Technical University of Munich
Department of Mathematics

Matching Gear Unit Representations

An Integer Optimization Framework

Master's Thesis
by
Sarah Gräßle

SUPERVISOR: *Prof. Dr. Stefan Weltge*
ADVISORS: *Dr. Michael Ritter, Dr. Moritz Keuthen*
SUBMISSION DATE: *27. November 2023*

I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

Munich, 27. November 2023

S. Gräßle

Sarah Gräßle

Zusammenfassung

Ziel dieser Masterarbeit ist es, zwei Darstellungen von Getriebedaten zu vergleichen. Um diese beiden Darstellungen zu vergleichen, wird ein ganzzahliges Optimierungsproblem aufgestellt. Um dieses Problem zu lösen, werden wir einen Algorithmus implementieren. Der vorgeschlagene Ansatz und die Implementierung funktionieren sehr gut und der Algorithmus liefert die erwarteten Ergebnisse. Außerdem werden wir das Optimierungsproblem mit dem Graphenisomorphie Problem in Verbindung setzen und seine Komplexität analysieren.

Abstract

This master's thesis aims to match two representations of gearbox data. To match these two representations we will set up an integer optimization problem. To solve this problem we will implement an algorithm. The proposed approach and implementation works really well and the algorithm yields the expected results. We will also connect the optimization problem to the graph isomorphism problem and analyze its complexity.

Contents

1 Introduction

This master's thesis was developed in cooperation with the company FVA Software & Service. This company was founded to promote the practical application of findings from the research projects of the German Research Institute for Drive Technology (Forschungsvereinigung Antriebstechnik e.V., FVA) [7]. FVA pools resources and research within the drive technology industry [21]. It is a large network within the industry that connects people, shares knowledge and helps to implement ideas [21]. FVA Software & Service works closely with leading German research institutes and companies from the drive technology industry to provide practical applications of the findings from FVA research projects as well as training and consulting [8].

In the drive technology industry exist many different pieces of software for transmission systems. Even though they are very different, the same data of gearboxes is often used. For simpler exchange of these data, the FVA initiated the development of an industry-wide standard. This project was realized by FVA Software & Service in cooperation with leading research institutes and drive technology companies. This standardized interface is called REXS. Even though it is standardized, there are some aspects that may differ between two REXS files of the same transmission. This is due to the very different pieces of software. Hence, one gearbox may have different representations. The objective of this thesis is to develop an algorithm that matches two representations of gearbox data. The goal is to integrate this tool into the website of FVA Service & Software and make it available to their customers.

What does matching two representations of gearbox data exactly mean? Since this is a thesis in mathematics, we are not going into detail about gearboxes themselves. For the purpose of this thesis it is enough to know that a gear box consists of many different elements. For example, an element could be a gear, a shaft or a bearing. An example of a gearbox can be seen in figure 1.1. Matching two gearbox representations means we want to match the individual elements of the first gearbox to elements of the second, e.g we want to match a gear to a gear and a shaft to a shaft. Assume that we want to match a gear to another element. We know that a gear can only be matched to a gear of the second model. How do we decide which specific gear to assign to it? During the matching process we consider the structure of the gearbox as well, i.e. we want to consider how the elements relate to each other. This means we also look at adjacent elements. For example, in order to match a gear to another one, we also want to match the shafts the gears are located on to each other. Another thing to consider for the assignment of elements is how similar they are. What exactly the similarity of two elements comprises will be determined later in this thesis.

In order to solve this assignment problem we will model it as a linear integer optimization problem in Section 3. An integer optimization problem has the form:

$$\begin{aligned} \max \quad & c^T x \\ \text{subject to} \quad & Ax \leq b \\ & x \geq 0 \\ & x \in \mathbb{Z}^n \end{aligned}$$

with $A \in \mathbb{R}^{m \times n}$, $b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix} \in \mathbb{R}$, and $c = \begin{pmatrix} c_1 \\ \vdots \\ c_n \end{pmatrix} \in \mathbb{R}$. The optimization variables are $x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}$. (cf. [5, 20])

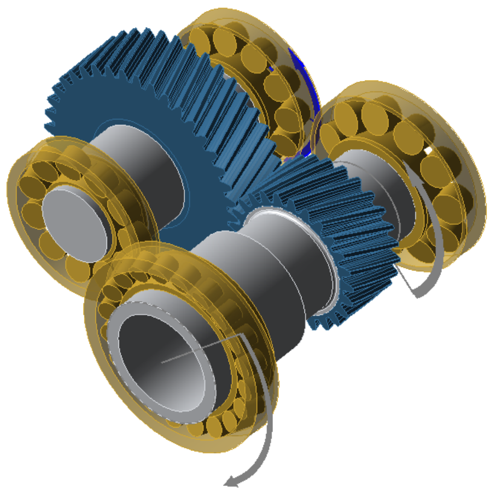
Figure 1.1 Example of a gearbox [16]

After setting up the integer optimization problem we implement and solve it with an optimization software.

The next chapter gives an introduction to the format that the gearbox data is stored in, namely REXS. Afterwards, we model the problem as an integer optimization problem. Next, the implementation of the algorithm is being addressed. Within that chapter, an overview of the used optimization software is given, the data structure in which the input data is stored is discussed, the development of the implementation and improvements of the optimization model are explained, the run time of the algorithm and possible performance improvements are analyzed, and lastly the output of the algorithm is explained. The chapter afterwards summarizes the final integer optimization problem after the adjustments were made in the previous chapter. In Chapter 6 the problem is related to the graph isomorphism problem and its complexity is analyzed. Before a conclusion is given in Chapter 8, the result of the algorithm is discussed on a few examples.

2 REXS

A way to save gearbox data is the *Reusable Engineering Exchange Standard (REXS)*. REXS is a standardized interface that was established for an industry-wide easy exchange of such data. It was introduced to make the usage of the same data across different tools and applications easier (cf. [14]). Because of this standardized aspect we are working with REXS in this thesis.

Industry 4.0 is becoming increasingly important, but the drive technology industry is very heterogeneous. Above all, the software used in this sector is very diverse, since they all fulfill different tasks. Nevertheless, the same gearbox data is often used. To simplify the exchange between companies of this kind of data, the German Research Institute for Drive Technology (the Forschungsvereinigung Antriebstechnik e.V., FVA) initiated the development of an industry-wide standard. Thus, the development of REXS started in the year 2017 by FVA Software & Service. Research institutes like the Institute of Machine Elements and Machine Design (IMM) of the Technical University of Dresden, the Laboratory for Machine Tools and Production Engineering (WZL) of the RWTH Aachen University and the Gear Research Center (FZG) of the Technical University of Munich, and leading drive technology companies like SEW-Eurodrive, Schaeffler AG and ZF were involved in the development process. The goal of REXS was to design a format for the exchange of gearbox data between different Computer Aided Engineering (CAE) tools. It helps to reduce both the mistakes that happen during data exchange and the maintenance efforts in communication between different tools. REXS is an open source software. It defines a uniform nomenclature of the gearbox and its components. This is based on the detailed terminology developed by 25 project related committees of the FVA. (cf. [9, 10, 11, 12])

A REXS file can either be given in JSON or XML format. In this thesis we only work with the JSON format. An instance of a REXS file includes components and relations between these components. Apart from relations and components the file also includes information on which REXS version is used, which application generated the file and the date of creation of the REXS file. An example of a REXS file with the general structure can be seen here:

```json
{
"model": {
    "version" : "1.5",
    "applicationId" : "FVA Workbench",
    "applicationVersion" : "8.1.2",
    "date" : "2023-11-09T16:28:37+01:00",
    "relations": [...],
    "components": [...]
}}
```

Listing 2.1 Example of a REXS JSON file

A component represents an element of a gearbox and is structured as follows: it has an ID, a type and a list of attributes that describe the properties of this gearbox element. An attribute could be the number of teeth of a gear, for example. It may also have a non-mandatory name. This name is irrelevant for our problem though. The ID is an integer that uniquely identifies the component within this file. The type specifies what type of component it is, e.g. a shaft. There are 84 different kinds of component types defined (cf. [13]). The optional name is given by the user. An attribute also has an ID, a unit and a value. The attribute ID is a string that uniquely identifies it. The unit specifies the unit (preferably in the SI system) in which the value is given (cf. [6]). An example of a component is given here:

```
{
"model": {
    "version" : "1.5",
    "applicationId" : "FVA Workbench",
    "applicationVersion" : "8.1.2",
    "date" : "2023-11-09T16:28:37+01:00",
    "relations": [...],
    "components": [{
        "id" : 1,
                "type" : "gear_unit",
        "name" : "Getriebeeinheit [1]",
                "attributes":[{
            "id" : "reference_component_for_position",
            "unit" : "none",
            "reference_component" : 1
        }, {
            "id" : "gravitational_acceleration",
            "unit" : "m / s^2",
            "floating_point" : 9.81
        }]
    }]
}}
```

Listing 2.2 Example of a component

The relations are represented by references between components. Formally, a relation has a unique ID, a type and a list of references. There are many different types of relations. Some relations are ordered. Then they also have an order. An order is an integer that for example clearly identifies the individual bearing rows of a rolling bearing to specify in which order they are to be assembled (cf. [6]). The list of references is a list of the components that are connected via this relation. A reference is made up of the component ID and the role of the component within the relation. A reference may also include a hint that gives more information about the reference. This information is not relevant to us. A relation can connect two or three components (cf. [6]). This restriction comes from the specifications of REXS files (cf. [6]). For an example of a relation refer to listing 2.3.

```
{
"model": {
    "version" : "1.5",
    "applicationId" : "FVA Workbench",
    "applicationVersion" : "8.1.2",
    "date" : "2023-11-09T16:28:37+01:00",
    "relations": [{
        "id" : 10,
        "type" : "assembly",
        "refs" : [ {
            "id" : 42,
            "role" : "assembly",
            "hint" : "Shaft [42]"
        }, {
            "id" : 45,
            "role" : "part",
            "hint" : "Cylindrical gear [45]"
        }]
    }],
    "components": [...]
}}
```

Listing 2.3 Example of a relation

Let's define these objects formally. A REXS data model consists of a finite set of components C and a finite set of relations $\mathcal{R}$. An element $c \in C$ has the form $c = (id, type, attributes)$ where attributes is a list of attributes. An *attribute* has the form $attribute = (id, unit, value)$. It may also have an optional origin. The origin provides the possibility to mark an attribute value either as user defined or calculated (cf. [6]). It indicates where this value comes from. The origin is not mandatory and therefore not relevant for us. An element $r \in \mathcal{R}$ has the form $r = (id, type, order, \{refs\})$ where *refs* is a list of components: $refs = (c_1, c_2, c_3)$. If the relation has no order we define the order to be 0. For simplicity, from now on we shorten the notation $r = (id, type, order, \{refs\})$ to $r = (id, type, order,$
$c_1, c_2, c_3)$ where c_1, c_2, c_3 are the components corresponding to the references of the relation. If a relation has only two references, the last slot for the components is filled with a dummy component: $c_\circ$. A relation with two references thus is $r = (id, type, order, c_1, c_2, c_\circ)$.

The goal of this thesis is to match two of these REXS data sets in order to determine if they are representations of the same gearbox. Components are only paired with components and relations with relations. Furthermore, we only want to match components and relations of the same type. Optimally we would find matching components and relations in the second data set for all elements of the first one. But there might be a different number of components and relations in the two data sets due to small changes or differences in exporting the data from different software tools. Therefore, we desire to match as many components and relations as possible. To achieve this we will be using integer optimization.

3 Initial Mathematical Model

To model the problem described above we create an integer program. This integer program aims to match components and relations to their counterparts. It is desired to match similar components and relations of the same type. We have two REXS data sets, $\mathcal{M} = (C, \mathcal{R})$ and $\mathcal{M}' = (C', \mathcal{R}')$. C and C' are finite sets of components and $\mathcal{R}$ and $\mathcal{R}'$ are finite sets of relations of the models. To indicate whether components and relations are matched, we introduce binary variables. The variable $x_{c,c'} \in \{0,1\}$ indicates if a component $c \in C$ of the first data set is matched to a component $c' \in C'$ of the second data set. Similarly, there is a variable $z_{r,r'} \in \{0,1\}$ that indicates if a relation $r \in \mathcal{R}$ of the first data set is matched to a relation $r' \in \mathcal{R}'$ of the second data set.

The objective function consists of reward and penalty terms. There are two reward terms, T_1 and T_2. T_1 rewards the matching of similar components, T_2 the matching of similar relations. The reward is given by functions $f : C \times C' \to \mathbb{R}$ and $g : \mathcal{R} \times \mathcal{R}' \to \mathbb{R}$. The similarity is given by the value of the function. The greater the similarities of two matched components or relations, the greater is the reward. A more precise definition follows later in chapter 4.3. The terms explicitly read:

$$T_1 = \sum_{c \in C} \sum_{c' \in C'} f(c, c') x_{c,c'}$$
$$T_2 = \sum_{r \in \mathcal{R}} \sum_{r' \in \mathcal{R}'} g(r, r') z_{r,r'}.$$

Regarding the penalty terms, there are two terms each for components and for relations. The terms T_3 and T_4 punish if components of model $\mathcal{M}$ and $\mathcal{M}'$ are not matched. Accordingly, the terms T_5 and T_6 give a punishment if relations are not matched. The amount of punishment is determined by the parameters $\gamma_c, \gamma_{c'}, \delta_r$ and $\delta_{r'}$. Explicitly, the penalty terms are:

$$T_3 = \sum_{c \in C} \gamma_c \Big(1 - \sum_{c' \in C'} x_{c,c'}\Big)$$
$$T_4 = \sum_{c' \in C'} \gamma_{c'} \Big(1 - \sum_{c \in C} x_{c,c'}\Big)$$
$$T_5 = \sum_{r \in \mathcal{R}} \delta_r \Big(1 - \sum_{r' \in \mathcal{R}'} z_{r,r'}\Big)$$
$$T_6 = \sum_{r' \in \mathcal{R}'} \delta_{r'} \Big(1 - \sum_{r \in \mathcal{R}} z_{r,r'}\Big).$$

Additionally, there is also a penalty term to punish if components of different types are matched:

$$T_7 = \sum_{c \in C} \sum_{\substack{c' \in C': \\ c_{type} \neq c'_{type}}} \varepsilon \cdot x_{c,c'},$$

where c_{type}, c'_{type} is the type of the component $c \in C$ and $c' \in C'$, respectively. The goal is to maximize the total utility. Therefore, we add the reward terms and subtract the penalty terms. This then creates the objective function:

$$O = T_1 + T_2 - T_3 - T_4 - T_5 - T_6 - T_7.$$

Since we want to match as many components and relations as possible, we have to maximize the objective function.

Next, we describe the constraints of the matching problem. First, we need constraints that ensure that every component and relation is only matched to at most one other component or relation. This is done by having a constraint for each component of the first data set $\mathcal{M}$ that guarantees that it is only matched to at most one component of the second data set $\mathcal{M}'$:

$$\sum_{c' \in C'} x_{c,c'} \leq 1 \qquad \forall c \in C.$$

Equivalently, we get constraints that each component of the second data set is only matched to at most one component of the first data set:

$$\sum_{c \in C} x_{c,c'} \leq 1 \qquad \forall c' \in C'.$$

Similarly, we can derive the corresponding constraints for the relations:

$$\sum_{r' \in \mathcal{R}'} z_{r,r'} \leq 1 \qquad \forall r \in \mathcal{R},$$
$$\sum_{r \in \mathcal{R}} z_{r,r'} \leq 1 \qquad \forall r' \in \mathcal{R}'.$$

Lastly, we need to ensure that we only match relations if the corresponding components are matched:

$$z_{r,r'} \leq x_{c_i,c_i'} \qquad \forall r \in \mathcal{R}\ \forall r' \in \mathcal{R}'\ \forall i \in [3] : r = (c_1, c_2, c_3), r' = (c_1', c_2', c_3').$$

Therefore, the entire model is set up as follows:

Problem 3.1 (Optimization Model)**.** *Let $\mathcal{M} = (C, \mathcal{R})$ and $\mathcal{M}' = (C', \mathcal{R}')$ be REXS data sets. Then the optimization problem to match these is:*

$$\begin{aligned}
\max \quad & \sum_{c \in C} \sum_{c' \in C'} f(c,c') x_{c,c'} + \sum_{r \in \mathcal{R}} \sum_{r' \in \mathcal{R}'} g(r,r') z_{r,r'} \\
& - \sum_{c \in C} \gamma_c \Big(1 - \sum_{c' \in C'} x_{c,c'}\Big) - \sum_{c' \in C'} \gamma_{c'} \Big(1 - \sum_{c \in C} x_{c,c'}\Big) \\
& - \sum_{r \in \mathcal{R}} \delta_r \Big(1 - \sum_{r' \in \mathcal{R}'} z_{r,r'}\Big) - \sum_{r' \in \mathcal{R}'} \delta_{r'} \Big(1 - \sum_{r \in \mathcal{R}} z_{r,r'}\Big) \qquad (3.1) \\
& - \sum_{c \in C} \sum_{\substack{c' \in C': \\ c_{type} \neq c'_{type}}} \varepsilon \cdot x_{c,c'}
\end{aligned}$$

$$\begin{aligned}
\text{s.t.} \quad & z_{r,r'} \leq x_{c_i,c_i'} && \forall r \in \mathcal{R}\ \forall r' \in \mathcal{R}'\ \forall i \in [3] : && (3.2) \\
& && \quad r = (id, type, order, c_1, c_2, c_3), \\
& && \quad r' = (id, type, order, c_1', c_2', c_3') \\
& \sum_{c' \in C'} x_{c,c'} \leq 1 && \forall c \in C && (3.3) \\
& \sum_{c \in C} x_{c,c'} \leq 1 && \forall c' \in C' && (3.4) \\
& \sum_{r' \in \mathcal{R}'} z_{r,r'} \leq 1 && \forall r \in \mathcal{R} && (3.5) \\
& \sum_{r \in \mathcal{R}} z_{r,r'} \leq 1 && \forall r' \in \mathcal{R}' && (3.6) \\
& x_{c,c'} \in \{0,1\} && \forall c \in C, \forall c' \in C' && (3.7) \\
& z_{r,r'} \in \{0,1\} && \forall r \in \mathcal{R}, \forall r' \in \mathcal{R}' && (3.8)
\end{aligned}$$

where it is $c = (id, type, \{attributes\})$ for $c \in C \cup C'$,
and $r = (id, type, order, c_1, c_2, c_3)$ for $r \in \mathcal{R} \cup \mathcal{R}'$, $c_i \in C \cup C'$ for $i \in [3]$.

4 Implementation

In this chapter the implementation of the optimization model of Section 3 is discussed. First we give an overview of the utilized optimization software. Then, an explanation of how the input data was prepared for further use follows. Next, the development of the implementation and the improvement and refinement of the model is discussed. Afterwards it is explained how the run time was improved. Lastly, the model's output and processing is described. This project is implemented in Python. The entire implementation of this project can be found in the following GitHub repository: `https://github.com/fva-net/rexs-diff`.

4.1 SCIP Optimization Suite

There are several software tools that can be used to solve optimization problems, e.g. Gurobi and the SCIP Optimization Suite. Both these solvers can do similar things. For example they can both solve (integer) linear programs or mixed-integer programs. Since Gurobi is a commercial solver whereas SCIP is open source, we decided on using the SCIP Optimization Suite instead of Gurobi.

SCIP is a noncommercial optimization solver for mixed integer programming and mixed integer nonlinear programming. It also supports constraint integer programming. [4] Among others, SCIP utilizes branching, bounding and cutting approaches to solve an optimization problem (cf.[4]). The problem is divided into smaller subproblems that are solved recursively. For an introduction to integer optimization and branching methods refer to [5] and [20]. SCIP uses propagation to tighten domains of variables. [4] SCIP is implemented in C and provides one of the fastest non-commercial solvers for linear programming, mixed integer programming and mixed integer nonlinear programming. Different interfaces for several programming languages are available [4]. For example there are interfaces for C/C++, Python, Matlab, Julia and Java. In our case, the Python interface [19] is used. SCIP is very flexible because there are many additional user plugins [4]. A few are listed here:

- constraint handlers to implement arbitrary constraints
- variable pricers to dynamically create problem variables
- primal heuristics to search for feasible solutions with specific support for probing and diving
- node selectors to guide the search
- branching rules to split the different children
- presolvers to simplify the solved problem

4.2 Data Structure

In this section an overview of the implemented data structure is given. To organize the data of a REXS file, we utilize a class structure. Classes for the components, relations, attributes and references are defined.

An instance of the *Component class* has the following mandatory attributes: *id*, *type* and *attributes*, and the optional attribute *name*. *Attributes* is a list of instances of the class *Attribute*. An object of the class *Attribute* has the attributes *id*, *unit*, *parameter type* and *parameter value*. The parameter type specifies what kind of data type the parameter value is. The class *Relation* has the mandatory attributes *id*, *type* and *references*, as well as the optional attribute *order*. The attribute *references* is a list of instances of the class *Reference*. An object of this class has the obligatory attributes *id* and *role* and the optional attribute *hint*.

These classes are filled with the data from the REXS file. A REXS file is either provided as an XML or JSON file. Since there is a converter to transform XML into JSON files and vice versa provided by the REXS Database [15], the algorithm in this thesis will only be able to import JSON files.

4.3 Development of the Implementation

This section illustrates the development of the implementation. First, the import routine is discussed and afterwards several problems that arise during testing are analyzed and solved.

Import routine

First, we needed to prepare the input data in a way for it to be easily accessible and transmissible to SCIP. Hence, we implemented an import routine. The input data is given as REXS files. These are either in an XML or a JSON format. The focus was set on processing the JSON format due to an existing converter on the REXS Database [15] to transform XML files into JSON files. The data in the JSON file was parsed and then put into the class structure that was explained in Chapter 4.2.

The next step was to implement the optimization problem using the SCIP Optimization Suite and test the implementation with simple test data.

Matching same types of components

To start we set the distance functions of the components $f(c, c')$ and relations $g(r, r')$ universally to one and the penalty parameters $\gamma_c, \gamma_{c'}, \delta_r, \delta_{r'}, \varepsilon$ to zero. The objective function then only consisted of the reward terms T_1 and T_2:

$$\sum_{c \in C} \sum_{c' \in C'} x_{c,c'} + \sum_{r \in \mathcal{R}} \sum_{r' \in \mathcal{R}'} z_{r,r'}.$$

This was done as to start as easy as possible and then later add the penalties and refine the functions to make the results of the optimization problem more accurate.

In the beginning we started to match data from two identical files. With this approach we ensured that our optimization problem was implemented correctly and that the correct components and relations were matched. Since we used two identical files, we knew that the algorithm should match the components and relations onto themselves.

With this test data, the produced matching was correct. The next step was to change the IDs of the components in one of the files to ensure that the implementation was not relying on the components' IDs. In this process we noticed that the calculated matching was not always correct. Upon further inspection, it became clear that components were matched with components of a different type. But as stated in the beginning we only wanted to match components of the same type.

To rectify this problem, we added the penalty term T_7 to the objective function that punishes if two components of different types are matched. This penalty term was added in the objective function rather than adding it as a strict constraint. That was necessary because we did not want to force every component to be strictly matched to the same type. The intention was to permit, in special cases, the assignment of elements of different types. Especially bearings are affected as there are several types of bearings, e.g. rolling bearings, concept bearings and slide bearings and we wanted to have the option of allowing to match different bearings. Since this could be done by defining a more specific penalty term in the objective function, we added the condition of matching the same type into the objective function and not as a constraint. In the penalty term T_7 we set the parameter ε to 5. Hence, every mismatched component type was punished the same. It is possible to add different penalties for different mismatches later. For example, one could set a smaller punishment if two different bearings are matched. The objective function then reads:

$$\sum_{c\in C}\sum_{c'\in C'} x_{c,c'} + \sum_{r\in\mathcal{R}}\sum_{r'\in\mathcal{R}'} z_{r,r'} - \sum_{c\in C}\sum_{\substack{c'\in C':\\ c_{type}\neq c'_{type}}} 5\cdot x_{c,c'}.$$

With this added penalty term, the components were now matched to the same type.

Refining the similarity function

So far, all the parameters were always set to a fixed value. Next, we wanted to add a more precise similarity function for the similarity of components $f(c,c')$. Since we had only utilized the structure of the data and the different types of relations and components, we wanted to incorporate the attributes of the components. These were then considered in this function.

The idea was to reflect the fraction of equal attributes of two components in the similarity function. With this, the domain and image of the function is $f : C\times C' \to [0,1]$. The value of $f(c,c')$ indicates how many percent of the attributes match. Let n_c be the number of attributes of component c, then it is:

$$f(c,c') = \frac{\text{number of equal attributes}}{max(n_c, n_{c'})},$$

where $c\in C$ and $c'\in C'$.
Two attributes are considered to be equal when the attribute IDs and values are equal. Remember that the attribute ID is different from the component ID. A component ID is an integer that uniquely identifies a component. An attribute ID is a string that uniquely identifies an attribute of a component, e.g. *number_of_teeth*. This idea for the distance function was thoroughly tested with different sample data by manually changing or deleting attributes of components.

In this implementation of $f(c,c')$ we only compared components of the same type. Components that were not of the same type were first attributed the value 0. Therefore, the aforementioned penalty term in the objective function which penalized if two components of different types were matched, was void and was hence removed, i.e. ε was set to zero. Due to further testing, the value of the similarity function was changed to -10 to make sure that two components of different types were not matched. With that the objective function then was:

$$\sum_{c\in C}\sum_{c'\in C'} f(c,c')x_{c,c'} + \sum_{r\in\mathcal{R}}\sum_{r'\in\mathcal{R}'} z_{r,r'},$$

with

$$f(c,c') = \begin{cases} \frac{\text{number of equal attributes}}{max(n_c,n_{c'})} & \text{for } c_{type} = c'_{type},\ max(n_c,n_{c'}) \neq 0,\\ -10 & \text{for } c_{type} \neq c'_{type}. \end{cases}$$

Ordered relations

This setup was tested with some more test data from [16]. During testing with the 2-stage industry gearbox, it was noticed that there were ordered relations, e.g. ordered assembly. An ordered assembly relation clearly assigns in what sequence components have to be assembled. This is useful, for example, when

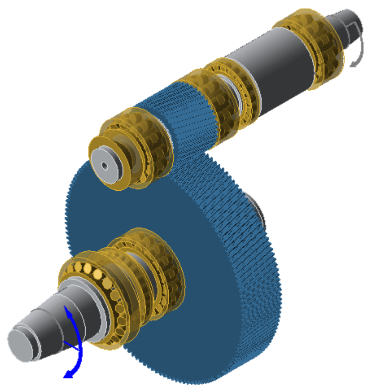

Figure 4.1 The Schaeffler marine transmission

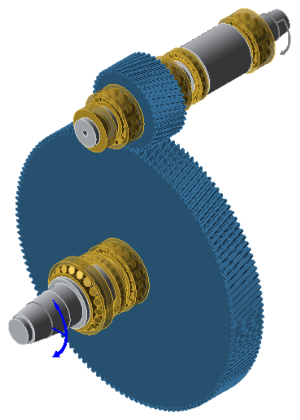

Figure 4.2 The modified Schaeffler marine transmission

assembling rolling bearings. It is desired to match relations of the same order. But since the order of relations has not been considered so far, relations of different orders were matched. To rectify this problem, we added constraints which ensured that relations can only be matched if they have the same type and order.

Matching different types of components

For some tests we modified the Schaeffler marine transmission, see figure 4.1 and figure 4.2, from [16]. In the modified version some attributes were changed. One can see in the figures that for example the sizes of the gears were altered. When matching these two transmissions, some components were not matched, even though they clearly should have been matched.

After analysis, the reason for not matching these components was a change in types of components during an update of the REXS version. Originally, it was of the type *coupling* but then was changed to be of the type *switchable coupling*.

As mentioned before it is possible that it is desired to match components of different types. Originally, we thought this was only wanted for bearings but as seen in this example it was also sought for couplings.

To make it possible for the algorithm to match these as well, we relaxed the strict condition that only components of the same type were allowed to be compared by defining the similarity $f(c, c')$ of components. We also calculated the similarities between related types to make it possible for them to be matched. Specifically, we allowed the type *coupling* to be matched with *switchable coupling*.

We also allowed something similar for bearings. At the time of writing there were the types *concept bearing, rolling bearing row, rolling bearing with catalog geometry, rolling bearing with detailed geometry* and *slide bearing*. In addition, we implemented a relaxation which allowed *concept bearings* to be matched with *rolling bearings with catalog geometry, rolling bearings with detailed geometry* and *slide bearings*. Moreover, we permitted the matching of *rolling bearings with catalog geometry* with *rolling bearings with detailed geometry*. *Slide bearings* were only allowed to be matched with *concept bearings* and of course with themselves.

Empty lists of attributes

When working on matching the Schaeffler truck planetary gearbox of the REXS Database [16] to a modified model, it was noticed that some components had an empty list of attributes. With the above definition of the similarity function $f(c, c')$, the value of these two components was zero or not well-defined if both components had no attributes. Hence, the algorithm did not match them even though we would want them to be matched. To rectify this we defined the value of the function $f(c, c')$ to be 1 when both components had no attributes and to be 0.1 whenever one of the components had no attributes. Setting it to 1 if both components had no attributes made sure they were able to be matched with each other. When both components had no attributes they were considered to be of the same similarity as when they had all of the same attributes. If only one component had no attributes, different values between 0 and 1 were tested. Setting it close to 1 resulted in matches that were not desired. Therefore we tried values closer to zero. Since it is possible we wanted to match a component with attributes to one without attributes, we created an incentive for this by setting it to 0.1. The similarity function then was:

$$f(c, c') = \begin{cases} \frac{\text{number of equal attributes}}{max(n_c, n_{c'})} & \text{for } c_{type} = c'_{type},\ max(n_c, n_{c'}) \neq 0, \\ 0.1 & \text{for } c_{type} = c'_{type},\ min(n_c, n_{c'}) = 0, \\ 1 & \text{for } c_{type} = c'_{type},\ min(n_c, n_{c'}) = max(n_c, n_{c'}) = 0, \\ -10 & \text{for } c_{type} \neq c'_{type}. \end{cases}$$

This makes sure that the algorithm can create a match in these situations, but the value is still low enough that in case there is a different component that is a better match, the algorithm prefers the better one.

4.4 Performance Improvement

With most of the example data from the REXS Database [16] the algorithm ran within a few minutes. The exceptions were tests with the wind turbine data and manual transmission data. The run time for the wind turbine data was around 45 minutes and for the manual transmission about 8 hours. Since this was quite a long time, we wanted to optimize the run time. For this, we first needed to pinpoint where the algorithm needed a lot of time. After analyzing all steps of the implementation, we found that the most time was spent during the setup of the optimization problem, i.e. defining the objective function and the constraints. Solving the optimization problem itself compared to the time needed to set it up was quite fast. In detail, the most time was needed while defining the first and second sum of the objective function, T_1 and T_2. Since setting T_1 was coded using two nested for-loops, see listing 4.1, the first idea was to use the python package numpy as a faster method to multiply and sum up the two matrices. But this effort was to no avail. Due to the fact that SCIP builds a custom expression of the sum that is passed to the solver, it did not improve the performance. We therefore looked for another solution of the run time problem.

The values of the similarity function f are saved as matrix `f`. This matrix `f` had many elements with the value zero. Therefore, another idea was to use sparse matrices. But this idea failed because `x` is a matrix of optimization variables and could not be transformed into a sparse matrix.

To exploit all possibilities, an if-condition was added that only updated the objective function if the entry in the matrix was not zero, see listing 4.2. Surprisingly, this improved the run time of the algorithm significantly. For the before mentioned data, the run time was reduced from 45 minutes to 1 minute and from 8 hours to 5 minutes. With this change, the algorithm ran under a minute for most of the test data. This improvement reduced the run time for all test data, on average by 75%. The longer the run time of the original implementation was, the bigger was the reduction of the run time in percentage.

```
    T1 = sum(f_c[i][j] * x[i][j] for i in range(x_rows) for j in range(
        x_cols))
```

Listing 4.1 Original code to calculate T_1.

```
    T1 = 0
    for i in range(x_rows):
        for j in range(x_cols):
            if f_c[i][j] != 0:
                T1 += f_c[i][j] * x[i][j]
```

Listing 4.2 Improved code to calculate T_1.

After this improvement, the definition of the optimization problem did not consume the most time anymore. Instead, the most time consuming part was the solving of the problem. The question arose if it was possible to reduce the run time even further. In an attempt to do so, we tried to improve the time the solver needed. One idea to do that was to not define all of the optimization variables. Since a lot of the variables were set to zero during the optimization process anyways, e.g. because the corresponding components or relations were not of the same type, it was thought to be possible to decrease the time SCIP needed to solve the problem if there were less variables and therefore less constraints as well. This effort did not yield the expected results. In some cases the run time for setting up the optimization problem was reduced even more. Defining the optimization variables, objective function and the constraints was faster, but the optimization process itself was not. In some cases the optimization process was significantly slower than before. This indicated that defining all variables made the problem computationally easier. Even though the time needed to define the objective function and the constraints may be less, the time added during the optimization process increased the total run time significantly. On average, the run time was extended to nearly 600% . Therefore, we decided to remain with defining all variables.

After these improvement efforts and intensive testing, it became clear that there was no more improvement in performance time achievable. The preparation of the data was quite quick due to the change in defining the objective function. Also, we could not find a way to improve the run time of the definition of the optimization model. Most of the time was consumed during the actual optimization process, i.e. by the optimization solver SCIP. Hence, from our end there was not much that could be done to improve the performance. Most of the used time cannot be influenced by anything we can do.

In figure 4.3, the run time of data of different sizes is plotted. On the x-axis the number of elements is depicted. The y-axis gives the run time in seconds. The number of elements of a data set is the sum of components and relations. There is a lot more test data available with less than 400 elements. Unfortunately we do not have a lot of data sets with more elements. The run time is not linear in the size of the input but it does not seem to be exponential either. For a more accurate estimation of how the run time behaves with large data sets, more test with larger sets must be run.

In the final implementation the setting up of the optimization problem takes on average about 40% of the total run time. The most time is consumed during the solving of the optimization problem with on average approximately 60% of the run time. The other steps, i.e. importing the data and creating the JSON output file, take a negligible amount of time. The improvements that were done significantly reduced the run time of the final implementation.

4.5 Output

An important thing is how the resulting matching of components and relations is presented. The final goal is to provide this algorithm as an online tool for clients on the REXS-Database [15]. We want to give the customer a list of all matched components and another list of all unmatched components of the data.

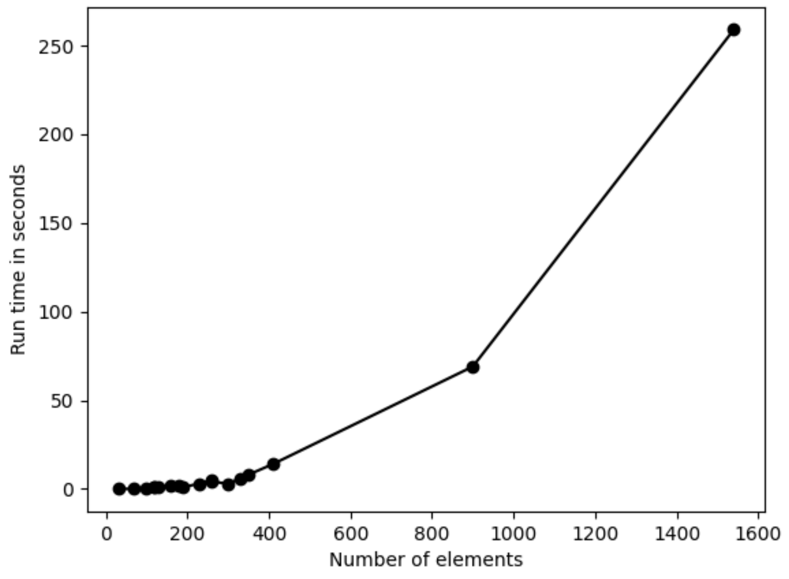

Figure 4.3 Run time of final implementation

We provide a JSON file with the resulting matching which makes it easy to process the data further and present it in an accessible way for the customers on the website.

This JSON file contains three lists of components. The first list contains all components that were matched. The second one contains all unmatched components of the first data set $\mathcal{M}$ and the third list contains all unmatched components of the second data set $\mathcal{M}'$. The data sets $\mathcal{M}$ and $\mathcal{M}'$ are called sets a and b in this JSON file. Additionally, it contains a fourth list "*warnings*". In this list any warnings or error messages are to be stored. The structure of the JSON file is then as seen below:

```
{
    "component_matches" : [...],
    "components_only_a" : [...],
    "components_only_b" : [...],
    "warnings" : [...]
}
```

Listing 4.3 Structure of the JSON file

An element of the first list has the four items *type*, *component_a*, *component_b* and *attributes*. *Component_a* has the ID and name of the component of the first data set, *component_b* the ID and name of the matched component of the second data set. *Attributes* is a combined list of all the attributes of the components. In addition to having the shared attributes, there is another value that indicates whether the attribute only appears in one of the two components or both. If it exists in both components it is also indicated whether the values are the same or whether they differ. If the values are different then they are specified for each component separately. An example for an element of this list that has the four possible attribute results is printed here:

```
{
    "component_matches" : [{
        "type": "gear_unit",
        "component_a": {
            "id": 1,
            "name": "Getriebeeinheit [1]"
        },
        "component_b": {
            "id": 3,
            "name": "Getriebeeinheit [1]"
        },
        "attributes":[{
                "result": "values_equal",
                "id": "reference_component_for_position",
                "unit": "none",
                "reference_component": 1
            },{
                "result": "only_a",
                "id": "direction_vector_gravity_u",
                "unit": "none",
                "floating_point": 0.0
            },{
                "result": "values_different",
                "id": "direction_vector_gravity_v",
                "component_a": {
                  "unit": "none",
                  "floating_point": 1.0
                },
                "component_b": {
                  "unit": "none",
                  "floating_point": 1.5
                }
            },{
                "result": "only_b",
                "id": "number_of_gears",
                "unit": "none",
                "integer": 1
            }]
    }],
    "components_only_a" : [...],
    "components_only_b" : [...],
    "warnings" : [...]
}
```

Listing 4.4 Example of an element in component_matches

The second and third lists have the same structure. The second list is for the unmatched components of the first data set and the third for the unmatched components of the second data set. Both are lists of components. Each component has a *type*, *id*, *name* and a list of attributes. Elements of these lists look like this:

```
    {
    "component_matches" : [...],
    "components_only_a" : [{
         "type": "helix_crowning",
         "id": 698,
         "name": "helix_crowning",
         "attributes": [{
               "id": "shape_of_helix_crowning",
               "unit": "none",
               "enum": "circular"
            }]
    }],
    "components_only_b" : [{
         "type": "shaft",
         "id": 716,
         "name": "Welle [716]",
         "attributes": [{
               "id": "reference_component_for_position",
               "unit": "none",
               "reference_component": 1
            },{
               "id": "support_vector",
               "unit": "mm",
               "floating_point_array": [
                  -90.71390357282,
                  0.0,
                  -51.038729438073]
            }]
    }],
    "warnings" : [...]
}
```

Listing 4.5 Example of an unmatched component

Lastly, we give an example of warnings:

```
{
     "component_matches" : [...],
     "components_only_a" : [...],
     "components_only_b" : [...],
     "warnings" : [{
          "warning": "The IDs of the components are not unique."
     },{
          "warning": "The optimization problem is infeasible."
     }]
}
```

Listing 4.6 Example of a warning

A JSON Schema was created to describe the JSON file. This JSON Schema can also be found in the GitHub Repository [17].

5 Final Mathematical Model

This chapter first discusses the final mathematical model that results from the adjustments discussed in Chapter 4.3. Lastly we describe the difference between having integer optimization variables and continuous optimization variables.

5.1 Optimization Model

The Optimization model that results from all the adjustments made in chapter 4.3 is as follows:

Problem 5.1 (Optimization Model)**.** *Let $\mathcal{M} = (C, \mathcal{R})$ and $\mathcal{M}' = (C', \mathcal{R}')$ be REXS data sets. Then the optimization problem is:*

$$\begin{aligned}
\max \quad & \sum_{c \in C} \sum_{c' \in C'} f(c,c') x_{c,c'} + \sum_{r \in \mathcal{R}} \sum_{r' \in \mathcal{R}'} z_{r,r'} && && (5.1)\\
s.t. \quad & z_{r,r'} \leq x_{c_i, c'_i} && \forall r \in \mathcal{R}\ \forall r' \in \mathcal{R}'\ \forall i \in [3]: && (5.2)\\
& && \quad r = (id, type, order, c_1, c_2, c_3), && \\
& && \quad r' = (id, type, order, c'_1, c'_2, c'_3) && \\
& \sum_{c' \in C'} x_{c,c'} \leq 1 && \forall c \in C && (5.3)\\
& \sum_{c \in C} x_{c,c'} \leq 1 && \forall c' \in C' && (5.4)\\
& \sum_{r' \in \mathcal{R}'} z_{r,r'} \leq 1 && \forall r \in \mathcal{R} && (5.5)\\
& \sum_{r \in \mathcal{R}} z_{r,r'} \leq 1 && \forall r' \in \mathcal{R}' && (5.6)\\
& x_{c,c'} \in \{0,1\} && \forall c \in C, \forall c' \in C' && (5.7)\\
& z_{r,r'} \in \{0,1\} && \forall r \in \mathcal{R}, \forall r' \in \mathcal{R}' && (5.8)
\end{aligned}$$

where it is $c = (id, type, \{attributes\})$ for $c \in C \cup C'$,
and $r = (id, type, order, c_1, c_2, c_3)$ for $r \in \mathcal{R} \cup \mathcal{R}'$, $c_i \in C \cup C'$ for $i \in [3]$.

First, we explain the optimization variables that are used. There are binary variables that indicate whether components and relations are matched. The variable $x_{c,c'}$ indicates whether a component $c \in C$ is matched to component $c' \in C'$. If the components are not matched, then $x_{c,c'} = 0$ and if they are matched, then $x_{c,c'} = 1$. Similarly, the variables for relations are defined. If a relation $r \in \mathcal{R}$ is matched to a relation $r' \in \mathcal{R}'$, it is indicated by the variable $z_{r,r'}$. Again, this is a binary variable. Therefore, if the relations are matched, then $z_{r,r'} = 1$, and $z_{r,r'} = 0$ if they are not.

Next, we give a more detailed explanation of the objective function and the constraints of this problem. To begin with, we are going into more detail about the objective function. The objective function consists of two parts. The first part $\sum_{c \in C} \sum_{c' \in C'} f(c,c') x_{c,c'}$ is a reward term if two components are matched. The second part does the same for relations but with a uniform reward of 1 for every pair of matched relations. The amount of the reward is defined by the function $f(c,c')$. This function indicates how similar the two components are. If the components are of the same type, it calculates the relative amount of attributes:

$$f(c,c') = \frac{\text{number of equal attributes}}{max(n_c, n_{c'})},$$

where n_c is the number of attributes of component c. If the components are of different types, the value of this similarity function is set to -10. Components can also not have any attributes which leads to the function not being well defined. Therefore, in that case, the value of the similarity function is set to 0.1 if either of the components has no attributes and to 1 if both have no attributes. All in all the similarity function is then $f : C \times C' \longrightarrow [0, 1]$ with:

$$f(c,c') = \begin{cases} \frac{\text{number of equal attributes}}{max(n_c, n_{c'})} & \text{for } c_{type} = c'_{type}, max(n_c, n_{c'}) \neq 0, \\ 0.1 & \text{for } c_{type} = c'_{type}, min(n_c, n_{c'}) = 0, \\ 1 & \text{for } c_{type} = c'_{type}, min(n_c, n_{c'}) = max(n_c, n_{c'}) = 0, \\ -10 & \text{for } c_{type} \neq c'_{type}. \end{cases}$$

Now, we discuss the five constraints. The first condition (5.2) ensures that only relations are matched if the corresponding components are matched as well. The constraints (5.3) and (5.4) make sure that a component can only be matched to at most one other component. Similarly, the last two constraints (5.5) and (5.6) guarantee that a relation is matched to at most one other relation.

5.2 Integer vs Continuous Variables

In the optimization process SCIP first runs several rounds of presolving. During the presolving process heuristics are used to delete redundant variables as well as constraints, as well as to find a feasible solution. Depending on the input data, SCIP often solves a relaxed version of the problem afterwards, i.e. the problem is solved with continuous variables instead of binary variables.

We notice that in many tests, the solver already found an integer solution with the relaxation. Therefore, the question arises if the linear program with continuous variables always results in an integer solution.

As a way to ascertain if it is true that the relaxed version results in an integer solution for all data sets, the optimization process is executed using continuous variables instead of integer ones. It is discovered that this observation is not universally true. Using other example data sets, the solution obtained using continuous optimization variables is not integer. This is the case for the example discussed in Section 7.2. In this example some components are allocated fractionally to different components. In this instance the objective value for the continuous variables is the same as for the binary variables. But after inspecting other sample data, there are several data sets where the objective value for the fractional solution is better than the one for the integer solution. Hence, we found instances where the fractional solution is better than the integer solution.

A small example where the fractional solution is better than the integer solution follows now.

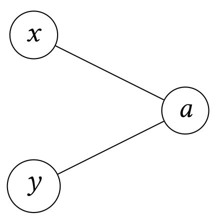

Figure 5.1 Data set 1

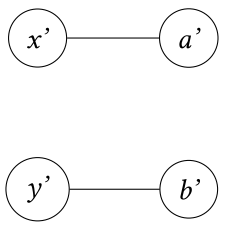

Figure 5.2 Data set 2

Suppose the first data set consists of 3 components. Two of those components are equal. We depict this example as a graph, see figure 5.1. The components are the nodes and the edges between them are the relations. The components x and y are the same. We want to match this data set to the one seen in figure 5.2.

Here we add another component. This new component *b'* is equal to the component *a'*. Also, the relations are modified. The relations in both data sets are of the same type. The components *x*, *y* ,*x'* and *y'* are all the same. The components *a*, *a'* and *b'* are the same but different from the other four. When Matching these two data sets using the integer program we get an optimal value of 4. The component matches can be seen in table 5.1. The relation that connects *x* and *a* is matched to the relation that matches *x'* and *a'*.

component of dataset 1	component of dataset 2
x	*x'*
y	*y'*
a	*a'*

Table 5.1 Component matches

Running the algorithm with continuous variables we get another result. The components and relations are matched fractionally. The component *x* is matched to *x'* and *y'* with a fraction of 0.5 each. Similarly, the component *y* is matched to *x'* and *y'* with a fraction of 0.5 each. Likewise, the component *a* is matched with a fraction of 0.5 to *a'* and *b'*. The relations are also matched fractionally. The relation connecting component *x* to *a* is matched to both relations of the second data set, each with a fraction of 0.5. The same is true for the relation connecting component *x* and *a*. This relation is also matched to both relations in the second data set with a fraction of 0.5 each. This matching then has an objective value of five. Therefore, this example shows that the use of continuous variables results in an better objective value than using binary ones.

Therefore, we can conclude that we were simply lucky in our initial sample data sets. Thus, the question if the linear program with continuous variables always results in an integer solution, can be answered with no.

After analyzing why some data sets get a better solution with the continuous variables than other sets, it is discovered that the data sets with better fractional solutions have several identical components. If there are identical components in the data set the algorithm matches these components to each other fractionally. The corresponding relations are matched fractionally as well.

6 Complexity of the Integer Program

In this chapter we discuss the complexity of the mathematical model. We want to use the graph isomorphism problem to make statements about the complexity of our problem. In the graph isomorphism problem we are given two graphs $G = (V, E)$ and $G' = (V', E')$. We want to determine whether these two graphs are isomorphic or not.

Definition 6.1 (Isomorphism of graphs). *An isomorphism of two graphs $G = (V, E)$ and $G' = (V', E')$ is a bijection $g : V \longrightarrow V'$ that preserves the adjacency of vertices, i.e. $\{u, v\} \in G$ if and only if $\{g(u), g(v)\} \in G'$. Two graphs are called isomorphic if there exists an isomorphism between them.*

It is not known if this problem is solvable in polynomial time or if it is NP-complete. In 1983 Babai and Luks proposed an algorithm to solve this problem. This algorithm has a run time of $2^{O(\sqrt{n \log n})}$ [3]. For a very long time this was the best known algorithm to solve the graph isomorphism problem. In recent years Babai proposed a quasi-polynomial time algorithm that has a run time of $2^{O((\log n)^c)}$ for a fixed $c > 0$ [2]. The quasi-polynomial time claim was refuted by Harald Helfgott because there was a flaw in the proof [18]. But the quasi-polynomial time was restored in 2017 by Babai who published a correction [1].

To make claims about the complexity of our problem, we reduce the graph isomorphism problem to our gear matching problem, i.e. given an instance of the graph isomorphism problem, we want to transform it into an instance of our problem. With "our problem" we mean the question if there exists a matching between two data sets that have the same amount of components and relations.

Proposition 6.2. *There is a reduction from the graph isomorphism problem to our gear matching problem.*

Proof. Suppose we have an instance of the graph isomorphism problem: $G = (V, E)$ and $G' = (V', E')$.
For our gear matching problem we need the two data sets $\mathcal{M} = (\mathcal{C}, \mathcal{R})$ and $\mathcal{M}' = (\mathcal{C}', \mathcal{R}')$ and a similarity function $f : (\mathcal{C}, \mathcal{C}') \longrightarrow [0, 1]$. We can interpret the vertices of the graphs G and G' as the components and the edges as the relations:

$$\mathcal{M} = (\mathcal{C} = V, \mathcal{R} = E) \text{ and } \mathcal{M} = (\mathcal{C}' = V', \mathcal{R}' = E').$$

Note that graphs do not associate a type with vertices. Therefore, we define the components $\mathcal{C}$ and $\mathcal{C}'$ to be all of the same arbitrary type with zero attributes. Similarly, since graphs do not associate types with edges, we define the relations $\mathcal{R}$ and $\mathcal{R}'$ to be all of the same arbitrary type. Because edges in graphs connect exactly two vertices, the relations in $\mathcal{M}$ and $\mathcal{M}'$ connect exactly two components as well. Since all components are of the same type and have zero attributes, the similarity function is one everywhere:

$$f(c, c') = 1 \quad \forall c \in \mathcal{C}, c' \in \mathcal{C}'.$$

Next, we need to determine for which solution of the gear matching problem a graph isomorphism exists. On the one hand, if the objective value is equal to the sum of the number of components and relations, i.e. $|\mathcal{C}| + |\mathcal{R}|$, then there exists an isomorphism of the graphs. On the other hand, if the objective value is less than this sum, then there exists no isomorphism of the graphs. □

With that we know that out problem is at least as hard as the graph isomorphism problem.

The other direction, reducing our gear matching problem to the graph isomorphism problem, is not possible. Assume we have an instance of the gear matching problem with $\mathcal{M} = (\mathcal{C}, \mathcal{R})$, $\mathcal{M}' = (\mathcal{C}', \mathcal{R}')$ and

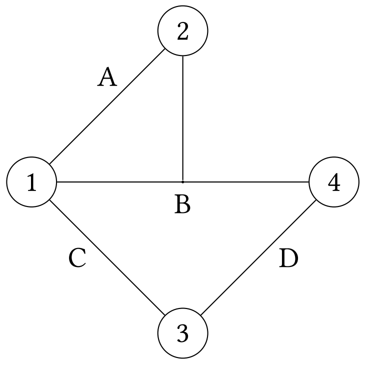

Figure 6.1 Representation as a hypergraph

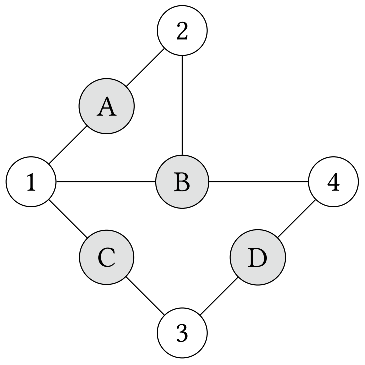

Figure 6.2 Representation as a bipartite graph

similarity function $f(c, c')$. First, we interpret this data as a graph. The components correspond to the vertices and the relations are represented by edges. For every relation we add an edge between the components that are part of that relation. Hence, $H = (V, E)$ with $V = C$ and $E = \{\{u, v, w\} | r = (type, id, u, v, w) \in \mathcal{R}\}$. Since a relation could connect more than two components, there are edges in this graph that connect more than 2 vertices. This graph is then a hypergraph. A small example is depicted in figure 6.1. This example has four components that are labeled with the numbers one through four. The relations are labeled with letters. There are also four relations. Relation B connects three components, namely components 1, 2 and 4. This hypergraph can be represented by a normal graph $G = (W, F)$ in the following way: Let the vertex set be $W = V \dot{\cup} E$ and the edge set s $F = \{\{v, e\} | v \in V, e \in E, v \in e\}$. The resulting graph H' is a bipartite graph. The hypergraph in figure 6.1 transformed into a bipartite graph is depicted in figure 6.2. The grey vertices labeled with letters correspond to the relations. Therefore, we can represent the data sets $\mathcal{M} = (C, \mathcal{R})$, $\mathcal{M}' = (C', \mathcal{R}')$ as normal graphs G and G'. Since the components and relations have additional information, like the type and attributes, the vertices in these graphs G and G' need to have this information. Since this information is taken into account when matching the components and relations, it must also be considered during the creation of an isomorphism between these graphs. But in a normal graph isomorphism problem there is no possibility to consider extra information like this. Therefore, we cannot reduce our gear matching problem to a graph isomorphism problem.

7 Examples

In this chapter three examples are being discussed. First, there is a simple example to understand the results of the algorithm. The other two examples are a bit more complex. They are data sets from the REXS Database [15].

7.1 Simple Example

First, we are looking at a small example. This example has two shafts, a bearing, and a cylindrical stage connecting the two shafts. For a picture see figure 7.1. Even though it looks like there are only these five components, there are actually a few more that are not shown in the figure. These may include some material or manufacturing settings. Overall, we have 21 components. Between these components there are 22 relations in total.

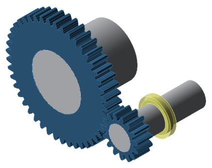
Figure 7.1 Original model

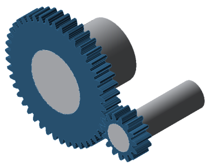
Figure 7.2 Modification 1

To begin, we match the data set to itself. We expect that every component and relation is matched to itself. This expectation became true.

Next, we delete a component of the original data set. As can be seen in figure 7.2, the deleted component is the bearing. By deleting a component, all corresponding relations are deleted as well. In this case the bearing was only part of one relation, therefore in the new model, we now have 20 components and 21 relations. As expected, exactly this component and relation are not matched. Everything else is matched to itself, since we have not changed anything else.

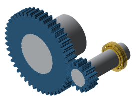
Figure 7.3 Modification 2

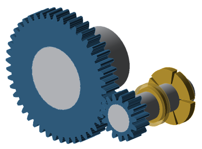
Figure 7.4 Modification 3

After deleting the bearing, we add a different kind of bearing on the same shaft as the deleted bearing was on, see figure 7.3. This new bearing is of a different type than the one we deleted. It is a rolling bearing. A rolling bearing has balls or rolls within it. These rolling elements are components of the gearbox as well. Therefore this modified gearbox now has more than the one component added. Actually there are now 33 components versus the 21 of the original one. With adding these components, associated relations are also added. In total there are now 34 relations. The original data set has 22 relations. From these numbers we expect some unmatched components and relations, namely the newly added ones that are associated to the rolling elements of the bearing. We also expect that the original bearing is matched to the rolling bearing and every other component to be matched with itself. This is exactly what happens when the algorithm is run.

Then we add a component to the original data set. We add another bearing on the smaller shaft, see figure 7.4. In this modified version we have 22 components and 23 relations. Similar as before, we expect this new component and relation to not have a match. As expected, these two elements are the only ones not matched. Every other element is again matched with itself.

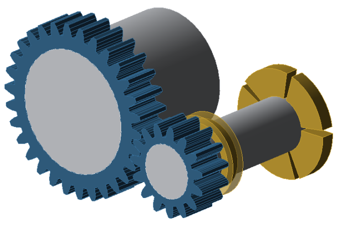

Figure 7.5 Modification 4

Afterwards, we change some attributes to make the example a little bit more difficult. For example, the number of teeth on the gears as well as the positions of the bearings are changed, see figure 7.5. Now we want to match this new model with the model in figure 7.4. Looking at the pictures, we expect that the components are matched to themselves, more specifically to their modified versions. Even though some attributes of the components are changed, this is the best way of matching the components. The resulting matching is exactly what was expected.

7.2 Two Stage Industry Gearbox

In this section we are looking at the 2 stage industry gearbox from the REXS Database [15]. This data model is compared to a modification of it where random attributes of random components are changed and components and relations are deleted and added. For example the size of some gears is altered, as well as the number of teeth of gears.

The original model has 158 components and 171 relations. The modified model has 178 components and 197 relations. As one can easily see from these numbers not all relations and components are able to be matched by the algorithm. Since we have deleted and added components (and therefore also the corre-

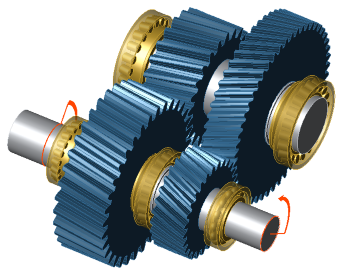
Figure 7.6 2 Stage Gearbox

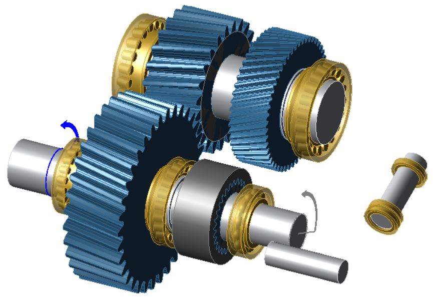
Figure 7.7 Modified 2 Stage Gearbox

sponding relations), we expect that there are unmatched elements in both data sets.

As expected, not all components and relations are matched. Of the original model only 8 components are not matched, whereas from the modified model 28 are not matched. Let's have a look at why these specific components are not assigned to other components.

Since we deleted some components from the original model, it suffices to say that these deleted components do not occur in the modified model. Therefore, the deleted components do not have a corresponding component in the modified model. Hence, these components are not matched during the optimization process.

But we also included some new components in the modified data set. Therefore, these additional components do not have a corresponding one in the original model.

Let us have a closer look why the unmatched components are not matched. First we analyze the unmatched components of the original data set. Some of these components are of the type *helix crowning*. In the modified data set there are no components of this type. Therefore these components do not a have a match. The same is actually true for all unmatched components of the original data set. There are no components in the second data set that have the same type. Hence, they cannot be matched. Let us have a look at another component that could have been matched due to its type. There are two major reasons why a component is unmatched, its similarity and its relation to other components. It could be the case that the other components of the same type have a higher value of the similarity function. On the other hand, if there is a component that has a high value of similarity, and the components are not matched, there must be another reason for that. In this case the bearing is not matched because its relation to other components does not align with the ones of the possible matched component. Here, the shafts where they are on, are not matched.

7.3 Schaeffler Marine Transmission

The last example is also an example from the REXS Database [16]. This example is provided by Schaeffler for the database. Schaeffler AG is a leading company in the drive technology industry. Similar as before we use the original data set available on the database and match it to a modified version of it. In the modified version some additional components are added, some are deleted and attributes are changed. See figure 7.8 for the original gearbox and figure 7.9 for the modified one.

The original data set has 53 components and 53 relations. The modified gearbox has 104 components and 115 relations. From these numbers alone we know that there are quite a few elements that remain

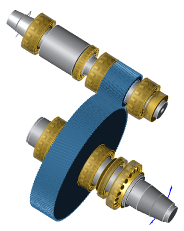
Figure 7.8 Original Marine Transmission

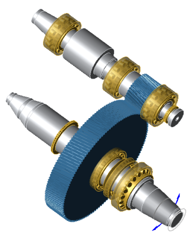
Figure 7.9 Modified Marine Transmission

unmatched. After running the tool, there is one component of the first gearbox and more than 64 of the second one that are not matched. Let's have a closer look why these components are not matched. We analyze this on a few exemplary components. First, we will look at the unmatched component of the first data set. This component is of the type *external_load*. In the second data set are two components of the same type. Both of these components are matched to other *external_load* components. So why are the components matched that way? To understand this we examine the similarity between these components. To make it easier to reference these components, we refer to them by their ID. The *external_load*-components of the first data set have IDs 1, 2, and 3. The unmatched one has ID 1. The components of the second data set have IDs 4 and 5. Component 2 is matched to component 4 and component 3 is matched to component 5. In table 7.1 the result of the similarity function is depicted.

ID	ID	Similarity
1	4	0.14285714285714285
1	5	0
2	4	0.2857142857142857
2	5	0.14285714285714285
3	4	0.7142857142857143
3	5	1

Table 7.1 Similarity of the Components

In this table we can see that the components with ID 3 and 5 are equal. The two components which share the second most similarity are components 2 and 4. Since the component 1 has only a very small similarity with the others, this component is not matched, whereas the others are matched according to their similarities. Another thing to consider is the relations that connect these components. After analyzing the relations, we know that all components have a relation of the type *assembly* that connects them to a shaft. We also realize that in the first data model there are 3 relations that connect an *external_load* to a shaft, whereas in the second one there are only two. The components 2 and 3 are both on the same shaft. The same applies to components 4 and 5 of the second gearbox. They are both situated on the same shaft. Therefore it also makes sense from a relation point of view to match these two shafts to each other and the corresponding relations. Because of these reasons the component with ID 1 is not matched, and the components with ID 2 and 3 are matched to the components with ID 4 and 5.

8 Conclusion

In conclusion, we constructed the given problem as an integer optimization problem and examined its complexity. For this purpose the problem presented in this thesis was related to the graph isomorphism problem. Through reductions it was concluded that the problem presented in this thesis is at least as hard as the graph isomorphism problem. Furthermore, we implemented this problem and used the SCIP Optimization Suite to solve it for several different instances. In this process we defined a similarity function that describes how similar two components are using the relative amount of equal attributes. Then, we improved the performance of the algorithm. If an even more reduced run time of the algorithm is desired, it might be worth looking into alternative software to solve the optimization problem. In the end, we were able to present a software that compares two different representations of a gearbox and matches elements of these gearboxes according to their similarity and the structure of the gearbox. With our approach of solving an integer optimization problem we achieved very good results.

Bibliography

[1] László Babai. “Fixing the upcc case of split-or-johnson”. In: *manuscript on Babai’s* (2017).

[2] László Babai. “Graph isomorphism in quasipolynomial time”. In: *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing* (2016).

[3] László Babai and Eugene M Luks. “Canonical labeling of graphs”. In: *Proceedings of the fifteenth annual ACM symposium on Theory of computing* (1983).

[4] Zuse Institute Berlin. *SCIP Optimization Suite Website.* 2023. URL: `https://scipopt.org/` (visited on 05. 07. 2023).

[5] Michele Conforti, Gérard Cornuéjols, and Giacomo Zambelli. *Integer Programming.* Springer International Publishing Switzerland, 2014.

[6] FVA GmbH. *Aufbau der REXS Objekte.* 2023. URL: `https://www.rexs.info/spec/1.5/de/root-rexs-objects.html` (visited on 29. 10. 2023).

[7] FVA GmbH. *FVA Software & Service.* 2023. URL: `https://www.fva-service.de/en/company/about-us/` (visited on 13. 11. 2023).

[8] FVA GmbH. *FVA Software & Service Website.* 2023. URL: `https://www.fva-service.de/` (visited on 13. 11. 2023).

[9] FVA GmbH. *FVA Techblog 13.* URL: `https://www.fva-service.de/de/software/techblog/techblog13/` (visited on 24. 10. 2023).

[10] FVA GmbH. *FVA Techblog 18.* URL: `https://www.fva-service.de/de/software/techblog/techblog18/` (visited on 24. 10. 2023).

[11] FVA GmbH. *FVA Techblog 2.* URL: `https://www.fva-service.de/de/software/techblog/techblog2/` (visited on 24. 10. 2023).

[12] FVA GmbH. *REXS - Standardisiertes Getriebemodell.* 2018. URL: `https://www.fva-service.de/de/news/news-liste/news-detail/rexs-standardisiertes-getriebemodell/` (visited on 24. 10. 2023).

[13] FVA GmbH. *REXS Datenbank - Komponenten.* 2023. URL: `https://database.rexs.info/rexs/component/list?page=0&size=2147483647&sort=componentId,asc` (visited on 22. 11. 2023).

[14] FVA GmbH. *REXS Website.* 2023. URL: `https://rexs.info/rexs_en.html` (visited on 03. 07. 2023).

[15] FVA GmbH. *REXS-Database.* 2023. URL: `https://database.rexs.info/dashboard` (visited on 02. 08. 2023).

[16] FVA GmbH. *REXS-Examples.* 2023. URL: `https://database.rexs.info/rexs/example/list` (visited on 14. 07. 2023).

[17] Sarah Gräßle and Sarah Douglas. *Github Repository "rexs-diff".* 2023. URL: `https://github.com/fva-net/rexs-diff`.

[18] Harald Helfgott. *Graph isomorphism in subexponential time.* 2017. URL: `https://valuevar.wordpress.com/2017/01/04/graph-isomorphism-in-subexponential-time/` (visited on 20. 11. 2023).

[19] Stephen Maher et al. “PySCIPOpt: Mathematical Programming in Python with the SCIP Optimization Suite”. In: *Mathematical Software – ICMS 2016.* Springer International Publishing, 2016, pp. 301–307.

[20] Alexander Schrijver. *Theory of Linear and Integer programming*. Wiley, 2000.

[21] Forschungsvereinigung Antriebstechnik e. V. *FVA Profil*. URL: `https://fva-net.de/2022/profil/` (visited on 13. 11. 2023).