DHBW
Duale Hochschule
Baden-Württemberg
Mannheim

DLR

Clustering of RCE Workflow Graphs

Bachelorarbeit

für die Prüfung zum

Bachelor of Science

des Studiengangs Informatik

der Dualen Hochschule Baden-Württemberg Mannheim

von

Dominik Schneider

Abgabe am 14. September 2020

Bearbeitungszeitraum:	22.06.2020 – 14.09.2020
Matrikelnummer, Kurs:	6740511, TINF17ITIN
Abteilung:	Institut für Softwaretechnologie: Intelligente und Verteilte Systeme
Ausbildungsfirma:	Deutsches Zentrum für Luft- und Raumfahrt e. V. (DLR)
Betreuer der Ausbildungsfirma:	Dr. Alexander Weinert
Gutachter der Dualen Hochschule:	Ulf Runge

Erklärung

Ich versichere hiermit, dass ich meine Bachelorarbeit mit dem

Thema

Clustering of RCE Workflow Graphs

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.*

* falls beide Fassungen gefordert sind

Köln, den 14. September 2020

Abstract

RCE is an integration environment which allows to create automated workflows orchestrating multi-disciplinary simulation tools in a distributed manner. A workflow consists of components representing tools and connections between these components. The components can be grouped by users within the GUI by creating colored labels. This requires specialist knowledge and is a fully manual task. We investigate the feasibility of automating this task by applying graph clustering methods on such workflows.

To this end, we model graphs based on workflows by adopting components as vertices and connections as edges whereby we transfer connection properties to edge weights. We examine three different hierarchical clustering algorithms: edge betweenness, spectral bisection and agglomerative clustering. Additionally, we apply four different metrics to stop the algorithms when a cluster is found: cluster density, global clustering coefficient, average local clustering coefficient and modularity. We examine different mappings of edge weights in combination with the mentioned algorithms and metrics.

As groups in workflows have no canonical definition we evaluate our approach qualitatively. We consider 27 results of 1 008 parameter combinations as useful. The most expedient approach across multiple workflows is the edge betweenness algorithm with the modularity metric with an undirected graph representation. The scores for the metrics and the mapping vary across workflows and do not enable us to draw general conclusions. We show that our approach is feasible, whereas we remark that a quantitative study is necessary to validate our results in general.

Kurzfassung

RCE ist eine Integrationsumgebung für multidisziplinäre Simulationstools mit automatischer und verteilter Ausführung. Ein Workflow besteht aus Components, welche die Simulationstools repräsentieren, und Connections, welche die Components verbinden. Components können in der Benutzeroberfläche mit Hilfe von farbigen Labels gruppiert werden. Dieser Vorgang benötigt Fachwissen und wird rein händisch durchgeführt. In dieser Arbeit wird die Anwendung von Graph Clustering zur automatisierten Gruppierung von Workflows untersucht.

Hierfür wird ein auf dem Workflow basierender Graph modelliert. Eine Component wird als Knoten übertragen und eine Connection als Kante, wobei Eigenschaften einer Connection als Kantengewichte übertragen werden. Drei verschiedene hierarchische Clustering Algorithmen werden untersucht: Edge Betweenness, Spectral Bisection und Agglomerative Clustering. Zusätzlich werden vier verschiedene Metriken zur Erkennung von Clustern getestet: Cluster Density, Global Clustering Coefficient, Average Local Clustering Coefficient und Modularity. Diese werden in Kombination mit drei Arten der Gewichtung evaluiert.

Die Ergebnisse werden qualitativ bewertet, da keine kanonische Definition für Gruppierungen innerhalb von Workflows existiert. 27 von 1 008 verschiedenen Kombinationen führen zu nutzbaren Ergebnissen. Der Edge Betweenness Algorithms mit der Modularity Metrik und einer ungerichteten Graphrepräsentation scheinen am erfolgversprechendsten zu sein. Der angewendete Wertebereich für die Metriken sowie die Art der Gewichtung schwanken über die verschiedenen Workflows hinweg und lassen keine Rückschlüsse zu. Die Arbeit zeigt, dass Graph Clustering erfolgreich auf RCE Workflows angewendet werden kann. Jedoch ist eine quantitative Studie notwendig, um die Ergebnisse im Allgemeinen zu bestätigen.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

DLR German Aerospace Center

GUI Graphical User Interface

RCE Remote Component Environment

1 Introduction

The software RCE (Remote Component Environment) allows orchestrating different tools, mainly simulation tools, in a distributed manner. [1] A user can integrate their own software tools into RCE and connect multiple such tools to create executable workflows. A workflow consists of tools, also called *components*, which are chained together by *connections*. In order to increase the readability of large workflows, the user can create groups of components that form a logical unit by marking them with a color.

In order to support the user in determining such logical units, it would be useful to automatically detect components belonging together. We examine workflows originating from the field of aeronautical engineering where workflows with around 200 components and more are used to simulate an aeroplane as whole. Manually marking groups requires specialist knowledge about each part of a workflow which often consists of multiple disciplines. This restricts this task to a small group of associates who have to repeat the task with each change of the workflow.

To this end it is necessary to model RCE workflows as a data structure that easily allows identifying groups. We chose a graph as data structure. Workflows and graphs have in common that they consist of *vertices* (components) and *edges* (connections). Thus, it is fairly straightforward to model an RCE workflow as graph. We map connection properties to edge weights. Nevertheless, we do not know which properties are important and what is the most expedient scale of weights. Therefore, we examine how to choose a mapping which supports our approach.

We will identify groups automatically in an RCE workflow by using graph clustering. In general, *clustering* is the term for creating additional information of data by

grouping them. Retrieving this information by applying clustering on a graph is called *graph clustering*. Graph clustering describes the process of clustering vertices while taking the edge structure into account. [2] There exists a lot of research on graph clustering which allows adapting different already proven methods. We focus on methods which require as little information as possible about the workflow from the user to achieve a high degree of automation. As we do not know whether our approach is feasible with satisfying results, we examine methods which differ from each other to gather an overview over their practicability. The emphasis will be on hierarchical clustering methods which vary in terms of function and expected results. Additionally, we examine different metrics to identify clusters in combination with the algorithms to regulate the outcome.

As there does not exist an approach like this for RCE workflows yet, this work is to be seen as a feasibility study. Therefore, we will experiment with multiple combinations of parameters such as mappings, algorithms and metrics to identify correlations. In doing so we focus on a qualitative analysis at the end of our approach instead of a quantitative one due to the boundary conditions of a feasibility study.

First, we explain the necessary preliminaries for approach in Chapter 2. Afterwards we deepen the details of our motivation, the general problem and the used procedure of our approach in Chapter 3. In the main Chapter 4 we present our concepts and implementations to perform our approach. Subsequent to this we evaluate the fabricated results in Chapter 5. Closing this work we summarize our approach and present ideas for future work whereby we give pointers to related work in Chapter 6.

2 Preliminaries

This chapter describes the preliminaries for this work. In Section 2.1 we present the software RCE which provides the data for the graph clustering. We show the notations and definitions of graph theory used in this work in Section 2.2. At last we present graph clustering and different clustering methods in general in Section 2.3.

2.1 RCE

RCE (Remote Component Environment) is a tool for engineers and scientists to simulate and analyze complex multi-disciplinary systems, such as aircraft or satellites, in a workflow-based manner. To this end, RCE allows to create automated workflows which contain multiple disciplinary tools from different working groups. It also makes it possible to execute these workflows on a distributed network. For example the process of designing and evaluating a new type of airplane with different working groups with different know-how like aerodynamics or aeroelastics can be supported by a workflow with different simulation tools to simulate the airplane as a whole. For this purpose the individual tools are chained together so that simulation results are used as input data for other simulations. [1] RCE is an open source software, based on the Eclipse Rich Client Platform and was developed from 2006 through 2010 by Frauenhofer SCAI and the German Aerospace Center (DLR). Since 2010, DLR has been developing the software.

RCE has a variety of features and explaining all of them would be out of scope of this work. We present the ones relevant for understanding the context of the work. The software can be executed with or without a graphical user interface (GUI). RCE does

not provide its own simulation programs (also referred to as tools in the following), but it allows the user to integrate the existing tools as *workflow components*. The integrated tools can be shared with team members via a peer-to-peer network created by RCE and thus do not need to be re-integrated on each instance. Different tools can be linked via standardized inputs and outputs such that results are automatically fed into the next tool. Every so-called *connection* has a data type dependent on what information are transferred through the connection. The resulting flow definitions are referred to in RCE as *workflows*. By chaining the components, the effects of individual changes of the simulation parameters can be tracked more quickly than by executing the simulations manually and preparing the resulting data for further usage by hand. In addition to the user's own tools, RCE includes ready-made components that can be used to create specific structures within a workflow, such as loops. Furthermore, there are components which can process data, create different initial situations for a workflow or find extrema of a target function.

One feature of RCE is the capability to create labels inside a workflow within the GUI. A label is a colored rectangular marking which does not have an impact on the execution of a workflow. Instead, these labels can be used to add contextual information for users such as comments or descriptions. One possible and common use case is to group components on a semantic level by marking a group with a label. Potential groups of components are, e.g., components which read data for processing, simulation components, loops for optimization, components which persist results, or any group which is defined by a user created context. Figure 2.1 shows a part of a workflow whose components a grouped by different labels. There are five different labels with different colors. As can be seen it is possible that labels can overlap or contain each other which allows for hierarchical labeling.

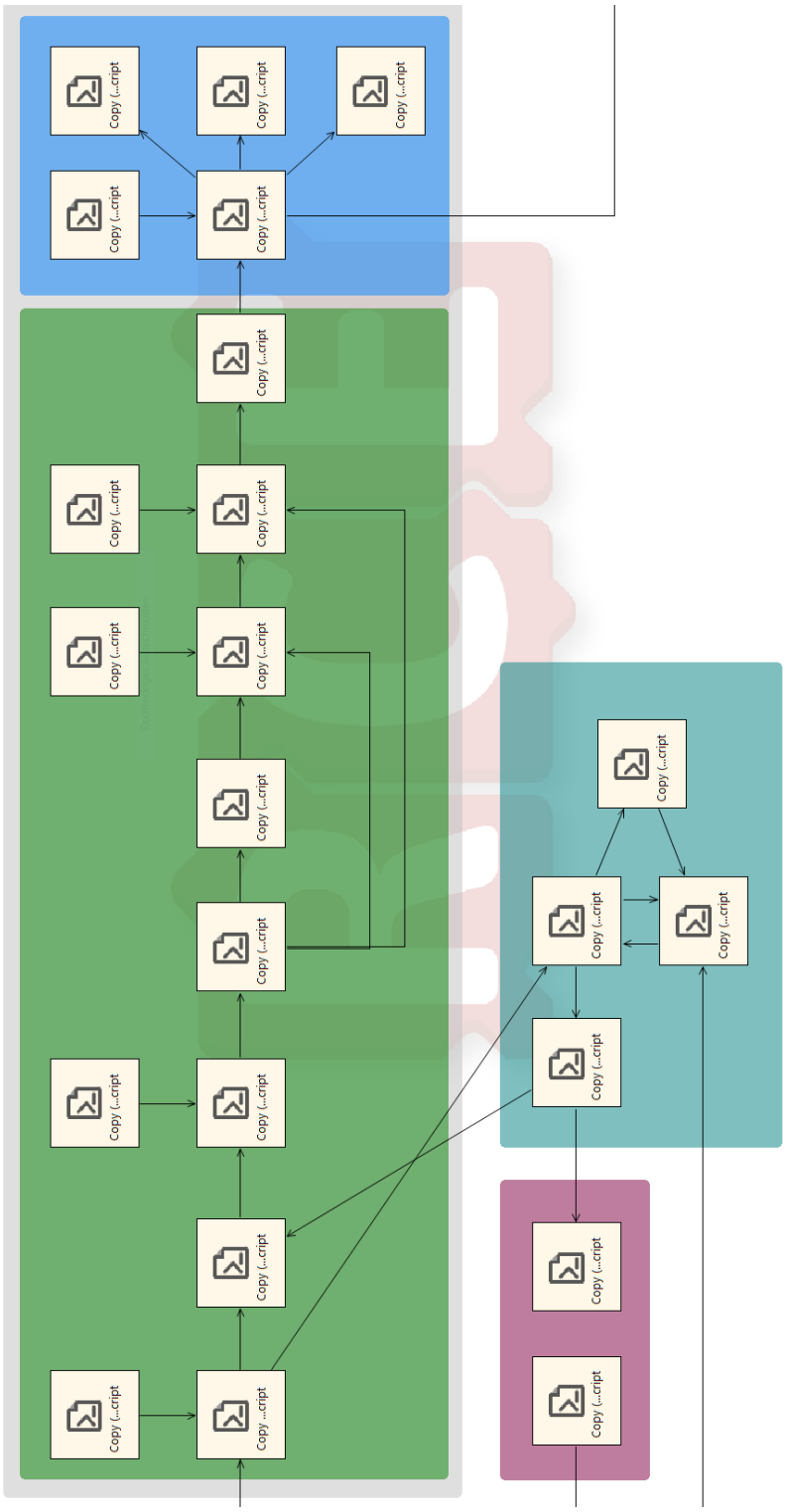


Figure 2.1: Picture section of an adapted RCE workflow with user labeled component groups (note: original components are replaced by dummy components).

2.2 Graph Theory

In this section we define some mathematical notations and general terms of graph theory as basis for the following considerations.

A *graph* $G = (V, E)$ consists of a finite set of *vertices* V and a set of *edges* $E \subseteq V \times V$. If for each edge $(v, v') \in E$ we have that $(v', v) \in E$ holds true then we call G *undirected*. A *weighted graph* $G = (V, E, \omega)$ consists of a graph (V, E) and a weight function $\omega \colon E \to \mathbb{Q}_{>0}$. A vertex u is the *successor* of a vertex v if $(v, u) \in E$ holds true. An *adjacency list* is a list containing a list of successors for all $v \in V$.

We use upper-case letters to name matrices and the associated lower-case letters with a vertical and a horizontal index to reference an element within the matrices. In addition to the adjacency list we define the *adjacency matrix* with the dimensions $|V| \times |V|$ of a graph $G = (V, E, \omega)$ where $V = \{v_0, \ldots, v_{n-1}\}$ and $a_{ij} = \begin{cases} \omega(v_i, v_j), & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases}$. We denote the transpose of a matrix A as A^T.

A path $v_0 \cdots v_k$ from v_0 to v_k is a sequence of vertices with $(v_i, v_{i+1}) \in E$ for each i with $0 \leq i < k$. We denote if a path exists between two vertices v and v' as $v \rightsquigarrow v'$. A graph is *connected* if for each pair of vertices $(v, v') \in V$ $v \rightsquigarrow v'$ and $v' \rightsquigarrow v$ hold true. We define the weight of a path $v_0 \cdots v_k$ within a graph $G = (V, E, \omega)$ as $\sum_{0 \leq i < k} \omega(v_i, v_{i+1})$. A path $v_0 \cdots v_k$ is a *shortest path* if there exists no other path between v_0 and v_k with a smaller weight. For a graph (V, E) the above definitions are defined as they would be for a graph (V, E, ω), where $\omega(e) = 1$ for all $e \in E$.

We define two graphs $G' = (V', E')$ and $G = (V, E)$. G' is a *subgraph* of graph G if $V' \subseteq V$ and $E' \subseteq E$. An *induced subgraph* is an subgraph (V', E') with $E' = E \cap (V' \times V')$. A *component* is a connected induced subgraph.

We define the *density* of a graph $G = (V, E)$ as $\delta(G) = \frac{2|E|}{\binom{|V|}{2}}$, if G is undirected and as $\delta(G) = \frac{|E|}{\binom{|V|}{2}}$ otherwise. A graph G is *complete* if $\delta(G) = 1$, i.e., if $E = V \times V$. A complete induced subgraph is called a *clique*. A clique, where no vertex can be added without losing the clique property, is called a *maximal clique*.

A *cut* $C = (S, T)$ is a partition of the vertex set V of a graph (V,E) into two nonempty sets S and $T = V \setminus S$. Each cut is uniquely defined by a set S. We define the *cut set* of C as $\gamma(C) = \{(v, v') \in E \mid v \in S, v' \in T\}$. In an unweighted graph we define the *cut size* as $c(C) = |\gamma|$ and $c(C) = \sum_{x \in \gamma} \omega(x)$ in a weighted graph with a weight function $\omega \colon \gamma \to \mathbb{Q}_{>0}$. A *minimal cut* is a cut with the minimal cut size of all possible cuts within a graph.

2.3 Graph Clustering

In this section we describe the preliminaries for our applied methods in the context of graph clustering.

Schaeffer [2] gives a good introduction to graph clustering and a general overview over the topic with its methods and difficulties. The term *clustering* describes the process of grouping data to reveal the underlying structure. The found groups are called *clusters* or *communities*. Applying clustering on a graph is called *graph clustering*. In detail, graph clustering describes the process of grouping vertices together while taking the edge structure into account. The goal is to have more edges within a cluster than between the clusters.

The problem with graph clustering is the ambiguity of the term cluster. Schaeffer shows that there are numerous definitions of what a cluster is and none of them is universally accepted. For that reason it is out of scope for this work to give a canonical definition of a cluster. In particular, a cluster should be connected, which means two things:

1. Two vertices should only be clustered if there exists a path between them.
2. The vertex set C should be connected in itself, i.e., at least on path between to vertices should only visits vertices in C.

The first point is equivalent to the definition of "connected" in terms of graph theory, whereas the latter one is a stronger definition. Schaeffer points out that it is generally agreed that a cluster should be dense as described in Section 2.2. The *cluster density*

$\delta(C)$ of a cluster C within an undirected graph $G = (V, E)$ where $C \subseteq V$ is defined as $\delta(C) = 2|\{(v, u) \mid v \in C, u \in C\}|/\binom{|C|}{2}$ and as $\delta(C) = |\{(v, u) \mid v \in C, u \in C\}|/\binom{|C|}{2}$ otherwise. According to Schaeffer, a "good" clustering consists of clusters where the cluster density is higher than the graph's density. Based on the already provided properties of a cluster there are multiple definitions for a cluster dependant on the use-case. Two extreme ones are:

- a connected component as the most general possible case of a cluster.
- a maximal clique as a very strict possible case of a cluster.

Whether two vertices belong together is usually based on a *similarity measure* which shows the "strength" of the bonding. There exist approaches which allow a vertex to be part of multiple clusters dependent on a membership degree. Such *fuzzy* relations are not part of our approach.

Schaeffer differentiates between *global* and *local* clustering methods. Methods of the former one cluster each vertex of a given graph whereas the latter one only cluster a subset of vertices of a graph. Which method is used depends on the number of vertices of a graph. According to Schaeffer global methods are able to deal with "a few millions of vertices on sparse graphs", which is far more than the graphs used in our approach have. Local clustering methods are used above this limit, e.g., clustering a graph representation of the World Wide Web. For our approach we use only global clustering methods for the mentioned reasons.

Fortunato [3] gives an overview over several "traditional" methods of graph clustering. These methods are *graph partitioning*, *hierarchical clustering*, *partitional clustering*, and *spectral clustering*. The aim of graph partitioning is to divide a graph into k groups of a predefined size. The constraint of this partitioning is that the number of edges between the groups has to be minimal. Hierarchical clustering methods do not require the number of clusters to find. As the name suggests, they produce hierarchical clusters which reveals information about the multilevel structure of the graph. There are two groups of hierarchical clustering algorithms, the *divisive* and the *agglomerative* clustering methods. Whereas the former ones start with the whole graph and iteratively split the graph into groups, the latter ones start

with each vertex as one cluster and iteratively merge them to larger groups. Which groups to split or to merge is decided by a similarity measure. Partitional clustering requires a given number k of clusters to find. Each vertex is treated as point within a metric space and a distance measure based on a similarity measure is defined. Afterwards the distance measure is minimized or maximized given the constraint of splitting the graph into k clusters. Spectral methods work by using eigenvectors and eigenvalues of matrices to partition a graph. The initial set of points, i.e., vertices, are transformed into a space "whose coordinates are elements of eigenvectors" [3]. Afterward standard clustering methods like the k-means (description in the next paragraph) algorithm are used to cluster the points. The advantage of this approach is that the transformation "makes the cluster properties of the initial data set much more evident" [3]. For our approach we use hierarchical and spectral clustering methods for the reasons mentioned in Section 4.2.

To apply graph clustering we rely on or mention several general known algorithms. The first algorithm is Dijkstra's shortest paths algorithm [4]. This algorithm uses a breadth first search to determine the shortest path between a given start and an end vertex. We do not use the k-means [5] algorithm for our approach, but it is used for other clustering methods. The algorithm only works within any desired metric space. The general procedure is to randomly select k points which then are the centers of each cluster. Afterwards each point is assigned to the closest cluster center. Finally the cluster centers are recalculated for each cluster. Both the last mentioned steps are repeated until the centers of the clusters do not change.

3 Methodology

In this chapter we present our general approach using graph clustering for automatic labeling of RCE workflows. First, we explain our motivation in Section 3.1. Afterwards we describe the main problems to solve in the context of workflows and clustering in Section 3.2. We show our strategy how to solve these problems in Section 3.3. At last we present a metric to evaluate our results in Section 3.4.

3.1 Motivation

RCE is used in a wide range of fields of engineering subjects, e.g., ship construction or aeronautical engineering. Developing these systems consists of lots of intermediate steps involving several specialist departments. Each of these departments has its own tools which all have their own functionalities. Within each department, the employees know how their tool works and how it has to be used. The other involved parties do not have this knowledge. That is why combining several tools and chaining them together in a workflow has to be a well coordinated task. This holds true particularly if the main task is something like creating a workflow which simulates an aircraft as a whole. The workflows resulting from such approaches can be large. One of the workflows used in this work derives from such an aircraft project and consists of about 160 components and 370 connections. Working with workflows of this size would be difficult if there was no possibility of marking parts of the workflows with labels as described in Section 2.1. These labels improve the usability by showing which components belong together. The disadvantage of labels is that they have to be placed manually. It is necessary that at least one employee knows about the

structure of the whole workflow and the semantical meaning of each component. In addition to that, labels introduce maintenance effort in order to keep up with changes of the workflow structure.

As usability is an important part of software development, there is always the effort to improve the software according to the user's needs. In this case creating labels automatically would save time at first creation of the workflow and reduce the needed maintenance effort. Because of this we examine whether RCE workflows can be labeled automatically without further knowledge of the semantical meanings of the used components.

The first step is to model a workflow in a well-known data structure where groups can be identified by algorithms. The second step is to apply such algorithms to find groups. The latter procedure is called clustering. The structure of an RCE workflow resembles that of a graph: components can be seen as vertices and connections as edges. Due to this resemblance we chose a graph as canonical data structure we described in Section 2.2. Additionally, clustering of graphs has been a well-researched topic in graph theory as can be seen in the variety of graph clustering methods shown by Schaeffer [2]. For those reasons we have decided to use graph clustering as approach to generate labels automatically. The scientific issue to which this work is dedicated to can be summarized as the following question: Is it possible to use graph clustering to generate RCE workflow labels automatically and if it is, under which circumstances is it possible?

3.2 Problem

There are two major problems which are to be solved in this work. The first one is to create a graph on basis of a workflow to enable graph clustering. The second problem is finding clusters, which in general are defined ambiguously themselves and additionally are used to find groups which have no definition at all. We describe the problems in the mentioned order.

A workflow as it is can not be used for graph clustering. Although a workflow and a graph have obvious similarities regarding the general structure, i.e., components and connections resemble vertices and edges, a graph can not be created directly from a workflow. To use graph clustering algorithms on RCE workflows, the workflow structure has to be extracted and mapped to a graph. While components will be used as vertices without changes, the connections have to be adjusted. Connections inside an RCE workflow have several properties, which can not be directly mapped to the properties of the edge representing them:

- Each connection has a data type.
- Each connection has a constraint.
- There can be multiple connections between a pair of components.

There is only one property which can directly be mapped to an edge property:

- A connection has a direction.

Since these properties are described in Section 4.1, we will not go further into details at this point. The basic idea for the mapping is to use weights for the former mentioned properties. For each edge the mapped weights of the connection properties are summed up. The naive approach would be to keep one edge for each connection which results in a multigraph. Using this approach would lead to incompatibilities with many graph clustering algorithms. Thus, we decide to add up all the edges' weights corresponding to the connections between a pair of components which results in one single edge. The difficulty of using weights for the approach is finding a good mapping: The graph resulting from the mapping has not only to represent the structure of the workflow, but rather the idea behind the workflow. As graph clustering algorithms use edge weights to find clusters, a bad mapping will lead to bad results. The problem is to find a mapping representing the idea behind a workflow as one precondition of this approach is to find groups without knowledge of the semantical meaning of the workflow. Solving this issue might be achieved by assuming general concepts and properties of workflows in the context of simulations. In Section 4.1 the latter approach is described in detail.

For the following introduction to graph clustering and cluster properties we recall an excerpt from Section 2.3. Graph clustering is the term for grouping vertices taking into account the edge structure of a graph. This process aims to create clusters where as many edges as possible are inside a cluster and as few as possible connect the cluster to the outside. In general, there is no universally accepted definition of clusters in graphs. It is common sense that a cluster should be *connected*. [2] That means two things:

1. Two vertices should only be clustered if there exists a path between them.
2. The vertex set C should be connected in itself, i.e. at least on path between to vertices should only visits vertices in C.

The first point is equivalent to the definition of *connected* in the context of graphs (see Section 2.2). The second point is a more strict form of the first point and intuitively means that a cluster with this property is "more" self-contained than a cluster with the property connected defined by the first point. In general, it is agreed that a cluster is a good cluster if the subgraph induced by a cluster is dense. [2] With this definition it is possible to gather an idea about what a cluster is. Nevertheless this definition is ambiguous and always has to be adapted to a particular use-case. In addition to that, the groups we try to identify in a workflow have no definition at all. They are created by users and it is possible that different users might mark groups differently. Due to this it is definitely impossible to find the correct solution, it is only possible to find any possible solution. This ambiguous combination makes it difficult to determine under which circumstances a clustering can be successful, especially since the clustering depends on the mapping which itself introduces problems as described. Chapter 4 describes which procedures including different mappings and several clustering algorithms are used to find a general approach to identify groups in workflows.

3.3 General Procedure

The general procedure of the idea behind this approach consists of two major steps:

1. Creating a graph representing the used workflow.
2. Applying graph clustering algorithms to the created graph.

If this approach shows that graph clustering can be used to determine labels for groups, there would be a third step to complete the procedure to be an RCE feature:

- Creating labels based on the found clusters and drawing them inside the graphical representation of the workflow.

If this approach shows that labels can not be found with acceptable results, there is no need to map the clusters to labels. As the focus of this work strictly lies on determining the feasibility of using the mentioned two major steps for creating usable labels, the third step is not part of this work. We describe the different combinations of mappings, graph clustering algorithms and evaluation workflows in detail in Chapter 4 and Chapter 5 respectively. In the following we give an overview over the covered subjects.

To create a graph representing a workflow there has to be a mapping rule. This rule mainly consists of transferring workflow properties to graph properties. There will be no "correct" mapping, only possible mapping solutions. A mapping should satisfy several requirements:

- The mapping considers the simulation background of workflows.
- The mapping is adjusted to a specific graph clustering algorithm.
- The mapping is general so it can be applied to different kinds of workflows.

To conclude this means that the mapping should not be focused on workflow details. Instead it should have the semantics of simulation workflows embedded in order to be independent from a specific workflow. Despite that the mapping can be dependent on a clustering algorithm because in a productive use-case there will be only one algorithm as part of automatic labeling. We present different approaches how to find a good mapping in Section 4.1.

The goal of this work is to find a graph clustering algorithm that produces clusters that are usable for labeling an RCE workflow. To achieve this goal we examine

different graph clustering algorithms. These differ from one another to cover the wide field of these algorithms. We chose to implement and evaluate three different algorithms:

1. Edge Betweenness Clustering
2. Spectral Clustering
3. Agglomerative Clustering

All of the mentioned algorithms are part of global clustering algorithms. We chose them over local clustering methods as these are used for extremely large graphs [2] which is not our use-case. Each algorithm works hierarchically. While the edge betweenness and the spectral clustering algorithms are divisive algorithms, the agglomerative clustering algorithm works as the name suggests. We decided to use hierarchic clustering methods because RCE workflows can be labeled hierarchically themselves. It is possible that user create overlapping groups which are ignored by this approach. Spectral clustering works very differently compared to the other algorithms, as it is based on linear algebra operations whereas the other two work with general graph related methods. We describe further details of the used algorithms and why we chose them in Section 4.2.

3.4 Evaluation Metric

All of the developed mappings with the applied algorithms will lead to results as the algorithms are designed to terminate independent of the resulting clusters. In the end the results have to be evaluated in the context of real usage of the algorithms. This leads to an evaluation from the point of view of an RCE user. We assume that all properties of the resulting clustering would be compared to manually labeled groups. As our motivation is to improve the usability, this comparison is the base of our metric. The properties which can be evaluated in this context on first sight are:

- Number of different clusters.
- Position and size of each cluster.

Both metrics have to be evaluated to decide whether the approach was successful or not. That is due to the fact that there may be cases where one metric might indicate a good result meanwhile the other one might not. For example there could be the correct number of clusters identified but the clusters found do not match the desired groups. The opposite case would be if there were one or two well matching clusters but there are eight groups to be found. At this point we anticipate our evaluation to remark that our evaluation is only qualitative, due to the restricted number of workflows with labelings available for our approach. Additionally, the labelings we use are partly created by ourselves as some workflows did not have a labeling before. Therefore, our results are rather recommendations for clustering approaches of new workflows than generally valid insights.

As described in Section 4.2.1 there also metrics to evaluate one or multiple clusters without the context of our use-case. These metrics are based on density and other graph properties. They are part of the algorithms to determine when a cluster has been found. It might be the case that the applied algorithms find good clusters as judged by those metrics, but are not good in terms of our self-defined metrics. As we orientate ourselves with this approach towards the use-case described in Section 3.1, we use our self-defined metrics to evaluate the final results.

4 Clustering

In this chapter we present our approach to apply well-known graph clustering techniques to RCE workflows. As described in Section 3.3, the first step is to create a graph based on the workflow. We show this mapping in Section 4.1. Afterwards we have to apply the selected graph clustering algorithms to our approach on the created graph. First we lay out our criteria for the selection of the clustering algorithms and explain their theoretical background in Section 4.2. At last we show our own developed graph clustering tool including the implementations of the chosen clustering algorithms in Section 4.3.

4.1 Mappings

In this section we present our ideas to create a graph based on an RCE workflow. Although a workflow looks like a graph at first sight, for example see Figure 2.1 on Page 5, there are differences which prevent a direct adoption of the workflow structure. Before we analyze the differences between a graph and a workflow, we describe the properties of the latter.

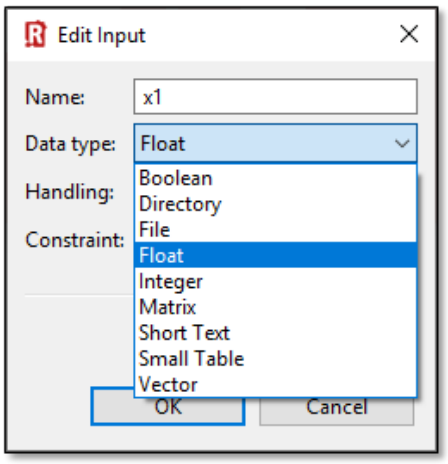
As described in Section 2.1 a workflow consists of components and connections. Components are executable units for different purposes. Connections are used to transfer information between components. Whereas components have no relevant properties for our approach, connections have different configurable properties which influence our mapping. Figure 4.1 shows all properties of an RCE connection:

- Each connection has a data type (4.1a).

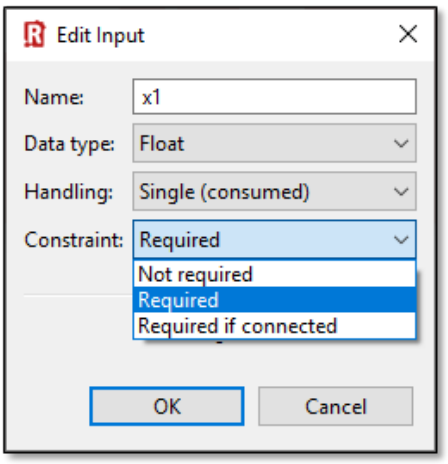
- Each connection has a *constraint* (4.1b).
- Multiple connections can have the same start and end component (4.1c).
- Each connection has a direction.

Theoretically, the former two mentioned properties are not connection properties. Instead they are the properties of a component's *input*. Each component can have multiple inputs and *outputs* where a connection connects an output to an input. As we want to abstract the structure of a workflow we treat these properties as connection properties. We leave out the *handling* property of a connection. This property displays whether the transfered date of a connection can be used a single time, multiple times or can change during the workflow runtime. In case of the changing date we do not know how many times it changes or whether it changes at all. Thus, using this information requires semantical knowledge of the workflow leading to the exclusion from our approach.

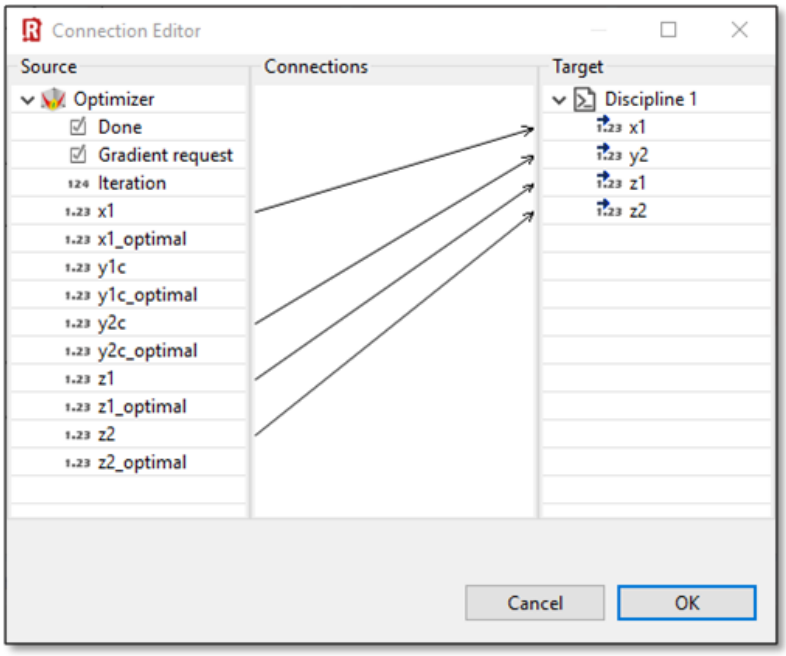
The purpose of connections is to transport data, so there are nine different data types for the information exchange. The spectrum ranges from simple data types like `Integer` or `Boolean` to complex data types like `Matrix` or `File`. Constraints are RCE specific properties. A constraint describes the importance of a connection, as it defines the start condition for the component's execution. The four constraints are: "Required", "Not required", "Required if connected" and "None". "Required" means that the component cannot be executed without the data of this connection. "Not required" means that the component is executed without the data of this connection. Theoretically, "Required if connected" cannot be a connection property as a connection can not have a property if it is not connected, i.e., the connection does not exist. As mentioned before, we ignore this to obtain a consistent handling of the properties. The meaning of this constraint is that if the connection exists, the component cannot be executed before the data of this connection is received. The constraint "None" exists due to technical reasons within RCE and so has no high importance to us. Although the workflow editor of RCE always shows only one connection in the workflow overview, there can be multiple connections between a pair of components as a component can have multiple in- and outputs. All connections



(a) Connection data types.



(b) Connection constraints.



(c) Multiple connections.

Figure 4.1: RCE screenshots of configurable connection properties.

have a direction as their purpose is to transfer information from a defined start point to a defined end point.

Our goal is to preserve most of the information of the workflow when creating a graph. As we have no constraint of the components to take care for, we simply create a vertex for each component. Within our implementation, we preserve the unique ID and the name of each workflow component in the related vertex, where the ID is used within our algorithm implementations (for example how described in Section 4.3.4) to achieve a deterministic behavior. Using a directed graph preserves the direction property of each connection. The data type and the constraint property of a connection have to be mapped to a different graph property. Using a directed and weighted graph enables us to modify the weights according to our weighting of the properties. Although an adoption of multiple connections between two components is possible in terms of graph theory, it is not practical as in general graph clustering algorithms do not work with such so-called multigraphs. We solve this issue by combining the edges between a pair of components to one edge where the edge weights are summed up.

Finding a "good" mapping is no simple task. We want to develop a mapping which works with different workflows because we cannot presume which workflows are used in production. Nevertheless, we can adapt the mapping to specific clustering algorithms, as there would be only one algorithm used for an RCE feature. At this point we anticipate our used clustering algorithms. The edge betweenness (Section 4.2.1) and agglomerative (Section 4.2.3) clustering algorithm are both based on shortest paths. As in our approach shortest paths are defined by the sum of edge weights, we can directly influence the behavior of the algorithms by a chosen mapping. The spectral clustering algorithm (Section 4.2.2) also works for weighted graphs, but we cannot influence it in the same way we can do it with shortest paths, as the calculations are more complex. We rely on the spectral clustering approaches of von Luxburg [6]. The weighting of these show the similarity between vertices: the higher the weight, the higher the similarity. In contrast to this, our agglomerative implementation described in Section 4.3.6 clusters vertices together which have short shortest paths, so the weights inside a cluster are low. The case is different with

edge betweenness: frequently visited edges along shortest paths are removed as they are seen as inter cluster connections. Thus, we assume that the edge weights cannot fully control where the shortest path lie, as the graph's structure with assumed few inter cluster edges has a large impact on the algorithm. This given structure will lead to high betweenness scores of those few inter cluster edges as there are not many paths between clusters, so they have to include the mentioned edges. Newman [7] assumes that approaches with the edge betweenness algorithm we implemented will lead to poor results if similar vertices have an edge with a low weight. This means that the distance between those edges is short. He remarks that edges between similar vertices will attract shortest paths and with that a high edge betweenness score which leads to their removal what is definitely undesired. We agree partly with his remark although we think that the graph structure with the few inter cluster edges has more influence than the edge weights. We will pick up this subject in our evaluation in Section 5.3.

As workflows can differ a lot from each other, we will make several assumptions about the information flow to generate several mappings. A naive approach is to map data types with probably more information than other data types to a high weight. For this we create an ascending ordering in terms of the assumed information amount a data type contains: `Boolean`, `Integer`, `Float`, `Vector`, `Directory Reference`, `Short Text`, `Small Table`, `Matrix` and `File Reference`. In addition to that we map a constraint to a high weight, if it shows high importance of a connection. This results in this ascending ordering: "None", "Not required", "Required if connected" and "Required". As "None" has no importance for our approach, we always weight it with zero. We weight the data types from 12 to 20 and the constraints from 3 to 5 as we assume that the data type of a connection is more important than the constraint. We choose the range of the data types in the range between 10 and 20 to avoid that a data type can have more than twice the weight of another data type. Due to the different meanings of high weights within the algorithms, we will evaluate the same mapping in reverse. The naive approach would be to switch the high weights with the low weights. The problem with this approach is that it ignores the behavior with multiple connections between two components: Even if the weights are reversed, a pair of components with many connections between them will result in an edge with

a high weight due to summing up the weights of each connection. That is why we use the reciprocal value of the resulting edge weight as final weight which guarantees the desired behavior. We will test both mappings with each algorithm as we have to include the semantical background of simulation workflows. We assume two possible behaviors of the information flow:

1. Clusters exchange much information within a cluster for calculations which results in high weights within a cluster.
2. Clusters exchange much information between them which results in high weights between the clusters.

We describe the necessity of testing both mappings with the agglomerative clustering algorithm. Our implementation of the agglomerative clustering algorithm clusters vertices together which have a short shortest path between each other, i.e., low weights. In case of a high information exchange within a cluster the connections with data types which contain much information should have low weights to achieve a clustering of the associated vertices. With a high information exchange between clusters, it behaves reversely: the connections with data types which contain much information are between clusters and should have high weights to prevent a clustering of the associated vertices. In case that our assumptions are not correct we test a mapping where each edge in the resulting graph is weighted with 1, which means that the algorithms behave on this graph as if it were an unweighted graph.

There might be other useful approaches for mappings, but we only test the three mentioned ones to avoid having too many combinations of parameters for the evaluation.

4.2 Algorithms

In this section we present the theoretical backgrounds of our chosen algorithms: edge betweenness clustering in Section 4.2.1, spectral clustering in Section 4.2.2 and agglomerative clustering in Section 4.2.3. We describe the methods to determine

a cluster with the first mentioned algorithm in Section 4.2.1. Nevertheless the mentioned methods are applied on all our implemented algorithms.

Why we chose the mentioned algorithms depended on several factors. On the one hand we want algorithms which are well-known in literature as our approach is a feasibility study and does not aim to apply very special and customized algorithms. In addition, the graphs we use are self-defined and variable so the algorithm has to be quite general to be applicable to our use-case. On the other hand we want a collection of algorithms which behave differently to evaluate which approaches work and which do not. An inevitable exclusion criterion for clustering algorithms is the requirement of given number k of clusters to find, as we do not want do use information beside the structure of a graph. Therefore, the graph partitioning and the partitional clustering approach described in Section 2.3 are not applicable to our use-case.

The workflows we work with have a size where we do not have to rely on local clustering methods which are used for very large graphs like the World Wide Web and only work on a subset of the graph, so we use global clustering algorithms which work on the full set of vertices of a graph [2]. In regard to the possibility of hierarchic labeling of RCE workflows, we chose to work with clustering algorithms which also work hierarchically. Besides that, hierarchically clustering algorithms make up a large part of graph clustering literature. The advantage of hierarchical clustering is that we have a valid clustering at each step of the algorithm containing all vertices and only the stopping condition determines what are the final clusters. This gives us more freedom on evaluating the different approaches. Additionally, hierarchical clustering algorithms work without defining the number of clusters in advance.

We chose the edge betweenness clustering algorithm because it is widely known and delivers good results [8]. The spectral clustering algorithm works in a completely different way by using linear algebra methods. Thus, it satisfies our goal to experiment with different behaviors. We remark that spectral clustering methods in general are not necessarily hierarchical, whereas our implementation is. Both the mentioned algorithms are divisive algorithms, so they split the graph starting with one cluster containing all vertices of a graph. Our third chosen algorithm, the agglomerative

clustering algorithm, works the other way round, starting with each vertex inside a single cluster. This enables us to contrast the two different kinds of algorithms within the family of hierarchically algorithms. We assume that we can gather a good overview over the application of graph clustering on RCE workflows with this selection of algorithms.

4.2.1 Edge Betweenness

Edge Betweenness itself is not a clustering algorithm, but rather than a measure of centrality for edges. The term is introduced by Newman and Girvan [8]. It is based on *Betweenness Centrality*, which is a score for vertices introduced by Freeman in his studies about communication in social networks [9]. For the remainder of this section, we fix an undirected graph $G = (V, E)$. Recall the definition of betweenness due to Freeman: Let $g(v)$ be the betweenness of a vertex $v \in V$ with

$$g(v) = \sum_{s \neq v \neq t} \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{4.1}$$

where σ_{st} is the number of all shortest paths between the vertices s and t and $\sigma_{st}(v)$ is the number of shortest paths from s to t through v. Freeman's approach is about finding actors in a social network, i.e., vertices which can control communication between two persons. Vertices with a high betweenness centrality score have a high probability that messages pass through them.

Newman and Girvan port this idea to their approach of detecting communities inside a network. They assume that communities of a network are highly connected within each other over a high amount of edges between the members of the community but are connected to other communities over less edges. Based on Freeman's Betweenness Centrality, edges connecting communities have a high probability that information flows through them. Figure 4.2 shows a graph showcasing the assumed structure in Newman and Girvan's approach. Newman and Girvan's idea is to find and remove those inter-cluster edges to find communities inside of a network. Based on this thought and referring to betweenness centrality, the edge betweenness $g(v, v')$ of an

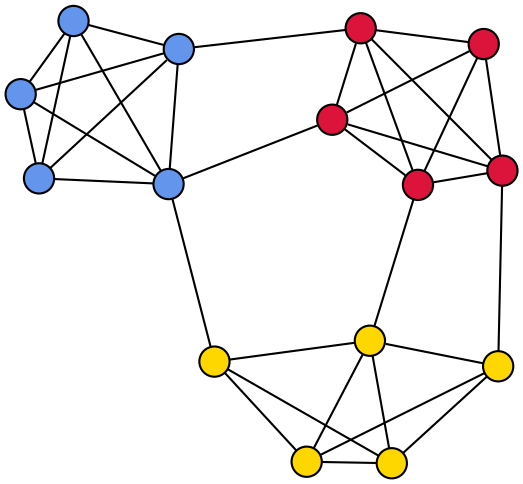


Figure 4.2: Graph with highly connected communities and few inter-cluster edges.

edge $(v, v') \in E$ is defined as

$$g(v, v') = \sum_{s \neq t \wedge v \neq v'} \frac{\sigma_{st}(v, v')}{\sigma_{st}} \tag{4.2}$$

where σ_{st} is the number of all shortest paths between the vertices s and t and $\sigma_{st}(v, v')$ is the number of shortest paths from s to t through the edge (v, v').

The algorithm to cluster a network, i.e., a graph, as formulated by Newman and Girvan is as follows [8, p. 5]:

1. We calculate the edge betweenness of every edge in the network.
2. We remove the edge with the highest betweenness score, or randomly choose one such if more than one edge ties for the honour.
3. We recalculate betweenness scores on the resulting network and repeat from step 2 until no edges remain.

As Newman and Girvan point out, the recalculation in step 3 is crucial for the

algorithm in the case of multiple inter-cluster edges. In that case it is not guaranteed that all inter-cluster edges receive a high betweenness score as in most cases the shortest path between vertices of different clusters will contain only one inter-cluster edge. With recalculation it is possible that several inter-cluster edges receive high scores after other inter-cluster edges with high scores are removed and the shortest paths between clusters change.

We remark that Newman and Girvan defined their algorithm for undirected and unweighted graphs. Nevertheless we can apply this algorithm to directed and weighted graphs as the edge betweenness score is calculated with shortest paths in a graph which can also be derived from directed, weighted graphs. In the case of weighted graphs, Newman [7] presented an edge betweenness algorithm designed for this type of graphs. Although he shows that his approach leads to good results, we decided against using his approach. The reason for that is the requirement of a weighting where a high value indicates a strong relationship between vertices. As we experiment with different mappings where edge weights may indicate a relationship of vertices in a different way, we cannot fulfill this requirement. Nevertheless, Newman's approach can be part of future work in the context of our approach.

The algorithm removes all edges of the graph on which the algorithm is applied. Analyzing the order in which edges are removed can reveal the underlying structure of the graph. Nevertheless, our aim is to find clusters and not analyzing the structure of the graph. Thus, we want to stop the algorithm when a cluster is found.

As described in Section 2.3, it is no simple task do determine when a cluster is found due to the ambiguity of the cluster term's definition. Additionally, the structures of the used workflows for this approach differ and so may do the cluster structure. To determine when a cluster is found we need a score which is calculated after each removal of an edge splitting the graph, as this changes the structure of the graph. We experimented with different metrics to measure the quality of the clusters. Intuitively, the first approach was using the *global clustering coefficient* defined by Luca and Perry [10] and the *local clustering coefficient* defined by Watts and Strogatz [11]. Both are used to determine which vertices cluster together and are already implemented by our graph library (further details in Section 4.3.2). The

global clustering coefficient depends on the number of triplets in a graph and only works with undirected graphs. A triplet is a group of vertices consisting of three nodes where at least two are connected. With exactly two connected vertices it is called an open triplet and with all three connected it is called a closed triplet. We denote the number of closed triplets in the graph as t_c and the number of all triplets as t_a. The global clustering coefficient C is defined as

$$C = \frac{t_c}{t_a} \ . \tag{4.3}$$

The local clustering coefficient is determined per vertex and shows how close its adjacent vertices are to being a clique. We define the neighborhood N_v of a vertex v as $N_v = \{v' \mid (v, v') \in E \vee (v', v) \in E\}$. The local clustering coefficient of a vertex is defined as number of edges in its neighborhood divided by the maximum possible number of edges in this neighborhood. Formally, the local clustering coefficient C_v for directed graphs is defined as

$$C_v = \frac{|\{(s,t) \in E \mid s,t \in N_v\}|}{|N_v|(|N_v| - 1)} \tag{4.4}$$

and as

$$C_v = \frac{2|\{(s,t) \in E \mid s,t \in N_v\}|}{|N_v|(|N_v| - 1)} \tag{4.5}$$

for undirected graphs as there are half the maximum possible edges. To apply the local clustering coefficient on a group of vertices instead of a single vertex, the average local clustering coefficient for a group of vertices V' [12, p. 142] is defined as follows:

$$\overline{C} = \frac{1}{|V'|} \sum_{v \in V'} C_v \tag{4.6}$$

Newman proposed a quality measure for clusters of undirected and unweighted graphs called *modularity* [13]. The idea behind this quality measure is not to evaluate the found clusters themselves rather than evaluating the clusters in combination with the structure of the group of clusters as a whole. This quality measure is widely known in literature and is used by several clustering approaches [13–17] which is the reason why we chose to include this metric into our feasibility study.

Newman [13] defined the modularity Q for the $k \times k$ symmetric matrix E of a graph split into k clusters as

$$Q = \sum_{i=1}^{k} (e_{ii} - {a_i}^2) \tag{4.7}$$

where e_{ij} is the fraction of edges connecting cluster i to cluster j and $a_i = \sum_j e_{ij}$ which is the fraction of edges connecting to cluster i. As the graphs we use are not only undirected and unweighted ones, we need to adapt the modularity metric to our approach. How we transform an directed graph into an undirected graph is described in Section 4.2.2. To apply the modularity metric, we always use the simple method of transformation if the input graph is not already undirected. Schaeffer [2] formulated a version of modularity for weighted graphs over a clustering into k clusters $C_1 \dots C_k$ of a graph $G = (V, E, \omega)$ as follows:

$$Q = \sum_{i=1}^{k} \varepsilon_{ii} - \sum_{\substack{i \neq j \\ i,j \in \{1 \dots k\}}} \varepsilon_{ij} \tag{4.8}$$

where

$$\varepsilon_{ij} = \sum_{\substack{(v,u) \in E \\ v \in C_i, u \in C_j}} \omega(v, u) \ . \tag{4.9}$$

She describes the behavior of the quality measure to be resulting in high scores if more edge weights are inside the clusters and less outside. We cannot directly apply this approach as the maximum and minimum score is dependent on the size of the edge weights of a graph which prevents us from evaluating different graphs with a fixed spectrum of goal scores. We solve this problem simply by dividing Q by the sum of all edge weights in G:

$$Q' = \frac{Q}{\sum_{(v,u) \in E} \omega(v, u)} \tag{4.10}$$

This results in a minimum value of -1 and a maximum value of 1 for each graph.

In addition the above mentioned metrics we experimented with the cluster density metric described in Section 2.3 to determine a final cluster. The cluster density metric, the average local clustering coefficient and the global clustering coefficient

delivers scores in $[0, 1]$. We assume that applying these metrics on clusters leads to scores closer to 1, whereas applying them on a whole graph leads to scores closer to 0. For the modularity metric we also consider only the mentioned interval, as the scores within this interval mean that more edges are within than between clusters.

All four metrics are used within our implementations of the edge betweenness, the spectral (Section 4.2.2) and the agglomerative (Section 4.2.3) clustering algorithm. The former two algorithms require a special algorithm implementation to apply the modularity metric. The details are described in Section 4.3.4 for edge betweenness clustering and in Section 4.3.5 for spectral clustering respectively.

4.2.2 Spectral Methods

Spectral clustering methods are linear algebra methods applied on graph properties to achieve a clustering of a graph. To be precise, the eigenvalues and eigenvectors of graph related matrices are used, e.g., the adjacency matrix. Donath and Hoffman [18] introduced these procedures in their original paper from 1973. In the same year, Fiedler [19] presented his results on applying spectral methods on the *Laplacian* matrix to which we will return later. Since then spectral methods have been examined on different types of graphs with mixed results. [20] As this work is no guide to applied spectral methods nor has the goal to explain the used linear algebra techniques in detail, we refer to von Luxburg's guide for spectral clustering methods [6] and Chung's monograph on spectral graph theory [21] for a thorough introduction on the subject. The latter one is recommend by a wide range of papers related to spectral clustering methods on which we rely. [2, 6, 22] Nevertheless we give a high-level overview over the used techniques and the necessary preliminaries.

The already mentioned Laplacian matrix is the base for several spectral clustering algorithms. Given an undirected, weighted graph $G = (V, E, \omega)$, the unnormalized Laplacian matrix L is defined as $L = D - A$, where D is the *diagonal* matrix and A is the adjacency matrix of the graph G. We recall the definition of the adjacency matrix as remainder for this section. The adjacency matrix A with the dimensions

$|V| \times |V|$ is defined with

$$a_{ij} = \begin{cases} \omega(v_i, v_j), & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases} . \tag{4.11}$$

According to von Luxburg [6] the diagonal matrix D with the dimensions $|V| \times |V|$ is defined with $d_1, \ldots, d_{|V|}$ on the diagonal where

$$d_i = \begin{cases} \sum_{j=1}^{|V|} \omega(v_i, v_j), & \text{if } (v_i, v_j) \in E \\ 0, & \text{otherwise} \end{cases} . \tag{4.12}$$

One of the most common general spectral clustering algorithm [6, p. 399] uses the Laplacian matrix as defined above. The algorithm has the aim to find k clusters, where k has to be known beforehand. Summarized, the algorithm works as follows: First the k first eigenvectors of the Laplacian matrix are calculated. These eigenvectors are used as columns of a new matrix. The rows of this new matrix are then treated as points in $\mathbb{R}^k$ and are clustered by the standard k-means algorithm (see Section 2.3). Theoretically, this algorithm could be applied to our use-case. The reason why we chose not to use this algorithm is its requirement of a given k as number of clusters. That is due to our motivation not to use information from the workflow given by a user.

Another approach to use spectral clustering methods on graphs is to take advantage of the so-called Fiedler vector. Fiedler showed that the second smallest eigenvalue and the corresponding eigenvector, the Fiedler vector, of the Laplacian matrix has properties which can be used to draw conclusions about the graph's structure. [19] Each component of the Fiedler vector has a corresponding vertex of the Laplacian matrix' graph. He called this eigenvalue the "algebraic connectivity" as it equals 0 if and only if the graph is not connected. The Fiedler vector can be used to split the graph into multiple clusters. [2] If there are only two resulting clusters this is called *spectral bisection* [23]. Spielman and Teng [24] describe the idea of spectral partitioning as finding a splitting value s to which the several components of the Fiedler vector are compared against and dependent on the result of the comparison

are put in one of the resulting clusters. For example let a Fiedler vector $\vec{z}$ be $\begin{pmatrix} 2 \\ 3 \\ -1 \end{pmatrix}$ and $s = 0$. For spectral bisection the comparison is a simple size comparison [24], whereas the comparison for resulting clusterings with more than two clusters require a different approach [2]. The latter case is not part of our approach so we only focus on the size comparison. According to that all values of the Fiedler vector which are at least 0 are put in one cluster, the remaining values are put in the other cluster respectively. The result of our example consists of the two sets $\{2, 3\}$ and $\{-1\}$. As each component of the Fiedler vector corresponds to a vertex of a given graph, the resulting clustering is derived from the resulting sets of the comparison.

As the value s directly affects the result of a clustering, there exist several approaches to find a value s to achieve "good" clusterings. Spielman and Teng present several approaches to determine s which are often used in literature. Among others these are the *ratio cut* [25] and the *sign cut*. With the first one the goal is to achieve a split which results into equal sized partitions with regard to the number of vertices in each cluster. The splitting of the ratio cut behaves like a minimal cut. The goal of a resulting equipartition in combination with the minimal cut introduces additional complexity. This new problem containing a balancing between the two mentioned constraints is NP-hard [6]. The sign cut uses $s = 0$, i.e., the sign of each eigenvector component is the determining factor. Von Luxburg [6] shows that the sign cut is a relaxation of the ratio cut for $k = 2$. We decided to use the sign cut as relaxation of the ratio cut in combination with an iterative execution of spectral bisection on each resulting subgraph according to Schaeffer [2]. This enables us to find more than two resulting clusters. As described in Section 4.3.5 this approach allows reusing the recursive algorithm of the edge betweenness implementation. This leads to better comparability between the methods presented in this thesis, as only the method of identifying the subgraphs is different. The general procedure however, remains constant over all methods. This includes the applied metrics as stopping condition of the algorithm. As mentioned in Section 4.2.1, all the metrics cluster density, global clustering coefficient, average local clustering coefficient and modularity are used where modularity requires a special implementation described in Section 4.3.5.

The used Laplacian matrix for the described procedure is based on an undirected

graph. As RCE workflows are directed, applying spectral methods on a directed graph representation of a workflow would improve the evaluation as there would be more results to compare. The problem is that the methods for spectral clustering of undirected graphs are not directly applicable to directed graphs, as the methods for undirected graphs require a symmetric adjacency matrix which directed graphs do not deliver. [3] Van Lierde [22] gives an overview over several approaches to enable spectral clustering on directed graphs. When we decide which method to use with directed graphs, it is important that our implementation for undirected graphs can still be used for the clustering. Otherwise we can not compare the results of the clusterings as we do not know whether the differences are introduced by the additional information of a directed graph or the different algorithm. Chung [26] presents an approach to create a symmetric Laplacian matrix from a directed graph. To create such an symmetric Laplacian matrix he denotes that the directed graph has to be aperiodic. That mean, that the directed graph does not contain cyclic paths whose lengths do have a greatest common divisor greater than 1. As we cannot guarantee that all our used graphs fulfill this requirement, this method is not applicable for our use-case.

Another obvious possibility is the conversion of a directed into an undirected graph. The naive approach would be, if there exists two edges between a pair of vertices, to sum up the edges, and in every case drop the direction information. A better approach would be to use the edges' direction in a different way to prevent a complete loss of the contained information.

Satuluri and Parthasarathy [27] presented a conversion called *bibliometric symmetrization*, where the direction information is taken into account. With this conversion there does not have to be an edge between two vertices although there was an edge in in the original graph. The idea behind this approach is that only edges are present which are between vertices which share the same incoming and outgoing edges, i.e., with the same start, respectively the same end vertex. The resulting undirected adjacency matrix A_u created from the original adjacency matrix A is defined as

$$A_u = AA^T + A^TA \tag{4.13}$$

where AA^T is called the *bibliographic coupling* matrix [28] and A^TA the *co-citation strength* matrix [29]. The bibliographic coupling matrix measures the number of common outgoing edges, whereas the co-citation strength matrix measures the number of common incoming edges. If all edges from A should be present in the symmetrized matrix, matrix A is replaced with $A + I$ before the symmetrization, where I is the identity matrix. For our approach we use both mentioned methods, the simple approach with dropping the direction information and the bibliometric symmetrization approach, for creating an undirected graph from a directed graph to evaluate whether the direction information from the workflow is relevant for the clustering. In case of the bibliometric symmetrization we replace A with $A + I$ to preserve the original structure of the graph and view the new edges as transformed information given by the "directed" property of the original graph. We evaluate both the simple transformation by combining edges and the bibliometric symmetrization in Section 5.3.

4.2.3 Agglomerative Clustering

Schaeffer [2] gives a good introduction to agglomerative clustering algorithms. She describes agglomerative clustering as the counterpart to divisive clustering algorithms which split a graph top-down whereas agglomerative clustering algorithms work the other way round by "merging singleton sets of vertices iteratively into clusters". She refers to this approach as the *pairwise nearest neighbor* method. One requirement for this approach is a so-called similarity function to score the similarity between sets of vertices. In addition to that it was to be decided whether only a singleton set of vertices is merged with another singleton or larger set or both sets to be merged can be larger sets. The first one results in a single flat clustering, whereas the latter one results in a hierarchical clustering.

The problem with a naive implementation of an agglomerative clustering algorithm is that calculating all similarity scores for each pair of vertices results in an overall runtime of $O(|V|^3)$. [30] In literature there exist multiple approaches with improved runtime of agglomerative clustering algorithms, for example by Hopcraft et al. [30],

Newman [13] and Clauset et al. [31]. Nevertheless we decided to rely on our own naive implementation as the number of vertices of our used graphs does not exceed 300 vertices. It is not necessary to use a specialized algorithm, whose implementation might introduce complications. Further details of the implementation are described in Section 4.3.6.

As we decided to use our own naive implementation, we had to specify the properties of the algorithm. First, we had to choose a similarity function. The similarity function cannot score the semantical similarity of vertices because one requirement for our approach is that only the structure of the workflows is taken into account. As we can manipulate the edge weights of the graph by changing the mapping, we decided to use the shortest path between two vertices as similarity measure. We cluster vertices together whose shortest path has a small weight. In the context of similarity it is important to decide whether the resulting clustering is flat or hierarchical. Since workflows can be labeled hierarchically and the other chosen algorithms create hierarchical clusterings, the agglomerative clustering algorithm should behave equally. This leads to the question, how the similarity function is applied on clusters which contain multiple vertices.

One has multiple solutions for this general clustering problem. The most common solutions are single-linkage, complete-linkage and average-linkage. Given an undirected graph $G = (V, E, \omega)$ and two clusters $C_0 \subset V$ and $C_1 \subset V$ where $C_0 \cap C_1 = \emptyset$, we define $\Phi = \{\phi(v_0, v_1) \mid v_0 \in C_0, v_1 \in C_1\}$ where $\phi(v_0, v_1)$ denotes the weight of a shortest path (see Section 2.2) between v_0 and v_1. If single-linkage is used, the similarity of the two clusters is $\min(\Phi)$, whereas it is $\max(\Phi)$ for complete-linkage. The similarity measure for average-linkage is calculated by $(1 \div |\Phi|) \sum_{x \in \Phi} x$. Complete-linkage is not useful for our use-case, as we want to detect highly connected groups of vertices, which have short paths between each other. We decided to use average-linkage rather than single-linkage as we assume that average-linkage reflects the similarity of two groups better. In case of clustering a undirected graph, we can use the similarity measure without further actions. Contrasting that is the case of clustering an directed graph, where the similarity measure is not applicable directly as the measure is not symmetric. The average shortest path between two clusters

is dependent on which cluster is the start point and which the end point of the paths. That means that the similarity of two clusters C_i and C_j differs from the similarity of C_j and C_i. Like in the case of spectral clustering methods, we decided to transform a directed graph into an undirected graph by the two methods described in Section 4.2.2.

As the agglomerative algorithm works the other way round compared to divisive algorithms, it is not possible to reuse the recursive implementation of the edge betweenness clustering algorithm. The agglomerative algorithm works on the whole set of vertices, whereas the edge betweenness algorithm only works with a subset of all vertices. Our iterative agglomerative clustering algorithm consists of the following steps:

1. We put each vertex of the graph into a single cluster.
2. We calculate the similarity based on average-linkage of shortest paths for each combination of clusters.
3. We merge the pair of clusters with the smallest similarity score into a new one. If the new cluster satisfies the stopping condition, we ignore the cluster for further calculations and consider it as final. Otherwise we treat the new cluster as one of the others.
4. We repeat all steps since step 2 until all clusters are final or only one cluster remains.

When to stop the algorithm is decided by the same metrics used by the other chosen algorithms: cluster density, global clustering coefficient, average local clustering coefficient and modularity.

4.3 Workflow Clustering Tool

In this section we present our worklfow clustering tool which is responsible for each graph clustering calculation, the configuration and the preserving of results. First,

we describe the tool itself in Sections 4.3.1 to 4.3.3. Afterwards we explain our implementations of each selected clustering algorithm in Sections 4.3.4 to 4.3.6.

4.3.1 Requirements

We implemented all procedures mentioned in this chapter. This includes the mapping described in Section 4.1 and the application of different algorithms shown in Section 4.2. The tool has been developed with the following feature requirements:

- Reads and processes workflow (`.wf`) files.
- Works with different mappings.
- Creates a graph on the basis of a workflow and a mapping.
- Applies different implemented graph clustering algorithms.
- Creates a graphical representation of the graph including the result of a clustering.
- Saves all provided data as well as the resulting data in a reusable way to enable re-running a clustering attempt.

4.3.2 Technology

To minimize development effort, the tool is written in the object-oriented programming language Java. Thus, we use the same programming language as the main software RCE. Using the same programming language allows reusing parts of the software stack of RCE. This is the case for the Jackson Project [32] Java library which is used to parse the workflow `.wf` files which are written in the JSON [33] format. One additional library we use, which is not used in RCE, is JGraphT [34]. This library is used to realize a graph data structure. Besides that JGraphT allows to create `.dot` language output from a graph. This output is processed by the visualization tool Graphviz [35] to render a graphical representation of the graph. For all linear algebra calculations we use the library jblas [36]. We use the build

management tool Apache Maven [37] in order to include the libraries Jackson Project, JGraphT and jblas.

4.3.3 General Implementation

We implemented the tool as a command line tool without any GUI. As the developed software is the main tool of this scientific approach, the requirement of repeatable execution is one of the most important ones. Thus, a single configuration file is the only way to interact with the tool and provides all necessary information for execution. The configuration file is written in the JSON format to avoid using different libraries for file input handling. As the tool has to handle multiple input and output files, the tool operates on a predefined directory structure which the tool creates by itself. It is designed to execute a set of tasks in a predefined order, starting from reading the configuration file to writing the output. In the following we describe the mentioned properties and functions in detail, without considering the code level implementation details as the focus of this work lies on the different algorithms.

File Handling

The tool requires a specific directory structure and will create it, if it is not available. This structure is used to order the used files and to preserve all information of each run. In addition to the configuration file, there are multiple directories for input and output data. The directory `workflows` contains `.wf` files and the `output` directory stores all generated data. As the tool is designed to run once with the specified settings for one clustering, each run creates a subdirectory in the `output` directory. The generated subdirectory of a run is named after the time of execution, i.e., a time stamp, the workflow name and the used clustering algorithm. To fulfill the requirement of reproducibility, all used and generated files are stored in this subdirectory. This contains the main configuration file, the workflow file, the `.dot` language output, and the rendered graphical representation of the graph with the clusters found by the used algorithm.

Configuration

The `config.json` configuration file written in the JSON format is the only possibility to control the execution of the Graph Clustering Tool. Using only one file enhances the user experience as all settings can be seen at a glance. We tried to find a balance between enough options for a detailed configuration and a preferably minimal set of key configurations which exert the greatest influence on the algorithms. The motivation behind this is that the complexity of the evaluation increases with the number of combinations of different options.

The configuration file consists of general settings, workflow settings and mapping settings. Listing 4.1 shows the contents of the configuration file. The general settings contain the path to the Graphviz executable. The workflow settings contain the name of the workflow, the used algorithm including several settings regarding the algorithm and several graph properties. The two major groups of settings are the metric and the graph properties settings. These settings strongly influence the resulting clustering as they control what the algorithms regard as a cluster and how the graph for the clustering is structured based on the directed weighted graph created from the workflow file. The tool can calculate scores of the following metrics: cluster density, global clustering coefficient, average local clustering coefficient and modularity. The graph structure can be configured by manipulating the edge weights: All edge weights can be set to one or the edge weights can be inverted, which means that the reciprocal value of the original edge weight is used. In addition to that the graph can be transformed into an undirected graph. As described in Section 4.2.2 the undirected graph can be created by dropping the direction information and summing up edge weights between two vertices or by bibliometric symmetrization. The mapping settings show which workflow property is mapped to a certain weight of the graph representation.

Listing 4.1: Example configuration file.

```
{
  "general": {
    "graphvizExecutablePath": "dot"
  },
  "workflow": {
  "filename": "workflow_1.wf",
  "algorithm": "edgeBetweenness",
  "score": "modularity",
  "minScore": "0.8",
  "oneWeighted": true,
  "invertedWeighted": false,
  "directed": false,
  "undirectedKind": "simple"
  },
  "mapping": {
    "datatype": {
      "Boolean": 12,
      "DirectoryReference": 15,
      "FileReference": 20,
      "Float": 13,
      "Integer": 12,
      "Matrix": 19,
      "ShortText": 17,
      "SmallTable": 18,
      "Vector": 14
    },
    "constraints": {
      "Required": 0,
      "NotRequired": 0,
      "RequiredIfConnected": 0,
      "None": 0
    }
  }
}
```

Execution

The general procedure of the program consists of multiple steps described as follows.

1. Checking the working directory status and creating missing directories and files.
2. Reading the `config.json` configuration file.
3. Reading the workflow `.wf` file.
4. Creating an object representation of the workflow.
5. Creating a directed weighted graph on basis of the object representation of the workflow and the configuration.
6. Creating a graph based on the directed weighted graph according to the configuration.
7. Setting up and running the graph clustering algorithm.
8. Creating a new subdirectory for the output.
9. Creating the `.dot` language output and writing it to file.
10. Calling the Graphviz executable to process the `.dot` language output file.
11. Copying all processed files to the new subdirectory.

The tool is designed to be executed for one combination of parameters at once. During each execution the number of clusters found, the minimum size and the maximum size of the clusters, and the runtime of only the clustering process itself are written to the command line output.

4.3.4 Edge Betweenness Implementation

Our implementation of edge betweenness closely follows the theoretical description of the algorithm in Section 4.2.1. There are two major tasks of the implementation:

- Calculating the edge betweenness score.
- Repeating the cutting of edges until a predefined quality of the clusters is reached.

Both steps are covered in the described order.

The naive first approach was to calculate the edge betweenness score by calculating all acyclic paths in the directed graph between a pair of vertices and a shortest path between those vertices. As the weight of the shortest path is known, all shortest paths can be found by filtering all found paths with this weight. Both the methods of finding all paths and a shortest path are given by JGraphT. Afterwards the score for each edge of the shortest paths between the two vertices is calculated as all relevant edges and the number of shortest paths are given. This calculation is repeated for each pair of vertices in the graph to calculate the edge betweenness score for each edge. Although the calculation itself was possible and correct, the approach itself has some disadvantages:

- It works only with directed graphs, thus limiting the possibilities for the evaluation. There is no given method to calculate all paths between a pair of vertices in an undirected graph.
- The runtime makes the approach infeasible for realistic graphs.

We further illustrate the latter point in the following. As we have to calculate the edge betweenness scores with each pair of vertices in the graph, the calculation time increases quadratically with the number of vertices. Testing this specific implementation of the algorithm on a graph with ten vertices results in a runtime of under ten seconds. Using the same implementation on a larger graph with 162 vertices results in a runtime of about 45 minutes. The execution took place on a system with an Intel i7-6600U processor with up to 2.6 gigahertz per core and 16 GB RAM without parallelization. At this point it is important to recall that the scores

have to be recalculated after each cutting of an edge. Keeping this in mind leads to the conclusion that the algorithm can not be used in this form as this calculation is only an auxiliary one and is repeated several times. The runtime for a complete clustering would be too long to experiment with the algorithm.

An idea to improve the runtime was to use parallelization. We experimented with multiple, in our case four was stable on the testing machine, worker threads. Each thread calculates all scores for all combinations for one start vertex with different end vertices. This approach led to a runtime of about 20 minutes on the same 162-vertices-graph. Even with this major improvement the runtime is still too long for the clustering.

The next idea was to adapt an algorithm that computes the conceptually similar measure of betweenness centrality and that is known for its efficiency. The version of the betweenness centrality score implemented in JGraphT is based on Brandes' [38] improved version of the original algorithm. Even for the large 162-vertices-graph the JGraphT implementation of Brandes' algorithm calculates the betweenness centrality score in seconds. As edge betweenness and betweenness centrality are conceptually closely related due to their usage of number of shortest paths, the idea was to adapt Brandes' algorithm to our use-case. Brandes' algorithm consists of two parts:

- Finding all shortest paths for each pair of vertices based on Dijkstra's shortest path algorithm (see Section 2.3).
- Calculating the scores for each vertex.

To use the algorithm for our use the case we replace the latter point with the calculation of the score for each edge on the found shortest paths. Listing 4.2 shows the calculation of the edge betweenness score formulated as pseudocode. Brandes' algorithm is implemented to be executed for each single vertex of the graph as start vertex. By iterating through all vertices of the graph, all shortest paths are considered. After the shortest paths are found, the following information can be used for further calculations:

- The number of shortest path from the start vertex to each vertex.

Listing 4.2: Edge betweenness score calculation.

```
1  first part of Brandes' algorithm [38, p. 10]
2  ...
3  P //predecessor list
4  σ //number of shortest paths
5  edgeScores[(v, v')] ← 0, (v, v') ∈ E
6  foreach w ∈ V
7    bfsQueue ← empty queue;
8    enqueue w → bfsQueue;
9    do
10     tmp ← pop bfsQueue;
11     localPredecessorList ← P[tmp];
12     enqueue localPredecessorList → bfsQueue;
13     foreach p ∈ localPredecessorList
14       edgeScores[(p, tmp)] ← edgeScores[(p, tmp)] + (1 / σ[w]);
15     end
16   while bfsQueue not empty
17 end
```

- The predecessor of each vertex on a shortest path. A vertex only has multiple predecessors if the vertex is part of multiple shortest paths.

The list of predecessors of each vertex can be used as an adjacency list. A graph represented by such an adjacency list made of a predecessor list contains all shortest path of the respective start vertex. In combination with a breadth first search with every vertex in the graph as end vertex of a shortest path, all edges which are part of shortest paths outgoing from the start vertex are found. As the number of shortest paths to each vertex from the start vertex is given, the score for each edge can be calculated. Due to the iteration through several start vertices each edge score can be updated multiple times whereas the new score value is added to the previous one.

To complete the algorithm, we have to implement the clustering itself. The main method of our final implementation is shown in Listing 4.3 formulated as pseudocode. As the theoretical algorithm splits a graph into induced subgraphs and repeats this process for each subgraph, we decided to implement the algorithm in a recursive manner. In addition to that the described procedure by Newman and Girvan has to be adapted to real usage because of the structure of the used workflows and the fact that the algorithm has to stop when a cluster is found. The points introducing a

Listing 4.3: Simplified main method of the edge betweenness clustering.

```
1  clusterMap[]
2
3  int recursiveEdgeBetweennessClustering(graph, clusterCount, minClusterScore)
4    clusterScore ← getClusteringScore(graph);
5    if clusterScore >= minClusterScore then
6      clusterMap[++clusterCount] ← graph;
7      return clusterCount;
8    end
9    edgeToCut ← getEdgeToCut(graph);
10   if edgeToCut == null then
11     clusterMap[++clusterCount] ← graph;
12     return clusterCount;
13   end
14   removeEdge(edgeToCut) → graph;
15   if graph is connected then
16     return recursiveEdgeBetweennessClustering(graph, clusterCount, minClusterScore);
17   else
18     if clusterScore < getAverageChildClusterScore() && getAverageChildClusterScore()
           < minClusterScore then
19       foreach getSubGraphs(graph)
20         clusterCount ← recursiveEdgeBetweennessClustering(subGraph, clusterCount,
               minClusterScore);
21       end
22     else
23       clusterMap[++clusterCount] ← graph;
24     end
25     return clusterCount;
26   end
27 end
```

difference to the theoretical algorithm are:

- In some cases multiple edges have to be removed to split the graph.
- Determinism in the context of the chosen edge to cut in case of multiple edges with the same score.
- Checking the quality of a subgraph and deciding whether a good cluster is found or not.

The most striking point related to the structure of the workflow is that cutting an edge does not lead to a new cluster in most cases. Thus, the algorithm has to check whether the resulting graph is still connected. In the case of a still connected graph another edge has to be cut until the resulting graph is not connected anymore.

As described by Newman and Girvan [8], if multiple edges have the same score, an arbitrary edge is chosen. The first naive implementation contained a Java built-in sorting of the edges to find the edge with the highest score. There were no further constraints regarding the case of multiple edges with the same score. That led to different behavior and results when debugging rather than executing the program as the Java object IDs differ between the modes. An Java object ID is an unique ID for an object created during execution. Without going further into details at this point, that problem shows that there has to be a deterministic choice of the edge to cut to fulfill one of the main requirements of the clustering tool: reproducibility. It is not important which sorting is used rather than that the sorting delivers the same result in each execution to enable a meaningful evaluation.

For our use-case we decided to sort the edges by the following edge properties in the mentioned order:

1. Edge betweenness score (descending)
2. Workflow component ID of the start vertex (lexicographically)
3. Workflow component ID of the end vertex (lexicographically)

Note that the workflow component IDs are unique and JGraphT can distinguish between start and end vertex even in undirected graphs which makes this specific sorting possible.

The last major difference to the algorithm is the stopping condition for the algorithm. In our use-case this is when a cluster is found. As our implementation is recursive, the stopping condition is not only important for the logical stopping of the algorithm. It is also the stopping condition for the recursive descent.

Given a graph G, our algorithms cuts the chosen edge, producing subgraphs G_1 and G_2. If we have $C(G) < (C(G_1) + C(G_2))/2$ and $(C(G_1) + C(G_2))/2 < C_{\min}$, where the function C is the applied metric and $C_{\min}$ the configured minimum score of the metric, the algorithm is applied on both the subgraphs G_1 and G_2. Otherwise the algorithm stops and saves G as the result of this clustering. In this case the algorithm ascends. At first sight, the second condition $(C(G_1) + C(G_2))/2 < C_{\min}$ seems to be incorrect, as it leads to saving the parent cluster instead of executing the algorithm on the children clusters although the children exceed the minimum score on average and should be saved. First experiments with this condition showed that it prevents too many and very small clusters. This led us to the decision to use this condition for the final algorithm. Nevertheless, we evaluate different values for the goal score.

There are edge cases which require special treatment, especially since graphs based on workflows differ from the graphs assumed by Newman and Girvan. Our described implementation is based on the assumption that the initial graph is connected and not a desirable cluster, so that at least one cut is made and afterwards the metric is applied to evaluate the subgraphs. Using this implementation on a graph which is not connected initially and consists of one large and several small and highly connected subgraphs, the metric evaluation result of the subgraphs after one cut (which is most probably made in the large subgraph) will show that the clustering is quite good due to the same weighting in the average calculation of the highly connected small subgraphs and the one large subgraph. In this case the algorithm would stop with a result which is not usable as the algorithm stopped in the first iteration although there may be clusters hidden in the large subgraph. This problem can be solved by

analyzing the connectivity of the graph before the clustering algorithm is applied. Then we apply the clustering algorithm on each subgraph of the graph, so that each subgraph is treated as an initial graph. Combining this with a check whether the graph already fulfills the desired cluster property prevents that an edge is removed without reason.

As described in Section 4.2.1, we experimented with multiple clustering metrics. The clustering coefficient based metrics are implemented in JGraphT, whereas the cluster density metric had to be implemented by us. We will not describe further details of the implementation, as the implementation is straightforward from the mathematical description in Section 2.3. The same applies to the calculation of the modularity metric described in Section 4.2.1. As the global clustering coefficient is only defined for undirected graphs, we use the simple transformation of a directed graph to apply this metric. Although the average local clustering coefficient is defined for undirected and directed graphs, we have to use an undirected graph representation for the calculation. This is due to an issue in JGraphT which denies executing the score calculation method with undirected graphs.

The application of the modularity metric is not as straightforward compared to the other metrics. The main difference is the scope of the metric: Modularity evaluates the whole clustering, whereas we use the other metrics to evaluate single clusters. Using our recursive implementation for the clustering leads to the problem that we cannot access the other clusters during the recursive descent as the scope of one recursive step is only one subgraph. An idea to solve this problem was to let the algorithm split the graph until only clusters with one single vertex exist. We save the results in form of a tree, where the original graph is the root node and the single-vertex-clusters are the leaf nodes. This tree consists of several levels, each level representing an additional split of the elements of the parent nodes, i.e., the parent graph.

One problem with this approach is that not all levels of the tree contain all vertices of the original graph, as resulting subgraphs do not have to be equal sized so that not all subgraphs are fully split on the same level. This problem can be solved by filling the levels with already fully split clusters so that each level contains every vertex of

the original graph. Afterwards we apply the modularity metric on each level starting with the root node level until the score falls below the configured goal score. This procedure is possible due to the decreasing of the modularity score when the number of clusters increase while the number of vertices inside the clusters decrease.

Applying the modularity metric on each level leads to high jumps between the scores, as the number of clusters doubles on each level until the first clusters are completely split. It would be possible to use combinations of large clusters of a level closer to the root node and smaller clusters of the following level so that all vertices of the original graph are contained to prevent these jumps of the score. The problem is that there exist a lot of different combinations. In addition to that this approach can distort the results as the original idea of the edge betweenness clustering is not considered: Cutting edges one by one in a fixed order given by the edge betweenness score. In the tree, the order information is completely lost. As we see, the approach of sticking with the recursive implementation leads to more and more difficulties and calculation effort.

This led us to the decision to use a completely new iterative implementation which works on a global scope, i.e., all clusters are accessible on each iteration step. This enables us to apply the modularity metric on each split and to keep the order of the edge betweenness scoring. An advantageous side effect is that we do not have to split the graph fully until only single-vertex-clusters exist. Listing 4.4 shows the iterative implementation of the edge betweenness algorithm with modularity scoring. The implementation is straightforward.

1. We removes edges based on the edge betweenness score iteratively until the number of connected subgraphs increases.
2. Afterwards we calculate the modularity score of the clustering.
3. If the modularity score is still to high and the original graph is not fully split, we repeat all steps since step 1.

We note that like our recursive implementation of the edge betweenness algorithm we execute the algorithm on each connected subgraph of the original graph. The subgraphs are then treated like the “original” graph. Additionally, the modularity

Listing 4.4: Simplified iterative edge betweenness algorithm for the modularity metric.

```
1  getClustersByEdgeBetweennessModularity(graph, graphToCut, minScore)
2    while getConnectedSetsNumber(graphToCut) < getVerticesNumber(graph)
3      connectedSets ← 1;
4      isConnected ← true;
5      while isConnected
6        removeEdge(getEdgeToCut(graphToCut));
7        isConnected ← connectedSets == getConnectedSetsNumber(graphToCut);
8      end
9      score ← getModularityScore(graphToCut, getConnectedGraphs(graphToCut));
10     if score <= minScore then
11       break;
12     end
13   end
14   return getConnectedGraphs(graphToCut);
15 end
```

score is always calculated on a copy of the original graph as we need all original edges for the calculation of the edge betweenness score.

4.3.5 Spectral Methods Implementation

As described in Section 4.2.2, using spectral bisection enables us to reuse the recursive main algorithm of the edge betweenness clustering described in Section 4.3.4. Nevertheless, the recursive algorithm has to be adapted and the spectral methods have to be implemented.

At first we describe the implementation of the spectral methods. For all vector and matrix operations we use the jblas library (see Section 4.3.2). After the graph has been created according to the settings of the configuration file, the Laplacian matrix has to be created. At this point we have to mention that in every case the graph is created as an undirected graph independent of the settings in the configuration file, as the following procedure requires an undirected graph, i.e., a symmetric adjacency matrix. To create the Laplacian matrix, the adjacency matrix A and the diagonal matrix D of the given graph are required. The implementation is straightforward as the adjacency matrix has the dimensions $|V| \times |V|$ and only contains entries which are the edge weights given by a graph. Given the adjacency matrix, creating

the diagonal matrix is also straight forward and is realized by the mathematical description in Section 4.2.2. Afterwards, we create the Laplacian matrix given by $L = D - A$.

The main part of the spectral method is the bipartition of the graph. Listing 4.5 shows the spectral bisection as simplified pseudocode. At first the eigenvalues with

Listing 4.5: Simplified spectral bisection.

```
1  splitGraphBySpectralBisection(graph)
2    clusterOne ← empty set;
3    clusterTwo ← empty set;
4    fiedlerValue ← getFiedlerValueFromGraph(graph);
5    fiedlerVector[] ← getFiedlerVectorFromGraph(graph);
6      foreach v ∈ V
7        if fiedlerVector[v] >= 0 then
8          put v → clusterOne;
9        else
10         put v → clusterTwo;
11       end
12     end
13   return clusterOne, clusterTwo;
14 end
```

the corresponding eigenvectors are calculated. The next step is to split the graph. As there exists one component of the Fiedler eigenvector (the eigenvector associated to the Fiedler eigenvalue) for each vertex of the graph, we use the values of the component of each vertex to determine to which of the two resulting clusters a vertex belongs. We are using the sign cut, so only the sign of the component value determines the cluster. This last step concludes the bisection.

We already mentioned that the recursive algorithm used for the edge betweenness clustering can be reused. Before the recursive algorithm starts we also check whether the graph is connected and if not we execute the recursive algorithm on each connected subgraph of the graph. What is reused of the recursive algorithm itself is the validation of the subgraphs after the clustering, in this case the spectral bisection, has finished. Before the spectral bisection is called, two checks are made. The first check is used to validate if the graph already fulfills the desired quality and stops the algorithm if it does. The second check is to check if the graph is connected because

it is possible that the spectral bisection delivers clusters which themselves are not connected. If the check detects that the graph is not connected, it calls the recursive algorithm on each connected subgraph so that the next execution of the spectral bisection starts with a connected graph.

Reusing our recursive implementation of the edge betweenness clustering algorithm means that the metrics used to stop the algorithm behave equally. This means that applying the modularity metric requires a new iterative algorithm for the reasons mentioned in Section 4.3.4. The algorithm is inspired by the iterative edge betweenness algorithm. The main difference is that the iterative spectral clustering algorithm iterates through each cluster and splits it by spectral bisection with the modularity score calculation afterwards, whereas the iterative edge betweenness algorithm uses the edge betweenness score to remove edges which results in more and more connected subgraphs with which then the modularity score is calculated. Thus, we forgo a detailed explanation of the implementation.

4.3.6 Agglomerative Clustering Implementation

As described in Section 4.2.3, we have to implement the agglomerative clustering algorithm without reusing the main part of the edge betweenness algorithm like the implementation of the spectral clustering algorithm does. The main difference between this implementation and the edge betweenness clustering implementation is that the agglomerative clustering algorithm works on the whole set of clusters to enable joining them, whereas the edge betweenness algorithm works on a subset of clusters as the subsets are to be split. This leads to an iterative implementation which is described in the following. Listing 4.6 shows the simplified pseudocode of our iterative agglomerative clustering algorithm.

For the sake of readability we list the steps of the algorithm already described in Section 4.2.3 again.

1. We put each vertex of the graph into a single cluster.

Listing 4.6: Simplified iterative agglomerative clustering algorithm.

```
1  getClustersByAgglomerative(graph, scoreType, minScore)
2    localClusteringMap[] ← empty map;
3    localFinishedClusteringMap[] ← empty map;
4    idCounter ← 0;
5    foreach v ∈ V:
6      clusterList ← empy list;
7      put v → clusterList;
8      localClusteringMap[++idCounter] ← clusterList;
9    end
10   foundClusters ← 0;
11   while size(localClusteringMap) > 1
12     minShortestPathWeight ← 0.0;
13     clusterPairs ← empy list;
14     clusterIdList ← getAllClusterIds(localClusteringMap);
15     for (i; i < size(cluserIdList); i++)
16       for (j = i + 1; j < size(cluserIdList); j++)
17         clusterOne ← localClusteringMap[clusterIdList[i]];
18         clusterTwo ← localClusteringMap[clusterIdList[j]];
19         result ← getAverageShortestPathWeight(clusterOne,clusterTwo);
20         if (result < minShortestPathWeight || minShortestPathWeight == 0) then
21           minShortestPathWeight ← result;
22           clusterPairs ← empy list;
23           put clusterOne, clusterTwo → clusterPairs;
24         else
25           put clusterOne, clusterTwo → clusterPairs;
26         end
27       end
28     end
29     firstCluster ← getFirstClusterFromSorting(clusterPairs);
30     secondCluster ← getSecondClusterFromSorting(clusterPairs);
31     mergedCluster ← mergeClusters(firstCluster, secondCluster);
32     finishCluster;
33     if (scoreType == MODULARITY) then
34         allClusters ← getAllClustersWithMergedCluster(mergedCluster)
35         finishCluster ← getModularityScore(allClusters) >= minScore;
36     else
37         finishCluster ← getClusteringScore(mergedCluster) <= minScore;
38     end
39     if (finishCluster) then
40       localFinishedClusteringMap[++foundClusters, mergedCluster];
41       remove firstCluster → localClusteringMap;
42     else
43       replace mergedCluster → localClusteringMap[firstCluster];
44     end
45     remove secondCluster → localClusteringMap;
46   end
47   if (size(localClusteringMap) == 1) then
48     putAll localClusteringMap → localFinishedClusteringMap;
49   end
50   return localFinishedClusteringMap;
51 end
```

2. We calculate the similarity based on average-linkage of shortest paths for each combination of clusters.
3. We merge the pair of clusters with the smallest similarity score into a new one. If the new cluster satisfies the stopping condition, we ignore the cluster for further calculations and consider it as final. Otherwise we treat the new cluster as one of the others.
4. We repeat all steps since step 2 until all clusters are final or only one cluster remains.

The algorithm is only executed on undirected graphs due to the similarity function described in Section 4.2.3. Like the implementations of the edge betweenness clustering algorithm and the spectral clustering algorithm, the main part of the algorithm is only executed on connected subgraphs of the original graph. As described in step 4, the whole algorithm runs in a loop which stops when there is no or only one cluster left to cluster.

The implementation of the first step is trivial. The set of vertices is ordered after the component ID and afterwards each vertex is put in a cluster with a unique ID.

The next step is to apply the similarity measure on all combinations of clusters. The calculation is executed as described in Section 4.2.3. At this point it is important to note the possibility of multiple cluster combinations with the same score. This occurs frequently in graphs where all edges are equally weighted. As reproducibility is an important requirement of our approach, there has to be a deterministic behavior which clusters are to be merged in that case. We decided to order the candidate pairs for a merge after the following criteria:

1. The average cluster size of each pair (ascending)
2. The smallest component ID of the first cluster
3. The smallest component ID of the second cluster

We can distinguish between the first and the second cluster of a pair of clusters as each cluster has an unique ID and the similarity score is calculated for each pair once by iterating through an cluster list ordered by the cluster ID. The first criterion

differs from the other two criteria as it influences the clustering in a semantical way whereas the other two criteria are only used to force a deterministic ordering. Ordering ascending after the average cluster size of a pair of clusters is used to prevent that some clusters enlarge through several iterations only because of a lucky combination of small component IDs whereas other clusters stay small because their components have larger component IDs. At this point we note that this semantical ordering strongly influences the results of the clustering and may even lead to bad results as we force an equal growth for all candidates with an equal similarity score. As our approach is a feasibility study we accept this possible problem in return for a deterministic behavior which is important for our evaluation.

The last step inside the loop is the merging process. After the clusters to be merged are determined, we put the vertices of the second cluster into the first one. Afterwards the metric of the stopping condition is applied. If the new cluster fulfills the configured score, the cluster is put aside and considered as one final cluster. We have to mention that the comparison of the score, except in case of modularity, is reverse to the comparison within the divisive algorithms. This is due to the different starting points. As an example, we take a closer look at the cluster density metric. The divisive algorithms start with the full graph, so the density is low in comparison to the final clusters. During the recursive descent, the density increases, so the algorithm stops when the goal score is exceeded. The agglomerative algorithm starts with one vertex per cluster. As the density of a group of vertices is defined as the number of edges within the group divided by the maximum number of possible edges, the density for one single vertex is not defined as it would be a division by zero. So it is left to us to define that value. Regarding the next evolutionary step of a cluster, two connected vertices will have have a high score of one. Within realistic graphs, the density will decrease if further nodes are added. This leads us to the decision to define the density of one vertex to be one, which enables a monotone decrease of the density. The algorithm stops when the calculated score falls below the goal score. The global and the average local clustering coefficient have the same decreasing behavior of the cluster density metric. In contrast to the edge betweenness and the spectral clustering algorithm (see Section 4.2.1 for further details about modularity and its application), we do not require a special implementation to apply the modularity

metric as our implementation is already working on the global scope of a given graph. The only difference is that the modularity score increases during the agglomerative merging process and requires an inverted comparison of the modularity score and the goal score.

If one cluster is left over at the end of the algorithm, it is considered a final cluster although it does not fall below the goal score. We remark that this implementation does not guarantee to always result in connected clusters, as not all clusters are final at the same time. Due to ignoring final clusters for further calculations, it is possible that clusters are merged where a final cluster lies between them.

5 Evaluation

In this chapter we evaluate our approach. First, we show the used workflows in Section 5.1. Afterwards we recall the different parameters we use for our parameter study in Section 5.2. Finally, we present our results in Section 5.3.

5.1 Workflows

In this section we present the workflows used for the evaluation of our approach. We have three different workflows which are different in terms of the size, the general structure and the use-case. We give a short overview over each one as follows.

First workflow This workflow is the smallest of the three workflows and consists of 10 components with 78 connections leading to a directed graph with 10 vertices and 20 edges shown in Figure 5.1. It is part of the example workflows which are delivered with each RCE installation. The workflow represents the optimization of two disciplines with given constraints. Thus, it is divided into three parts: the center which controls the optimization process and one side for each discipline where the calculation takes place. Both sides are already labeled and we add one label around the center so that each component is included in a group. This results in three clusters, where the minimum cluster size is two and the maximum cluster size is four.

Second workflow This workflow consists of 151 components with 359 connections leading to a directed graph of 151 vertices and 194 edges shown in Figure 5.2. For our experiments we removed 11 components and with them 10 connections from the original workflow comprising 169 components and 369 connections.

The removed components were not connected to the main part of the workflow which led to fully independent clusters of at most four components in our experiments. As none of these groups has any connection to any other group we removed them. Clustering these does not give any insights and only increases the number of found clusters. This workflow is the only one which is handmade by project partners and fully labeled according to a real use-case. The workflow is obtained from partners from the aerospace field of DLR and used for the simulation of an airplane as a whole. The labeling is not clear in every case, i.e., to which group a component belongs. Thus, we have to estimate the correct membership for those cases. This is the only workflow which contains not connected clusters. For the evaluation we define the number of clusters as 23 where the minimum cluster size is one and the maximum cluster size is 27.

Third workflow This workflow is the largest of the three and consists of 262 components with 702 connections leading to a directed graph of 262 vertices and 311 edges shown in Figure 5.3. This workflow is also created by project partners, albeit not by hand. Instead, it is generated by a plugin made for RCE. The workflow is like the second workflow used in the context of aerospace research, but this one especially in the context of ecological passenger flights. In contrast to the second workflow one can see that the structure is more uniform and consists of recurring group structures. Unfortunately, the workflow is not labeled, so we have to label the workflow ourselves. We follow the graphical representation of the workflow in RCE and create groups of components which are positioned close to each other. This leads to 45 clusters where the minimum clusters size is one and the maximum cluster size is nine.

As the three workflows differ in structure, size, and use-case, we can gain insights about how our algorithms behave on differently structured, workflow-based graphs and whether there are combinations of parameters which can achieve good results in general. Nevertheless, we have to remark that the labels used as reference are changed or completely created by us. Thus, we cannot make general statements but only recommendations for the real usage of the results of our approach.

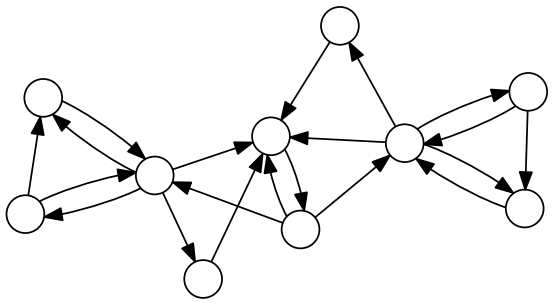


Figure 5.1: Directed graph representation of the first workflow.

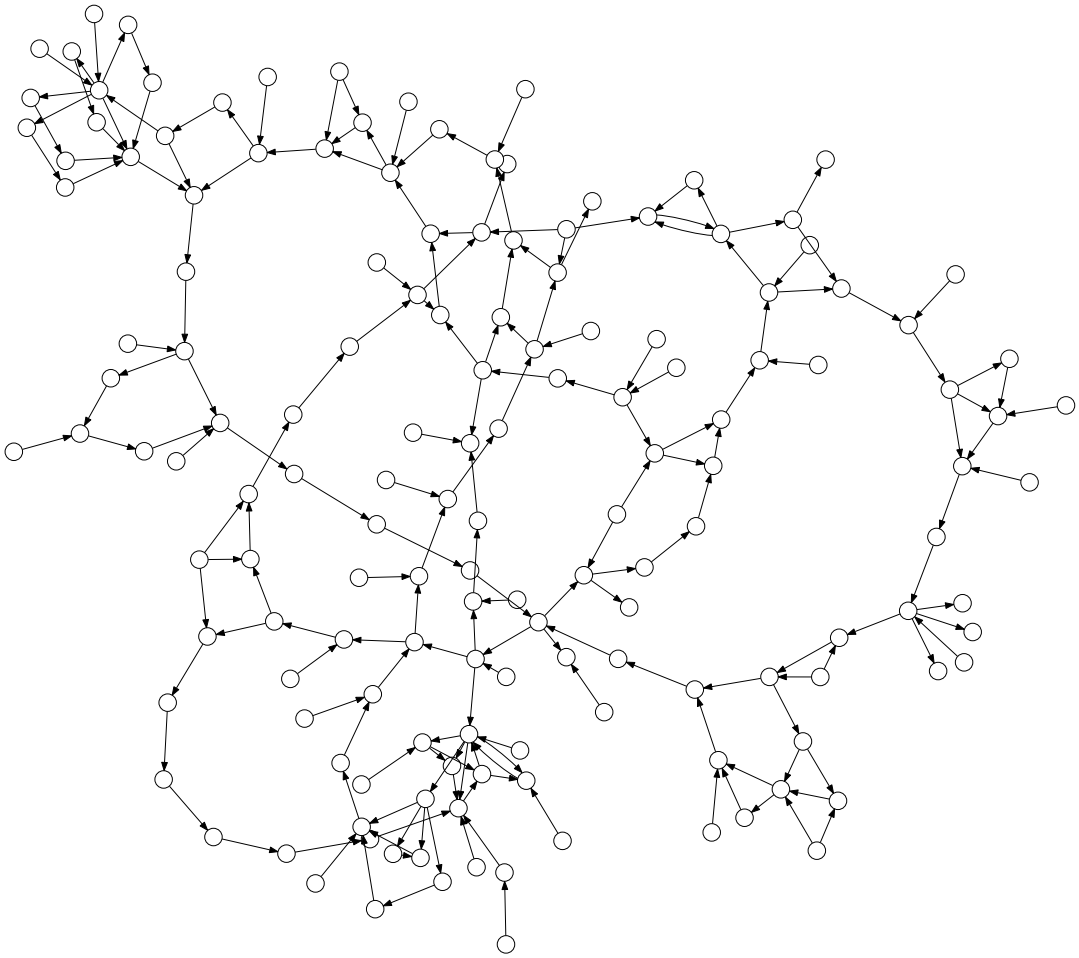


Figure 5.2: Directed graph representation of the second workflow.

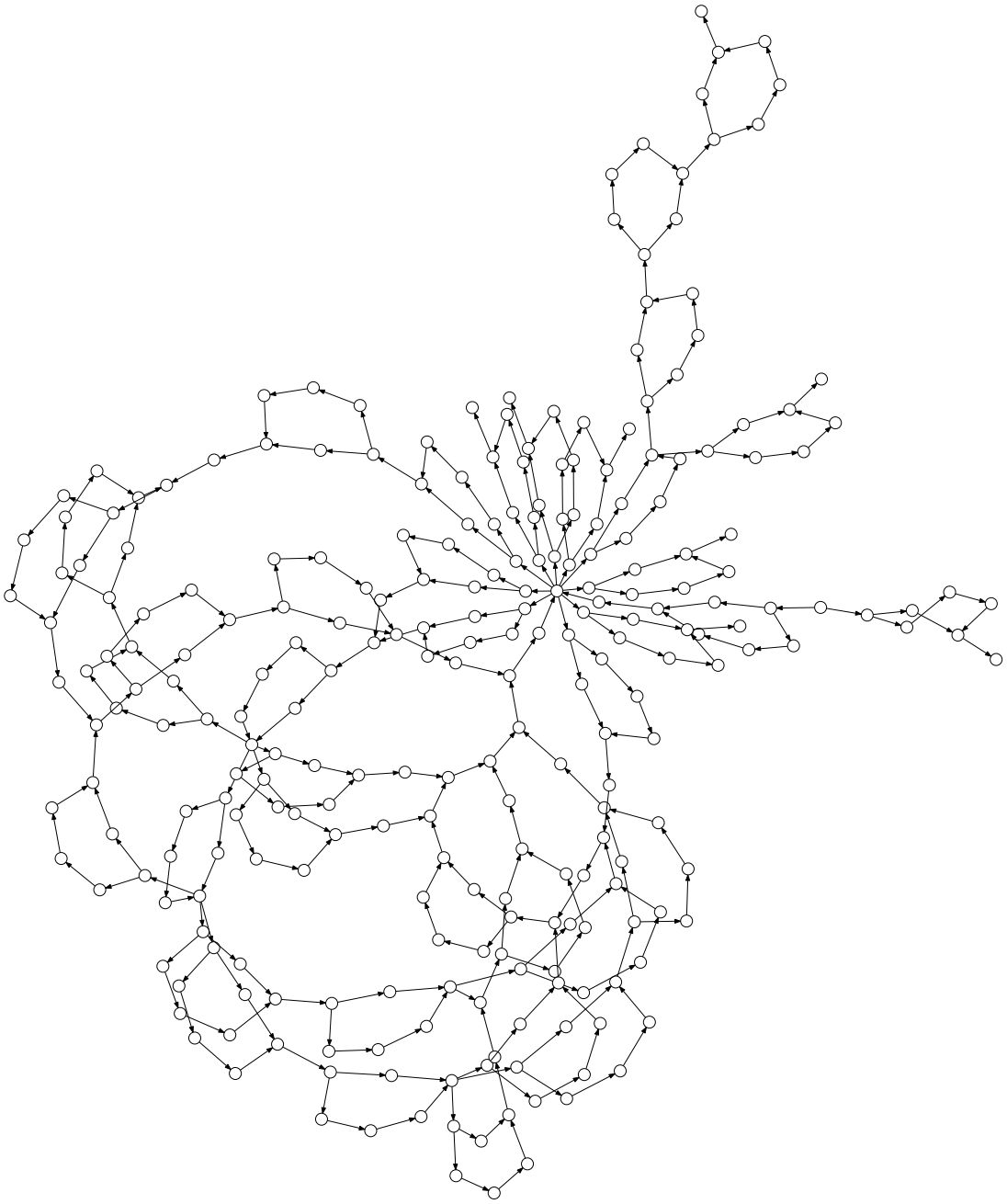


Figure 5.3: Directed graph representation of the third workflow.

5.2 Evaluation Data

In this section we present how we collect the data for the evaluation. As our goal is to examine the feasibility of graph clustering methods applied to workflows, we experiment with different algorithms, parameters, and workflows. First, we present the different combinations of settings.

We experiment with three different algorithms:

- Edge betweenness clustering (Section 4.2.1)
- Spectral clustering (Section 4.2.2)
- Agglomerative clustering (Section 4.2.3)

Each of these algorithms can be executed with four different metrics (see Section 4.2.1) defining when a cluster is found:

- Global clustering coefficient
- Average local clustering coefficient
- Cluster density
- Modularity

We recall that each score of the metrics is in $[0, 1]$ except for the modularity metric whose scores are in $[-1, 1]$. Nevertheless, we treat this metric like the other ones as only scores greater than 0 lead to more edges within a cluster rather than between them. As the workflows have different structures and the metrics work differently, we will experiment with the four different score values 0.2, 0.4, 0.6, and 0.8 as termination condition. For each workflow we create a graph representation. One of the major parameters of this generation is the direction property. We experiment with three different kinds of graphs regarding the direction of edges:

- Directed graphs (edge betweenness only, Section 4.2.1)
- Simple transformed graphs (Section 4.2.2)
- Bibliometric symmetrization transformed graphs (Section 4.2.2)

Another property we can influence is the mapping of the graph (see Section 4.1):

- Ascending correlation of weight and assumed information content of data types and constraints.
- Descending correlation of weight and assumed information content of data types and constraints.
- All edges are weighted with one.

In combination with the three presented workflows in Section 5.1 there are 1 008 different combinations of execution parameters. For each execution we collect the resulting number of found clusters, the minimum size of the clusters, the maximum size of the clusters and the runtime of the clustering itself.

5.3 Results

In this section we present the results of our graph clustering approach with RCE workflows. We remark that we only perform a qualitative analysis. As there is no canonical definition for a group of workflow components belonging together and different users can group components differently, we only can give recommendations about methods producing clusterings which might be useful for RCE users. We use the number of clusters, the minimum and the maximum size of the clusters of each workflow described in Section 5.1 as reference for the following considerations.

First, we take a closer look at the general distribution of the number of found clusters dependent on the algorithm. As the reference of the number of clusters varies between workflows, we have to examine the results for each workflow independently. Figure 5.4 shows the described distribution. One can see that for each workflow the number of found clusters is not focused on a single number, but instead has a wide range. The exception is the number one which is a frequent result among each algorithm for workflow one whereas for workflow two and three the spectral and the edge betweenness algorithm show this result.

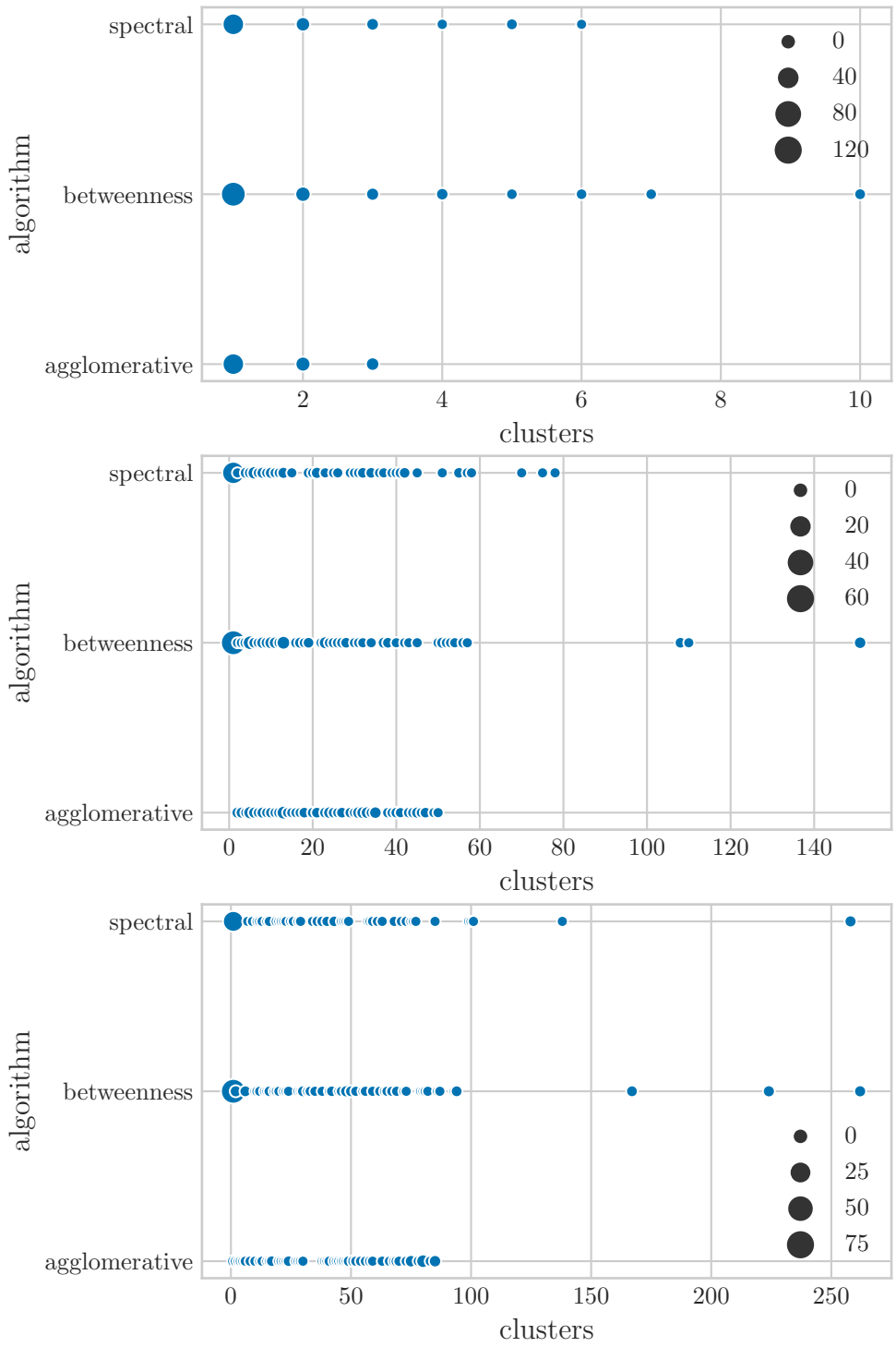


Figure 5.4: Number of found clusters split by algorithm for workflow one to three.

As the desired number of found clusters does not occur more often than other values for any workflow, we decide to focus on those combinations of parameters of results which roughly have the desired properties and roughly match the desired labeling. With this procedure we avoid overlooking good results induced by picking the apparently optimal parameters and only looking at the left over parameters of this preselection. Table 5.1 shows the filter we apply on the result set. The number in

workflow	found clusters	min. size	max. size
1	2 - 4 (3)	2 (2)	4 (4)
2	20 - 26 (23)	1 (1)	35 (27)
3	41 - 49 (45)	1 (1)	15 (9)

Table 5.1: Applied filter on all results for each workflow.

brackets illustrates our defined reference value. The selected filter values are chosen to lead to a low number of results as we compare each clustering to the reference one manually. This step is necessary as we do not know whether similar basic data of clusterings also lead to similar results. Applying the filter leads to 29 results for the first, 23 for the second, and 13 for the third workflow. We compare each of these results against our reference clustering of each workflow.

We decide whether a result is useful by its assumed usefulness for an RCE user. If a user can not rely on the results and has to refactor many clusters there is no advantage in using an automated clustering. After the manual comparison there are 15 results left of the first, 8 of the second and 4 of the third workflow. In the following we refer to these results as the "final result set".

We remark that for the first workflow the combination of the agglomerative algorithm with the simple direction transformation and the weighting where all edges are weighted with one results in clusterings where not all vertices are part of a cluster. There are five results with this error. The error does not occur with the other two workflows. The first workflow is the only workflow where more than half of the results remain after the comparison. Keeping this in mind we conclude that appropriate basic data of an RCE workflow's clustering does not mean that the clustering itself is useful.

Figure 5.5 shows the occurrence of different combinations of algorithms and metrics of the final result set for each workflow. As the maximum occurrence of a combination is four, our evaluation statements are only tendencies and might not be representative. One can see that each algorithm has at least one result for each workflow except for the spectral clustering algorithm which has none for the third workflow. The average local and the global clustering coefficient lead to results only for workflow one and only with the agglomerative clustering algorithm. Nevertheless, both combinations occur four times. Considering all workflows, the agglomerative algorithm mostly leads to results with the cluster density metric. For the edge betweenness and the spectral clustering algorithms this is the case with the modularity metric, whereas in the case of the edge betweenness algorithm modularity is the only metric which leads to results. There are two combinations which lead to results in any workflow. These are the agglomerative algorithm with the cluster density metric and the edge betweenness algorithm with the modularity metric. The former one occurs five times overall and the latter one six times.

Both are candidates for a general recommendation as first approach for new workflows, but there is a major difference in the runtime of the algorithms. Table 5.2 shows the average runtime in milliseconds for each algorithm for workflow one to three. The calculations are made on a system with an Intel i7-5700HQ with up to 3.5

workflow	agglomerative	edge betweenness	spectral
1	315	151	587
2	43 915	2 690	1 408
3	260 441	4 375	-

Table 5.2: Average runtime in milliseconds of each algorithm for workflow one to three of the final result set.

gigahertz per core and 16 GB RAM. The execution took place within a Linux Ubuntu Sever 18.04.5 virtual machine with eight GB RAM available. Each combination of parameters is calculated within one process without parallelization by our own implementations. For workflow one the differences are negligible, as the runtime never exceeds one second. For the larger workflows two and three the differences are enormous, especially for workflow three. While the runtime of the edge betweenness

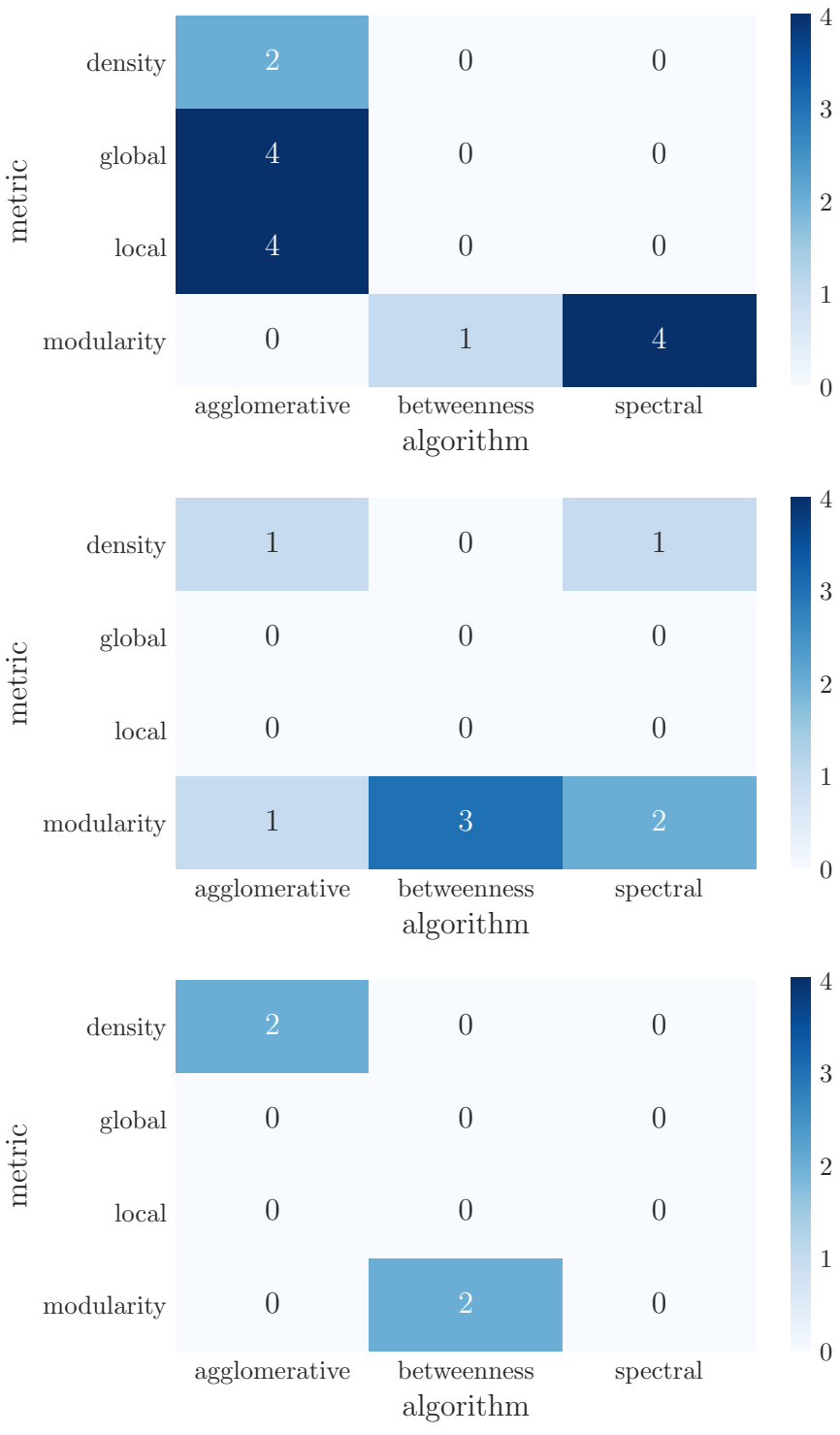


Figure 5.5: Heatmap with occurrence of algorithm and metric combinations for workflow one to three of the final result set.

algorithm is always under five seconds, the agglomerative implementation exceeds 40 seconds for workflow two and four minutes for workflow three. This makes the agglomerative algorithm impractical for usage as an RCE feature. Nevertheless, we remark that a more optimized variant of the agglomerative algorithm could improve the overall runtime. Given our own implementations, we conclude that using the edge betweenness algorithm with the modularity metric is a promising approach for clustering RCE workflows as it has a short runtime and leads to usable results across differently structured workflows. It remains to investigate how the other parameters influence the result.

Metric and Score

Figure 5.6 shows the occurrence of different combinations of metrics and scores of the final result set for each workflow. We focus on the cluster density and the modularity metric as these lead to results in every workflow. For the two large workflows the cluster density occurs only with the value 0.4 whereas for workflow one the score is double the size with 0.8. Across all three workflows modularity has the most results with a score of 0.4 and 0.6 and only two results have a score of 0.2. It appears that modularity is more workflow independent than cluster density as the scores are focused on the range between 0.4 and 0.6 whereas the score for cluster density differs more.

Direction

We experiment with directed and undirected graphs where one option for the undirected graph is a simple transformation from a directed graph and the other option is a transformation by bibliographic symmetrization. In the final result set only one result is achieved with a directed graph and none with the bibliography symmetrization. Although, about 45 % of the filtered results before the comparison with the reference are achieved by bibliographic symmetrization. The comparison of the bibliographic symmetrization results with the references reveals that the results are absolutely not usable as the clusters are not connected in most cases and the distribution

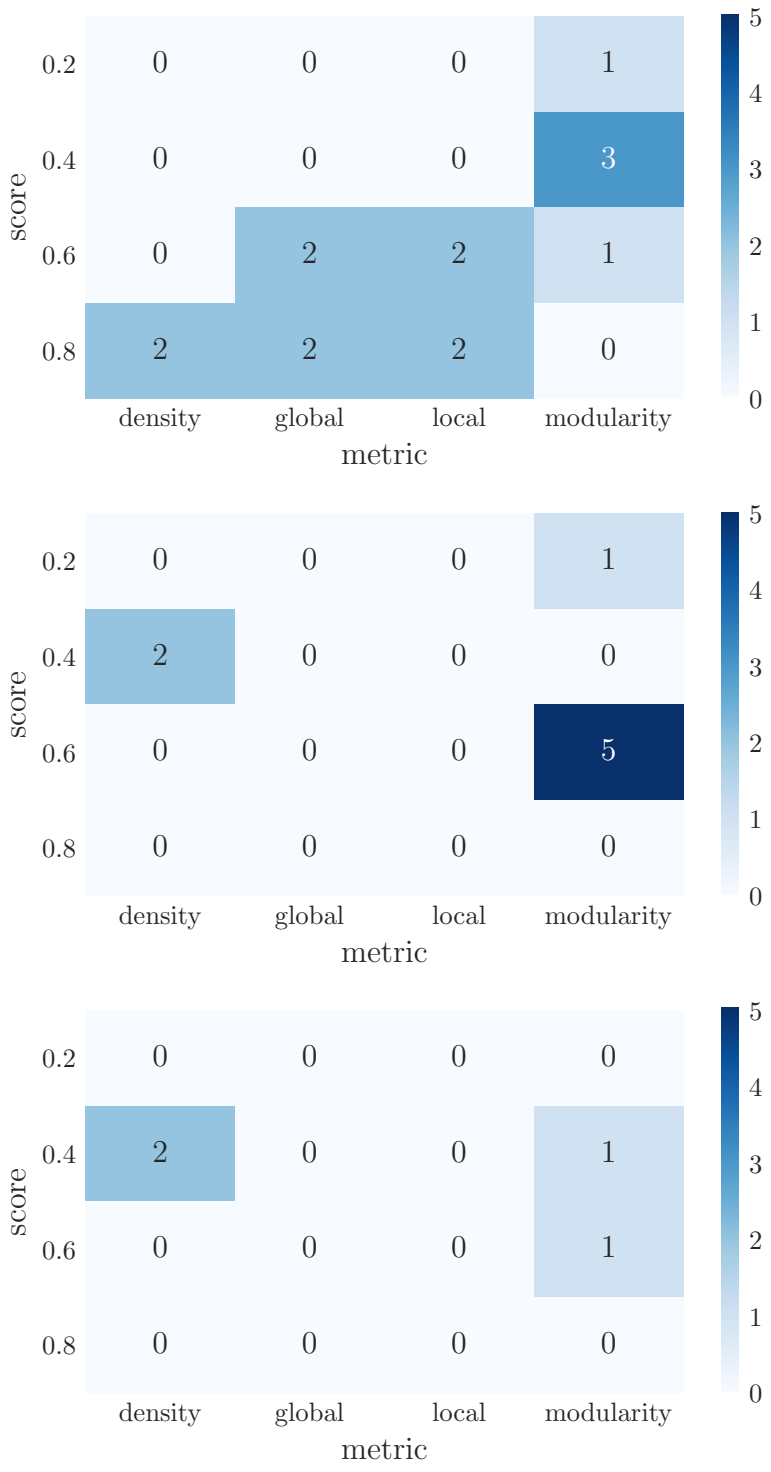


Figure 5.6: Heatmap with occurrence of metric and score combinations for workflow one to three of the final result set.

appears to be random. We have to remark that only the edge betweenness algorithm works with a directed graph so in the end there are less results with this property than results with undirected graphs without any filtering. Nevertheless, as there is only one result with a directed graph we assume that directed graphs are not suitable for our approach. To conclude, the simple transformation from a directed graph representation of an RCE workflow appears to be the most expedient kind of direction information handling.

Mapping

The last point of our evaluation is the behavior of different mappings in combination with different algorithms. Figure 5.7 shows the occurrence of different combinations of algorithms and mappings of the final result set for each workflow. In contrast to the other parts of our evaluation, the distribution of the mapping does not enable us to draw specific conclusions. As described in Section 4.1 we assume that either a lot of information are transported within a cluster or between them. If this was correct, the same algorithm should only work with the ascending or the descending mapping but not with both as the information of the relation strength is inverted. The one-weighted graph is exempted as all relations between vertices are treated equally. The spectral algorithm appears to confirm this theory for workflow one and two, as there are only results for the one-weighted graph and one mapping with weights different from one. Sadly there are no results with the spectral algorithm for workflow three to prove or disprove this. The agglomerative algorithm shows the opposite: The ascending as well as the descending mapping lead to results for workflow one and three. We assumed that the edge betweenness algorithm is not influenced that much by the different mappings. This seems to be true for workflow two, as all kinds of our mappings lead to results. We do not observe this behavior for the other two workflows, where either the ascending or the descending mapping and the one weighted mapping lead to results.

The algorithm and mapping combinations which work for every workflow are the agglomerative algorithm with descending mapping and the edge betweenness algorithm with the one-weighted mapping. We remark that although these combinations work

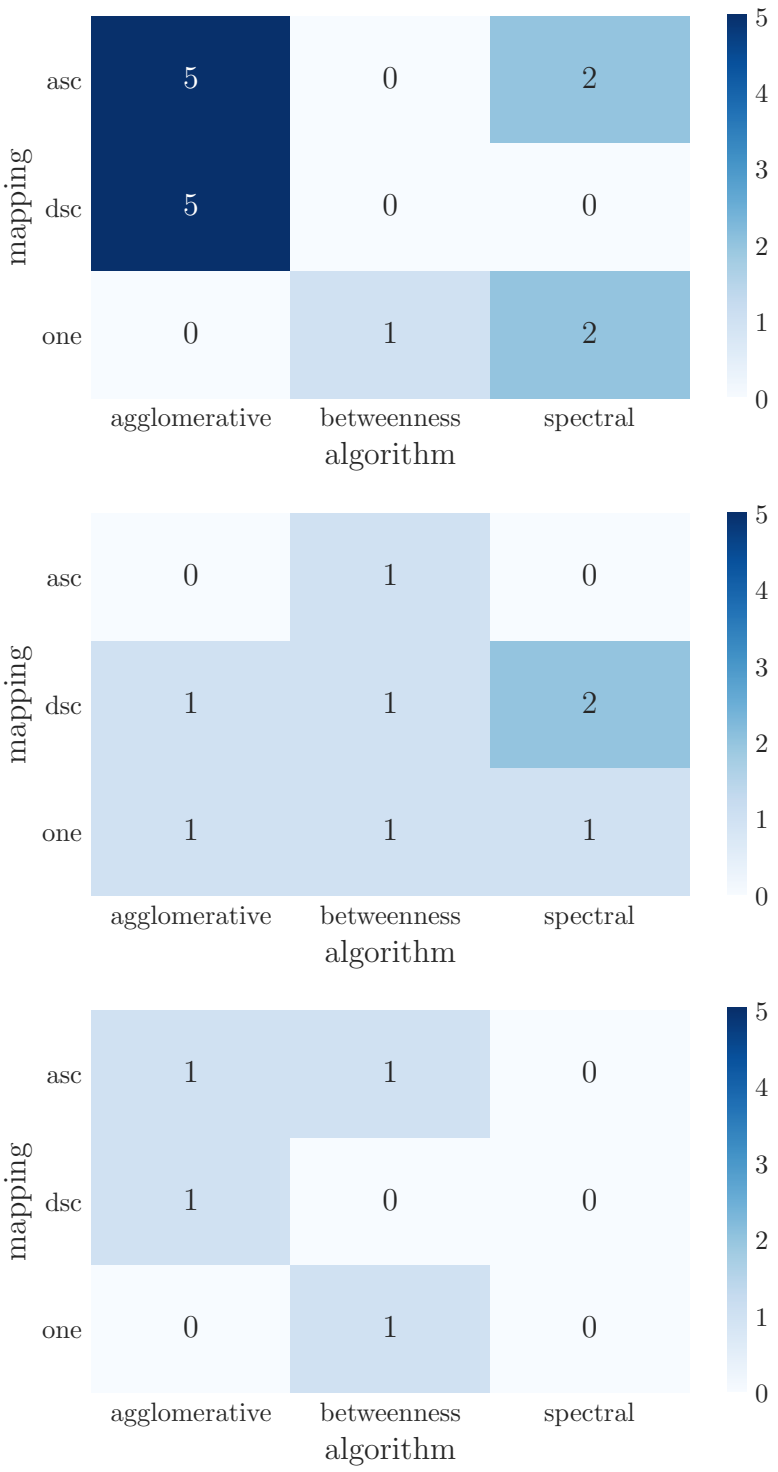


Figure 5.7: Heatmap with occurrence of algorithm and mapping combinations for workflow one to three of the final result set.

for every workflow, they differ regarding the other parameters. To conclude, there are certain combinations which appear to be expedient, but the results are inconclusive so we recommend to experiment with different mappings for new workflows.

Conclusion

Summarizing the intermediate results, for new workflows we recommend the approach of using the edge betweenness algorithm with the modularity metric and an undirected graph created by the simple transformation. The score and the mapping are more likely to vary, but we recommend starting with a score between 0.4 and 0.6 and the one-weighted mapping.

To answer the question whether the application of graph clustering to RCE workflows is possible with good results: yes, it is. There are good results across different combinations of parameters. Nevertheless, we can not give an answer about under which circumstances it is possible in general, or if it is possible in general at all. We remind the reader that our recommendations are based on a small data set of results where we partly created references values ourselves. The next step would be to evaluate our recommendations on a larger set of different workflows labeled by different users to validate or refute our results. Finally, we present one good result for each workflow in Figures 5.8 to 5.10, where vertices sharing a color belong to the same cluster.

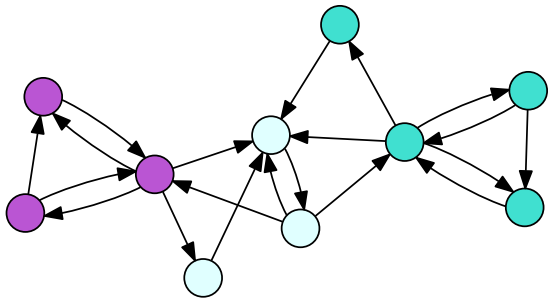


Figure 5.8: Directed graph representation with calculated clusters of the first workflow.

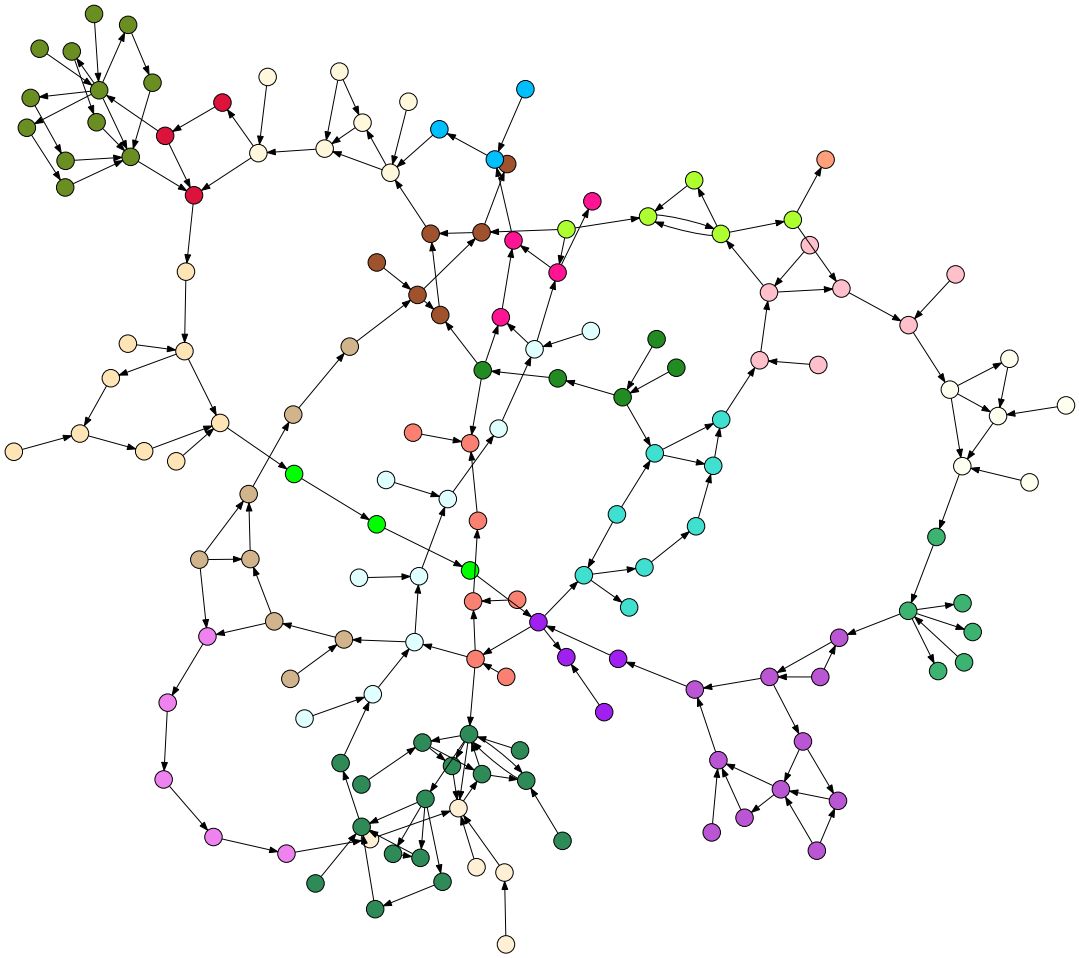


Figure 5.9: Directed graph representation with calculated clusters of the second workflow.

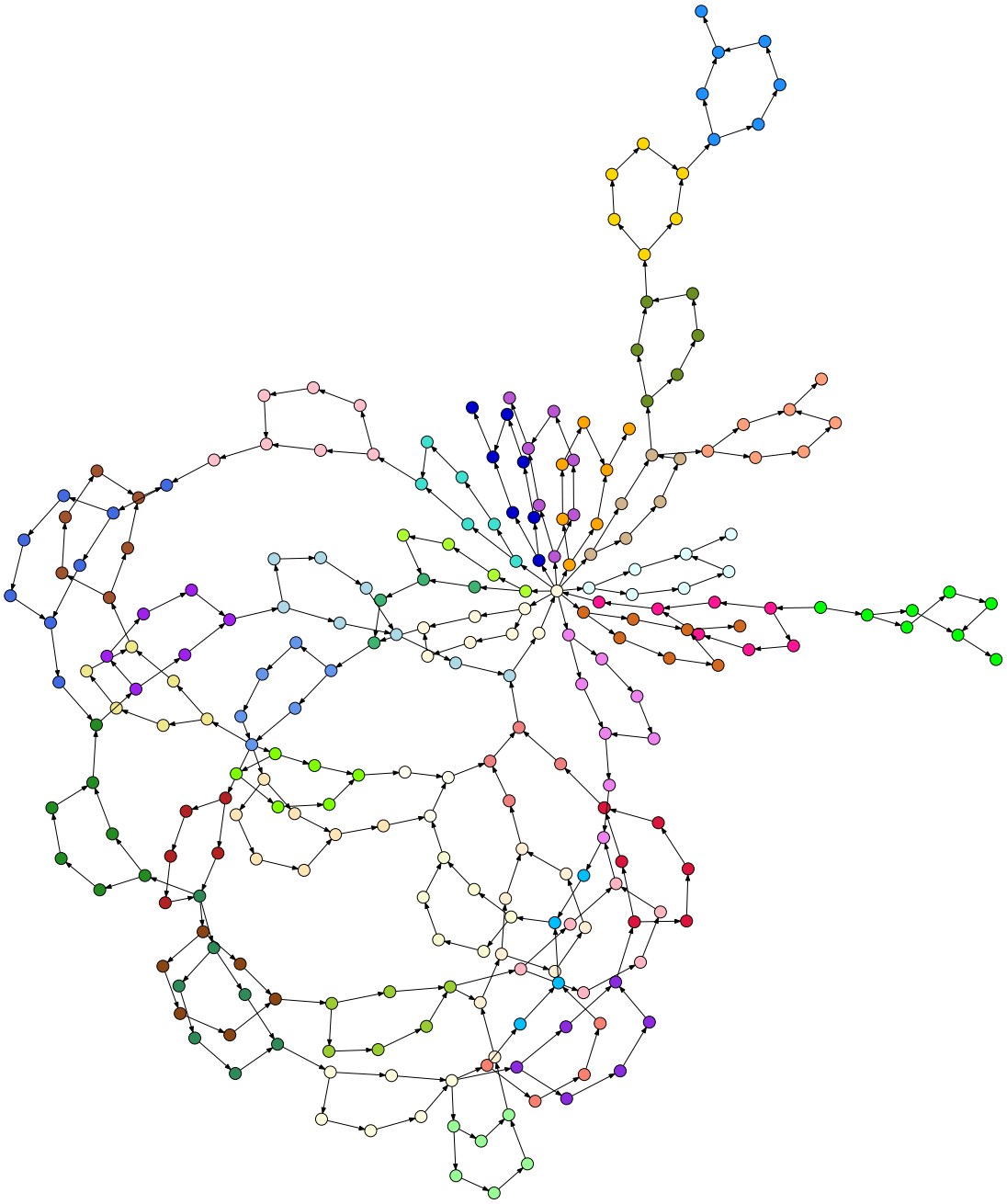


Figure 5.10: Directed graph representation with calculated clusters of the third workflow.

6 Conclusion

In this chapter we summarize our approach and recapitulate our results. We start with the summary in Section 6.1. Afterwards we give an overview over related work in Section 6.2. Based on our results we present our ideas for future work in Section 6.3.

6.1 Summary

Our approach is part of the development of the integration environment RCE [1]. This software allows to create automated workflows orchestrating multiple by third parties integrated simulation tools for multi-disciplinary systems. A workflow is an RCE in-house data structure which resembles a graph due to its main parts consisting of workflow components which are linked by so-called connections. These workflows can be presented visually within the GUI of RCE where users can group tools with colored labels. We evaluated an automation approach for this manual and time-consuming task by applying methods adapted from graph clustering theory.

This procedure requires the creation of graphs representing the workflow. We mapped workflow tools to vertices and connections to edges. For the latter we created multiple mappings from the data types and the requirement constraints to edge weights. The edge weights are leaned on the assumed amount of information content and the importance of a connection. We directly carried over the direction property of each connection, whereas multiple connections between workflow tools are combined and the resulting edge weight matches the sum of the single weights. In addition to the directed graph representation we used undirected graph representations by

either combining multiple directions between two vertices to one undirected edge or by bibliographic symmetrization [27]. For our experiments we used three different weight mappings: ascending, the reciprocal value of the ascending mapping and a one-weighted mapping.

For automation purposes we forwent clustering approaches requiring the number of clusters to be found as input. We used hierarchical algorithms: the divisive edge betweenness clustering algorithm [8], another divisive algorithm based on spectral bisection, and a general implementation of agglomerative clustering. To terminate the algorithms when a cluster is found we experimented with the cluster density, the global clustering coefficient [10], the average local clustering coefficient [11, 12], and the modularity metric [13].

We evaluated our approach on three different workflows: a small RCE example workflow, a large workflow created by hand, and an even larger workflow which was created automatically. For each workflow we ran the mentioned algorithm with a set of different combinations of parameters resulting in 1 008 outputs. After a manual qualitative evaluation of these outputs, we considered 27 of these as usable results. The combination of parameters which appear to be the most expedient across the mentioned workflows is the edge betweenness algorithm with the modularity metric and an undirected graph representation, where multiple directions between two vertices are combined to one undirected edge. The score for the metrics and the mapping tend to vary more and did not enable us to draw general conclusions.

Our approach shows that graph clustering is applicable to RCE workflow graphs with usable results, whereas we could not determine a general approach as we lacked a representative amount of labeled workflows for evaluation.

6.2 Related Work

Our approach belongs to the group of applied graph clustering methods on structures which can be represented as graphs. As graph clustering is a well-researched subject, we do not give an exhaustive overview but instead provide pointers to different

subsections of the subject. Fortunato [3] gives an overview over major fields of applied graph clustering: biological and social networks. Junker and Schreiber [39] present multiple approaches to analyze biological networks such as protein-protein interaction networks. Wasserman and Faust [40] show different methods and applications to analyze social networks. We frequently cite Newman whose graph clustering works also derives from his work with collaborative networks [41]. He influenced the whole scientific field by the publications with Girvan of the modularity metric [13] and the edge betweenness algorithm [8] which we use both for our approach. There are multiple algorithms [15, 16, 42] which are based on modularity as well as there are multiple approaches based on edge betweenness [38, 43, 44].

Yoon et al. [43] use a similar approach like us to apply Brandes' [38] fast betweenness algorithm to calculate the edge betweenness score for the clustering of a metabolic network. Tanaka and Tatebe [45] use graph partitioning to minimize the data flow in distributed scientific workflows resulting in an improved overall runtime of the workflows. Jung et al. [46] use hierarchical clustering to group business processes with the goal to assist process design or reengineering.

In addition to clusters within a workflow, there exist approaches to cluster groups of similar workflows. Santos et al. [47] examine different clustering techniques to organize the increasing number of workflows of different systems and their generated provenance. Jung and Bae [48] use hierarchical clustering to cluster process models represented by so-called weighted complete dependency graphs.

6.3 Future Work

The main problem of our approach was the lack of a representative number of labeled workflows. We partly had to label the workflows ourselves which influences the results, although we tried to prevent this. We could only do a qualitative evaluation which did not even lead to explicit results in every case. One possible next step could be to setup a user study with RCE users and a representative number of workflows, provided that the users know the context of the workflows. By examining the results from different users on the same set of workflows one could observe whether the

labelings are similar or whether they differ widely. Afterwards one could compare clusterings based on our recommendations with the different labelings to evaluate them on a representative scale. In combination with a quantitative study one could examine whether or not a general approach exists.

Our approach shows that each of our mappings lead to results, nevertheless we could not recognize correlations. Either the mapping is irrelevant for the approach or rather not representative for the structure of the workflow, where the latter option appears to be more probable to us. Possible next steps could be to experiment with stronger mappings, i.e., where the differences between the weights of the data types or the constraints are larger. One could also use a mapping where the weights are adapted to the probability of occurrence of each property for each workflow. This would result in a mapping where each edge weight is influenced by the entirety of all connections of a workflow without losing the unique properties of the associated connection. Another option could be to abandon the idea to use mappings according to the amount of information content and importance completely and use a different metric.

Our agglomerative clustering algorithm gives the most room for improvements. First, one could examine how the five wrong results arose only with one workflow (see Section 5.3). Additionally, one could improve the runtime of the algorithm as it is by far the slowest of our implemented algorithms. Besides that we focused on average linkage, whereas there are other linkage options we mentioned are left to evaluate. We already mentioned Newman's approach [7] for applying the edge betweenness algorithm to weighted networks, which might improve the accuracy of the results. Although we experimented with several different algorithms, there are other algorithms which may lead to good results. For example, we excluded all algorithms requiring the number of clusters to be found. It would be interesting to examine the results of, for example, partitional clustering and compare these results to our approach showing whether taking only the structure of the workflow into account is sufficient.

If a general approach is found, the final step would be to implement the graph clustering approach within RCE. This step is not trivial as the visual graph representation

has to be arranged according to the clustering so that a rectangular label can be drawn around the components of a group. Besides, the graphical representation has to be comfortable to use, which itself raises new questions and problems.

Bibliography

[1] Brigitte Boden et al. “RCE: An Integration Environment for Engineering and Science”. In: *arXiv preprint arXiv:1908.03461* (2019).

[2] Satu Elisa Schaeffer. “Graph clustering”. In: *Computer science review* 1.1 (2007), pp. 27–64.

[3] Santo Fortunato. “Community detection in graphs”. In: *Physics reports* 486.3-5 (2010), pp. 75–174.

[4] E. W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (12/1959), pp. 269–271.

[5] J. MacQueen. “Some methods for classification and analysis of multivariate observations”. In: *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*. Vol. 1. 14. Oakland, CA, USA. 1967, pp. 281–297.

[6] Ulrike von Luxburg. “A tutorial on spectral clustering”. In: *Statistics and computing* 17.4 (2007), pp. 395–416.

[7] M. E. J. Newman. “Analysis of weighted networks”. In: *Phys. Rev. E* 70 (5 11/2004), p. 056131.

[8] M. E. J. Newman and M. Girvan. “Mixing patterns and community structure in networks”. In: *Statistical mechanics of complex networks*. Springer, 2003, pp. 66–87.

[9] Linton C. Freeman. “A set of measures of centrality based on betweenness”. In: *Sociometry* (1977), pp. 35–41.

[10] R. Duncan Luce and Albert D. Perry. “A method of matrix analysis of group structure”. In: *Psychometrika* 14.2 (1949), pp. 95–116.

[11] Duncan J. Watts and Steven H. Strogatz. “Collective dynamics of ‘small-world’networks”. In: *nature* 393.6684 (1998), pp. 440–442.

[12] Andreas Kemper. *Valuation of network effects in software markets: A complex networks approach*. Springer Science & Business Media, 2009.

[13] M. E. J. Newman and M. Girvan. “Finding and evaluating community structure in networks”. In: *Phys. Rev. E* 69.2 (2004), p. 026113.

[14] M. E. J. Newman. “Fast algorithm for detecting community structure in networks”. In: *Phys. Rev. E* 69.6 (2004), p. 066133.

[15] Luca Donetti and Miguel A. Munoz. “Detecting network communities: a new systematic and efficient algorithm”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2004.10 (2004), P10012.

[16] Haifeng Du et al. “An algorithm for detecting community structure of social networks based on prior knowledge and modularity”. In: *Complexity* 12.3 (2007), pp. 53–60.

[17] Leon Danon et al. “Comparing community structure identification”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2005.09 (2005), P09008.

[18] William E. Donath and Alan J. Hoffman. “Lower bounds for the partitioning of graphs”. In: *Selected Papers Of Alan J Hoffman: With Commentary*. World Scientific, 2003, pp. 437–442.

[19] Miroslav Fiedler. “Algebraic connectivity of graphs”. In: *Czechoslovak mathematical journal* 23.2 (1973), pp. 298–305.

[20] Santo Fortunato and Darko Hric. “Community detection in networks: A user guide”. In: *Physics reports* 659 (2016), pp. 1–44.

[21] Fan R. K. Chung and Fan Chung Graham. *Spectral graph theory*. 92. American Mathematical Soc., 1997.

[22] Hadrien Van Lierde et al. *Spectral clustering algorithms for directed graphs*. 2015.

[23] Stephen Guattery and Gary L. Miller. “On the quality of spectral separators”. In: *SIAM Journal on Matrix Analysis and Applications* 19.3 (1998), pp. 701–719.

[24] Daniel A. Spielman and Shang-Hua Teng. “Spectral partitioning works: Planar graphs and finite element meshes”. In: *Proceedings of 37th Conference on Foundations of Computer Science*. IEEE. 1996, pp. 96–105.

[25] Lars Hagen and Andrew B. Kahng. “New spectral methods for ratio cut partitioning and clustering”. In: *IEEE transactions on computer-aided design of integrated circuits and systems* 11.9 (1992), pp. 1074–1085.

[26] Fan Chung. “Laplacians and the Cheeger inequality for directed graphs”. In: *Annals of Combinatorics* 9.1 (2005), pp. 1–19.

[27] Venu Satuluri and Srinivasan Parthasarathy. “Symmetrizations for clustering directed graphs”. In: *Proceedings of the 14th International Conference on Extending Database Technology*. 2011, pp. 343–354.

[28] Maxwell Mirton Kessler. “Bibliographic coupling between scientific papers”. In: *American documentation* 14.1 (1963), pp. 10–25.

[29] Henry Small. “Co-citation in the scientific literature: A new measure of the relationship between two documents”. In: *Journal of the American Society for information Science* 24.4 (1973), pp. 265–269.

[30] John Hopcroft et al. “Natural communities in large linked networks”. In: *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. 2003, pp. 541–546.

[31] Aaron Clauset, M. E. J. Newman, and Cristopher Moore. “Finding community structure in very large networks”. In: *Phys. Rev. E* 70.6 (2004), p. 066111.

[32] *Jackson Project*. 2020. URL: `https://github.com/FasterXML/jackson`.

[33] ISO/IEC 21778:2017. *The JSON data interchange syntax*. Standard. Geneva, CH: International Organization for Standardization, International Electrotechnical Commission, 11/2017.

[34] Dimitrios Michail et al. “JGraphT—A Java Library for Graph Data Structures and Algorithms”. In: *ACM Trans. Math. Softw.* 46.2 (05/2020).

[35] Emden R. Gansner and Stephen C. North. “An open graph visualization system and its applications to software engineering”. In: *SOFTWARE - PRACTICE AND EXPERIENCE* 30.11 (2000), pp. 1203–1233.

[36] Mikio L. Braun. *jblas-Linear Algebra for Java*. 2015. URL: `http://jblas.org/`.

[37] *Apache Maven*. 2020. URL: `https://maven.apache.org/`.

[38] Ulrik Brandes. “A faster algorithm for betweenness centrality”. In: *Journal of mathematical sociology* 25.2 (2001), pp. 163–177.

[39] Björn H. Junker and Falk Schreiber. *Analysis of biological networks*. Vol. 2. John Wiley & Sons, 2011.

[40] Stanley Wasserman, Katherine Faust, et al. *Social network analysis: Methods and applications*. Vol. 8. Cambridge university press, 1994.

[41] M. E. J. Newman. “The structure of scientific collaboration networks”. In: *Proceedings of the national academy of sciences* 98.2 (2001), pp. 404–409.

[42] Leon Danon, Albert Diaz-Guilera, and Alex Arenas. “The effect of size heterogeneity on community identification in complex networks”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2006.11 (2006), P11010.

[43] Jeongah Yoon, Anselm Blumer, and Kyongbum Lee. “An algorithm for modularity analysis of directed and weighted biological networks based on edge-betweenness centrality”. In: *Bioinformatics* 22.24 (2006), pp. 3106–3108.

[44] Ruth Dunn, Frank Dudbridge, and Christopher M. Sanderson. "The use of edge-betweenness clustering to investigate biological function in protein interaction networks". In: *BMC bioinformatics* 6.1 (2005), p. 39.

[45] Masahiro Tanaka and Osamu Tatebe. "Workflow scheduling to minimize data movement using multi-constraint graph partitioning". In: *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. IEEE. 2012, pp. 65–72.

[46] Jae-Yoon Jung, Joonsoo Bae, and Ling Liu. "Hierarchical clustering of business process models". In: *International Journal of Innovative Computing, Information and Control* 5.12 (2009), pp. 1349–4198.

[47] Emanuele Santos et al. "A first study on clustering collections of workflow graphs". In: *International Provenance and Annotation Workshop*. Springer. 2008, pp. 160–173.

[48] Jae-Yoon Jung and Joonsoo Bae. "Workflow clustering method based on process similarity". In: *International Conference on Computational Science and Its Applications*. Springer. 2006, pp. 379–389.