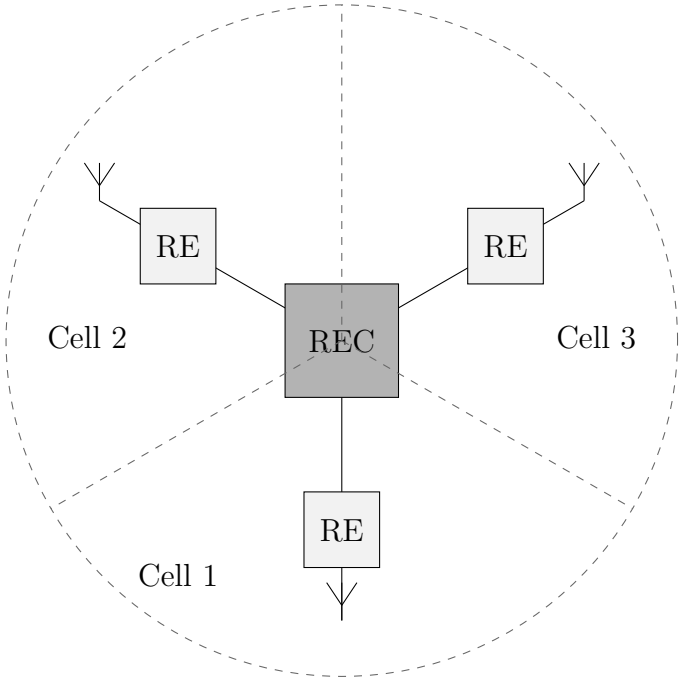


Pattern Recognition of RBS Configuration Topologies in Radio Access Networks

Master's Thesis in Computer Science:
Algorithms, Languages and Logic

JOACIM LINDER

ELLINOR RÅNGE

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Master's thesis

Pattern Recognition of RBS Configuration Topologies in Radio Access Networks

Joacim Linder

Ellinor Råange

CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Department of Computer Science and Engineering
Machine Learning, Algorithms & Computational Biology
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Pattern Recognition of RBS Configuration Topologies in Radio Access Networks
JOACIM LINDER
ELLINOR RÅNGE

Supervisor:	Richard Johansson, Department of Computer Science and Engineering
Advisors:	Krister Bergh, Ericsson
	Staffan Ehnebom, Ericsson
Examiner:	Alexander Schliep, Department of Computer Science and Engineering

Master's Thesis
Department of Computer Science and Engineering
Machine Learning, Algorithms & Computational Biology
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg, Sweden
Telephone +46 31 772 1000

Cover: A simple illustration of a radio base station visualising its main components and coverage.

Typeset in LaTeX
Gothenburg, Sweden 2017

Abstract

This thesis explores how graph-based representations for cluster evaluation may be applied on configuration topologies of radio base stations. It presents experiments performed using a Weisfeiler-Lehman subtree kernel and a graph embedding method based on graph edit distance computations. These techniques are compared to the performance of simpler baseline methods.

Experiments are performed on a data set which have been categorised at two levels of detail by RBS configuration experts at Ericsson, and evaluation is performed using both intrinsic and extrinsic metrics. The Weisfeiler-Lehman subtree kernel shows promising results and has a reasonable runtime when executed on the provided data set. The graph embedding technique, on the other hand, gives less promising results even when compared to the baseline methods.

Keywords: pattern recognition, radio access networks, radio base stations, configuration topologies, graph kernels, graph embeddings, clustering.

Acknowledgements

We would like to express our gratitude towards our supervisor Richard Johansson, who has given us useful feedback and support throughout this work. We are also greatful for the support from our advisors at Ericsson, Staffan Ehnebom and Krister Bergh, who generously shared their expert knowledge of RBS configurations and hepled us design experiments. We would also like to acknowledge our examiner Alexander Schliep for his helpful feedback and remarks during midterm discussion.

Finally, we would like to give a special thanks to our fellow Master's thesis workers at our department at Ericsson, Ludvig Helén and Alexander Persson.

Joacim Linder and Ellinor Rånge, Gothenburg, June 2, 2017

Contents

1
Introduction

This thesis investigates how graph-based pattern recognition can be applied to a specific problem in telecommunications. The data in this research can be represented as graphs, since graphs can model relations between different components of an object in the data. Two graph-based data representations for pattern recognition will be explored, namely graph embeddings and graph kernels. These representations enable the use of pattern recognition algorithms developed for feature vectors. The pattern recognition algorithms evaluated in this thesis will use unsupervised learning to divide the data into clusters. For this reason, cluster evaluation will be an important subject in this thesis, in addition to the graph embedding and the graph kernel representations.

In this chapter, we will present the context of the research, to motivate what makes this research relevant to Ericsson. We will then specify the problem further and list some previous work, relevant to this thesis.

1.1 Context

Ericsson is a global telecommunications company which provides products and services for mobile and landline network operators. An operator provides services for mobile users in areas with different levels of mobile traffic, ranging from rural areas to busy city environments. The operator needs to consider mountains, tall buildings and other obstacles of the areas it wants to cover as well as provide services in indoor and outdoor environments. The various numbers of conditions which the operator needs to consider require different solutions of coverage and capacities.

A Radio Access Network (RAN) is a part of a mobile telecommunication network which connects a user to the core network. A RAN contains a collection of Radio Base Stations (RBSs), where each RBS is configured using an object-oriented managed interface. A part of the managed interface defines the configuration topology of an RBS, in particular how the RBS equipment is connected and how the RBS functions are using the RBS equipment. An RBS can be configured by the operator to meet different needs of coverage and capacities, which is why a RAN can contain a number of distinct types of RBS configuration topologies.

Ericsson wants to be able to decide what types of RBS configuration topologies exist in a RAN. This information should be useful to gather statistics and to optimise testing, so that all types of configuration topologies have corresponding test cases. Today, the elements of the RBS configuration topology are studied by visualising the so called management information tree structure. Deciding if two

RBS configuration topologies are equal is difficult in the management information tree because the relations between entities are not explicitly visualised in this tree structure. However, relations between entities can be visualised by analysing attributes of the entities in the management information tree and creating a configuration topology graph.

To decide if RBS configuration topologies are equal, or different, is important in the optimisation of test cases. Unknown configurations ought to be identified in order to introduce test cases for such, and identifying (unknown and known) configurations is hence of great value.

1.2 Problem

The purpose of this Master's thesis is to investigate the performance of graph-based pattern recognition methods applied to RBS configuration topologies. The objective of the methods is to divide a set of configurations topologies into clusters, so that each cluster contains configuration topologies which are similar within the scope of this thesis.

A configuration topology of an RBS describes how RBS equipment is connected and how RBS functions are using the RBS equipment. Within the scope of this thesis, the connections between entities in the configurations are of great importance, since two RBSs with equal equipment can be considered different if they have different topologies. Furthermore, the difference between the coupling of entities between two RBS configuration topologies can be more significant than the difference between the actual distribution of entity types. Due to this importance of relations between entities, a graph representation of an RBS configuration is preferred to model its topology.

There are some ways of configuring RBSs, which are known and tested by Ericsson, and recommended to its customers for different usages. However, Ericsson has no knowledge how the operators actually configure their radio base stations. There might be configurations invented by the operators, which Ericsson has not previously seen nor tested. Also, the customers might have configured their RBSs incorrectly or abnormally. Therefore, clustering is the appropriate approach of analysis, as it tries to find patterns on its own rather than classifying them according to known classes. Hence, with help of clustering the goal of this thesis is to investigate methods of finding differences in configurations in a radio access network. Thus, the research questions of this thesis are posed as:

1. To which extent can pattern recognition methods correctly find clusters in a set of RBS configuration topologies?
2. Do the performances of more complex algorithms compensate for longer runtimes, compared to more simple algorithms?
3. Are the algorithms able to perform well, even if they are not specifically tailored to the problem domain?

1.3 Related Work

The research in this thesis is primarily based on two topics in the field of graph-based pattern recognition: graph kernels and graph embeddings.

There have been a great number of graph kernels proposed since the first graph kernel was introduced by Gärtner et al. [1]. Two types of graph kernels designed for comparison of large graphs are graphlet kernels [2] and Weisfeiler-Lehman graph kernels [3]. Both compare graphs by enumerating substructures within the graphs.

Shervashidze et al. [2] propose a graph kernel based on a distribution of subgraphs of a fixed size k, so called graphlets. The authors also propose two theoretical speedups to cope with the prohibitively expensive enumeration of all possible graphlets. However, the proposed graphlet kernels only consider graphs with unlabelled vertices.

A family of graph kernels based on Weisfeiler-Lehman's test of isomorphism was introduced by Shervashidze et al. [3]. In contrast to the above-mentioned graph kernel, these kernels compare graphs based on their vertex labels. One of these graph kernels, the subtree kernel, turns out to be competitive to state-of-the-art graph kernels in terms of both accuracy and runtime.

Both mentioned studies describe experiments on a number of different graph data sets and compare their findings to various graph kernels. Experiments are performed using supervised learning and the chosen classifiers are support vector machines (SVM) [4].

Riesen and Bunke [5] propose an approach to embed graphs into feature vectors in terms of graph edit distance to selected prototypes. The dimension of the extracted feature vectors is decided by the number of prototypes and the performance of the embedding method depends on both the number of prototypes and the chosen prototype selection strategy. Another challenge that arises in this method is the complexity of graph edit distance, which grows exponentially in the number of vertices. Therefore, an approximation of the distance is desirable to avoid prohibitively expensive computations. Despite these challenges, this method offers a flexibility of choice of graph (directed or undirected, any type of vertex labels etc.).

Another embedding method proposed by Li et al. [6] extracts a number of topological and label attributes from a graph to a feature vector. Features include the average edge degree, average path length and label entropy. This approach should be efficient on large graphs and experiments are compared to a number of state-of-the-art graph kernels.

As for graph kernels, both articles mentioned above conduct experiments using supervised learning. In general, there seems to be a lack of examples of unsupervised learning in graph-based pattern recognition.

1.4 Limitations

There are possibly thousands of different configurations, where most of them consist of additional units in order to control e.g. a power supply of an RBS. This thesis does not include these and only focuses on those units which control and

affect the actual functionality of outdoor RBSs. Indoor RBSs have a insignificant diversity and are hence neglected in the scope of this thesis.

The work will not include designing new algorithms, but rather utilising already existing ones. In addition to baseline methods, we concentrate on one kernel method and one more advanced embedding method. Evaluating additional methods would be outside the scope of the thesis.

1.5 Thesis Outline

The remainder of this thesis will proceed as follows. Chapter 2 briefly discusses the domain of the problem, with examples of aspects in the variety of RBS configurations. Chapter 3 then starts by presenting some basic graph theory and its application in this thesis. The chapter then continues describing some ways of representing graph data for machine learning, followed by a presentation of methods chosen for this research. Chapter 3 finally presents some ways of clustering different representations of data, and how to evaluate the clustering.

The methodology and implementation of the theory is described in Chapter 4. In Chapter 5 the results from the executed methods described in Chapter 4 are shown, and discussed upon in Chapter 6.

2
Background

In order for the reader to gain some knowledge of the domain in this thesis, this chapter presents a short introduction to what a radio base station is, as well as highlighting differences of the configurations of radio base stations.

2.1 Radio Base Stations

Radio Base Stations (RBSs) are nodes in large radio access networks providing the possibility to call friends, family and colleagues, and access the Internet. Each RBS has a number of antennas, which cover a certain area. These coverage areas are called *cells* and decide to which RBS a user connects with its cell phone or mobile device. The RBS then connects the user to the core network.

The RBSs consist of a radio equipment controller (REC) which is the main equipment controlling the RBS. These directly control radio equipment (RE), which in turn is connected to antennas. In Figure 2.1, a simple RBS configuration covering three cells is visualised. The figure demonstrates that each antenna with a corresponding RE is used by one cell. In addition, each cell uses a part of the REC.

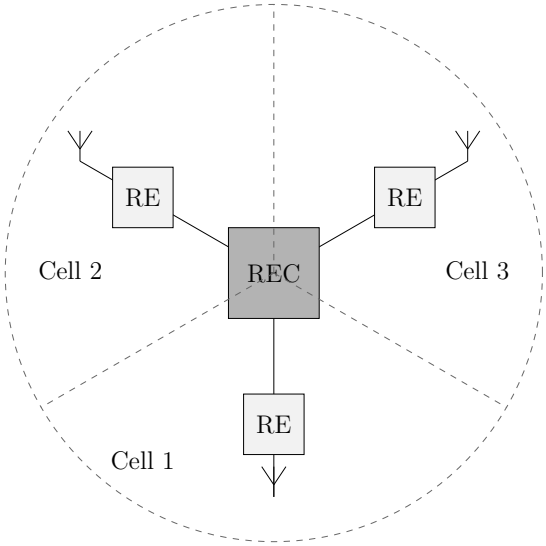


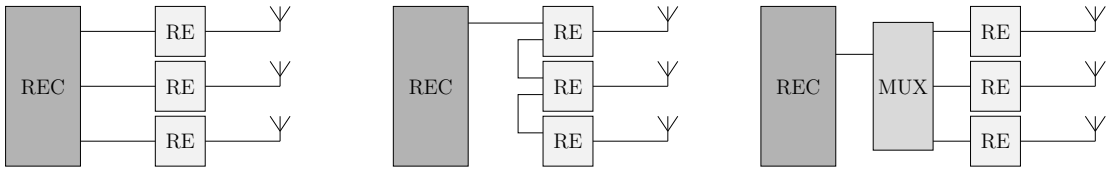
Figure 2.1: A simple example of an RBS visualising its main components and coverage. Antennas are marked as ⍱.

In this thesis, the configurations of RBSs are of most interest. As there are different circumstances where the RBSs are located, the operators want e.g. dif-

ferent coverage and capacities. An extension of the configuration in Figure 2.1 with six REs and six antennas could e.g. imply the same number of cells, where each such cell is covered with two different frequencies. It could also mean that there are six cells. There could hence be different functionality depending on the configurations of the RBSs.

Not only the functionality can vary between RBSs, but also the way they are connected. Only looking at the equipment, the configuration in Figure 2.1 could be visualised as in Figure 2.2a. In this case the three REs are star-coupled from the REC, distributing the load on three ports on the REC. There might also be several connections between REC and an RE, in order to distribute the load of one RE on several connections.

Capacities and coverage are however not the only things to take into consideration, but also the costs of the different equipment. There are cases when the antennas and REs are situated on high towers and the REC at the base of the tower. In such cases the operators might want to reduce their cost of cables and cascade-couple the REs in order to do so (Figure 2.2b). The operator then only connects one long cable from the REC up to an RE, and connects the REs by shorter cables. The downside of this is that the load of the cascading REs is put on one port of the REC.



(a) Three REs star-coupled with an REC.

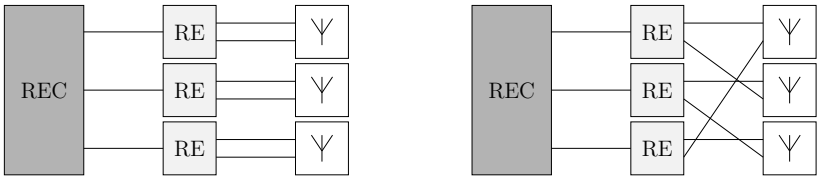
(b) Three REs cascade-coupled from an REC.

(c) Three REs connected to an REC via a MUX.

Figure 2.2: Examples of simple RBS configurations visualising some differences in their couplings, where antennas are marked with ⍱.

For larger configurations, there might be more REs than there are ports on the REC. Then a multiplexer (MUX) can be used, which is an add-on enabling the use of more REs.

There are however not only different ways of connecting REs to RECs, but also different ways of connecting antennas to REs. In the simple examples presented so far, there has only been one cable connecting an RE with an antenna. It is also possible to draw two cables from an RE to an antenna in order to meet other capacities (Figure 2.3a). With two cables from each RE, there is also the possibility to connect an RE to more than one antenna (Figure 2.3b). In this way, if one RE becomes faulty, a cell will not be lost, but the performance of two cells will decrease.



(a) Straight-connected antennas

(b) Crossed-connected antennas

Figure 2.3: Examples of simple RBS configurations visualising some differences in the connections of antennas and REs.

As the figures presented in this chapter are simple examples, they illustrate the main concepts of the differences in RBS configurations. The configurations contain additional subcomponents, such as ports, which do not affect the concepts of the differences, but are needed to be considered when analysing the configuration topologies. Such analysis is done representing the RBS configurations as *graphs*.

3
Theory

In this chapter, we introduce the theory on which we have based our research and that will be of relevance throughout this thesis. The chapter starts by introducing some basic graph theory and relevant notations in Section 3.1. Then it proceeds in Section 3.2 by introducing different ways of representing graph data in the field of machine learning. There the concepts relevant to the different representations are explained, and the baseline methods used in this thesis are presented. In Sections 3.3 and 3.4, two more complex methods are presented, which build upon two different representations of graph data. In addition, these sections include how to set and modify these methods, as well as techniques to speed up their computation times. In Section 3.5, different methods of clustering data are presented, and finally in Section 3.6, metrics of evaluating the clustering both intrinsically and extrinsically are presented.

3.1 Graph Theory

Graphs are data structures which describe a set of entities (vertex set) and a set of binary relationships between these entities (edge set). We say that a graph is directed if its edges are represented by ordered pairs of vertices, and undirected if they are represented by unordered pairs. Additionally, both vertices and edges can be labelled, meaning that there are labelling functions which map a vertex or an edge to a label. For the purpose of this thesis, the edge labelling function is omitted and all graphs are considered undirected.

Definition 3.1 (Graph)**.** *A graph $G = (V, E, \ell)$ is an ordered triplet, where V is a set of vertices, $E \subseteq V \times V$ is a set of edges and $\ell : V \to \Sigma$ is a labelling function from the set of vertices to a set of equipment and function labels Σ.*

For simplicity we may denote by $V(G)$ and $E(G)$ the vertex set and edge set, respectively, of graph G. Also, when considering a set of graphs $\mathcal{G}$, the labelling function may be joint for all graphs in $\mathcal{G}$, i.e. $\ell : \bigcup_{G \in \mathcal{G}} V(G) \to \Sigma$.

In this thesis, RBS configurations will be described as graphs, where each vertex in the graph has a symbolic label corresponding to what type or instance of equipment or function the vertex represents. An edge represents either a cable connecting two pieces of equipment, an application of functionality or an "ownership"-relation between for example a piece of equipment and its ports. The type of an edge is easily distinguished by the vertices the edge connects. In Section 4.1, the specific representation of RBS configurations used during experiments is explained.

3.1.1 Notation

For clarity of the remainder of this thesis, let us state and define some concepts used in this thesis.

We denote by the *size* of a graph G the number of vertices in G, i.e. $|V(G)|$. The *neighbourhood* $\mathcal{N}(v)$ of a vertex v is the set of nodes to which v is connected by an edge, i.e. $\mathcal{N}(v) = \{v' \,|\, (v, v') \in E\}$. A *star* is a structure within a graph which contains a vertex and the vertex's outgoing edges. Hence, for a given graph $G = (V, E, \ell)$ and a vertex $v \in V$, a star S_v is defined as

$$S_v = \Big(v, \{(v, u) \,|\, (v, u) \in E\}, \ell\Big).$$

A *walk* is a sequence of vertices in a graph, in which consecutive vertices are connected by an edge, and a *path* is a walk which consists of distinct vertices only. We call a graph *connected* if there exists a path between any pair of vertices in the graph. A *subgraph* $G' = (V', E', \ell')$ of a graph $G = (V, E, \ell)$, written $G' \subseteq G$, is a graph such that $V' \subseteq V$, $E' \subseteq E \cap V' \times V'$ and $\ell'(v') = \ell(v')$ for every $v' \in V'$. With *subtree* we mean a connected subgraph of a graph, which contains no cycles, but has an designated root vertex. Two graphs $G = (V, E, \ell)$ and $G' = (V', E', \ell')$ are said to be *isomorphic* if there exists a bijective mapping $f : V \to V'$ such that $(u, v) \in E$ if and only if $\Big(f(u), f(v)\Big) \in E'$, and $\ell(v) = \ell'\Big(f(v)\Big)$ for every $v \in V$. This means that two graphs are isomorphic if they are structurally identical, i.e. if each vertex in the first graph has a one-to-one correspondence to a vertex in the second graph having the same label and edge structure as the first, and vice versa.

3.2 Data Representation for Machine Learning

Machine learning is a field in computer science that has gained much attention in recent years. The key idea is that machines should learn patterns in a data set by using learning algorithms and be able to come to a desirable conclusion when exposed to new data.

Two main learning approaches in machine learning are supervised and unsupervised learning. In the former, the algorithms use a training data set, where each element is associated with a specific class. For example, recognition of handwritten digits, where each element in the training set is associated with the corresponding digit. In unsupervised learning, the elements in the data set are not associated with a specific class. Due to paucity of knowledge of how the operators configure their radio base stations, and to be able to identify unknown configurations, unsupervised learning will be used in this project by reasons of its ability to find structures which are not predefined. This opposed to supervised learning, which learns from imitating given structures.

In some machine learning applications, it may be desirable to represent elements in a data set as graphs. A graph can represent an object, not only by what entities an object has, but also the relations between the entities. However, graph representations give rise to a few challenges. First of all, algorithms for graph matching are computationally demanding. Second, there is a paucity of

mathematical structure in the graph domain. As a consequence, there is a lack of machine learning algorithms that are applicable directly on graphs.

In this work, we are looking at approaches which enable graph data to be applied on machine learning algorithms designed for vectors and *kernels*.

3.2.1 Vector Representation

Feature vectors are common data structures to represent objects in machine learning. A feature vector represents an object by n numerical measurements of the object, called *features*.

Graph embeddings can be thought of as (feature) vector representations of graphs. To represent a graph as an n-dimensional feature vector a mapping

$$\phi : \mathcal{G} \to \mathbb{R}^n$$

is defined. A graph embedding method can be trivially simple or very complicated depending on which algorithm is used to extract the features. The features could also vary from counting specific substructures in the graphs to a comparison with other objects. Once a graph has been mapped into a feature vector it can be used on any available learning algorithm designed for vector input.

Two Simple Graph Embedding Methods

One simple embedding method is the one producing a feature vector containing the number of occurrences of each label in a graph. If there are n discrete possible labels, then we have a mapping to a feature space $\mathbb{R}^n$, where each dimension is the occurrence of a specific label. Due to its simplicity, this method will be used as a baseline in this thesis.

Given a graph $G = (V, E, \ell)$, let $\#(\sigma_i, G)$ denote the number of vertices with label $\sigma_i \in \Sigma$ in G, i.e. $\#(\sigma_i, G) = \Big|\{v \in V \,|\, \ell(v) = \sigma_i\}\Big|$. The *baseline vector* ϕ_B is then computed as

$$\phi_B(G) = \Big(\#(\sigma_1, G), \#(\sigma_2, G), \ldots, \#(\sigma_n, G)\Big). \tag{3.1}$$

Note that this approach does not take any relations between entities in account.

An extended version of the previous method also captures relations between vertices. This method outputs a feature vector containing the number of occurrences of each type of instance of entities in a configuration (as for the previous method), as well as the number of occurrences of edges between pairs of vertices of any labels in the graph. Given a graph $G = (V, E, \ell)$, let $\#(\sigma_i \leftrightarrow \sigma_j, G)$ denote the number of edges between vertices with labels $\sigma_i, \sigma_j \in \Sigma$ in G, i.e.

$$\#(\sigma_i \leftrightarrow \sigma_j, G) = \Big|\Big\{(u, v) \in E(G) \,\Big|\, \ell(u) = \sigma_i, \ell(v) = \sigma_j \,\vee\, \ell(v) = \sigma_i, \ell(u) = \sigma_j\Big\}\Big|.$$

Combining these occurrences with $\phi_B(G)$ gives the feature vector of what will be referred to as the *extended baseline*:

$$\phi_{B^+}(G) = \Big(\phi_B(G), \#(\sigma_1 \leftrightarrow \sigma_1, G), \#(\sigma_1 \leftrightarrow \sigma_2, G), \ldots, \#(\sigma_{|\Sigma|} \leftrightarrow \sigma_{|\Sigma|}, G)\Big). \tag{3.2}$$

A more sophisticated graph embedding method will be introduced in Section 3.3.

3.2.2 Kernels

Graph kernels are tools for representing graph data and have seen rapid development since first introduced by Gärtner et al. [1]. As they respect and exploit graph topology, but restrict themselves to comparing substructures of graphs which are computable in polynomial time, they are an attractive way of comparing graphs.

Graph kernels, and other kernel methods, can be applied in high dimensional feature spaces without explicitly computing the feature map. This is often referred to as the *kernel trick*, due to which kernel methods are computationally attractive. The explicit computation for graph kernels would be costly, since graph kernels often extract features by counting substructures of the graphs.

Formally, a kernel method uses a positive semidefinte kernel function $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ on some input space $\mathcal{X}$. This function can be seen as first applying an implicit mapping $\phi : \mathcal{X} \to \mathcal{H}$ to the inputs, where $\mathcal{H}$ is a Hilbert space, and then taking the scalar product of the resulting mappings, i.e. $\kappa(x, x') = \left\langle \phi(x)\,,\, \phi(x') \right\rangle$ for $x, x' \in \mathcal{X}$.

As the feature map ϕ is implicit, there is no constraint of it being unambiguous. Assume $\mathcal{X} \subseteq \mathbb{R}^2$ and for $\boldsymbol{x} = (x_1, x_2) \in \mathcal{X}$ we have $\phi(\boldsymbol{x}) = (x_1^2, x_2^2, \sqrt{2}x_1x_2)$. Then the kernel function will compute:

$$\begin{aligned}
\kappa(\boldsymbol{x}, \boldsymbol{x}') &= \left\langle \phi(\boldsymbol{x})\,,\, \phi(\boldsymbol{x}') \right\rangle \\
&= (x_1^2, x_2^2, \sqrt{2}x_1x_2) \cdot (x_1'^2, x_2'^2, \sqrt{2}x_1'x_2') \\
&= (x_1x_1')^2 + (x_2x_2')^2 + 2x_1x_1'x_2x_2' \\
&= (x_1x_1' + x_2x_2')^2 \\
&= \langle \boldsymbol{x}\,,\, \boldsymbol{x}' \rangle^2.
\end{aligned} \tag{3.3}$$

Note that if we instead would have defined the feature map in a 4-dimensional feature space as $\phi(\boldsymbol{x}) = (x_1^2, x_2^2, x_1x_2, x_2x_1)$ we would have got the same result. So, there is no explicit feature map present in kernel functions, and hence, it does not need to be computed.

One reason to use kernel methods and move the computations to higher dimensions can easily be motivated when working with linearly inseparable data. As the data are not linearly separable in some input space the implicit mapping to a higher dimensional feature space enables the data to be linearly separated in the feature space. In such cases, it is easier to group the data by separating them by hyperplanes (Figure 3.1). Another reason is that kernel methods are also applicable directly on the input, as exemplified in (3.3), leaving the explicit feature map uncomputed.

In the field of machine learning, there are several different graph kernels defined. Despite their differences, they can be categorised into three classes: graph kernels based on walks [1, 7] and paths [8], graph kernels based on limited size subgraphs [2], and graph kernels based on subtree patterns [9].

The first class, graph kernels based on walks and paths, compares two graphs by the number of matching pairs of random walks or paths in the graphs. As an example, Borgwardt and Kriegel [8] present a shortest path kernel, which counts the pairs of shortest paths having the same lengths and the same source and target labels in two graphs.

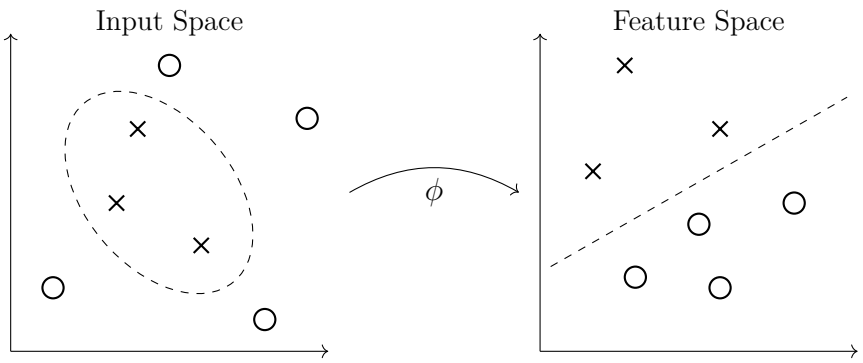


Figure 3.1: The idea of feature mapping: transform linearly inseparable input data to linearly separable data in feature space via ϕ. Crosses and circles represent data points of different categories and the dashed lines represent decision boundaries.

In the second class, graph kernels based on limited size subgraphs, we find kernels based on so called graphlets, where the graphs are represented as the numbers of occurrences of each possible subgraph of size $k \in \{3, 4, 5\}$ in the graphs [2].

A kernel from the third class, subtree kernels, is used in this thesis, and is presented in Section 3.4. The very first subtree kernel, defined by Ramon and Gärtner in 2003 [9], compares two graphs G and G' by iteratively comparing the neighbourhoods of two vertices $v \in V(G)$ and $v' \in V(G')$. In other words, for every pair of vertices $v \in V(G)$ and $v' \in V(G')$, the kernel counts all pairs of matching substructures in subtrees pattern with v and v' as roots. This kernel has later been refined for applications in, inter alia, chemoinformatics and handwritten digit recognition [3].

3.3 Graph Embedding Using Dissimilarities

Dissimilarity space embeddings use dissimilarity measures as features for pattern recognition. In this sense, objects are represented by vectors of dissimilarities to prototype objects, rather than absolute features. The idea is that objects, in our case graphs, which are regarded similar should have similar differences to the prototypes.

Graph embedding using dissimilarities (GEUD) [5] embeds a graph G into a feature vector using prototype selection and graph edit distance computations. Let $\mathcal{G}$ be a set of input graphs, $\mathcal{P} = \{p_1, ..., p_n\} \subseteq \mathcal{G}$ be a set of n prototype graphs and $d(G, p_i)$ be a function computing (an approximation of) the graph edit distance between graph G and prototype $p_i \in \mathcal{P}$. Then the mapping

$$\phi_n^{\mathcal{P}} : \mathcal{G} \to \mathbb{R}^n \tag{3.4}$$

is defined as the function

$$\phi_n^{\mathcal{P}}(G) = \Big(d(G, p_1), d(G, p_2), \ldots, d(G, p_n)\Big). \tag{3.5}$$

Graph edit distance measures the dissimilarity between graphs by the minimum

amount of distortion it takes to transform one of the graphs so that it is isomorphic to the other. This means that a vector representation of a graph will contain the dissimilarities of this graph to each prototype in the prototype set, in terms of these distortions.

This method gives rise to three challenges. The first is that the graph edit distance problem is NP-hard [10], which requires some approximation to be computationally feasible. The second and third challenges are how to choose proper prototypes and how many to choose. In this section we will describe methods to overcome each of these challenges.

3.3.1 Graph Edit Distance

Graph edit distance (GED) is a measure of dissimilarity between two graphs which has the benefit that it can be applied on any type of graph, but has the downside of being computationally demanding. In GED a dissimilarity measure is the minimum cost of edit operations which transforms a source graph G to a target graph G'. Edit operations on a graph include deletion and insertion of vertices and edges, and substitution, i.e. relabelling, of vertices.

Definition 3.2 (Edit Operations)**.** *Let $G = (V, E, \ell)$ be the source graph and $G' = (V', E', \ell)$ be the target graph. Furthermore, let $v \in V$, $(v, u) \in E$, $v' \in V'$, $(v', u') \in E'$ and ε denote an empty vertex or edge. Then an edit operation ϵ can be any of the following operations:*

- $v \to v'$ *substituting v for v'*
- $v \to \varepsilon$ *deleting v*
- $\varepsilon \to v'$ *inserting v'*
- $(v, u) \to (v', u')$ *substituting (v, u) to (v', u')*
- $(v, u) \to \varepsilon$ *deleting (v, u)*
- $\varepsilon \to (v', u')$ *inserting (v', u')*

Let $\gamma(G, G') = \{\epsilon_1, \epsilon_2, \ldots, \epsilon_k\}$ be an *edit path* which transforms G to G' and let $c(\epsilon_i)$ be the cost of edit operation i. An example of an edit path is found in Figure 3.2.

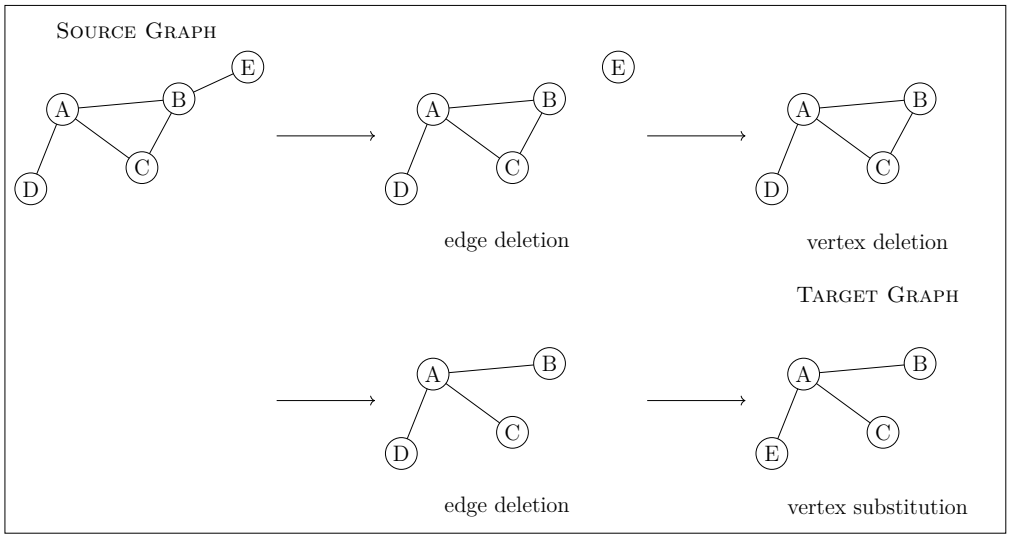


Figure 3.2: Example of an edit path which transforms the source graph to the target graph.

Definition 3.3 (Graph Edit Distance)**.** *Let G be the source graph and G' be the target graph. Then the graph edit distance between G and G' is defined as*

$$d(G, G') = \min_{\gamma(G,G') \in \Gamma(G,G')} \sum_{\epsilon \in \gamma(G,G')} c(\epsilon), \tag{3.6}$$

where $\Gamma(G, G')$ denotes the set of all edit paths which transforms G to G'.

Setting the Cost of Edit Operations

The cost function $c(\epsilon)$ can be defined on any kind of graph structure definition, such as directed or undirected graphs with any type of vertex or edge labels, such as symbolic or numerical values. This is what makes GED flexible. Before we establish cost functions for edit operations on stars, we introduce cost functions for edge and vertex edit operations. The presented costs are rudimentarily based on the concepts and possibilities of RBS configurations. This means that there is a cost of changing equipment of different types, but no weight is put on the difference of the types, i.e. there either is a difference or there is not. Obviously, there may be changes of equipment that are more substantial than others, but this would require more knowledge of the problem domain. This will not be investigated further.

Let $G = (V, E, \ell)$ be the source graph and $G' = (V', E', \ell)$ be the target graph and assume $v \in V$ and $v' \in V'$. Also, let E_v denote the set of outgoing edges from $v \in V$, i.e. $E_v = \{e \in E \,|\, e = (v, u)\}$. Then the costs of substituting E_v with $E_{v'}$, deleting E_v and inserting $E_{v'}$ are:

$$\begin{aligned} c(E_v \to E_{v'}) &= \big||E_v| - |E_{v'}|\big| && \text{substitution} \\ c(E_v \to \varepsilon) &= |E_v| && \text{deletion} \\ c(\varepsilon \to E_{v'}) &= |E_{v'}| && \text{insertion.} \end{aligned}$$

Since the vertex labels are symbolic in this thesis the cost function is defined so that there is no cost for substituting two vertices with equal labels

$$c(v \to v') = \begin{cases} 0 & \text{if } \ell(v) = \ell(v') \\ 1 & \text{otherwise} \end{cases}$$
$$c(v \to \varepsilon) = c(\varepsilon \to v') = 1$$

We then have the cost of substituting, deleting or inserting of stars

$$\begin{aligned} c(S_v \to S_{v'}) &= c(v \to v') + c(E_v \to E_{v'}) && \text{star substitution} \\ c(S_v \to \varepsilon) &= c(v \to \varepsilon) + c(E_v \to \varepsilon) && \text{star deletion} \\ c(\varepsilon \to S_{v'}) &= c(\varepsilon \to v') + c(\varepsilon \to E_{v'}) && \text{star insertion} \end{aligned}$$

Constraints on the cost functions need to be added so that the cost of a path that transforms a source graph to a target graph is a true edit distance. These constraints apply to both vertex and edge operations, hence we for now denote by element either vertex or edge. Any edit operation ϵ either adds or does not add any cost to the total cost. This requires that $c(\epsilon) \geq 0$, for any edit operation ϵ. Further constraints are added so that no path contains an insertion with a subsequent deletion of the same element

$$\begin{aligned} c(a \to \varepsilon) &> 0 && \text{deleting } a \\ c(\varepsilon \to a) &> 0 && \text{inserting } a. \end{aligned}$$

Finally, constraints to avoid unnecessary substitution of elements in a path are added,

$$\begin{aligned} c(a \to b) &\leq c(a \to c) + c(c \to b) && \text{substituting } a \text{ to } b \\ c(a \to \varepsilon) &\leq c(a \to b) + c(b \to \varepsilon) && \text{deleting } a \\ c(\varepsilon \to a) &\leq c(\varepsilon \to b) + c(b \to a) && \text{inserting } a, \end{aligned}$$

where a, b and c are all either vertices or edges.

Computing the exact graph edit distance is usually carried out by an implementation of the A^* search algorithm [11], whose computational complexity is exponential in the number of vertices of the graphs [12].

Approximations of GED

An approximation of GED solvable in polynomial time by means of bipartite graph matching (BP) was introduced by Riesen and Bunke [12]. BP transforms GED into an instance of the linear assignment problem. The method defines an $(n + m) \times (m + n)$ cost matrix C, given a source graph G of size n and a target

graph G' of size m. An approximation of the optimal edit path $\gamma(G, G')$ is found by using Munkre's algorithm [13] on C. The cost matrix is defined so that element c_{ij} of the upper left $n \times m$ quadrant of C is the cost of substituting a star S_{v_i} in G to a star S_{v_j} in G'. The element c_{ii} of the $n \times n$ quadrant in the upper right corner is the cost of removing S_{v_i}, and c_{jj} in the bottom left $m \times m$ quadrant is the cost of inserting star S_{v_j}. Non-diagonal elements of the upper right and bottom left quadrants are set to infinity since any star only can be deleted and inserted at most once. Element c_{ij} in the bottom right $m \times n$ quadrant is the cost of inserting and deleting a non-existent element, i.e $\varepsilon \to \varepsilon$, which is zero. BP has the same complexity as Munkre's algorithm, which depends on the size of the cost matrix, i.e. $\mathcal{O}((m+n)^3)$. More formally the cost matrix C have the following structure:

$$C = \left[\begin{array}{cccc|cccc} c_{11} & c_{12} & \cdots & c_{1m} & c_{1\varepsilon} & \infty & \cdots & \infty \\ c_{21} & c_{22} & \cdots & c_{2m} & \infty & c_{2\varepsilon} & \cdots & \infty \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \ddots & \infty \\ c_{n1} & c_{n2} & \cdots & c_{nm} & \infty & \cdots & \infty & c_{n\varepsilon} \\ \hline c_{\varepsilon 1} & \infty & \cdots & \infty & 0 & 0 & \cdots & 0 \\ \infty & c_{\varepsilon 2} & \ddots & \vdots & 0 & 0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \infty & \vdots & \ddots & \ddots & 0 \\ \infty & \cdots & \infty & c_{\varepsilon m} & 0 & \cdots & 0 & 0 \end{array}\right].$$

When executing Munkre's algorithm of C an approximation of the optimal edit path $\gamma(G, G')$ is found with a total cost, $\text{EditCost}(G, G') = \sum_{\epsilon \in \gamma(G,G')} c(\epsilon)$, which is an approximation of the GED.

A BP approach with faster computation was introduced by Serratosa [14]. This method has a complexity of $\mathcal{O}(\max(m, n)^3)$ and gives the same result as in [12]. A new cost function and a new cost matrix C^* are defined. The only limitations are that the sum of the cost for removal and inserting stars is larger than substitution. The trick is to redefine the cost function

$$\text{EditCost}'(G, G') = \text{EditCost}(G, G') - \text{CostInsertDeleteStar} \tag{3.7}$$

where CostInsertDeleteStar is the sum of costs of inserting stars in G' and deleting stars in G. An $n \times m$ cost matrix C^* is defined to solve EditCost$'$:

$$C^* = \begin{bmatrix} c_{11} - (c_{1\varepsilon} + c_{\varepsilon 1}) & c_{12} - (c_{1\varepsilon} + c_{\varepsilon 2}) & \cdots & c_{1m} - (c_{1\varepsilon} + c_{\varepsilon m}) \\ \vdots & \vdots & & \vdots \\ c_{n1} - (c_{n\varepsilon} + c_{\varepsilon 1}) & c_{n2} - (c_{n\varepsilon} + c_{\varepsilon 2}) & \cdots & c_{nm} - (c_{n\varepsilon} + c_{\varepsilon m}) \end{bmatrix}.$$

The true EditCost is then computed as the cost of the approximation of the optimal path found by Munkre's algorithm added with CostInsertDeleteStar.

3.3.2 Prototype Selectors

As previously stated, a big challenge in using graph embedding using dissimilarities is to choose the number of prototypes and which prototypes to use. In this section, we consider the number of prototypes being chosen to be n, and present some

strategies of choosing these n prototypes. For the interested reader, a complete list of prototype selection strategies is presented in [5].

In order to easier specify these strategies, let us first define some special graphs.

Definition 3.4 (Set Median Graph)**.** *Let $\mathcal{G}$ be a set of graphs. The set median graph G_{mdn} is then the graph whose sum of graph edit distances to all graphs in $\mathcal{G}$ is minimal, and is defined by*

$$median(\mathcal{G}) = G_{mdn} = \underset{G \in \mathcal{G}}{\operatorname{argmin}} \sum_{G' \in \mathcal{G}} d(G, G').$$

Definition 3.5 (Set Centre Graph)**.** *Let $\mathcal{G}$ be a set of graphs. The set centre graph G_c is then the graph for which the maximum distance to all other graphs in $\mathcal{G}$ is minimal, and is defined by*

$$centre(\mathcal{G}) = G_c = \underset{G \in \mathcal{G}}{\operatorname{argmin}} \max_{G' \in \mathcal{G}} d(G, G').$$

Both these special graphs are located in the centre of the graph set. The difference is that the set centre graph can be seen as the geometrical centre of the graph set, while the set median graph also takes the densities in the graph set into account.

Targetsphere Prototype Selector (tps) [5] The main idea of the targetsphere prototype selection strategy is to distribute the prototypes as uniformly as possible in terms of the distance from the set centre graph. It is done by conceptualising $n - 1$ equidistant spheres around the set centre graph, and choosing a prototype closest to each sphere.

First the set centre graph $G_c = centre(\mathcal{G})$ is determined and selected as prototype. After G_c is found, the graph furthest away from G_c is located, i.e. the graph $G_f \in \mathcal{G}$ with the maximal distance to G_c, and selected as prototype. Denote the distance between G_c and G_f as $d_{\max}$, i.e. $d_{\max} = d(G_c, G_f)$, and let each $r \in \{iw\}_{i=1}^{n-2}$ be the radius of a sphere around G_c, where $w = \frac{d_{\max}}{n-1}$ is the distance between each sphere. Then choose the graph closest to each sphere to be a prototype, that is

$$p = \underset{G \in \mathcal{G}}{\operatorname{argmin}} \, | \, d(G_c, G) - iw \, | \qquad \text{for } i \in \{1, ..., n-2\}.$$

For a pseudo-code description see Algorithm 1.

The targetsphere prototype selector hence distributes the prototypes uniformly in terms of the distance from the set centre graph, but it does not take into account the distance between each prototype. Depending on the graph set, this prototype selector might distribute the prototypes well, but it might also only choose the prototypes in between the set centre graph and the graph furthest away, leaving suitable prototype graphs in other directions in the graph space unchosen.

Algorithm 1 Targetsphere prototype selector (tps)

Input: set of N graphs $\mathcal{G}$
Output: set of n prototypes $\mathcal{P} \subseteq \mathcal{G}$

Initialise $\mathcal{P}$ as $\emptyset$
Let $G_c = centre(\mathcal{G})$
Let $G_f = \underset{G \in \mathcal{G}}{\operatorname{argmax}}\, d(G, G_c)$
Let $d_{\max} = d(G_c, G_f)$, and set $w = \dfrac{d_{\max}}{n-1}$
$\mathcal{P} = \mathcal{P} \cup \{G_c, G_f\}$
$\mathcal{G} = \mathcal{G} \setminus \{G_c, G_f\}$
for each $i \in [1, n-2]$ **do**
    $p = \underset{G \in \mathcal{G}}{\operatorname{argmin}} \mid d(G, G_c) - iw \mid$
    $\mathcal{P} = \mathcal{P} \cup \{p\}$
    $\mathcal{G} = \mathcal{G} \setminus \{p\}$
end for

Spanning Prototype Selector (sps) The main idea of the spanning prototype selector is to find prototypes uniformly distributed over the graph set. It starts by finding the set median graph $G_{mdn} = median(\mathcal{G})$, and adds it to the set of prototypes $\mathcal{P}$. Then, until n prototypes are found, the next prototype is the graph furthest away from the graphs already in $\mathcal{P}$, i.e.

$$p = \underset{G \in \mathcal{G}}{\operatorname{argmax}} \min_{p' \in \mathcal{P}} d(G, p').$$

For a pseudo-code description see Algorithm 2.

In contrast to the tps, the sps considers all distances to the already selected prototypes in the selection, where the tps only considers the distances to the set centre graph. The purpose and result of the spanning selection strategy is hence to span the space of the graph set.

Algorithm 2 Spanning prototype selector (sps)

Input: set of N graphs $\mathcal{G}$
Output: set of n prototypes $\mathcal{P} \subseteq \mathcal{G}$

Initialise $\mathcal{P}$ as $\emptyset$
Let $G_{mgn} = median(\mathcal{G})$
$\mathcal{P} = \mathcal{P} \cup \{G_{mdn}\}$
$\mathcal{G} = \mathcal{G} \setminus \{G_{mdn}\}$
while $|\mathcal{P}| < n$ **do**
    $p = \underset{G \in \mathcal{G}}{\operatorname{argmax}} \min_{p \in \mathcal{P}} d(G, p)$
    $\mathcal{P} = \mathcal{P} \cup \{p\}$
    $\mathcal{G} = \mathcal{G} \setminus \{p\}$
end while

3.3.3 Choosing the Number of Prototypes

The graph embedding method in (3.5) depends on the choice of number of prototypes and the prototypes selected. This means that the number of prototypes needs to be set a priori. This kind of parameter is usually referred to as meta-parameter.

Choosing the proper number of prototypes when using unsupervised learning differs from when using supervised learning. Bunke and Riesen [15] perform experiments using supervised learning. In these experiments the true classes of the data are known, and one can then simply choose the number of prototypes as the number giving the lowest classification error or the highest accuracy.

In unsupervised learning, one can use clustering *validation indices* (Section 3.6) instead of the classification error to decide the best choice of number of prototypes. Experiments using clustering algorithms are presented in [16], where for each validation index considered the number of prototypes giving the best result is chosen. The experiments have been performed on a fixed number k of clusters, as this value is known in the data sets they have used. Generally, this value is unknown when using unsupervised learning, which will be explained in Section 3.5.

The number of clusters k in a set of RBS configuration topologies will vary from operator to operator and we can hence not use a fixed value of k when choosing the number of prototypes. The number of prototypes can be chosen after performing experiments over an interval of the number of clusters, where the number of prototypes is chosen as the one having the best average validation index score over the interval.

3.4 Weisfeiler-Lehman Kernels

In this section, we present a subtree kernel based on the 1-dimensional Weisfeiler-Lehman (WL) test of graph isomorphism. This subtree kernel was introduced by Shervashidze et al. [3] and uses this test of isomorphism as a similarity measure, as opposed to GEUD which instead is a measure of the cost of making graphs isomorphic. The idea is that graphs are most similar if they have many matching vertex labels, which in turns of this test implies that their subtrees match. This is an interesting approach, since it ought to be able to capture important differences of connections in, and topologies of, RBS configurations.

This subtree kernel ought to be more suitable for this domain than a graphlet kernel or a shortest path kernel. Even if graphlet kernels are good at capturing subpatterns in unlabelled graphs, it would not be feasible to apply one on the labelled graphs considered in this thesis, as they are not efficient for labelled graphs [2]. Also, a shortest path kernel jeopardises to disregard cycles which might occur in the cascading of radio equipment.

The WL test of isomorphism proceeds in iterations and tests whether two given graphs G and G' are isomorphic. During each iteration, indexed by i, the following steps are performed:

Algorithm 3 One iteration of the 1-dimensional Weisfeiler-Lehman test of graph isomorphism

Input: graphs $G = (V, E, \ell)$, $G' = (V', E', \ell)$ and number of iterations h
Output: label functions l_0, ..., l_h
Comments: $M_i(v)$ are multiset-labels, i.e. multisets of labels for a vertex v, Σ^* is a set of strings made up by letters in Σ

1. Multiset-label determination
   - for each $v \in V \cup V'$ do
     - if $i = 0$, let $M_i(v) = \{l_0(v)\} = \{\ell(v)\}$
     - if $i > 0$, let $M_i(v) = \{l_{i-1}(u) \mid u \in \mathcal{N}(v)\}$
2. Sorting each multiset
   - for each multiset-label $M_i(v)$, sort the elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$
   - if $i > 0$, for each string $s_i(v)$, add $l_{i-1}(v)$ as prefix to $s_i(v)$
3. Label compression
   - sort the strings $s_i(v)$ for all $v \in V \cup V'$, in ascending order
   - map each string $s_i(v)$ to a new compressed label, using a function $f : \Sigma^* \to \Sigma$ such that $f(s_i(v)) = f(s_i(v'))$ iff $s_i(v) = s_i(v')$
4. Relabeling
   - for each $v \in V \cup V'$, set $l_i(v) = f(s_i(v))$

The key idea of Algorithm 3 is to enhance the vertex labels by the sorted set of vertex labels of neighbouring vertices, and compress these enhanced labels into new, short labels [3]. The algorithm terminates after iteration $i < h$ if the newly created labels are not identical in G and G', i.e. $\{l_i(v) \mid v \in V\} \neq \{l_i(v') \mid v' \in V'\}$, which means that the two graphs are not isomorphic. If the sets are identical after h iterations it means either that G and G' are isomorphic, or that the algorithm could not determine that they are not isomorphic.

Letting for now n denote the maximal number of vertices in a G or G', and m the maximal number of edges in G or G', i.e. $n = \max(|V|, |V'|)$ and $m = \max(|E|, |E'|)$, the time complexity of running Algorithm 3 in h iterations would be $\mathcal{O}(hm)$ as each step in the algorithm can take $\mathcal{O}(m)$ time. A complexity analysis can be found in [3].

From each iteration i of the WL algorithm (Algorithm 3) we obtain a new labeling function $l_i(v)$ for each vertex v. Recall that vertices in G and G' will get identical new labels if, and only if, they have identical multiset labels. We can therefore think of one iteration of the WL relabeling as a function $r\Big((V, E, l_i)\Big) = (V, E, l_{i+1})$ which depends on the considered set of graphs and transforms all graphs in the set similarly [3]. Let us extend this thought in terms of graphs.

Definition 3.6. *For a given graph $G = (V, E, \ell)$, the i^{th} iteration Weisfeiler-Lehman rewriting is denoted $G_i = (V, E, l_i)$, where $G_0 = G$ and $l_0 = \ell$, and is computed using Algorithm 3.*

We note that G_0 is the original graph and that $G_i = r(G_{i-1})$ is the i^{th} relabeling.

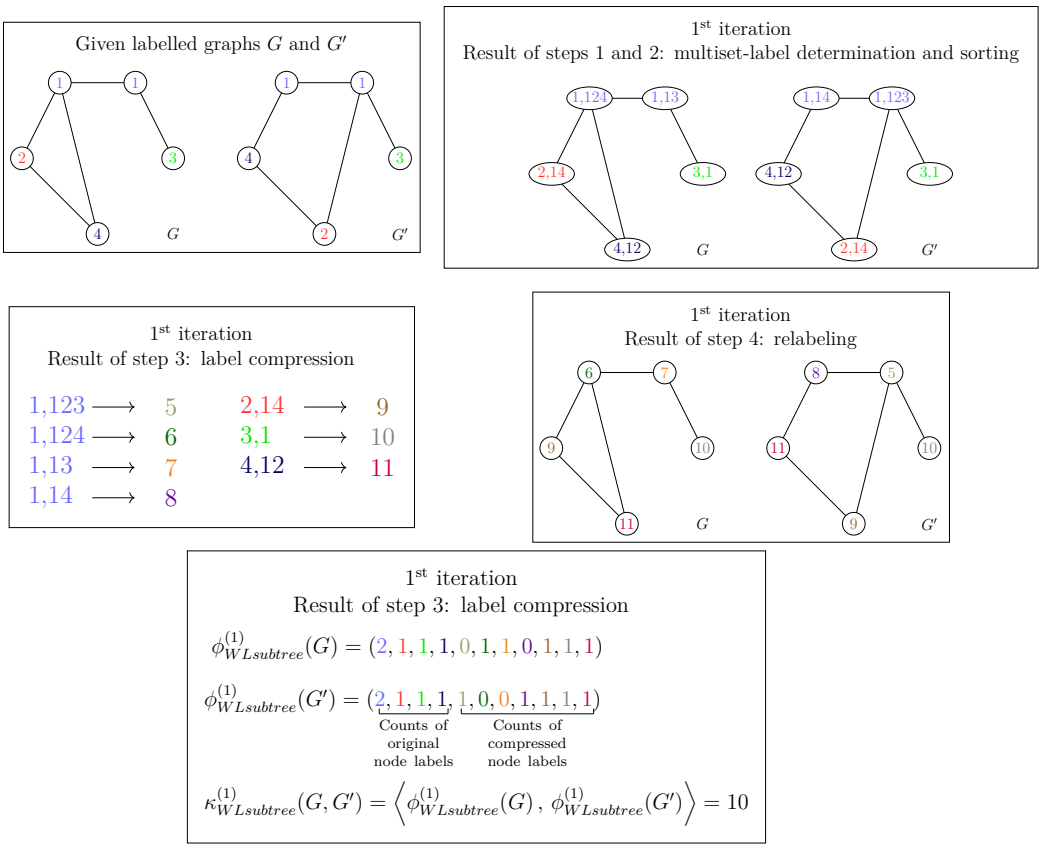


Figure 3.3: Illustration of the computation of the Weisfeiler-Lehman subtree kernel with $h = 1$ for two graphs G and G'. Here $1, 2, ..., 11 \in \Sigma$ are considered labels.

Based on the WL algorithm, we consider the general Weisfeiler-Lehman kernel as Definition 3.7.

Definition 3.7. *Let κ be any kernel for graphs, which we refer to as the base kernel. The Weisfeiler-Lehman kernel with h iterations and base kernel κ is then defined as*

$$\kappa^{(h)}_{WL}(G, G') = \sum_{i=0}^{h} \kappa(G_i, G'_i). \tag{3.8}$$

3.4.1 The Weisfeiler-Lehman Subtree Kernel

There exist various numbers of Weisfeiler-Lehman kernels [3]. In this section we present the WL subtree kernel. In order to define this kernel we first need to introduce some notations. Consider the two graphs G and G' and let $\Sigma_i \subseteq \Sigma$ be the set of labels which occur at least once in G or G' at the end of iteration i of the WL algorithm. Let Σ_0 be the set of original node labels of G and G'. Assume that all Σ_i are disjoint, and w.l.o.g. that every $\Sigma_i = \{\sigma_{i1}, \sigma_{i2}, ..., \sigma_{i|\Sigma_i|}\}$ is ordered. Also define a map $\#_i : \Sigma_i \times \{G, G'\} \to \mathbb{N}$ computing the number of occurrences of a letter $\sigma_{ij} \in \Sigma_i$ in graph G or G'.

Definition 3.8. *The Weisfeiler-Lehman subtree kernel on two graphs G and G'*

with h iterations is defined as

$$\kappa^{(h)}_{WLsubtree}(G, G') = \left\langle \phi^{(h)}_{WLsubtree}(G)\,,\, \phi^{(h)}_{WLsubtree}(G') \right\rangle, \tag{3.9}$$

where

$$\phi^{(h)}_{WLsubtree}(G) = \Big(\#_0(\sigma_{01}, G), ..., \#_0(\sigma_{0|\Sigma_0|}, G), ..., \#_h(\sigma_{h1}, G), ..., \#_h(\sigma_{h|\Sigma_h|}, G)\Big)$$

We note that the WL subtree kernel counts common labels in the two graphs.

The WL subtree kernel in (3.9) is indeed a special case of the general WL kernel in (3.8).

Theorem 1 (Shervashidze et al. [3])**.** *Let V and V' be the sets of vertices in graphs G and G', respectively. Furthermore, let κ_δ defined as*

$$\kappa_\delta(G, G') = \sum_{v \in V} \sum_{v' \in V'} \delta\Big(\ell(v), \ell(v')\Big),$$

be the base kernel, where δ is the Dirac kernel, which tests for equality, that is, it is 1 if its arguments are equal, and 0 otherwise. Then for all graphs G and G' $\kappa^{(h)}_{WL}(G, G') = \kappa^{(h)}_{WLsubtree}(G, G')$.

Proof. We note that the base kernel κ_δ is a function counting pairs with matching vertex labels. Hence, we easily notice that

$$\begin{aligned}
\kappa^{(h)}_{WL}(G, G') &= \sum_{i=0}^{h} \kappa_\delta(G, G') \\
&= \sum_{i=0}^{h} \left(\sum_{v \in V} \sum_{v' \in V'} \delta\Big(l_i(v), l_i(v')\Big) \right) \\
&= \sum_{i=0}^{h} \left(\sum_{j=1}^{|\Sigma_i|} \#_i(\sigma_{ij}, G) \cdot \#_j(\sigma_{ij}, G') \right) \\
&= \left\langle \phi^{(h)}_{WLsubtree}(G)\,,\, \phi^{(h)}_{WLsubtree}(G') \right\rangle \\
&= \kappa^{(h)}_{WLsubtree}(G, G').
\end{aligned}$$

□

3.4.2 The Weisfeiler-Lehman Subtree Kernel on *N* Graphs

Up to now, we have only considered computing the Weisfeiler-Lehman subtree kernel on pairs of graphs. In most cases, there is a set of N graphs which we want to compare. An obvious and naïve way of computing the WL subtree kernel on these N graphs is to compute the kernel on every pair of graphs. Shervashidze et al. [3] propose a faster algorithm to simultaneously process all N graphs. This algorithm is presented in Algorithm 4.

Algorithm 4 The Weisfeiler-Lehman subtree kernel computation on N graphs
Input: a set $\mathcal{G}$ of N graphs

- for $i = 0$ to h do
  - for every graph $G \in \mathcal{G}$ simultaneously do
    1. Multiset-label determination
       - for each $v \in V(G)$, let $M_i(v) = \{l_{i-1}(u) \mid u \in \mathcal{N}(v)\}$
    2. Sorting each multiset
       - for each multiset-label $M_i(v)$, sort the elements in $M_i(v)$ in ascending order and concatenate them into a string $s_i(v)$
       - for each string $s_i(v)$, add $l_{i-1}(v)$ as a prefix to $s_i(v)$
    3. Label compression
       - map each string $s_i(v)$ to a compressed label using a hash function $f : \Sigma^* \to \Sigma$ such that $f(s_i(v)) = f(s_i(v'))$ iff $s_i(v) = s_i(v')$
    4. Relabeling
       - for each $v \in V(G)$, set $l_i(v) = f(s_i(v))$

The result of running Algorithm 4 is an $N \times N$ kernel matrix K such that $K_{ij} = \kappa^{(h)}_{WLsubtree}(G^i, G^j)$ for each $G^i, G^j \in \mathcal{G}$.

Let for now n denote the maximal number of vertices in a graph in $\mathcal{G}$, and m the maximal number of edges in a graph in $\mathcal{G}$, i.e. $n = \max_{G \in \mathcal{G}} |V(G)|$ and $m = \max_{G \in \mathcal{G}} |E(G)|$, then running the naïve implementation in h iterations would result in $\mathcal{O}(N^2hm)$ time, but this algorithm only consumes $\mathcal{O}(Nhm + N^2hn)$ time, where the first term dominates the overall runtime in practice [3]. This can be accomplished by explicitly computing $\phi^{(h)}_{WLsubtree}$ for each $G \in \mathcal{G}$ and only then taking the pairwise scalar product. For a more profound time complexity analysis, we refer to [3].

3.5 Clustering Methods

Clustering algorithms aim to partition a data set into *clusters* so that all elements which fall into the same cluster are considered similar. This means that it is equally important that any two elements which are dissimilar are assigned to two different clusters. What makes two elements similar or dissimilar may vary depending on the application. Therefore, it is important that the data are represented so that patterns which are relevant in the application are captured.

In more technical terms, a clustering algorithm takes a data set $\mathcal{X}$ as input and outputs a set of clusters $\boldsymbol{C}$. In this work, we only allow *disjoint* clusters that cover the whole data set. In other words, $C_m \cap C_n = \emptyset$ for any pair of clusters $C_m, C_n \in \boldsymbol{C}$, where $m \neq n$, and $\bigcup_{C \in \boldsymbol{C}} C = \mathcal{X}$.

There exist numerous clustering algorithms, such as density-based clustering [17] and hierarchical clustering [18]. In this work, we will use centroid-based clustering, which opposed to density-based clustering and hierarchical clustering, requires that the number of clusters k to be predefined. We will use k-means for feature vectors and kernel k-means for kernels. Both algorithms partition $\mathcal{X}$ into

k clusters. A method for choosing the value of k which best represents the true number of clusters in $\mathcal{X}$ is presented in Section 3.5.4.

3.5.1 *k*-Means Clustering

The k-means clustering method is an unsupervised learning method which aims to divide an input set of patterns into k clusters so that the within-cluster sum of squared distances to the cluster's centroid is minimised.

Given a set of input data $\mathcal{X}$, where each element in $\mathcal{X}$ is a d-dimensional feature vector, the objective is to find a partitioning of k clusters $\boldsymbol{C} = \{C_1, .., C_k\}$ which minimises this error. Formally,

$$\boldsymbol{C} = \underset{\boldsymbol{C}}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{\boldsymbol{x} \in C_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2, \tag{3.10}$$

where $\boldsymbol{\mu}_i$ is the centroid of cluster C_i and is computed as

$$\boldsymbol{\mu}_i = \frac{1}{|C_i|} \sum_{\boldsymbol{x} \in C_i} \boldsymbol{x}. \tag{3.11}$$

Finding the optimal solution to the objective function is NP-hard even for $k = 2$ [19]. A widely used algorithm to find an approximate solution to k-means is Lloyd's algorithm [20], which has polynomial runtime. This method first initialises the cluster centroids randomly, then proceeds to assign elements in the data set to their closest centroid and then updates the centroids with respect to its cluster's elements. This is repeated until there is no change in the cluster assignments. This means that the algorithm is sensitive to the centroid initialisation. Often the algorithm is executed a number of times and then chooses the cluster assignment $\boldsymbol{C}$ with the smallest cluster error. To speed up the convergence, an extension of this algorithm uses a randomised seeding technique for the centroid initialisation, which is called k-means++ [21].

3.5.2 Kernel *k*-Means Clustering

When comparing entities in the input space $\mathcal{X}$ using a kernel (see Section 3.2.2), instead of comparing them as feature vectors, the clustering is performed slightly differently. Similar to the ordinary k-means one wants to find a partitioning which minimises the clustering error:

$$\boldsymbol{C} = \underset{\boldsymbol{C}}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \in C_i} \|\phi(x) - \mu_i\|^2, \tag{3.12}$$

where $\phi : \mathcal{X} \to \mathcal{H}$ is some implicit mapping from the input space to some feature space.

To utilise the so-called *kernel trick* we want to express the Euclidean distance using scalar products. Let N_i be the number of elements which have been assigned to cluster C_i, then we may write the cluster centroid as

$$\mu_i = \frac{1}{N_i} \sum_{y \in C_i} \phi(y). \tag{3.13}$$

This gives the expression

$$\|\phi(x) - \mu_i\|^2 = \Big\langle \phi(x)\,,\, \phi(x) \Big\rangle - \frac{2}{N_i} \sum_{y \in C_i} \Big\langle \phi(x)\,,\, \phi(y) \Big\rangle + \frac{1}{N_i^2} \sum_{y,z \in C_i} \Big\langle \phi(y)\,,\, \phi(z) \Big\rangle. \tag{3.14}$$

As explained in Section 3.2.2 a kernel function κ can be seen as a scalar product, and we can hence compute this expression in terms of κ as

$$\|\phi(x) - \mu_i\|^2 = \kappa(x,x) - \frac{2}{N_i} \sum_{y \in C_i} \kappa(x,y) + \frac{1}{N_i^2} \sum_{y,z \in C_i} \kappa(y,z). \tag{3.15}$$

The objective function can thus be written as

$$\underset{\boldsymbol{C}}{\operatorname{argmin}} \sum_{i=1}^{k} \sum_{x \in C_i} \left(\kappa(x,x) - \frac{2}{N_i} \sum_{y \in C_i} \kappa(x,y) + \frac{1}{N_i^2} \sum_{y,z \in C_i} \kappa(y,z) \right). \tag{3.16}$$

An algorithm for kernel k-means is described in Dhillon et al. [22].

3.5.3 Global (Kernel) k-Means Clustering

Most algorithms for solving the k-means and kernel k-means problems suffer from the dependence of the final solution on the initial partitioning. To deal with the initialisation problem for k-means the global k-means was proposed in [23], and for kernel k-means the global kernel k-means was proposed in [24]. These are incremental-deterministic algorithms which employ the (kernel) k-means as a local search procedure. In terms of clustering error, these algorithms obtain near-optimal solutions, and hence avoid getting trapped in poor local minima.

The procedure and idea behind these algorithms are principally the same. In the following lines the global kernel k-means is presented, but for the k-means consider instead the centroids of the input clusters and let $\phi(\boldsymbol{x}) = \boldsymbol{x}$. Assume one wants to solve the M-clustering problem, i.e. divide a data set $\mathcal{X}$ into M clusters $C_1, C_2, ..., C_M$. The global kernel k-means then starts by solving the 1-clustering problem, i.e. putting all data points into the same cluster. The algorithm then proceeds solving the 2-clustering problem by executing a kernel k-means algorithm N times, once for each element in the data set. During the n^{th} execution, the initial clusters considered are $\{x_n\}$ and $\mathcal{X} \setminus \{x_n\}$. The solution chosen for the 2-clustering problem is then the clustering obtained from the execution with the lowest clustering error E_2^n, given by

$$E_k^n = \sum_{i=1}^{k} \sum_{x \in C_i} \|\phi(x) - \mu_i\|^2. \tag{3.17}$$

Generally, for the k-clustering problem, let $(C_1^*, C_2^*, ..., C_{k-1}^*)$ denote the solution from the $(k-1)$-clustering problem and assume $x_n \in C_i^*$. The kernel k-means algorithm is then executed N times with $(C_1^*, ..., C_i = C_i^* \setminus \{x_n\}, ..., C_{k-1}^*, C_k = \{x_n\})$ as initial clusters for the n^{th} execution. The resulting clustering corresponding to the lowest error $E_k^* = \min_n E_k^n$ is then chosen as solution for the k-clustering problem. The described procedure is then repeated until $k = M$.

3.5.4 Choosing k

As previously mentioned, a drawback of k-means and kernel k-means is that k needs to be predefined. There are a few methods of choosing k. In this section, we will explain the method of choice.

The Elbow Method

The Elbow Method [25] can be used to determine the best choice of the number of clusters k in a data set, by visualising the curve of the clustering error E_k to the number of clusters k. Given a partition $\boldsymbol{C} = \{C_1, \ldots, C_k\}$, the clustering error is computed as

$$E_k = \sum_{i=1}^{k} \sum_{x \in C_i} \|\boldsymbol{x} - \boldsymbol{\mu}_i\|^2, \tag{3.18}$$

where $\boldsymbol{\mu}_i$ is the cluster centroid of cluster C_i. Given a maximum number of clusters $k_{\max}$, the Elbow method computes the clustering error for each $k \in \{1, 2, ..., k_{\max}\}$. By plotting E_k against k, the best choice of k will be where the plotted curve stops dropping most significantly, i.e. where it has the smallest upper angle. The method has been given its name since k will resemble an elbow in the plot. However, this elbow might be difficult to determine if the drop is not evident.

When a kernel method has been used, the kernel trick is applied to compute the error. That is, the clustering error is determined in the implicit feature space, and the method is otherwise applied as described above.

3.6 Clustering Evaluation

Clustering evaluation can be divided into two categories: intrinsic and extrinsic metrics. The former measures the compactness of the clusters and how well they are separated from each other. This means that intrinsic measures do not need to take "true" categories of a given set into account, and as a consequence of this we get no measure of how well the clustering algorithm divides the input set into the correct categories. Extrinsic metrics, on the other hand, compare clusters from a clustering algorithm with a set of categories. This set of categories, often referred to as a gold standard, is usually set by human experts in the domain.

In the remainder of this section we present a couple of different intrinsic and extrinsic metrics, and explain what they measure.

3.6.1 Intrinsic Metrics

For all these validation indices, we assume an input data set $\mathcal{X}$ and a distance measure $d(x, x')$ between two elements $x, x' \in \mathcal{X}$ such that $d(x, x) = 0$ and $d(x, x') \geq 0$ for every $x, x' \in \mathcal{X}$.

Dunn Index This index takes into account the distances between clusters and the diameter of clusters, and was first introduced by Dunn [26]. Assume that $\mathcal{X}$

have been partitioned into k distinct clusters, $\boldsymbol{C} = \{C_1, \ldots, C_k\}$, where $\bigcup_{i=1}^{k} C_i = \mathcal{X}$. The Dunn Index is then calculated as

$$\mathrm{DU}(k, \boldsymbol{C}) = \frac{\min\limits_{1 \leqslant i < j \leqslant k} \mathrm{dist}(C_i, C_j)}{\max\limits_{1 \leqslant i \leqslant k} \mathrm{diam}(C_i)}, \tag{3.19}$$

where $\mathrm{dist}(C_i, C_j) = \min_{x \in C_i, y \in C_j} d(x, y)$ is the distance between clusters C_i and C_j, and $\mathrm{diam}(C_i) = \max_{x, y \in C_i} d(x, y)$ is the diameter of cluster C_i. The idea is that a compact and well-separated (CWS) cluster should have a smaller diameter within the cluster than the distance to other clusters. The Dunn index can take values between 0 and infinity and a value $\mathrm{DU}(k, \boldsymbol{C}) > 1$ indicates that the partitions are CWS.

The Dunn index is hence a measure of how distinct and obvious the clusters are. By measuring the maximal diameter of the clusters and the smallest distance between two clusters, this metric gives an indication of the data are clearly separable. This metric is beyond doubt good when dealing with data that can be perceivably clustered, but its score is decreased when having outliers in a cluster.

Silhouette Score This score is computed as the average Silhouette coefficient of all entities in a data set. It measures how well the elements have been assigned to their rightful cluster. The Silhouette coefficient was first introduced by Rousseeuw [27]. Assume that we have divided $\mathcal{X}$ into a set of clusters. Assume that $x \in \mathcal{X}$ have been assigned to a cluster C, then the Silhouette coefficient of x is computed as

$$s(x) = \frac{b(x) - a(x)}{\max\Big(a(x), b(x)\Big)}, \tag{3.20}$$

where $a(x)$ is the average distance from x to all other entities in C and $b(x)$ is the lowest average of distances from x to all elements in a different cluster than C. We can say that $a(x)$ is a measurement of how dissimilar x is to all other elements in C, and $b(x)$ how dissimilar x is to the second-best cluster of choice C', which is the closest cluster to x, to which x has not been assigned. Thus, if the Silhouette coefficient is close to 1, x has been assigned to the correct cluster, and if the Silhouette coefficient is close to -1, x should probably been assigned to C'. The Silhouette score is then the average Silhouette coefficient of every $x \in \mathcal{X}$, where a result of 1 indicate well-clustered entities, 0 overlapping clusters and -1 poor-clustered entities.

Compared to the Dunn index this metric does not consider the clusters directly, but instead the assignment of their elements. It is hence a measurement of the certainty of the assignment of the data to their correct clusters. Outliers do not necessarily affect the score, as this metric mainly focuses on data points at the borders of the clusters close to other clusters.

3.6.2 Extrinsic Metrics

As extrinsic metrics we have chosen to use Purity, Inverse Purity and the BCubed [28] variant of van Rijsbergen's F measure [29]. These metrics measure how the

set of clusters $\boldsymbol{C} = \{C_1, C_2, \ldots, C_k\}$ computed after clustering conforms with the gold standard set $\boldsymbol{L} = \{L_1, L_2, \ldots, L_m\}$ with manually set categories, of the input data $\mathcal{X}$ with N elements. For a more comprehensive review of extrinsic metrics we recommend Amigó et al. [30] for further reading.

Amigó et al. [30] present four formal constraints on evaluation metrics, which cast light on which aspects of the quality of a clustering are captured by different metrics. The first constraint checks for cluster homogeneity, i.e. if all clusters contain only data points which are members of a single category. The second checks for cluster completeness, which instead means that members of the same category must be assigned to the same cluster. Third is a constraint called "Rag Bag", which puts more penalty on having noise in a clean cluster than grouping additional noise to an already disordered cluster. Lastly, the forth constraint, called "Cluster size vs. quantity", prefers a small error in a large cluster to a large number of small errors in small clusters.

Purity This method focuses on the most common category of each cluster, and in what degree it intersects with the cluster. Purity is computed by taking the weighted average of the maximal intersection of some category and each cluster:

$$\text{Purity} = \frac{1}{N} \sum_i \max_j |C_i \cap L_j| \tag{3.21}$$

This method does not reward grouping elements from the same category together, but penalises the noise in a cluster. This means that having more and smaller clusters with elements only from one category is more profitable than having large clusters with little noise. In this way we can obtain maximal Purity by having one cluster per element. By results of experiments presented in [30] we note that this metric only fulfils the first constraint of cluster homogeneity.

This measurement hence indicates in what degree each cluster contains configurations sharing the same category, i.e. how pure the clusters are. It means that this metric is an indicator of how well the applied method (embedding/kernel), and clustering method, is able to separate the data according to the aspects of the considered domain.

Inverse Purity This measurement is computed just as Purity but with the difference that clusters and categories have swapped places in the computations:

$$\text{Inverse Purity} = \frac{1}{N} \sum_j \max_i |C_i \cap L_j|. \tag{3.22}$$

In contrary to Purity, Inverse Purity rewards grouping elements together, but it does not penalise mixing elements from different categories. Hence, to simply use one single cluster for all elements yields the maximal value of Inverse Purity. By results of experiments presented in [30] we note that this metric only fulfils the fourth constraint.

As this method is the inverse of Purity, it indicates in what degree the elements sharing the same category have been assigned to the same cluster, and not split into several clusters. Hence, this metric is an indicator of how well the applied

method is able to locate the correct similarities with respect to the aspects of the considered domain.

BCubed F Measure Unlike Purity and similar metrics, which independently compute the quality of each cluster and category, BCubed [28] metrics decompose the evaluation process of estimating the *precision* and *recall* associated to each element of the input set. The precision associated to an element represents how many elements from its cluster belong to its category, and the item recall represents how many elements in the same category appear in its cluster.

Formally, if $L(x)$ and $C(x)$ denote the category and cluster, respectively, of an element $x \in \mathcal{X}$, we can define the correctness of the relation between two elements $x, x' \in \mathcal{X}$ as

$$\text{Correctness}(x, x') = \begin{cases} 1 & \text{iff } L(x) = L(x') \wedge C(x) = C(x') \\ 0 & \text{otherwise} \end{cases}.$$

That is, two elements are correctly related if they share category and appear in the same cluster. The BCubed precision of an item is then the ratio of items which share the same category in the item's cluster. The BCubed precision of the entire data set is then the average precision of all items in the set, and defined as

$$\text{BCubed Precision} = \frac{1}{N} \sum_{x \in \mathcal{X}} \frac{1}{|\mathcal{X}_{C(x)}|} \sum_{x' \in \mathcal{X}_{C(x)}} \text{Correctness}(x, x'),$$

where $\mathcal{X}_{C(x)}$ denotes the set $\left\{ x' \in \mathcal{X} \,\middle|\, C(x') = C(x) \right\}$. Analogously, the BCubed recall of an item is the ratio of items which appear in the same cluster of those items which share the same category. The BCubed recall for the entire data set is hence defined as

$$\text{BCubed Recall} = \frac{1}{N} \sum_{x \in \mathcal{X}} \frac{1}{|\mathcal{X}_{L(x)}|} \sum_{x' \in \mathcal{X}_{L(x)}} \text{Correctness}(x, x'),$$

where $\mathcal{X}_{L(x)}$ denotes the set $\left\{ x' \in \mathcal{X} \,\middle|\, L(x') = L(x) \right\}$.

According to results of experiments presented in [30] the BCubed precision fulfils constraints 1 and 3, and the BCubed recall fulfils constraints 2 and 4, so an intuitive idea is to combine both these evaluation metrics into a single metric fulfilling all four constraints. Using a standard way of combining metrics, called Van Rijsbergen's F measure, this can be done. This F measure is generally computed as

$$F(R, P) = \frac{1}{\alpha \left(\frac{1}{P}\right) + (1 - \alpha) \left(\frac{1}{R}\right)},$$

where R and P are two evaluation metrics and α is a relative weight of the metrics. In this thesis we consider R being the BCubed recall and P being the BCubed precision, and have $\alpha = 0.5$ meaning the harmonic mean of R and P. Thus, we consider

$$F = \frac{2 \cdot \text{BCubed Precision} \cdot \text{BCubed Recall}}{\text{BCubed Precision} + \text{BCubed Recall}}.$$

4
Experimental Setup

In this chapter we will present the data set we have been provided, how it has been categorised to a gold standard, as well as how we have implemented and performed experiments. The outline of this chapter corresponds to the order we have implemented modules and performed experiments.

4.1 Data Description

For this work, we have been given a set of configuration topologies from one of Ericsson's customers. The data are provided by the customer and contain data from real RBS configurations in a radio access network.

In order to be able to apply the presented methods, these configurations are parsed through a database from which relevant attributes, such as radio equipment and its ports, are extracted. For each configuration, these attributes are represented as vertices having a label corresponding to which piece of equipment it represents. Cables from one piece of equipment to another, as well as the "ownership"-relation of a piece of equipment and its ports into which cables are put, are represented as edges. An illustration of this can be seen in Figure 4.1.

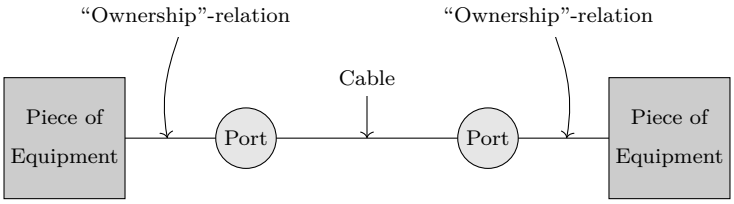


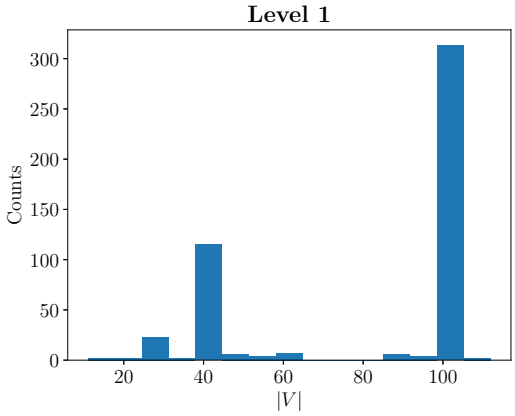
Figure 4.1: Illustration of what is represented as vertices and edges.

The data are processed on two levels of detail: the first level contains a shallow specification of hardware equipment along with its ports and relations, and the second level expands the first level by adding functionality such as cells into the model. Thus, the data representation has different sizes at the two levels.

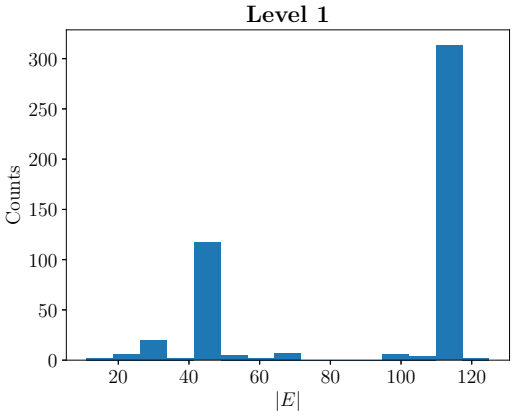
To provide some information of the data set, which consists of 486 RBS configurations, Table 4.1 shows some indication of the sizes of the configurations in the set at the different levels of detail, and Figure 4.2 visualises the distribution of these sizes.

Table 4.1: The maximal, minimal, average and median number of vertices and edges in the graph representations of the RBS configurations in the provided data set at the two considered levels of detail.

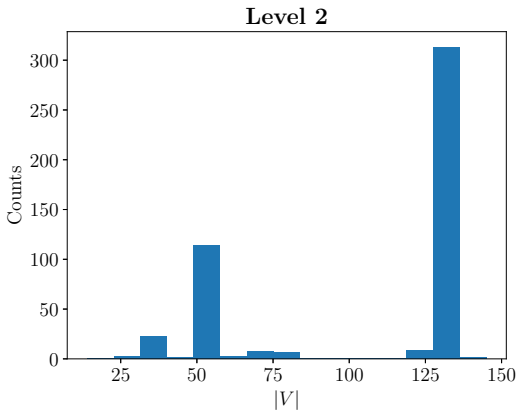
	Level 1				Level 2			
	Max	Min	Average	Median	Max	Min	Average	Median
Vertices:	112	11	79.20	99.0	145	14	102.55	129.0
Edges:	125	11	87.87	111.0	125	11	87.87	111.0



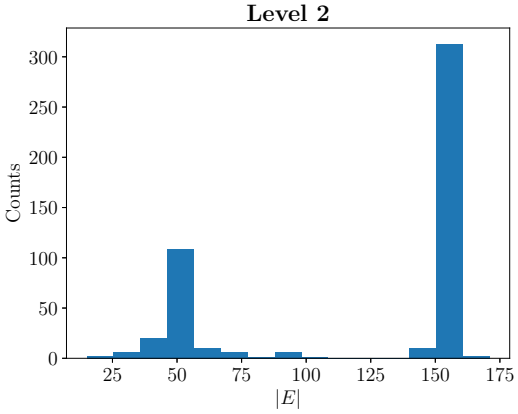
(a) Histogram of the number of vertices at level 1.



(b) Histogram of the number of edges at level 1.



(c) Histogram of the number of vertices at level 2.



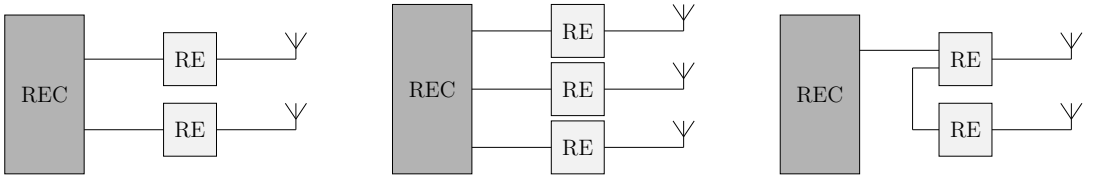
(d) Histogram of the number of edges at level 2.

Figure 4.2: Histograms of the number of vertices and edges in the configurations in the provided data set at the two considered levels of detail.

4.2 Manual Categorisation by Experts

In order to evaluate the clustering of the data set according to Ericsson aspects, the configurations have been visualised using the `matplotlib.pyplot` module in Python. The images obtained therefrom have then been given to two configuration specialists at Ericsson, who have categorised these configurations depending on

their topologies. As an example, the configuration shown in Figure 4.3a belongs to the same category as the configuration shown in Figure 4.3b, but not the same category as the configuration shown in Figure 4.3c. Even though the configurations in Figures 4.3a and 4.3c have the same pieces of equipment they do not have the same topology (due to the cascading), which the configurations in Figures 4.3a and 4.3b do, and are thus not considered similar.



(a) Two radios star-coupled with a REC.

(b) Three radios star-coupled with a REC.

(c) Two radios cascade-coupled from a REC.

Figure 4.3: Examples of simple RBS configurations visualising some differences of the considered categorisation. REC denotes radio equipment controller, RE denotes radio equipment and ᛉ denote antennas.

The occurrence of cascading, and the occurrence of MUXs, are two of four aspects taken into account at the first level of detail, giving a possible division of 16 categories. At the second level of detail, four additional aspects are considered, giving a theoretically possible division of 256 categories. Due to company secrets, we are not allowed to specify any other aspects. The provided data set was anyhow only sorted into four categories by the Ericsson specialists at the first level of detail, and five categories at the second level. At both levels there is one category containing only one RBS configuration, which can be seen in Table 4.2 along with the sizes of the other categories. Note that, if the data set would have been from more operators' networks, then the data would probably have been sorted into more of the 256 categories.

Table 4.2: The number of graphs assigned to each category on each level of detail.

(a) Level 1

Category	# Graphs
1	112
2	46
3	1
4	327

(b) Level 2

Category	# Graphs
1	112
2	34
3	12
4	1
5	327

The specialists concurred with this division as it is how they separate the configurations in their every-day work. Another division would hence be peculiar.

4.3 Data Representation

Once the RBS configuration topologies have been represented as graphs, we used the proposed methods to transform the set of graphs $\mathcal{G}$ into suitable data representations. The baseline methods and the graph embedding using dissimilarities method produce sets of feature vectors from $\mathcal{G}$, while the Weisfeiler-Lehman subtree kernel[1] produces a kernel matrix. For representations of and operations on arrays we have used the `numpy` package and for some algorithms we have used the `NetworkX` module for graph representation. We also measured the run time of computing these representations.

For the purpose of convenience we have given the produced sets the the names presented in Table 4.3.

Table 4.3: Names used for each graph representation produced by the methods used in this work at both levels of detail.

Method	Level 1	Level 2
Baseline	B_I	B_{II}
Extended Baseline	B_I^+	B_{II}^+
Graph embedding using dissimilarities (tps)	$GEUD_I^{tps}$	$GEUD_{II}^{tps}$
Graph embedding using dissimilarities (sps)	$GEUD_I^{sps}$	$GEUD_{II}^{sps}$
Weisfeiler-Lehman subtree kernel	WL_I	WL_{II}

The Weisfeiler-Lehman subtree kernel was computed for 1 iteration, as we believe that running another iteration of the test of graph isomorphism would not distinguish the configurations any further. This belief is based on that at an early stage we computed the kernel with 2 iterations, which yielded the same result as for 1 iteration, and hence only resulted in additional runtime. The baseline methods are straightforward, but in order to compute the graph embedding using dissimilarities we produce two sets of feature vectors at each level of detail, i.e. one for each prototype strategy. Recall that the number of prototypes needs to be set before computing the graph embedding, which will be explained in the section below.

4.3.1 Choosing the Number of Prototypes

The number of prototypes was chosen for both prototype selectors and levels of detail to create the representations $GEUD_I^{tps}$, $GEUD_{II}^{tps}$, $GEUD_I^{sps}$ and $GEUD_{II}^{sps}$. Hence, the procedure described below was repeated in order to create all these representations.

We decided to choose the number of prototypes from the list $n \in [5, 10, 20, 40, 80]$. The reason we decided to use these values was because how computationally demanding the prototype selectors are and that we wanted to explore different orders of magnitude. For each value of n we first generated $\mathcal{P}$ so that we could compute

[1]We have used an implementation of the Weisfeiler-Lehman subtree kernel available at GitHub. Link to the repository: `https://github.com/emanuele/jstsp2015/blob/master/gk_weisfeiler_lehman.py`

the graph embedding of all graphs in the data set. Once the representations of graph embeddings were computed, clustering was performed using k-means for $k \in \{2, 3, \ldots, 18\}$ and the Dunn index was computed for each clustering. The number of prototypes $n \in [5, 10, 20, 40, 80]$ which gave the best average Dunn index was chosen as representation for GEUD for each level of detail and prototype selector.

4.4 *k*-Means Usage

Once the sets were generated, we used them as input to the chosen clustering algorithms. We have used a k-means algorithm for the computed feature vectors and a kernel k-means algorithm for the computed Weisfeiler-Lehman subtree kernels.

For the k-means algorithm, we used the method `KMeans` available in the Python module `scikit-learn`. We have used the default arguments to this method, which means that the k-means++ algorithm has been executed. In addition, the method runs the algorithm 10 times and chooses the clustering which has given the smallest clustering error E_k.

To perform clustering using the Weisfeiler-Lehman subtree kernel we have implemented the global kernel k-means algorithm, as there were no available packages implementing this algorithm for arbitrary kernels. The reason of using the global kernel k-means algorithm instead of an "ordinary" kernel k-means algorithm in our case is that the rise of empty clusters is easier avoided. The algorithm was implemented as described in Section 3.5.3 with the adjustment of ignoring cluster initialisations yielding empty clusters.

4.5 Clustering Evaluation

We have used the Elbow method to investigate how well the methods are able to separate the data and what value of k was chosen when an elbow was evident. Since we have been provided a gold standard of the data set, we want to investigate whether or not the methods are able to choose the value of k corresponding to the number of categories at each level.

To select k the Elbow method was executed on all methods and at both levels of detail. Recall that this method is used to visually decide the best value of k, which means that we have generated plots of the clustering error against $k \in \{2, 3, \ldots, 18\}$ with a circle indicating which value of k that has been chosen, if evident.

We then proceeded to evaluate the clustering performance for each method. We used both intrinsic and extrinsic metrics, in order to evaluate how distinct the clusters are and measure how the clusters correspond to the categories.

Since we had a value of k which was indicated from the Elbow method, we used an interval surrounding this value and computed all intrinsic and extrinsic metrics in this interval. We wanted to use the same interval of k for all methods and at each level, so it would be easier to compare the methods. Hence, we evaluated the clustering performance at $k \in [k_{\min} - 1, k_{\max} + 1]$, where $k_{\min}$ and $k_{\max}$ are

the minimum and maximum values of k chosen using the Elbow method for all methods.

For the intrinsic metrics, we needed to compute the distance differently depending on the input has been feature vectors or kernels. The Euclidean distance was used for the vectors and the kernel trick was used for the kernels. All metrics, except the Silhouette score, were implemented by us. The Silhouette score was computed using the `silhouette_score` method in `scikit-learn`.

5
Results

In this chapter we present the results obtained from the different executions described in Chapter 4: first the executions of choosing the numbers of prototypes, then the executions of choosing the number k of clusters, and eventually the results of the clustering evaluation. Finally, the runtimes of generating the data representations are presented. The results are then discussed upon in Chapter 6.

5.1 Number of Prototypes for GEUD

The number of prototypes for both prototype selectors and at each level of detail was chosen considering the best average Dunn index score in Tables 5.1a–5.1d. Complete scores for each prototype selector and level with $k = 2, 3, ..., 18$ can be seen in Tables A.1–A.4.

Table 5.1: Average Dunn index scores for different numbers of prototypes using the targetsphere (tps) and spanning prototype selection (sps) strategies. The best performing numbers of prototypes, and corresponding scores, are marked in bold.

(a) Using tps at level 1

# Prototypes	Average Dunn Score
5	0.288254
10	**0.332445**
20	0.322803
40	0.322138
80	0.319137

(b) Using sps at level 1

# Prototypes	Average Dunn Score
5	0.349372
10	0.305997
20	0.315880
40	0.333604
80	**0.363447**

(c) Using tps at level 2

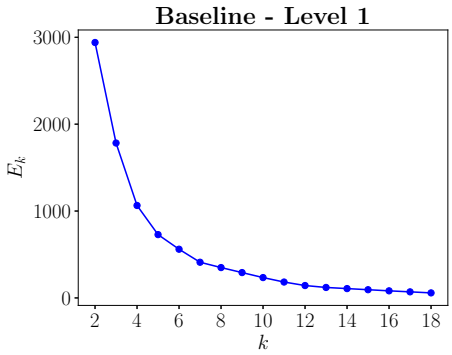
# Prototypes	Average Dunn Score
5	0.275140
10	0.295968
20	0.298037
40	**0.300540**
80	0.218416

(d) Using sps at level 2

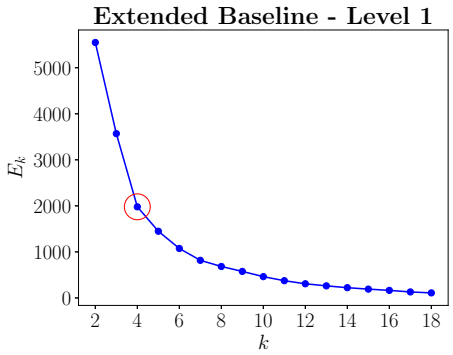
# Prototypes	Average Dunn Score
5	0.260252
10	0.278309
20	**0.346390**
40	0.319017
80	0.293630

5.2 The Elbow Method

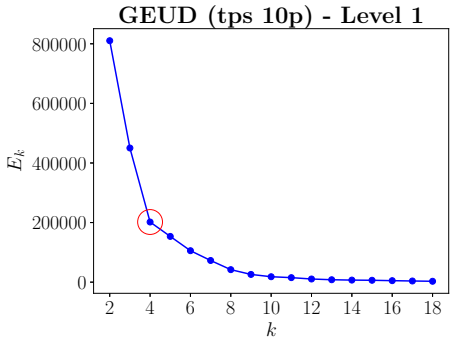
Figure 5.1 shows the elbow plots for all methods at level 1. In the baseline method in Figure 5.1a we cannot observe any elbow, since the curve of the clustering error drops smoothly as k increases. Hence, there is no marker in that plot indicating which value of k to choose. In Figures 5.1b–5.1e we observe elbows at $k = 4$, where the Weisfeiler-Lehman subtree kernel has the most significant elbow.



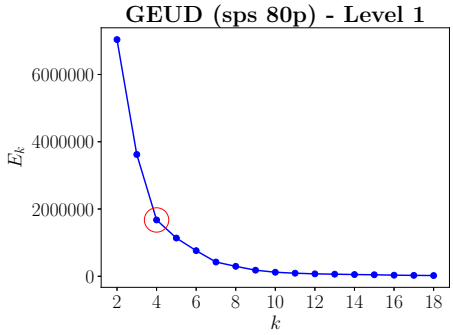
(a) Elbow plot of the simple baseline method.



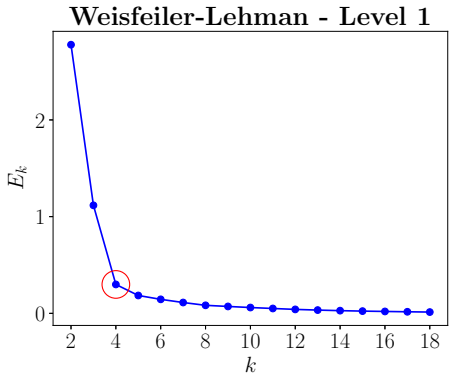
(b) Elbow plot of the extended baseline method.



(c) Elbow plot of the graph embedding using dissimilarities method with 10 prototypes chosen by tps.



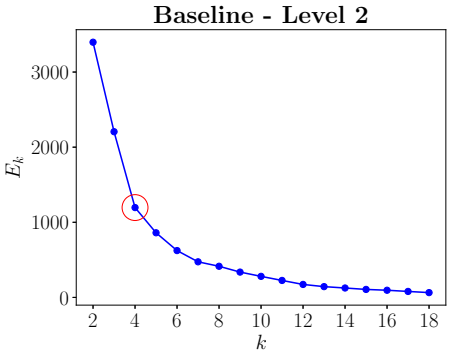
(d) Elbow plot of the graph embedding using dissimilarities method with 80 prototypes chosen by sps.



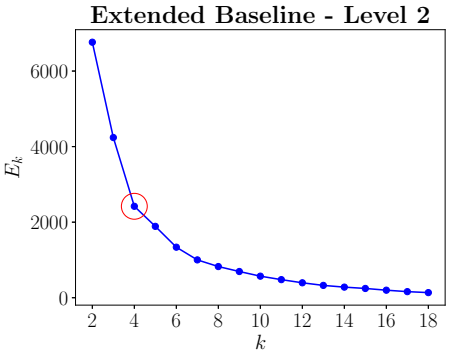
(e) Elbow plot of the Weisfeiler-Lehman method.

Figure 5.1: Elbow plots for the used methods at level 1. Red circles indicate the choice of k.

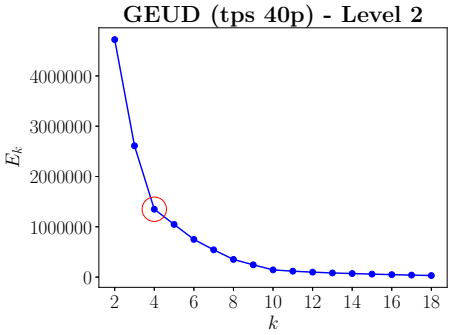
In the elbow plots for level 2, shown in Figure 5.2, we see that for each method the choice of k is also 4, even for the baseline method. As for level 1, the most significant elbow occurs in the elbow plot of the Weisfeiler-Lehman method.



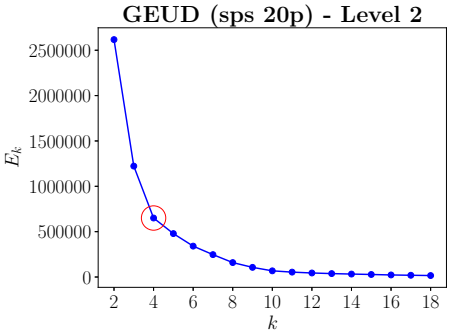
(a) Elbow plot of the simple baseline method.



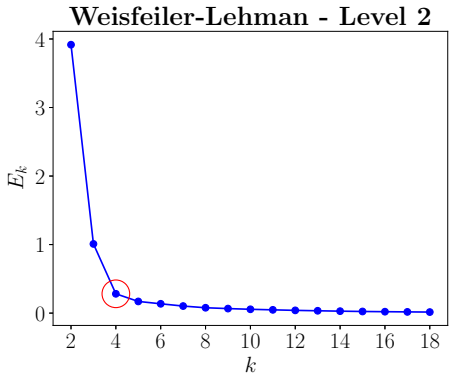
(b) Elbow plot of the extended baseline method.



(c) Elbow plot of the graph embedding using dissimilarities method with 40 prototypes chosen by tps.



(d) Elbow plot of the graph embedding using dissimilarities method with 20 prototypes chosen by sps.



(e) Elbow plot of the Weisfeiler-Lehman method.

Figure 5.2: Elbow plots for the used methods at level 2. Red circles indicate the choice of k.

5.3 Clustering Evaluation

In this section, we will present the results from the intrinsic and extrinsic metrics which we have used to evaluate the clustering performance of the proposed methods.

5.3.1 Dunn Index

Recall that the Dunn index measures how compact and well separated a set of clusters is, and can be seen as a relation between separation and compactness. Hence, the larger the Dunn index is the more distinct are the clusters.

Table 5.2: Dunn index scores for both levels of detail.

(a) Level 1

Method	$k = 3$	$k = 4$	$k = 5$
B_I	0.235702	0.105409	0.117851
B_I^+	0.219382	0.132686	0.169516
WL_I	**0.278746**	**0.177711**	**0.216469**
$GEUD_I^{sps}$	0.079020	0.132175	0.132175
$GEUD_I^{tps}$	0.127680	0.108372	0.108372

(b) Level 2

Method	$k = 3$	$k = 4$	$k = 5$
B_{II}	0.252439	**0.233762**	0.216295
B_{II}^+	0.099015	0.143101	0.152795
WL_{II}	**0.366401**	0.165873	**0.223819**
$GEUD_{II}^{sps}$	0.205938	0.072608	0.072608
$GEUD_{II}^{tps}$	0.171997	0.123505	0.123505

For Dunn index at level 1, we observe in Table 5.2a that the Weisfeiler-Lehman kernel outperforms all other methods for all considered values of k. The extended baseline has notably a better Dunn index than any of the GEUD methods, which are not consistently better than the baseline method.

At level 2, the Weisfeiler-Lehman kernel outperforms all methods for $k = 3$ and $k = 5$, but has the second highest score after the baseline method at $k = 4$. We observe that the baseline method outperforms both the GEUD methods and the extended baseline on all values of k. The extended baseline performs better than both GEUD methods at $k = 4$ and $k = 5$.

5.3.2 Silhouette Score

As described in Section 3.6.1 the Silhouette score is the average Silhouette coefficient in the data set and measures the certainty of the cluster assignment. If the Silhouette score is close to 1, this indicates that most elements have been assigned to the correct clusters. If it is close to 0, the majority of the elements are close

to a decision boundary, and if the score is closer to -1, this indicates that the majority of elements have been assigned to wrong clusters.

Table 5.3: Silhouette scores for both levels of detail.

(a) Level 1

Method	$k = 3$	$k = 4$	$k = 5$
B_I	0.887939	0.913413	0.913112
B_I^+	0.882862	0.908964	0.906685
WL_I	0.907507	**0.938726**	**0.952584**
$GEUD_I^{sps}$	**0.919010**	0.931832	0.928665
$GEUD_I^{tps}$	0.903168	0.926443	0.933783

(b) Level 2

Method	$k = 3$	$k = 4$	$k = 5$
B_{II}	0.880761	0.916542	0.914107
B_{II}^+	0.892784	0.905700	0.901169
WL_{II}	0.922258	**0.948196**	**0.956008**
$GEUD_{II}^{sps}$	**0.922748**	0.925078	0.928517
$GEUD_{II}^{tps}$	0.910148	0.925165	0.932733

The GEUD methods and the Weisfeiler-Lehman kernels have better scores than the baseline methods, and in general the Weisfeiler-Lehman kernel has the best score. It can also be observed that the baseline generally has a better score than the extended baseline. Overall, all methods perform well in terms of the Silhouette score, as most of their values are above 0.9.

5.3.3 Purity

Recall from Section 3.6.2, that this measurement measures how pure the clusters are, i.e. in what degree the clusters contain elements from only one category. A value of 1 thus indicates that all clusters only contain elements from one category.

Table 5.4: Purity measures for both levels of detail.

(a) Level 1

Method	$k = 3$	$k = 4$	$k = 5$
B_I	0.940329	0.973251	0.973251
B_I^+	0.940329	0.962963	0.962963
WL_I	**0.997942**	**0.997942**	**0.997942**
$GEUD_I^{sps}$	0.923868	0.965021	0.965021
$GEUD_I^{tps}$	0.930041	0.965021	0.965021

(b) Level 2

Method	$k = 3$	$k = 4$	$k = 5$
B_{II}	0.927984	0.960905	0.960905
B_{II}^+	0.921811	0.950617	0.950617
WL_{II}	**0.973251**	**0.995885**	**0.995885**
$GEUD_{II}^{sps}$	0.921811	0.923868	0.923868
$GEUD_{II}^{tps}$	0.921811	0.950617	0.923868

The Weisfeiler-Lehman subtree kernel has the highest Purity measure for all values of k at both levels of detail, where all values except one are over 0.99. It can also in Table 5.4 be observed that the baseline method performs better than both GEUD methods.

5.3.4 Inverse Purity

As described in Section 3.6.2 this measurment indicates in what degree elements of the same category are assigned to the same cluster. If the Inverse Purity is 1 it means that no elements of the same category are split into different clusters.

Table 5.5: Inverse Purity measures for both levels of detail.

(a) Level 1

Method	$k = 3$	$k = 4$	$k = 5$
B_I	0.942387	0.942387	0.934156
B_I^+	0.942387	0.952675	0.923868
WL_I	**1.000000**	**0.973251**	**0.952675**
$GEUD_I^{sps}$	0.971193	0.942387	0.921811
$GEUD_I^{tps}$	0.965021	0.942387	0.925926

(b) Level 2

Method	$k = 3$	$k = 4$	$k = 5$
B_{II}	0.942387	0.952675	0.934156
B_{II}^+	0.971193	0.942387	0.923868
WL_{II}	**1.000000**	**0.997942**	**0.977366**
$GEUD_{II}^{sps}$	0.965021	0.942387	0.942387
$GEUD_{II}^{tps}$	0.965021	0.942387	0.934156

As for the Purity measure, the Weisfeiler-Lehman subtree kernel has the best Inverse Purity measure of all observations. The GEUD methods have in general a lower score compared to one of the baseline methods.

5.3.5 BCubed F Measure

The BCubed F measure is, as described in Section 3.6.2, the harmonic mean of the BCubed precision and BCubed recall. Recall that the BCubed precision is the average ratio of items sharing the same category in the same cluster, and that BCubed recall is the average ratio of items appearing in the same cluster of those sharing the same category. A value of 1 thus indicates that the items are completely correctly clustered.

Table 5.6: BCubed F measures for both levels of detail.

(a) Level 1

Method	$k = 3$	$k = 4$	$k = 5$
B_I	0.914256	0.938125	0.919397
B_I^+	0.914256	0.925663	0.907104
WL_I	**0.997956**	**0.978471**	**0.957389**
$GEUD_I^{sps}$	0.917170	0.927439	0.906765
$GEUD_I^{tps}$	0.918685	0.927439	0.923072

(b) Level 2

Method	$k = 3$	$k = 4$	$k = 5$
B_{II}	0.904453	0.933646	0.915266
B_{II}^+	0.912523	0.918533	0.900354
WL_{II}	**0.979290**	**0.994060**	**0.973784**
$GEUD_{II}^{sps}$	0.910121	0.901432	0.901080
$GEUD_{II}^{tps}$	0.910121	0.919968	0.921299

The Weisfeiler-Lehman have the highest BCubed F measures for all values of k at both levels of detail. We can otherwise observe that the GEUD methods again shows lower values than the baseline methods in many of the observations.

5.4 Running Times

The computations presented in this thesis have been timed using the `time` module in Python giving the values presented in Table 5.7.

Table 5.7: Runtimes of computing the feature vectors and kernels for each method used. The values presented are rounded to two significant figures.

(a) Level 1

Method	Time (s)
Baseline	0.019
Extended Baseline	0.13
Weisfeiler-Lehman	280
GEUD-tps (10p)	1 800
GEUD-sps (80p)	13 000

(b) Level 2

Method	Time (s)
Baseline	0.026
Extended Baseline	0.26
Weisfeiler-Lehman	380
GEUD-tps (40p)	3 200
GEUD-sps (20p)	4 000

The two baseline methods are both relatively quick, and the time of computing the Weisfeiler-Lehman kernel matrix of the provided data set is also reasonable. Notable for the GEUD methods is that using the spanning prototype selector seems to be slower than using the targetsphere prototype selector, see Table 5.7b, where the tps selects 40 prototypes at shorter time than the sps selects 20 prototypes. Note that the size of the data set provided may be relatively small compared to other sets in this domain, which means that the runtime might have a significant impact of the performance when investigating larger sets of RBS configurations.

6
Discussion

In this chapter we discuss the results presented in Chapter 5 and try to argue why some methods perform better than others. In Section 6.3 we point out important findings and try to provide answers to our research questions.

6.1 The Elbow Method

The elbow plots in Figures 5.1–5.2 show that once an elbow could be observed, it was located at $k = 4$. This value corresponds to the number of categories in the gold standard at level 1, but not at level 2 where there are 5 categories. This means that increasing $k = 4$ to $k = 5$ does not significantly improve the clustering of the data at any of the levels of detail. Furthermore, this method does hence not seem to capture the additional aspects added in level 2, in this data set.

An evident elbow for B_I cannot be observed since the clustering error drops smoothly as k increases. At all other baseline methods an elbow is observed, however it is not very significant. This indicates that there might be no proper clustering in the data using this representation, meaning that either the data are poorly separable or that the data representation does not capture the important aspects of the data. Given the results of the validation indices, we consider the latter.

Both GEUD methods show significant elbows at $k = 4$. Where the GEUD using tps are a slightly more significant than GEUD using sps.

We finally observe that the Weisfeiler-Lehman subtree kernel has the most evident elbow at both levels of detail, which is a sign that the data are well clustered at $k = 4$. Compared to the other methods, we may say that the Weisfeiler-Lehman subtree kernels have the most distinct groups in its data, since the drop of the clustering error as k grows decelerates more abruptly compared to the other methods.

6.2 Clustering Evaluation

We will first present the intrinsic and extrinsic evaluations separately, and then summarise the performance of each method.

6.2.1 Intrinsic

Dunn Index

As previously mentioned, the Dunn index measures how compact and separated a set of clusters is. If a Dunn index score is above 1 the clusters are called compact and well-separated (CWS). This means that all clusters' diameters are smaller than the closest distance between any two clusters. As can be seen in the results in Table 5.2, no set of clusterings is CWS.

At both levels of detail we observe that the Weisfeiler-Lehman subtree kernel has the best results, with just one exception when this method is second best. This indicates that the Weisfeiler-Lehman subtree kernel in general is more compact and separated than all other methods at both levels. We also observe that the Dunn index for the Weisfeiler-Lehman subtree kernel are higher at level 2 compared to level 1, which indicates that the clusters are more distinct at the second level. This might depend on that adding the functionality into the model distinguishes the configurations further, and thus makes their clusters more distinct.

Additionally, we observe that the extended baseline has the second best value at level 1. However, the Dunn index of the extended baseline decreases at the second level and is consistently lower than the baseline. The reason for this could be that the extended number of dimensions causes noise which negatively affects the clustering. Hence, the extended baseline produces less distinct clusters at level 2 compared to level 1. The Dunn index of the baseline is actually higher at the second level.

The GEUD methods, on the other hand, had consistently lower scores than the extended baseline at level 1 and consistently lower scores than the baseline at level 2. Furthermore, we can see that GEUD using sps is generally better than GEUD using tps at level 1, while we observe the opposite at level 2. This is probably because the Dunn index for GEUD using sps generally decreases as we move to level 2, while the Dunn index increases for GEUD using tps.

In summary, we see that the Weisfeiler-Lehman subtree kernel, the extended baseline and GEUD using tps show better clusterings when using a more detailed data representation at level 2.

Silhouette Score

When looking at the Silhouette scores obtained in Table 5.3, we see that both the Weisfeiler-Lehman subtree kernel and the GEUD methods outperform the baseline methods. This means that these methods are better then the baselines when it comes to assigning the elements to their correct clusters. However, all Silhouette scores achieve a value above 0.88, which indicates that all methods perform fairly well.

The Weisfeiler-Lehman subtree kernel generally has the highest score, and we note that its Silhouette score increases from level 1 to level 2. Overall, we cannot see any significant increase or decrease of the Silhouette scores for the other methods when comparing level 1 to level 2.

As with the Dunn index, we see that GEUD using sps is generally better than GEUD using tps at level 1, and the opposite at level 2. However, both GEUD

methods achieve values close to each other at all observations.

We finally note that the extended baseline in majority of the time has the lowest score, which also could be because of the increase of dimensions, in relation to the baseline.

6.2.2 Extrinsic

Purity

Purity is a measurement of how pure the clusters are, which means in what degree each cluster contains elements sharing the same category. The Weisfeiler-Lehman method achieves the highest measure of all methods and for the actual number of categories (4 at level 1 and 5 at level 2) the method achieves measures over 0.99, which is impressively good.

We notice that the purity measure decreases for all methods from level 1 to level 2, which presumably depends on the increased number of categories taken into consideration. As there are more categories, there need to be more clusters in order for the clusters to be pure.

The GEUD methods achieve lower measures than the baseline method and also mostly than the extended baseline. This means that the extended baseline performs worse than the regular baseline. Also, we note that GEUD using sps performs worse than GEUD using tps, especially at level 2.

Overall, all methods perform well since they all achieve measures above 0.92. This means that they all manage to get somewhat pure clusters. Moreover, the Weisfeiler-Lehman method achieves above 0.99 on most observations, which is extraordinarily good.

Inverse Purity

Inverse Purity measures to which degree elements of the same category are assigned to the same cluster, meaning that a high Inverse Purity indicates that the majority of elements sharing the same category are found in the same cluster. All methods achieve measures above 0.92, which indicates that they all perform well.

Again, we see that the Weisfeiler-Lehman subtree kernel has the best values. This means that this method does not only produce fairly pure clusters, it also manages fairly well to not separate elements sharing the same category into different clusters. At $k = 3$ the Inverse Purity is 1, which means that no category has been split into different clusters. However, as k increases the score decreases since some elements of the same categories have been assigned to different clusters. A decrement is although expected as the number of clusters increases and exceeds the number of categories, since the elements of some category then need to be separated in order to avoid empty clusters. Additionally, we note that this method has a better score at level 2 compared to level 1, which can be explained by the increased number of categories. As the number of categories increases, the sizes of the categories decreases, which means that the elements sharing the same category are presumably not separated in the same extent.

The performance of the GEUD methods are comparable to the baseline methods.

BCubed F Measure

In this measure, a value of 1 indicates that the data set has been correctly clustered. Recall that it is the harmonic mean of two metrics measuring in what degree elements appearing in the same cluster are correctly related and in what degree elements sharing the same category are correctly related. Also, recall that two elements are correctly related if they share the same category and appear in the same cluster.

The Weisfeiler-Lehman subtree kernel has the best results for all values of k at both levels of detail and achieves a score above 0.95 at all observations. This suggests that the Weisfeiler Lehman subtree kernel yields a more correct clustering than the other methods.

For the other methods we can see that the GEUD methods have values similar to the baseline methods. Most values are above 0.90 which should be considered values of sufficiently high quality.

6.2.3 Summary

In these experiments, we could observe that the Weisfeiler-Lehman subtree kernel, with few exceptions, has the best results. The Dunn index scores indicate that the Weisfeiler-Lehman subtree kernel in general yields the most compact and well separated clusters of all methods, and at all values of k. Furthermore, the results of the Silhouette scores indicate a high certainty of the cluster assignments, since the score is above 0.9. In the extrinsic evaluations, we compared how the methods had clustered relative to the gold standard. The obtained clusters from the Weisfeiler-Lehman subtree kernel have generally the best Purity and Inverse Purity scores, as well as the additional BCubed F measures.

The GEUD methods do not show as much promise as the Weisfeiler-Lehman subtree kernel. These methods show generally low scores in the Dunn index compared to other methods, which indicates that they are generally less compact and separated than the other methods. The GEUD methods have higher Silhouette scores than the baseline methods, but have lower scores than the Weisfeiler-Lehman subtree kernel. In the extrinsic measurements, they show similar results as the baselines.

We believe that the GEUD methods could be improved either by more profound cost functions for graph edit distance computations, or by a more detailed graph representation. The costs could, in the former approach, be more adapted by putting more weight on substituting very dissimilar pieces of equipment and on deleting and inserting important types of equipment. The latter approach, with a more detailed representation, would force the GED algorithm to substitute vertex labels, rather than just edges, in order to capture differences in topologies.

In Figure 4.3, we can see that it would require more edit operations to transform the configuration in Figure 4.3a to the configuration in Figure 4.3b, since this would require insertions of both an RE and an antenna with corresponding edge insertions, while transforming the configuration in Figure 4.3a to the configuration in Figure 4.3c only will require one edge deletion and one edge insertion. Hence, GED will see the configuration in Figure 4.3a as more similar to the configuration

in Figure 4.3c than to the configuration in Figure 4.3b.

It is also important to take the running times into account. The GEUD methods need a much larger computation effort than the other methods which can be seen in Table 5.7. This is probably due to the costly computations of obtaining the prototype sets.

As for the baseline methods, we could observe that the extended baseline often has lower scores than the regular baseline. We think that this could be because of the additional dimensions cause noise. We also observe that none of the methods manage to correctly identify the configuration sharing category with no other configuration.

6.3 Conclusion

In this research we have explored two graph representation techniques found in literature, and implemented two simple baseline methods as additional graph representations. The purpose of the baseline methods was to compare the performances of the explored graph representations to the performances of these simpler methods.

We were also provided a data set that was categorised by Ericsson experts. This data set contained only around 5 categories of at least 256 estimated number of possible RBS configuration topologies.

We observed that the Weisfeiler-Lehman subtree kernel showed very good results. This is probably because this method compares RBS configurations by their topologies in subtrees and is thus able to capture differences of important topological attributes. Overall, this method shows excellent results for the extrinsic evaluations, which indicates that this method is able to capture the categories set by Ericsson experts, and is thus able to relatively well find correct clusters in a set of RBS configuration topologies. Another important aspect is that the Weisfeiler-Lehman subtree kernel is straightforward, in the sense that it does not require to be tailored to our application.

The GEUD methods are both significantly more computational demanding compared to the other methods and have often the worst results. Further investigations are needed in order to improve the GEUD methods. One of the strengths of this method is that the cost functions can be defined depending on the application, but this will require more domain knowledge in order to achieve desirable results. Even if an increase of the performance is achieved, this might not justify the runtime or additional work to tailor these methods. However, the GEUD methods could be relevant in other applications for RBS configuration topologies, where the level of distortions between RBS configurations are important.

We believe that our research motivates further investigation of clustering RBS configuration topologies. In further research different sets of RBS configuration topologies should be explored for further evaluation. In our setup the performance of the Weisfeiler-Lehman subtree kernel justifies the increased runtime compared to the baseline methods; this might not be the case for any larger sets of RBS configuration topologies. More importantly, further investigation on more diverse sets of RBS configuration topologies needs to be done to verify if the Weisfeiler-Lehman

subtree kernel can discriminate other types of RBS configuration topologies, which are not present in the provided data set. The clustering approach seems promising, especially when we only consider the equipment topologies.

Another approach, which we did not investigate in this thesis, is to define a set of substructures which captures important attributes of RBS configuration topologies. These substructures may then be extracted from the graphs to compute feature vectors where each dimension corresponds to one of these key substructures. An RBS configuration topology may hence be discriminated from another by these features. This however, would require some expert knowledge to define important substructures. On the other hand, this approach may be more efficient and accurate than the Weisfeiler-Lehman kernel.

We believe that future investigations of graph-based representation for clustering evaluation may result in a tool to discriminate RBS configuration topologies in RANs. Hence, Ericsson ought to be able to cover test cases which are present in live RANs.

Bibliography

[1] Thomas Gärtner, Peter Flach, and Stefan Wrobel. On graph kernels: Hardness results and efficient alternatives. In *Conference on Learning Theory*, pages 129–143, 2003.

[2] Nino Shervashidze, S. V. N. Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten M. Borgwardt. Efficient graphlet kernels for large graph comparison. In *International Conference on Artificial Intelligence and Statistics*, 2009.

[3] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, November 2011.

[4] Corinna Cortes and Vladimir Vapnik. Support-vector networks. *Mach. Learn.*, 20(3):273–297, September 1995.

[5] Kaspar Riesen and Horst Bunke. Graph classification based on vector space embedding. *IJPRAI*, 23(6):1053–1081, 2009.

[6] Geng Li, Murat Semerci, Bülent Yener, and Mohammed J. Zaki. Effective graph classification based on topological and label attributes. *Statistical Analysis and Data Mining*, 5(4):265–283, August 2012.

[7] Hisashi Kashima, Koji Tsuda, and Akihiro Inokuchi. Marginalized kernels between labeled graphs. In *Proceedings of the Twentieth International Conference on Machine Learning*, pages 321–328. AAAI Press, 2003.

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

[9] Jan Ramon and Thomas Gärtner. Expressivity versus efficiency of graph kernels. In *Proceedings of the First International Workshop on Mining Graphs, Trees and Sequences*, pages 65–74, 2003.

[10] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2 (1):25–36, 2009.

[11] N. J. Nilsson P. E. Hart and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems, Science,*

and Cybernetics, SSC-4(2):100–107, 1968.

[12] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7): 950–959, June 2009.

[13] J. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, March 1957.

[14] Francesc Serratosa. Fast computation of bipartite graph matching. *Pattern Recognition Letters*, 45:244–250, 2014.

[15] Horst Bunke and Kaspar Riesen. Recent advances in graph-based pattern recognition with applications in document analysis. *Pattern Recognition*, 44 (5):1057 – 1067, 2011.

[16] Kaspar Riesen. *Classification and Clustering of Vector Space Embedded Graphs*. PhD thesis, Universität Bern, September 2009.

[17] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of 2nd International Conference on Knowledge Discovery and Data Mining*, pages 226–231. AAAI Press, 1996.

[18] Ying Zhao, George Karypis, and Usama Fayyad. Hierarchical clustering algorithms for document datasets. *Data Min. Knowl. Discov.*, 10:141–168, March 2005.

[19] Daniel Aloise, Amit Deshpande, Pierre Hansen, and Preyas Popat. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning*, 75(2): 245–248, 2009.

[20] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theor.*, 28 (2):129–137, September 2006.

[21] David Arthur and Sergei Vassilvitskii. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics. ISBN 978-0-898716-24-5.

[22] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means: Spectral clustering and normalized cuts. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '04, pages 551–556. ACM, 2004.

[23] Aristidis Likas, Nikos Vlassis, and J.J. Verbeek. The global k-means clustering algorithm. *Pattern Recognition*, 36:451–461, 2001.

[24] Grigoris Tzortzis and Aristidis Likas. The global kernel k-means algorithm. In

International Joint Conference on Neural Networks, pages 1978–1985, 2008.

[25] Robert L. Thorndike. Who belongs in the family. *Psychometrika*, pages 267–276, 1953.

[26] J. C. Dunn. A Fuzzy Relative of the ISODATA Process and Its Use in Detecting Compact Well-Separated Clusters. *Journal of Cybernetics*, 3(3): 32–57, 1973.

[27] Peter Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *J. Comput. Appl. Math.*, 20(1):53–65, November 1987.

[28] Amit Bagga and Breck Baldwin. Algorithms for scoring coreference chains. In *In The First International Conference on Language Resources and Evaluation Workshop on Linguistics Coreference*, pages 563–566, 1998.

[29] CJ van Rijsbergen. *Information Retrieval. 1979.* Butterworth, 1979.

[30] Enrique Amigó, Julio Gonzalo, Javier Artiles, and Felisa Verdejo. A comparison of extrinsic clustering evaluation metrics based on formal constraints. *Information Retrieval*, 12:461–486, 2009.

A

Choosing the Number of Prototypes

A.1 Level 1

Table A.1: Dunn index scores for different numbers of prototypes using the targetsphere prototype selection strategy at level 1.

# Prototypes →	$n = 5$	$n = 10$	$n = 20$	$n = 40$	$n = 80$
$k = 2$	0.459015	0.405090	0.597625	0.898328	0.972987
$k = 3$	0.080099	0.127680	0.180233	0.078853	0.173276
$k = 4$	0.109243	0.108372	0.100812	0.089089	0.085196
$k = 5$	0.109243	0.108372	0.127721	0.183634	0.178020
$k = 6$	0.167866	0.165518	0.154307	0.129134	0.178020
$k = 7$	0.167866	0.181818	0.154307	0.158739	0.161033
$k = 8$	0.181818	0.181818	0.168509	0.141825	0.213371
$k = 9$	0.314805	0.322068	0.349048	0.317041	0.264573
$k = 10$	0.279389	0.206514	0.343163	0.355968	0.264573
$k = 11$	0.279389	0.471802	0.343163	0.329788	0.321579
$k = 12$	0.376654	0.397218	0.446159	0.359325	0.334495
$k = 13$	0.376654	0.391814	0.446159	0.377211	0.360045
$k = 14$	0.297761	0.428571	0.446159	0.364487	0.360045
$k = 15$	0.384655	0.397218	0.395293	0.410106	0.360045
$k = 16$	0.479934	0.482711	0.444399	0.377211	0.360045
$k = 17$	0.384655	0.656685	0.395293	0.410106	0.356583
$k = 18$	0.451278	0.618292	0.395293	0.495502	0.481449
Average	0.288254	0.332445	0.322803	0.322138	0.319137

Table A.2: Dunn index scores for different numbers of prototypes using the spanning prototype selection strategy at level 1.

# Prototypes →	$n = 5$	$n = 10$	$n = 20$	$n = 40$	$n = 80$
$k = 2$	0.550541	0.579094	0.566269	0.504935	0.478858
$k = 3$	0.069763	0.075794	0.078377	0.078819	0.079020
$k = 4$	0.101244	0.109931	0.125886	0.130966	0.132175
$k = 5$	0.111800	0.125215	0.128377	0.130966	0.132175
$k = 6$	0.147636	0.201673	0.159031	0.164746	0.167508
$k = 7$	0.182156	0.182107	0.179326	0.188002	0.179533
$k = 8$	0.222098	0.219806	0.172586	0.218241	0.249014
$k = 9$	0.324342	0.319561	0.329899	0.324610	0.322229
$k = 10$	0.396513	0.371764	0.357357	0.331665	0.285966
$k = 11$	0.396513	0.371764	0.420540	0.420432	0.410489
$k = 12$	0.396513	0.371764	0.357357	0.420432	0.410489
$k = 13$	0.396513	0.291158	0.371349	0.287407	0.221925
$k = 14$	0.407903	0.384983	0.357357	0.287407	0.595874
$k = 15$	0.582298	0.334126	0.403790	0.629883	0.316976
$k = 16$	0.710724	0.334126	0.193205	0.145209	0.699587
$k = 17$	0.710724	0.386978	0.488774	0.674331	0.711046
$k = 18$	0.232048	0.542099	0.680486	0.733214	0.785733
Average	0.349372	0.305997	0.315880	0.333604	0.363447

A.2 Level 2

Table A.3: Dunn index scores for different numbers of prototypes using the targetsphere prototype selection strategy at level 2.

# Prototypes →	$n = 5$	$n = 10$	$n = 20$	$n = 40$	$n = 80$
$k = 2$	0.532962	0.462485	0.642384	1.042531	1.066754
$k = 3$	0.173739	0.169024	0.168917	0.171997	0.168619
$k = 4$	0.069605	0.069605	0.063115	0.123505	0.079654
$k = 5$	0.069605	0.112773	0.098025	0.123505	0.079654
$k = 6$	0.109442	0.112773	0.098025	0.078831	0.050650
$k = 7$	0.157755	0.162287	0.141299	0.114012	0.073349
$k = 8$	0.167063	0.174402	0.154093	0.123896	0.079141
$k = 9$	0.276041	0.278806	0.264097	0.177985	0.158849
$k = 10$	0.276041	0.278806	0.264097	0.364923	0.171874
$k = 11$	0.318128	0.327438	0.329410	0.393229	0.172671
$k = 12$	0.247115	0.280656	0.351614	0.294296	0.162156
$k = 13$	0.247115	0.280656	0.326281	0.294296	0.183765
$k = 14$	0.310363	0.344305	0.326281	0.270820	0.254659
$k = 15$	0.362987	0.413960	0.351614	0.294296	0.162156
$k = 16$	0.413255	0.480384	0.454406	0.392732	0.254659
$k = 17$	0.473079	0.562540	0.516484	0.424163	0.363191
$k = 18$	0.473079	0.520558	0.516484	0.424163	0.231265
Average	0.275140	0.295968	0.298037	0.300540	0.218416

Table A.4: Dunn index scores for different numbers of prototypes using the spanning prototype selection strategy at level 2.

# Prototypes →	$n = 5$	$n = 10$	$n = 20$	$n = 40$	$n = 80$
$k = 2$	0.592168	0.617127	0.709322	0.582077	0.533460
$k = 3$	0.183813	0.199510	0.205938	0.190063	0.180039
$k = 4$	0.074305	0.074160	0.072608	0.077803	0.077864
$k = 5$	0.074305	0.074160	0.072608	0.083273	0.083304
$k = 6$	0.124197	0.111807	0.123828	0.161822	0.083304
$k = 7$	0.168014	0.155510	0.171207	0.173469	0.168546
$k = 8$	0.168014	0.155510	0.175877	0.173469	0.207596
$k = 9$	0.320892	0.290942	0.336915	0.322250	0.305825
$k = 10$	0.320892	0.307023	0.339241	0.322250	0.311000
$k = 11$	0.218176	0.259672	0.367218	0.348398	0.269587
$k = 12$	0.218176	0.269074	0.375986	0.348398	0.319613
$k = 13$	0.231789	0.269074	0.375986	0.348398	0.319613
$k = 14$	0.262759	0.338024	0.375986	0.394302	0.348349
$k = 15$	0.313957	0.387324	0.546476	0.489149	0.448117
$k = 16$	0.313957	0.387324	0.546476	0.489149	0.463515
$k = 17$	0.356631	0.387324	0.546476	0.443923	0.428625
$k = 18$	0.482243	0.447681	0.546476	0.475093	0.443354
Average	0.260252	0.278309	0.346390	0.319017	0.293630