, multiplex metrics can been used to distinguish real graphs from fake ones [32],

similarly to how other multiplex graph metrics such as the *multiplex participation coefficient* and the *pairwise multiplexity* have been used in [34] to differentiate real graph models from generated ones.

To improve the analysis, I plan to extend the typed local clustering coefficient metrics to include not only triangles, but also k-cycles and k-cliques. This will require a common generalization of the TCC and the k-local clustering coefficient [16] metrics. I also plan to experiment with the concept of *meta-paths* used in heterogeneous information networks [31], which look for paths containing a given sequence of node and edge types.

Acknowledgements

I would like to thank my supervisor Gábor Szárnyas for his friendly advice and enthusiasm. I would also like to thank József Marton for providing cloud virtual machines for the performance experiments.

Bibliography

[1] Panama Papers Q&A: What is the scandal about? *BBC*, Apr 2016. URL http://www.bbc.com/news/world-35954224.

[2] Adam Agocs, Dimitris Dardanis, Jean-Marie Le Goff, and Dimitris Proios. Interactive graph query language for multidimensional data in collaboration spotting visual analytics framework. *CoRR*, abs/1712.04202, 2017. URL http://arxiv.org/abs/1712.04202.

[3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Internet: Diameter of the world-wide web. *Nature*, 401(6749):130–131, September 1999. ISSN 00280836. DOI: 10.1038/43601.

[4] Alberto Aleta and Yamir Moreno. Multilayer networks in a nutshell. *CoRR*, abs/1804.03488, 2018. URL http://arxiv.org/abs/1804.03488.

[5] Ariful Azad, Aydin Buluç, and John R. Gilbert. Parallel triangle counting and enumeration using matrix algebra. In *IPDPS Workshops*, pages 804–811. IEEE Computer Society, 2015.

[6] A.L. Barabási and M. Pósfai. *Network Science*. Cambridge University Press, 2016. ISBN 9781107076266. URL https://books.google.hu/books?id=ZVHesgEACAAJ.

[7] Albert-László Barabási and Eric Bonabeau. Scale-free networks. *Scientific american*, 288(5):60–69, 2003.

[8] Federico Battiston, Vincenzo Nicosia, and Vito Latora. Structural measures for multiplex networks. *Phys. Rev. E*, 89:032804, Mar 2014. DOI: 10.1103/PhysRevE.89.032804. URL https://link.aps.org/doi/10.1103/PhysRevE.89.032804.

[9] Federico Battiston, Matjaz Perc, and Vito Latora. Determinants of public cooperation in multiplex networks. *New Journal of Physics*, 19(7):073017, 2017. URL http://stacks.iop.org/1367-2630/19/i=7/a=073017.

[10] Michele Berlingerio et al. Multidimensional networks: foundations of structural analysis. *World Wide Web*, 16(5-6):567–593, 2013. DOI: 10.1007/s11280-012-0190-4. URL http://dx.doi.org/10.1007/s11280-012-0190-4.

[11] Piotr Bródka, Pawel Stawiak, and Przemyslaw Kazienko. Shortest path discovery in the multi-layered social network. In *ASONAM*, pages 497–501, 2011. DOI: 10.1109/ASONAM.2011.67. URL http://dx.doi.org/10.1109/ASONAM.2011.67.

[12] Piotr Bródka, Przemyslaw Kazienko, Katarzyna Musial, and Krzysztof Skibicki. Analysis of neighbourhoods in multi-layered dynamic social networks. *Int. J. Computational Intelligence Systems*, 5(3):582–596, 2012. DOI:

10.1080/18756891.2012.696922. URL http://dx.doi.org/10.1080/18756891.2012.696922.

[13] Alessio Cardillo et al. Modeling the multi-layer nature of the European Air Transport Network: Resilience and passengers re-scheduling under random failures. *European Physical Journal-Special Topics*, 215(1):23–33, 2013.

[14] Luciano da Fontoura Costa et al. Analyzing and modeling real-world phenomena with complex networks: a survey of applications. *Advances in Physics*, 60(3):329–412, 2011. DOI: 10.1080/00018732.2011.572452.

[15] Paul Erdős and Alfréd Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60, 1960.

[16] Agata Fronczak, Janusz A Hołyst, Maciej Jedynak, and Julian Sienkiewicz. Higher order clustering coefficients in Barabási–Albert networks. *Physica A: Statistical Mechanics and its Applications*, 316(1):688 – 694, 2002. ISSN 0378-4371. DOI: https://doi.org/10.1016/S0378-4371(02)01336-5. URL http://www.sciencedirect.com/science/article/pii/S0378437102013365.

[17] Edgar N Gilbert. Random graphs. *The Annals of Mathematical Statistics*, 30(4): 1141–1144, 1959.

[18] Luke Harding. What are the Panama Papers? A guide to history's biggest data leak. *The Guardian*, Apr 2016. URL https://www.theguardian.com/news/2016/apr/03/what-you-need-to-know-about-the-panama-papers.

[19] Mohammad Al Hasan and Vachik S. Dave. Triangle counting in large networks: a review. *Wiley Interdiscip. Rev.: Data Mining and Knowledge Discovery*, 8(2), 2018.

[20] Cay S. Horstmann. *Java SE 8 for the Really Impatient*. Addison-Wesley Professional, 1st edition, 2014. ISBN 0321927761, 9780321927767.

[21] ICIJ. *Offshore leaks*, 2016. https://offshoreleaks.icij.org/.

[22] Benedek Izsó, Zoltán Szatmári, Gábor Bergmann, Ákos Horváth, and István Ráth. Towards precise metrics for predicting graph query performance. In *ASE*, pages 421–431. IEEE, 2013.

[23] P. Kaluza, A. Kölzsch, M.T. Gastner, and B. Blasius. The complex network of global cargo ship movements. *Journal of the Royal Society Interface*, 7(48):1093–1103, 2010.

[24] Przemyslaw Kazienko, Katarzyna Musial, and Tomasz Kajdanowicz. Multidimensional social network in the social recommender system. *IEEE Trans. Systems, Man, and Cybernetics, Part A*, 41(4):746–759, 2011. DOI: 10.1109/TSMCA.2011.2132707. URL http://dx.doi.org/10.1109/TSMCA.2011.2132707.

[25] Jeremy Kepner, David A. Bader, Aydin Buluç, John R. Gilbert, Timothy G. Mattson, and Henning Meyerhenke. Graphs, matrices, and the GraphBLAS: Seven good reasons. In *ICCS*, volume 51 of *Procedia Computer Science*, pages 2453–2462. Elsevier, 2015.

[26] Jeremy Kepner, Peter Aaltonen, David A. Bader, Aydin Buluç, Franz Franchetti, John R. Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, Scott McMillan, Carl Yang, John D. Owens, Marcin Zalewski, Timothy G. Mattson, and José E. Moreira. Mathematical foundations of the

GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference, HPEC 2016, Waltham, MA, USA, September 13-15, 2016*, pages 1–9. IEEE, 2016. DOI: 10.1109/HPEC.2016.7761646. URL https://doi.org/10.1109/HPEC.2016.7761646.

[27] Grzegorz Malewicz, Matthew H. Austern, Aart J. C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *SIGMOD Conference*, pages 135–146. ACM, 2010.

[28] Vincenzo Nicosia and Vito Latora. Measuring and modeling correlations in multiplex networks. *Phys. Rev. E*, 92:032805, 2015. DOI: 10.1103/PhysRevE.92.032805. URL http://link.aps.org/doi/10.1103/PhysRevE.92.032805.

[29] Steve Ressler. Social network analysis as an approach to combat terrorism: Past, present, and future research. *Homeland Security Affairs*, 2(2), 2006.

[30] Yousef Saad. *Iterative methods for sparse linear systems*. SIAM, 2003. ISBN 978-0-89871-534-7. DOI: 10.1137/1.9780898718003. URL https://doi.org/10.1137/1.9780898718003.

[31] Chuan Shi, Yitong Li, Jiawei Zhang, Yizhou Sun, and Philip S. Yu. A survey of heterogeneous information network analysis. *IEEE Trans. Knowl. Data Eng.*, 29(1): 17–37, 2017. DOI: 10.1109/TKDE.2016.2598561. URL https://doi.org/10.1109/TKDE.2016.2598561.

[32] Gábor Szárnyas, Zsolt Kovári, Ágnes Salánki, and Dániel Varró. Towards the characterization of realistic models: evaluation of multidisciplinary graph metrics. In *MoDELS*, pages 87–94. ACM, 2016. URL http://dl.acm.org/citation.cfm?id=2976786.

[33] Lei Tang, Xufei Wang, and Huan Liu. Community detection via heterogeneous interaction analysis. *Data Min. Knowl. Discov.*, 25(1):1–33, 2012. DOI: 10.1007/s10618-011-0231-0. URL http://dx.doi.org/10.1007/s10618-011-0231-0.

[34] Dániel Varró, Oszkár Semeráth, Gábor Szárnyas, and Ákos Horváth. Towards the automated generation of consistent, diverse, scalable and realistic graph models. In *Graph Transformation, Specifications, and Nets*, volume 10800 of *Lecture Notes in Computer Science*, pages 285–312. Springer, 2018.

[35] Shiguo Wang. A comprehensive survey of data mining-based accounting-fraud detection research. In *International Conference on Intelligent Computation Technology and Automation*, ICICTA '10, pages 50–53. IEEE Computer Society, 2010. ISBN 978-0-7695-4077-1. DOI: 10.1109/ICICTA.2010.831. URL https://doi.org/10.1109/ICICTA.2010.831.

[36] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393):440–442, 1998.

Appendix

A.1 Edge overlap

The implementations of the edge overlap using the edge list representation and EJML are shown in Listing 7 and Listing 8 respectively. It is clear that the former is significantly more intuitive and easier to implement and read than the latter.

```
double intersection = 0.0;
for (N node1 : edgesInCondType.keySet()) {
    Collection<N> nodes = edgesInCondType.get(n);
    for (N node2 : nodes) {
        if (edgesInType.containsEntry(node1, node2) | edgesInType.containsEntry(node2, node1)) {
            intersection += 1;
        }
    }
}
return intersection/numberOfEdgesCondType;
```

Listing 7: Edge overlap implementation with edge list representation

```
for (T condType : types) {
    DMatrixSparseTriplet tripletsB = adjacencyMatrices.get(condType);
    DMatrixSparseCSC B = ConvertDMatrixStruct.convert(tripletsB, (DMatrixSparseCSC) null);
    DMatrixRMaj rowSumB = new DMatrixRMaj(n, 1);
    CommonOps_DSCC.sumRows(B,rowSumB);
    double d = SimpleMatrix.wrap(rowSumB).elementSum();
    for (T type : types) {
        if (type != condType) {
            DMatrixSparseTriplet tripletsA = adjacencyMatrices.get(type);
            DMatrixSparseCSC A = ConvertDMatrixStruct.convert(tripletsA, (DMatrixSparseCSC) null);
            DMatrixSparseCSC C = new DMatrixSparseCSC(n, n, 0);
            CommonOps_DSCC.elementMult(A,B,C, null, null);
            DMatrixRMaj rowSum = new DMatrixRMaj(n, 1);
            CommonOps_DSCC.sumRows(C,rowSum);
            double n = SimpleMatrix.wrap(rowSum).elementSum();
        }
    }
}
```

Listing 8: Edge overlap implementation with EJML