A deep graph convolutional neural network aiding in finding feasible shunt plans

van de Ven, A.J.G.

Award date:
2018

Technische Universiteit
Eindhoven
University of Technology

Department of Industrial Engineering and Innovation Sciences
Operation Management & Logistics
Information Systems Research Group

A Deep Graph Convolutional Neural Network aiding in finding feasible shunt plans

Master Thesis

A.J.G. (Arno) van de Ven
Student number: 0971415

Supervisors:
dr. Y. (Yingqian) Zhang, Eindhoven University of Technology
dr. H. (Rik) Eshuis, Eindhoven University of Technology
dr. WJ. (Wan-Jui) Lee, Maintenance Development NS

Final Version

Eindhoven, September 2018

Abstract

A local search heuristic, currently under development at NS, evaluates all components of the Train Unit Shunting Problem simultaneously to create shunt plans. The heuristic requires an initial solution to start with. A previous study at NS contributed to this heuristic by predicting feasibility of an initial solution before applying the heuristic. This method relied on heavy feature engineering. This research project modified a machine learning based graph classification method to perform the same task without the need of heavy feature engineering. Results show that both methods are comparable in terms of classification accuracy. The graph classification method is also quite accurate predicting the order in which the local search heuristic should evaluate search operators. Accurately predicting the evaluation order theoretically leads to having less randomness in the search process and finding improvements faster. Although real effects could not been tested, it seems promising to guide the local search heuristic with the help of a graph classification method.

Keywords: Machine Learning, Applied Machine Learning, Convolutional Neural Networks, Graph Mining, Classification, Local Search Optimization

Executive summary

The Dutch Railways (NS) operates a daily amount of 4.800 domestic trains serving more than 1,2 million passengers each day. To ensure that these passengers arrive both safely and comfortably, trains need to be maintained and cleaned at regular intervals. Frequent tasks such as cleaning, washing, inspection and small maintenance are performed at service sites close to major stations. Here, NS is dealing with the so-called shunting activities. Shunting occurs whenever train units are temporarily not needed to operate a given timetable. Planning shunting activities on service sites is a major challenge to NS. It includes matching all arriving train units to departing train units according to the timetable. Maintenance and cleaning activities need to be scheduled and trains need to be routed over and parked on the tracks of the service site. The complex physical layout of service sites makes the parking and routing of trains a complex task. Maintenance and cleaning tasks are typically only possible on dedicated tracks on the service site. The main complexity follows from the interaction between the different components: matching, task scheduling, routing and parking. Adjustments in one component lead to complications in other components. This all is referred to as the Train Unit Shunting Problem, a difficult sequential decision making problem faced by NS.

As of today, human planners at NS manually generate a step-by-step plan, following a set of service site specific heuristic rules. This process is complex and time consuming. NS is expanding their fleet of train units by 37% in the next five years [23] and this calls for an automated planning approach. Management is questioning if the capacity of existing service sites is sufficient to handle the growing amount of train units. An automated planning approach is necessary to determine the current capacity of the service sites in order to consider any future investments.

Van den Broek [26] presented a local search heuristic that evaluates all components of the Train Unit Shunting Problem simultaneously to create shunt plans. The heuristic is able to generate feasible shunt plans for real-world problems which makes it useful to determine the capacity of service sites. The heuristic requires an initial solution to start with. A recent study [9] at NS shows that features, extracted from the initial solution, can be used to predict whether local search can find a feasible solution based on an initial solution. The main idea is that predicting feasibility before applying the local search heuristic will lead to a decrease in computation time determining the capacity of a service site. However, the method [9] relies on heavy feature engineering and extensive domain knowledge. This can be avoided given the way the local search heuristic represents shunt plans. The local search heuristic models the activities that take place on the service sites as nodes in a precedence graph. The heuristic makes small local changes to this graph to iteratively improve a shunt plan. Recent research on machine learning based graph classification has proven to achieve high accuracy predicting the class of an arbitrary graph [33]. Graph classification overcomes the need of extensive domain knowledge when predicting feasibility of an initial solution. These developments have lead to this research project and the following research question:

"How can machine learning based graph classification support the local search heuristic at NS?"

This research project is conducted following the six phases of the CRISP-DM methodology: business understanding, data understanding, data preparation, modeling, evaluation and deployment.

The first phase is *business understanding*. State-of-the-art literature about machine learning based graph classification has been reviewed. Creating a method from scratch would be too unrealistic considering the available time. Therefore has been chosen to modify the Deep Graph Convolutional Neural network (DGCNN) from Zhang et al. [33] to fit this research project. DGCNN transforms a graph into a feature vector by applying the Weisfeiler-Lehman subtree kernel. A neural network can be trained on these feature vectors to perform a classification task. But first, to perform this task, data has to be generated and prepared.

During the *data understanding* phase has been looked at how the data is generated. NS uses an instance generator to derive instances for the Train Unit Shunting Problem. Instances can be specified according to a set of input parameters. For this research project, instances are generated for a service site located near The Hague Central Station. After generating an instance, an initial solution generator transforms an instance into an initial solution that is represented as activity graph. Consequently, the local search heuristic is applied on an initial solution trying to find a feasible solution within a maximum runtime of 300 seconds. The outcome is the class label that belongs to that specific initial solution. This process is performed repeatedly to create a dataset of 10.000 graphs.

Three *data preparation* tasks have been performed: class balancing, data construction and data formatting. Class balancing has to be done because the negative class is outnumbered by the positive class. The heuristic was able to solve 7.205 initial solutions, while 2.795 initial solutions were not solvable. Balanced datasets have been created by both under- and oversampling. During the modeling phase it became clear that undersampling gave a slightly better performance. Also, the time to train DGCNN was significantly lower. Secondly, data construction. The level of detail of the node labels in the graphs had to be defined. The Weisfeiler-Lehman subtree kernel works by appending node labels of neighbouring nodes to the original node labels. Clearly, if the variety of original node labels becomes larger, the length of the resulting feature vector becomes larger. With too few original node labels, feature vectors will likely be very similar. Too many node labels results in very dissimilar feature vectors. Results of the modeling phase showed that performance increases as the number of node labels increases. Lastly, data formatting. DGCNN accepts graphs in the format of NetworkX. NetworkX is a Python library for studying graphs and networks. Graph data obtained from the initial solution generator is formatted into this data type.

During the *modeling and evaluation* phases, two contributions to the existing local search heuristic have been analyzed. The first contribution is predicting feasibility. The final classification model of DGCNN is able to predict feasibility of an initial solution with 65,1% certainty, after hyperparameter tuning. The domain knowledge based classifier Dai [9] created, achieves a performance of 66,35%. This slightly better performance is the result of heavy feature engineering assuming extensive domain knowledge. Whereas, on the contrary, DGCNN takes initial solutions directly as input. Notwithstanding the better performance, but given the fact that DGCNN does not rely on heavy feature engineering can be concluded that DGCNN is comparable to domain knowledge based models.

Besides predicting feasibility, analysis showed that prediction results on individual instances are not very similar between DGCNN and the domain knowledge based classifier. The percentage of predicted labels by DGCNN that matches labels predicted by domain knowledge based classifier is 70,2%. Therefore, it is possible that DGCNN performs better on one part of the instances, while the domain knowledge based classifier performs better on another part of the instances. Both classifiers have been combined by an ensemble learning technique, called stacking. Stacking is a technique to combine multiple base learners via a meta learner. The output of the base learners serves as input for the meta learner. Linear Discriminant Analysis has been used as meta learner. Accuracy increased to 67,7% using stacking. Being able to predict feasibility of an initial

solution before applying local search leads to a decrease in computation time when determining the capacity of a service site. This decrease in runtime will save 19,2% when determining the capacity of a service site. With 35 sites in the Netherlands and many different situations that have to be tested, this reduction can have quite an impact. This is the first way to contribute to the local search heuristic. The other contribution comes in the form of guiding the local search heuristic by predicting the best search operator to apply.

The local search heuristic attempts to find a feasible shunt plan by applying search operators in iterations to move through the search space. In total, 11 search operators are defined. In every iteration, the search operators are shuffled in random order. This will be the order in which the operators will be evaluated. DGCNN can replace random ordering by predicting in which order search operators should be evaluated. The output of DGCNN will be a vector of probabilities for each search operator. These probabilities determine the order in which search operators are evaluated. The operator with the highest probability will be evaluated first. Using DGCNN could be beneficial for two reasons:

1. Finding improvements faster;
2. Less randomness in search process.

Since three out of 11 operators account for 90% of the applied operators, two sequential DGCNN classifiers have been created. The first determines whether one of the three biggest operators should be applied, or one of the eight others. The second classifier depends on the outcome of the first. The average accuracy of both classifiers is 48,0% (predicting one of 11 operators directly results in an accuracy of 26,9%). Due to lack of time, the real effects of using DGCNN to determine the order of search operators could not be tested.

To conclude, this research project proves that machine learning based graph classification contributes to the existing local search heuristic without the need of extensive domain knowledge. Although guiding the local search heuristic has not been extensively studied, the results seem promising for further development.

Preface

This is the thesis 'A Deep Graph Convolutional Neural Network aiding in finding feasible shunt plans', a research project regarding the practical contribution of machine learning based graph classification in the Train Unit Shunting Problem. It has been written to fulfill the graduation requirements of the Operations Management & Logistics Program at Eindhoven University of Technology (TU/e).

This project was carried out at the request of NS, where I did an internship for the entire duration of my research. At the start of the project, the path I was going to take was not quite sure as there were many paths open to explore. The research was difficult, but conducting extensive research has allowed me to answer the research question that was formulated.

I would like to thank my supervisors for their excellent guidance and support during this process. Yingqian Zhang, my supervisor from TU/e, I would like to thank you for the interesting ideas and for helping me straighten my back in less motivating times. Wan-Jui Lee, my supervisor at NS, thanks for the occasional sparring sessions and the constructive feedback during the project. To my other colleagues at NS, I would like to thank you for your cooperation as well.

I would also like to thank Rens Peters for all those Wednesdays at the university that we kept each other posted about our projects and for supporting one another. My parents deserve a particular note of thanks: your interest and kind words have, as always, served me well.

I hope you enjoy this reading.

Arno van de Ven

Eindhoven, September, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Dutch Railways (NS) operates a daily amount of 4.800 domestic trains serving more than 1,2 million passengers each day. To ensure that these passengers arrive both safely and comfortably, trains need to be maintained and cleaned at regular intervals. Most of larger, technical maintenance and repair is done at specialized maintenance depots. Whereas more frequent tasks such as cleaning, washing, inspection and small maintenance are performed at service sites close to major stations. How is NS able to schedule its maintenance and service activities without causing problems to the day-to-day schedule on the Dutch rail network? Here, NS is dealing with the so-called planning of shunting activities. Shunting occurs whenever train units are temporarily not needed to operate a given timetable.

Planning movements and tasks on train service sites is a major challenge to the NS. It includes matching all arriving train units to departing train units according to the timetable. Maintenance and cleaning activities need to be scheduled and trains need to be routed over and parked on the tracks of the service site. Even without maintenance and cleaning activities, the complex physical layout of service sites makes the parking and routing of trains already a complex task: many tracks are dead-ended, overtaking is not possible, track- and train lengths vary. Maintenance and cleaning tasks are typically only possible on dedicated tracks on the service site. The main complexity creating a feasible shunt plan follows from the interaction between the different components: matching, task scheduling, routing and parking. Adjustments in one component lead to complications in other components causing difficulty in generating feasible plans. This all is referred to as the Train Unit Shunting Problem, a difficult sequential decision making problem faced by the NS.

As of today, human planners at NS manually generate a step-by-step plan, following a set of service site specific heuristic rules. This process is time consuming. Recent developments call for an automated planning approach. The NS is expanding their fleet of train units by 37% in the next five years [23]. Management is questioning if the capacity of existing service sites is sufficient to handle the growing amount of train units. To state it another way, enlarging existing service sites or building new sites will cost a huge amount of money. An automated planning approach is necessary to determine the current capacity of the service sites in order to consider any future investments. One way to determine the capacity of a service site is by solving a mathematical optimization model to find an optimal solution. However, mathematical models developed thus far have proven to be either too complex or too restrictive to generate feasible shunt plans in a reasonable amount of time. The models were able to construct shunt plans for small instances, but are unsatisfactory in more realistic cases. Furthermore, computation time rises quickly for larger instances and above all, due to its limited flexibility, the model is not capable of planning as many train units as human planners are able to do. Another way to determine the capacity is using an heuristic approach which gives sub-optimal solutions. A heuristic solution, currently under

development at NS, has been successfully applied to artificial and real-world planning problems. Van den Broek [26] presented a local search heuristic that evaluates all components of the Train Unit Shunting Problem simultaneously to create shunt plans. The heuristic is able to generate feasible shunt plans for real-world problems which makes it useful to determine the capacity of service sites. To determine the capacity of a specific service site, the heuristic is applied on many planning instances with different sizes. The instance size is expressed as the number of train units that needs to be processed on the specific service site. Instances with a small amount of train units are easy to solve, since there is sufficient space on the service site to schedule every train unit. As the instance size gets larger, it is more difficult to find feasible shunt plans. After running the heuristic a sufficient amount of time, the number of train units converges towards an amount for which the heuristic can solve 95% of the instances. This number is assigned as the capacity of the service site.

A local search approach requires an initial solution to start with. Van den Broek [26] states that any decent shunt plan, satisfying some hard constraints, suffices as initial solution. A recent study [9] at NS shows that features, extracted from the initial solution, can be used to predict whether the initial solution will lead to a feasible shunt plan. The main idea is that predicting feasibility before applying the local search heuristic will lead to a decrease in computation time determining the capacity of a service site. However, the method [9] relies on heavy feature engineering and extensive domain knowledge.

This can be avoided by the way the local search heuristic represents shunt plans. The local search heuristic models the activities that take place on the service sites as nodes in a precedence graph. The integrated search approach makes small local changes to this graph to iteratively improve a shunt plan. Recent research [33] on machine learning based graph classification has proven to achieve high accuracy predicting the class of an arbitrary graph. Graph classification methods using machine learning would overcome the need of extensive domain knowledge. These developments have lead to this research project.

1.1 Problem definition and research gap

Graphs arise very naturally in many situations. Examples vary from a social network graph of friends to road-map graphs of cities. There is an ongoing development to extract much more complicated information from graphs. This has lead to the field of graph mining. Whereas data mining is the process of finding patterns and correlations within large datasets to predict outcomes, graph mining is essentially the problem of discovering meaningful metrics to capture graph structure and designing algorithms that are able to calculate these metrics.

Within the field of graph mining, performing classification tasks on graphs is also quickly growing because the number and the size of graphs has been growing exponentially. This creates an opportunity to use machine learning algorithms for graph classification. Machine learning on itself is already a hot topic applied in a lot of ongoing development, but also stimulating completely new developments. Although it has been around since the early 1960s, the introduction of the internet, the exponential growth in data storage and processing, and the ever growing set of open source tools and research papers, has made it easier than ever to use machine learning to automate classification processes.

Recent developments on machine learning based graph classification has proven to be successful within the fields of bioinformatics and social networks. It has also been successfully applied on synthetically created datasets using a combination of mathematically understood random graph generation methods [5]. These developments showed to be accurate in both binary and multi-class graph classification tasks. Although graph classification shows promising results, not much research has been done on machine learning based graph classification (hereafter referred to as

'graph classification'). To the best of our knowledge, the first study was conducted in 2016 [22]. More importantly, no research has been done on graph classification to aid in solving optimization problems.

To conclude, graph classification seems to have promising applications, but its usefulness in optimization problems has not been studied yet. Current applications are designed to use arbitrary graphs as input. However, these methods are only tested on standard datasets within the fields of bioinformatics and social networks. A method designed for graph classification within optimization problems, specifically the Train Unit Shunting Problem, should be able to improve heuristic solutions.

1.2 Research scope

The first part of this research project is a literature review. The literature review consists of three parts: the scope of the first part lies on local search optimization. In particular, the implementation of a local search heuristic as a solution to the Train Unit Shunting Problem. The scope of the second and third part lies on graph theory. The second part will elaborate on graph classification and techniques to measure graph similarity. Whereas the third part will focus on machine learning in general, machine learning in optimization problems and machine learning on graphs.

As a succession to the literature review, which will elaborate thoroughly on the current research gaps and issues, the generation and preparation of graph structured data will be described. The data available in this research project is limited to graph data created by the local search heuristic [26]. Consequently, the scope of the modelling part is restricted to adapting an existing graph classification algorithm. Creating an algorithm for graph classification from scratch would be too unrealistic considering the available time.

1.3 Research questions

The introduction to the topic of this research and the problem definition defined in Section 1.1 motivate the following main research question:

"How can machine learning based graph classification support the local search heuristic at NS?"

The research project aims to demonstrate the effectiveness of graph classification by contributing to the local search heuristic at NS. The main contribution of this research project to this application, will be predicting whether local search can find a feasible solution based on an initial solution. Moreover, this research project will look into methods to combine domain knowledge based classifiers with a graph classification method to increase performance. Additionally, a brief analysis will be performed to check whether graph classification can be useful to guide the local search heuristic. Later in this report will be elaborated on this contribution. To answer the research question, multiple sub research questions are formulated that will help guiding and give direction answering the main research question.

1. How does the local search heuristic explore neighbourhoods?
2. How does the heuristic represent the states as graphs?
3. How does the heuristic create initial solutions?

4. Which graph classification methods already exist?
5. How to adapt an existing graph classification method to use graph data obtained from the local search heuristic?
6. How does a graph classification method perform compared to methods based on heavy feature engineering?
7. How to combine a graph classification method with a domain knowledge based classifier?
8. What is the added value of using a graph classification method during local search in an optimization problem like the planning and scheduling problem at NS?

The following three paragraphs will further elaborate on answering these research questions by defining the research more clearly in its relevance, design and goal.

1.4 Research approach

The research is split into several parts according to the cross-industry standard process for data mining cycle (Figure 1.1): business understanding, data understanding, data preparation, modelling, evaluation and deployment. The literature review is part of business understanding. It holds a critical view against related research about train unit shunting, graph classification and optimization problems in operations research. Additional to the literature review, the business understanding part consists of understanding the business goals at NS, such as understanding the capacity determination project. This first part of this research project answers the first four sub research questions.

Figure 1.1: CRISP-DM cycle

The second and third part of this research aims at understanding and preparing the data for the modelling part. Precedence graph data is a very primitive data type. However, the data not only contains precedence relations. It also incorporates information about specific train units, activity duration, tracks and service task related information. Graph classification methods are able to handle these kinds of additional information provided that this information is offered in the right format. Data preparation is needed to present the available data in the right format.

The fourth part is about development of the graph classification method. It will describe how the existing method is modified to fit this research project in order to make the most accurate predictions. A variety of graph classification methods are tested to determine which method provides the best predictions given the kind of graph data available in this project. The fourth part answers sub research questions five and six.

During the fifth part will be assessed what will be the added value of the method to the business at NS. The evaluation phase also involves assessing other results found during the research project. The last two sub research questions are answered in the fifth part of this research project. The sixth and last part is dependent on the results of the evaluation. If the results are satisfactory, a brief analysis will be performed to check whether graph classification can be useful to guide the local search heuristic and a strategy will be determined for deployment.

1.5 Research goal and relevance of research

The goal of this research is to modify a graph classification method, specifically designed to contribute to the local search heuristic. This involves representing available data in a format suitable to serve as input for a classification task. The foremost goal of this research project is to find out if graph classification can overcome the necessity of heavy feature engineering.

Scientifically, this research project closes a part of the gap that is currently existing in literature of machine learning in operations research. Taking knowledge from existing research in graph classification and apply this knowledge directly in an operations research problem and industry related application. Additionally, the research provides practical benefits for NS through its contribution at improving the local search heuristic. It does this by making the heuristic faster through predicting feasibility of initial solutions and making it more efficient if it is able to guide it to more promising areas.

1.6 Outline of report

This report is structured into seven parts, with every part being a separate chapter. The first chapter introduces the topic, problem definition and research design. The second chapter provides background information and contains a literature review, where relevant literature and theoretical concepts will be explained and elaborated. The third chapter will describe how data is obtained and prepared for modelling. The fourth chapter describes a graph classification method, the classification results and the comparison with a domain knowledge based model. The fifth chapter describes an ensemble learning technique that combining multiple classifiers and how the ensemble contributes to the local search heuristic. The sixth chapter describes elaborates on how the exact same method described in chapter 4 can be used to aid in guiding the local search heuristic. The seventh and last chapter of this report is the conclusion, which will include limitations and will end with the recommendations for further research.

Chapter 2

Background and literature review

The following chapter provides the background that is related to the research problem. Furthermore, it elaborates on theoretical concepts necessary to understand this research project. Additionally, it reviews the state-of-the-art literature and it gives an overview of the methods that are already proposed by other research. Finally, the last section concludes the limitations and further elaborates on the research gap that is existent in current literature. This chapter answers the sub research questions how local search explores neighbourhoods, represents shunt plans as graphs and creates initial solutions. Furthermore it describes which graph classification methods exist in literature.

2.1 Background: planning of train shunting activities

Shunt plans describe, for a 24-hour planning horizon: (1) the assignment of incoming trains to departures the next morning, (2) the order of completion of the service tasks, (3) the tracks on which the trains will be parked and (4) the exact route of each train movement. The arrival times are based on a fixed timetable, and service tasks are scheduled using their respective norm duration. Even without taking disturbances into account, finding a feasible shunt plan is a difficult problem, as it is referred to as strongly NP-hard [12].

Both the service and storage capacity on shunting stations is limited, due to the proximity to major stations, most of which are located in urban areas. Furthermore, train movements are highly constrained and trains should be parked such that each train movement has an unobstructed path to its destination. The infrastructure of a service site consists of a set of tracks which are connected by a number of switches. Figure 2.1 shows a small example shunting station.

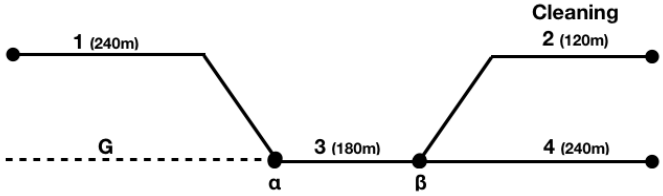


Figure 2.1: An example of a shunting station

Tracks approachable from both sides are called free tracks. On the contrary, last-in-first-out-tracks (lifo-tracks) are only accessible from one side, with the other side being blocked by a bumper.

Tracks 1, 2 and 4 in Figure 2.1 are lifo-tracks. Track 3 is a free track. Trains enter and exit the site over gateway-track G. The tracks are connected by the switches α and β. The length of the parking tracks is displayed in meters. A cleaning platform allows internal cleaning tasks to be performed on trains positioned on track 2.

To illustrate the Train Unit Shunting Problem, consider a simple scenario of three train units at the shunting station in 2.1. A train unit is a set of carriages that together form a self-propelling vehicle that can drive in both directions. Different types of train units exist, which on their hand consist of different subtypes indicating the number of carriages the train unit contains. Figure 2.2 shows one of the train units and their corresponding subtypes NS operates. Appendix A contains an overview of all train units and corresponding subtypes NS operates on a daily basis. The train units in the example scenario are of the VIRM type. Details of the train units are depicted in 2.1. The example shows two arriving and two departing trains, scheduled respectively according to Tables 2.2 and 2.3.

Figure 2.2: Different subtypes of train unit types VIRM with six and four carriages

Train Units	**Type**	**Service tasks**
1	VIRM-4 (108m)	Cleaning (34min)
2	VIRM-4 (108m)	Cleaning (34min)
3	VIRM-6 (162m)	None

Table 2.1: *The train units in the example scenario*

Arriving train	**Time**
(VIRM-4, VIRM-4)	12.00
(VIRM-6)	12.30

Table 2.2: *Arriving trains*

Departing train	**Time**
(VIRM-4)	12.45
(VIRM-4, VIRM-6)	14.00

Table 2.3: *Departing trains*

As mentioned in the introduction, the main complexity of creating a feasible shunt plan follows from the interaction between the different sub-problems: matching, task scheduling, routing and parking. In the **matching problem**, a set of arriving trains and a set of departing trains are given. The objective is to match each arriving train unit to a specific position in one of the departing train units such that: (1) the train subtypes match, (2) the train unit enters the service site before departure and (3) service tasks can be performed in the time the train is on the service site. When two train units, coupled at arrival, are assigned either to two different departing trains or in a different order to a single departing train, they have to be split on the service site. Similarly, if train units assigned to the same departed train are not coupled at arrival, a combine operation has to take place. The split and combine operations take several minutes. Therefore, it is usually preferable to find a matching that minimizes the number of changes to the train compositions.

In the **task scheduling problem** a set of service tasks together with their respective duration is given for each train unit. Each service task requires some resource for its entire duration and the availability of resources is usually limited. In the task scheduling problem, a schedule must be constructed such that all tasks are completed and all trains can depart on time.

The main objective in the **parking problem** is to place all shunt trains on the tracks such that at any moment in time the combined train length does not exceed the length of the track, while simultaneously ensuring that each shunt train has an unobstructed path when it has to depart. The order of trains on a free track depends not only on the order of arrival, but also on the arrival

side. Therefore, the arrival side has to be specified as well when a train is parked on a free track. An additional constraint is that parking, splitting and combining is not allowed on gateway-tracks.

The goal of the **routing problem** is to find for each train movement an unobstructed path that minimizes the movement duration. The duration is estimated based on the number of tracks, switches and required saw moves on the path. Due to safety regulations, train drivers, of whom there is often only one present at a service site, may only guide a single train movement at any time. Although these regulations are in reality often violated due to delays, an feasible shunt plan should follow the correct procedure.

Regarding the example scenario, no predefined matching is given, hence train units 1 and 2 can be freely assigned to either the first departure or the left position in the second departing train. The first arriving train has to split, there is no match for the entire train and the train is longer than the length of track 2, which provides access to the cleaning platform. Furthermore, one combine operation is necessary to form the composition of the second departure. With only a single cleaning platform, there has to be decided in which order the service tasks will be completed. Cleaning tasks have to be finished before a train needs to depart. Additionally, when the train units are not being cleaned, they have to be parked in such a way that the trains can depart on time without any crossings.

When constructing a feasible shunt plan for this example, train unit 2 can arbitrarily be assigned to the train departing at 12:45. The other two train units will be part of the second departure. As train unit 2 is the first to depart, it is scheduled to be cleaned before train unit 1. The individual shunting sub-problems seem trivial to solve. However, the interaction between these components will make most shunt plans infeasible. For example, while parking on track 3 is possible, it blocks virtually all routes on the service site. Furthermore, poorly parked trains might require multiple saw moves. A saw move is reversing the direction of a train. This move is time consuming and resources are limited. Only a single driver is available at a service site. Therefore, no other movements are possible when a saw move has to be performed. Planners try to avoid these moves because they can easily cause departures to be delayed in many heavily time-constrained cases. Service task scheduling in the example is determined entirely by the matching, as there is not enough time to clean both train units of the first arriving train before one of them has to depart. The matching is dependent on the parking and routing as well; switching the order in which train units 1 and 2 depart will result in an infeasible solution due to the small time-window between the first arrival and departure.

In a simplified example like this, a planner has a good overview of the whole plan and is able to deal with changes relatively easy. On the contrary, real-world scenarios consist of up to 21 tracks connected with many switches. As all sub-problems are interwoven to a great extent, making small adjustments in real-world scenarios have a snowball effect on the rest of the plan. Above all, in real-world scenarios some tracks might only be reachable by moving through some facility, which creates a dependency between routing and service task scheduling and this is just one example of interactions between components.

In 2005, Freling [11] introduced the Train Unit Shunting Problem and he tried to solve it mathematically. The Train Unit Shunting Problem is an optimization problem and its objective is to match the arriving and departing shunt units and park these shunt units on the shunt tracks such that the total shunting costs are minimal. Freling decomposed the problem in two steps formulating it as a Mixed Integer Problem. Since then, several approaches have been applied: column generation [11, 20, 13], constraint programming [19, 13], randomized greedy heuristics [13] and local search heuristics [16, 26]. Since this research project takes the local search heuristic approach by Van den Broek [26] as a starting point, only local search will be elaborated on in this chapter. Appendix B contains a detailed description of all other research approaches on the Train Unit Shunting Problem. The next section elaborates on the local search heuristic, successfully applied by Van den Broek.

2.2 An integrated local search algorithm

First an explanation on how the successful implementation of Van den Broek is integrated into the business at NS. The heuristic application is used at NS to determine the capacity of existing service sites. This is done by repeatedly running the heuristic for instances with a different amount of train units. Instances with a small amount of train units are easy to solve, since there is sufficient space on the service site to schedule every train unit. As the instance size gets larger, it is more difficult to find feasible shunt plans. After running the heuristic a sufficient amount of time, the number of train units converges towards an amount for which the heuristic can solve 95% of the instances. This number is assigned as the capacity of the service site.

Local search is a heuristic method that attempts to find the global optimum, a solution that minimizes some objective function. Starting from an initial solution, local search algorithms move through the search space, the set of all solutions, by iteratively applying small, local changes. The procedure of making a certain type of adjustment to a solution is referred to as a search operator. The neighbourhood of a solution is the set of all solutions reachable from the current solution through a single application of a search operator. The negative aspect of this heuristic is that it can get stuck in local optima when no improving solutions are present in the neighbourhood.

One extensively studied technique, solving the local optima problem, is Simulated Annealing [18, 8]. A stochastic search process that has seen many successful applications to other computational problems. Simulated annealing randomly selects an operator and evaluates the set of solutions according to an objective function. The selected solution is immediately accepted as the candidate solution for the next iteration if it is an improvement over the current solution. If the selected solution is worse, it is selected with a certain probability depending on the difference in solution quality and the progress of the search process. Accepting worse solutions is a fundamental property of simulated annealing because it allows for a more extensive search for the global optimal solution. Gradually, the probability of accepting a worse solution decreases. The algorithm ends when a stopping criterion, such as a maximum running time or time without improvements, is reached and returns the best solution found during the entire search.

The local search heuristic approach by Van den Broek [26] does this, except that it does not try to find the global optimum. The heuristic ends when a feasible solution has been found or maximum running time has been reached. A shunt plan is feasible if and only if it satisfies all hard constraint, meaning it has no crossings, no track lengths are violated and the plan does not have any arrival- and departure delays. A crossing is defined as the undesired scenario in which the path of a moving train is blocked by one or more other trains. Track lengths are violated when the total length of the trains parked on a track exceed the length of the track. An arrival delay occurs when a train arrives during the movement of another train. Due to the fact that simultaneous movements are not allowed, it would mean that the arriving train has to stand still on a gateway-track, which is also not allowed. Departure delays occur when a train cant leave in time.

The search neighbourhoods are grouped by the sub-problems matching, service task scheduling, routing and parking. Search operators are defined for each neighbourhood. These search operators are able to search the solution space by creating different solutions. During exploration of the search space, new solutions are created continuously. For a new solution, it has to be determined whether it is preferable over the current solution. A new solution is preferred over the current solution if the new solution is closer to being feasible. Note that feasibility is usually expressed as either infeasible or feasible. To be able to measure 'closer to being feasible', hard constraints are modelled as soft constraints and the total number of movements in a shunt plan is added as additional constraint. Weights are assigned to all these constraints in order to calculate the cost of a shunt plan. A new solution is preferable over the current solution if the cost has decreased. The cost of a shunt plan is computed with the following equation:

$$\begin{aligned} Cost(p) = W_{cr} * \sum_{m \in p} CR_p(m) \; + W_{tlv} * \sum_{\tau \in T} TLV_p(\tau) + \\ + W_d * \sum_{d \in p} delay_p(d) \; + W_a * \sum_{a \in p} delay_p(a) + \\ + W_m * |\{m \mid movement\, m \in p\}| \end{aligned} \tag{2.1}$$

where (next page)

$CR_p(m)$ = the number of crossings of one particular movement m in shunt plan p

$TLV_p(\tau)$ = length of shunt trains occupying track τ exceeds the length of track τ

$delay_p(d)$ = the delay of departure train d

$delay_p(a)$ = the delay of arriving train a

W_x = the assigned weights to all four constraints above

The total cost cost of a shunt plan is used to guide the local search heuristic. Appendix C shows the pseudo code of the local search heuristic by Van den Broek [26]. The first step is creating an initial solution based on a set of arriving and departing trains. An initial solution is essential to move through the search space. The next section will elaborate on the construction of an initial solution.

2.2.1 A heuristic to generate initial solution

Every local search requires an initial solution to start with. One of the properties of simulated annealing is the tendency to move away quickly from the initial position in the search space, as the probability of accepting worse solutions is high at the start of the search. A simple sequential algorithm is presented to construct the initial solution, starting with the matching sub-problem. A perfect matching between the arriving train units (AT) and departing train units (DT) is constructed such that an arriving train unit is matched to a position on a departing train such that all tasks can be finished. The arriving and departing train units form a bipartite graph. A bipartite graph is a set of graph nodes separated into two disjoint sets such that no two graph nodes within the same set are connected. Clearly, the arriving and departing train units form two disjoint sets. The algorithm of Hopcroft-Karp [14] is used to produce a perfect matching between arriving and departing train units. This is visualized in Figure 2.3 for the example given in Section 2.1. Train unit 2 is assigned to the first departing train. The other train units to the second departure train. Arriving trains can be split and combined to form new train compositions. These temporary compositions are referred to as shunt trains after the matching has been applied.

A service task schedule is constructed next in a greedy way. The tasks and their departure date of each shunt train result from the matching in the previous step. In order of increasing departure date, the tasks are added one by one to the schedule of a corresponding resource. If a task can be assigned to multiple resources, the resource with the smallest total processing time is selected. The order of tasks of the same shunt train is determined by the earliest starting time of the tasks in their resource schedule.

In the initial plan each shunt train will be parked between service tasks. There has to be decided where each shunt train will be parked between arrival, service tasks and departure. The track

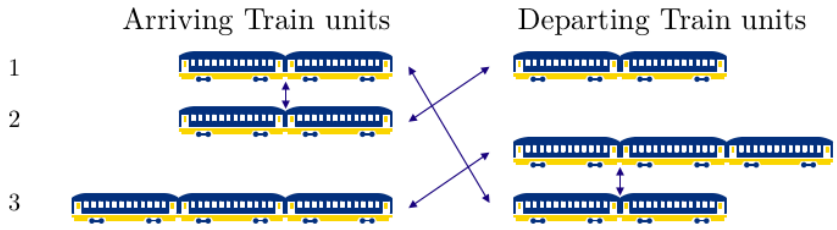


Figure 2.3: Hopcroft & Karp [14] matching algorithm example Section 2.1

for each parking interval is picked randomly from the tracks with sufficient free space during the entire interval, or assigned randomly if no such track is available. A movement node will be added for every activity on the service site. For each task, start and end time can be determined using the earliest starting time rule. These times are used to define a linear ordering on the movements.

Note that these initial solutions may not be feasible. The purpose of these initial solutions is that they contain all important features to be able to apply search operators to reach a feasible shunt plan. What is essential to any local search algorithm is a solution representation that properly captures all important aspects of the solution. The next section elaborates on the solution representation.

2.2.2 Solution representation in local search

The solution representation should properly captures all important aspects, while allowing for easy modification through the search operators. This is especially important for the Train Unit Shunting Problem with its tightly connected sub-problems. A shunt plan is modelled as a precedence graph. The graph for the example shunt plan constructed in Section 2.1 is shown in Figure 2.4. As can be seen, the graph structure for a simple example is already quite complicated.

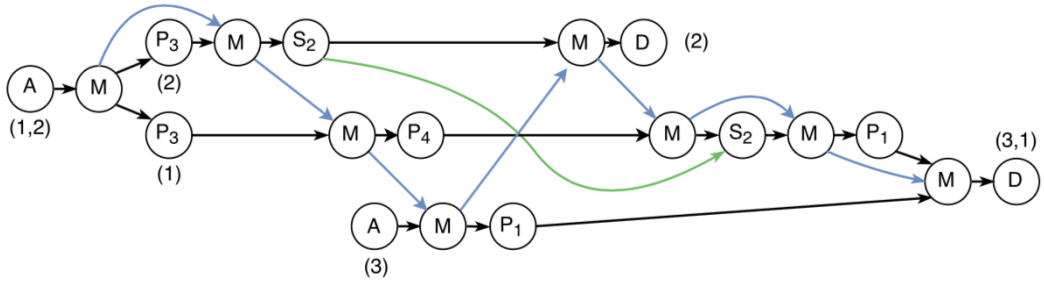


Figure 2.4: Activity graph of the example shunt plan in Section 2.1

Each activity is modelled as a node in a precedence graph to represent different characteristics. The resulting graph is a directed acyclic graph, where the edges indicate the precedence relations between the nodes. Multiple node types are distinguished based on their activity, each type having its own set of data associated to the node. The associated data is omitted from Figure 2.4 for clarity and described in Table 2.4. Table 2.4 shows all node types and corresponding data associated to the node.

All node types store information about the arrival side. This information is necessary to determine how shunt trains are positioned relative to each other. The edges in the graph express precedence

Node	Node type	Associated data	Subscript
A	Arrival	Arrival track, arrival time, train unit(s)	*None*
D	Departure	Departure track, departure time, train unit(s)	*None*
Sp	Split	Track, duration, train unit(s)	*None*
C	Combine	Track, duration, train unit(s)	*None*
S	Service task	Track, task type, duration, train unit(s)	*Specific task*
P	Parking	Track, duration, train unit(s)	*Parking track*
SM*	Saw Move	Track, duration, train unit(s); **not in Figure 2.4*	*None*
M	Movement	Start track, path, end track, duration, train unit(s)	*None*

Table 2.4: Node types, associated data and meaning of possible subscript

constraints of activities. The precedence constraints are either for the same shunt train or between a predecessor-successor-pair of shunt trains. The movement order (including splits and combines) and service orders are modelled explicitly in the activity graph (visualized as blue and green edges in Figure 2.4). As mentioned already, often there is only one train driver available at a service site. Considering this as a resource, train units are not allowed to move simultaneously. Therefore, movement order is explicitly modelled. The same accounts for service tasks assigned to the same resource. Solid, black edges represent the order of operations of one or more shunt trains. The blue edges determine the order of the movements, and the green edge indicates which service task is completed first. From this activity graph, the corresponding shunt schedule can easily be inferred. Table 2.5 shows a part of how the schedule inferred from Figure 2.4 would look like.

Start time	End time	Shunt train	Activity	Track(s)
12:00:00	12:02:30	(1,2)	Movement	Arrival –>3
12:02:30	12:04:30	(1,2) –>(1), (2)	Split	3
12:04:30	12:07:00	(2)	Movement	3 –>2
12:07:00	12:09:30	(1)	Movement	3 –>4

Table 2.5: Part of shunt schedule inferred from activity graph for example from Section 2.1

2.2.3 Feature extraction from solution representation

In a recent study at NS, Dai [9] proposed a method to predict if an initial solution will become feasible after applying the local search heuristic by Van den Broek [26], given a maximum computation time. Her approach involves extracting a broad category of features from an initial solution. extracting these features requires heavy features engineering based on extensive domain knowledge. Even though, her method proved that to be able to predict feasibility of an initial solution. As mentioned in Chapter 1, recent research [22, 33, 17] on graph classification has proven to achieve high accuracy predicting the class of an arbitrary graph without the need of extensive domain knowledge. Directly using the graph representation of an initial solution as input would overcome the need of extensive domain knowledge. Before moving on to recent developments on graph classification a formal description of graph classification is provided, followed by techniques necessary to perform graph classification.

2.3 Graph classification

Considering the problem of classifying graphs, a graph in this context is made up of nodes which are connected by edges. A graph is defined as $G = (V, E)$ having a finite set of nodes V and a finite set of edges E. Edges E are expressed as a tuple $\{u, v\}$ of nodes $u,\ v \in V$. In this research

project, a dataset D is used consisting of N graphs $G_i \in D$, where $i = 1, ..., N$ and $G_i = (V_i, E_i)$. Multiple node features are present on the nodes. Features include information about tracks, train units, duration's and activities. This information is explained in Table 2.4 in Subsection 2.2.2. Each graph $G_i \in D$ has a corresponding class $y_i \in C$ where C is the set of class labels given as $C = 1, ..., k$.

The goal of the graph classification task is to derive a mathematical formula f to perform $f : D \rightarrow C$ which can accurately predict the class label of each graph in the dataset. When f is derived using machine learning, a subset of D is used for which the class label is known. This subset D_{train} is referred to as the training set. The derived model is tested on the remaining subset of D, the test set D_{test}. The accuracy of the function is assessed by comparing the predicted label y_i' derived by $f(G_i)$ with the actual label y_i for all graphs in D_{test}. The problem at the core of learning classification functions on graphs is the ability to measure similarity between graph structures.

Probably the most natural measure of similarity between graphs is to check whether graphs are isomorphic. This means checking whether two graphs are the same graph, but drawn in a different way. Figure 2.5 should give an idea of the difficulty of telling if graphs are isomorphic. Despite the idea of checking graph isomorphism seeming quite intuitive, no efficient algorithms are known for it. The graph isomorphism problem is NP-hard, but has been neither proven NP-complete nor found to be solved by a polynomial-time algorithm [12]. Besides being computationally expensive, similarity measures based on graph isomorphism are too restrictive in the sense that graphs have to be exactly identical or contain large identical subgraphs in order to be deemed similar by these measures [28].

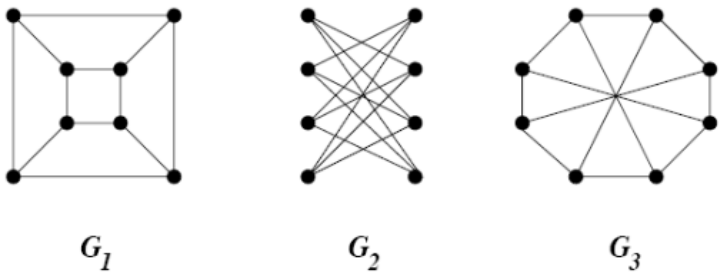


Figure 2.5: Isomorphic graphs G1, G2 and G3

The most common test of isomorphism is the Weisfeiler-Lehman test. The Weisfeiler-Lehman test proceeds in iterations and every iteration consists of four steps. These steps are best explained by showing them. Figure 2.6 shows two labeled graphs G and G' for which isomorphism has to be checked. In iteration 1, node labels of neighbouring nodes are appended to the original node labels for both graphs G and G'. Then, the appended labels are sorted alphabetically and compressed into new, short labels. The last step of an iteration consists of inserting the new labels into graphs G and G', which can be seen on top of iteration 2.

At the end of an iteration, the counts of the original node labels and the counts of the compressed node labels are represented as a feature vector. If those vectors are not identical after an iteration, the Weisfeiler-Lehman algorithm terminates. Meaning that the graphs are not isomorphic. If the sets are identical after n iterations, it means that graphs G and G' are isomorphic, or the algorithm has not been able to determine that they are not isomorphic. Figure 2.6 shows that graphs G and G' are still considered as isomorphic after one iteration. When another iteration is applied, the feature vectors of graphs and are different. The graphs are not isomorphic. The 1-dimensional Weisfeiler-Lehman test has been shown to be a valid isomorphism test for almost all graphs [4].

The Weisfeiler-Lehman test only checks for isomorphism. This makes this test not suitable per-

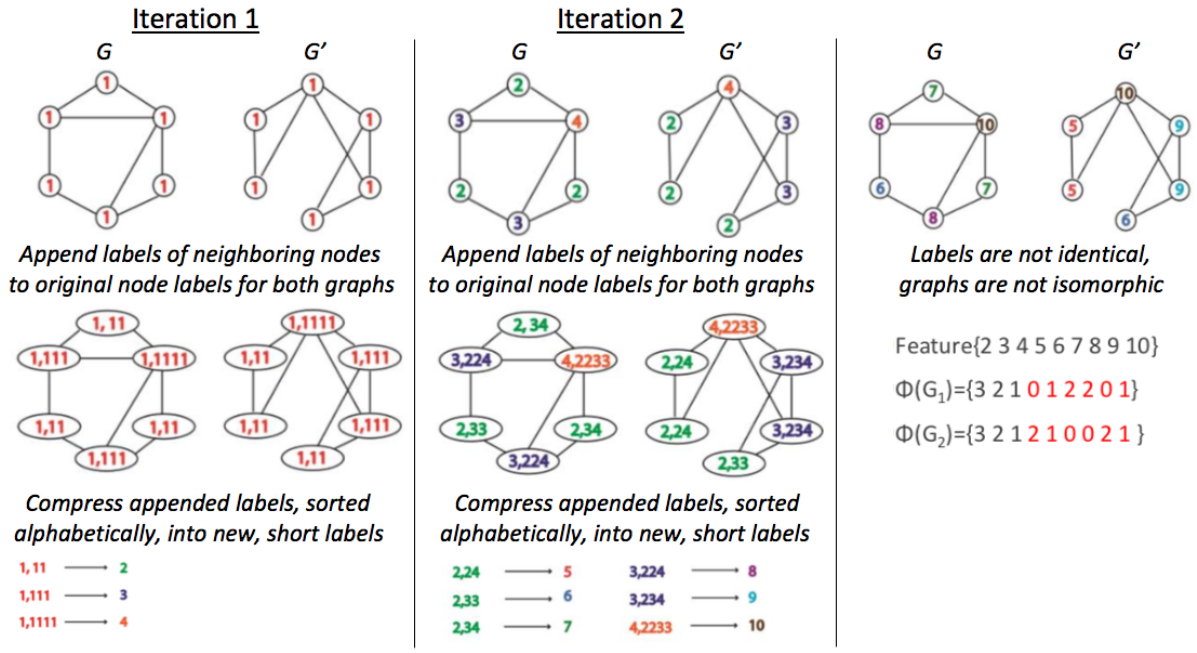


Figure 2.6: *Illustration of the computation of the Weisfeiler-Lehman test of isomorphism*

forming classification tasks. Instead of checking whether graphs are completely similar, several methods exist measuring to what extent two graph structures are similar. These kind of methods are suitable for classification problems. The methods can be divided into two categories:

1. Spatial features: **graph kernel methods** computing inner products on graphs;
2. Spectral features: **topological methods** deriving topological features from graphs

The next two sub sections zoom in on both categories, explaining how features are derived and elaborating on applications from literature.

2.3.1 Graph kernels

Graph kernels can be understood as functions measuring the similarity of pairs of graphs. Graph kernels restrict themselves to comparing substructures of graphs that are computable in polynomial time. Three classes of graph kernels have been defined in machine learning: graph kernels based on walks and paths, subgraphs and subtree patterns. Graph kernels on walks and paths, compute the number of matching pairs of random walks in two graphs. The second class, graph kernels based on subgraphs compares pairs of graphs by counting limited-size subgraphs of large graphs. The third class, subtree kernels, compares all matchings between neighbours of two nodes from two graphs. The applications of graph comparison with graph kernels are: function prediction of protein structures, analysis of semantic structures in Natural Language Processing and comparison of UML diagrams.

It is a general limitation of all graph kernels that they scale poorly to large, labeled graphs with more than 100 nodes. The efficient comparison of large, labeled graphs remained an unsolved challenge for almost a decade [28]. In 2011, Shervashidze et al. [28] presented a general definition of graph kernels that includes many previously known graph kernels. Their contribution is based

on the Weisfeiler-Lehman (WL) test of isomorphism. Their kernels are competitive in terms of accuracy with state-of-the-art kernels on several classification benchmark data sets. Moreover, the runtime of the kernels on large graphs outperform other kernels, also on discretely labeled graphs.

Figure 2.6 has shown that the two graphs in the example would be identified as non-isomorphic by the Weisfeiler-Lehman test after the second iterations, as their label sets are different. However, the WL subtree kernel [28], checks the similarity between graphs by using the WL algorithm. The degree to which the feature vector representations at the end of n iterations are similar measures the similarity between graphs. The intuition is that two graphs are similar if they have many common subtrees rooted at their nodes, which are characterized by labels. The WL subtree kernel counts the common labels until iteration h in order to compare two graphs. Classification algorithms use the same feature vectors to perform classification tasks. All graphs are transformed into feature vectors of fixed length using the WL subtree kernel where each feature vector from graph $G_i \in D$ has a corresponding class y_i.

2.3.2 Topological features

The other category of methods compares graphs based on topological features. Numerous features can be extracted from graphs. These features involve global graph features as well as local node features. Some examples of global features are total number of components, total number of edges and total number of triangles. The number of triangles is the number of nodes which form a triangle, with a triangle being a set of three nodes with an edge between every member. Local features are, among others, PageRank score [24], total degree and local clustering score. These features again form a feature vector representing each graph $G_i \in D$. extracting these features involve expensive multiplications with the eigenvector matrix of the graph Laplacian. Even though the computational burden, the technique has been used successfully for many graph mining tasks including graph similarity measurement [5], time series anomaly detection [2] and link prediction [1].

Li et al. [21] proposed a novel method of classifying graphs into feature vectors based on the extraction of global and label based features. After extracting the features, a Support Vector Machine (SVM) is used to classify the graphs. The work presents results on the classification of three different graph datasets including chemical compound graphs, protein graphs and cell graphs. The approach is shown to be more accurate than state of the art graph kernel based methods. However, the approach is not extended to datasets in which multiple classes may be present and is missing the potentially rich descriptive features at the node level.

Bonner et al. [5] proposed a method which is able to include descriptive features at the node level. They proposed a Deep Topology Classification (DTC) approach. DTC is an approach to the classification of massive complex graph datasets. DTC extracts both global and local features from a graph. Local features are extracted on node level, while global features are extracted on graph level. Local features on node level involve eigenvalues, PageRank and two-hop neighbours. For every local feature the following metrics are extracted: median, mean, standard deviation, variance, skewness, kurtosis, minimum and maximum value. Besides local features, some global features are extracted such as number of edges and number of triangles. These local and global features are used to create a feature vector of fixed length for each graph $G_i \in D$. They trained an Artificial Neural Network (ANN) with three hidden layers on these input vectors to classify the graphs. The results show that the combination of extracting local and global features is very effective. The approach is shown to have over 99% classification accuracy after k-fold cross validation across binary and multi-class datasets. They believe that the classification accuracy offered by the DTC approach will have many applications within the field of graph mining. Among these applications are the ones mentioned in Subsection 2.3.1. That's why the extraction of topological features from graphs in this research project seems very promising.

Both graph kernel methods and topological methods extract feature vectors such that pairs of graphs can be compared. These feature vectors can also be used to perform classification tasks. The next section will elaborate on machine learning techniques performing these classification tasks.

2.4 Machine learning on graphs

Many different machine learning algorithms are suitable to perform classification tasks. However, later in this chapter will be elaborated on one particular algorithm designed for graph classification. Therefore, this chapter will further elaborate on only neural networks and in particular, Convolutional Neural Networks.

2.4.1 Neural Networks

Neural networks are a set of algorithms, modelled after the human brain, designed to recognize patterns. They interpret sensor data through a kind of machine perception, labelling or clustering raw input. The patterns they recognize are numerical, contained in feature vectors, into which all real-world data must be translated. This real-world data can be for example images, sound, text or time series.

Neural networks are composed of interconnected, simple processing elements called artificial neurons. Neural networks can have one or more layers of neurons. Multi-Layer Perceptron (MLP) networks consist of an input layer, one or more hidden layers and an output layer. Each layer has a number of processing units and each unit is fully interconnected with weighted connections to units in the subsequent layer. The MLP transforms inputs to outputs through some non-linear functions. In Figure 2.7 a MLP with two hidden layers is depicted.

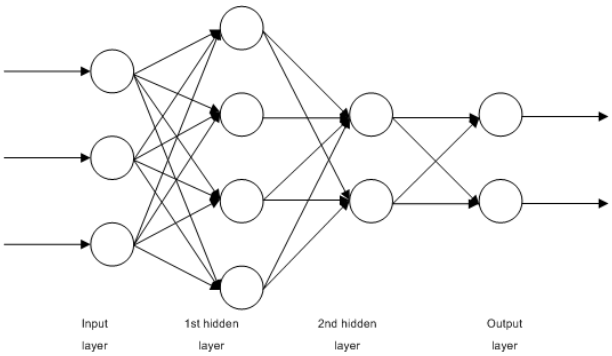


Figure 2.7: Example of a Multi-Layer Perceptron network with two hidden layers

Neural networks are typically designed to deal with data represented as feature vectors. In this research project we are dealing with graphs as input data, which lack input in this representation. Subsections 2.3.1 and 2.3.2 described how graph data can be transformed to a feature vector. But before we can dive into neural networks that can deal with graphs, it's necessary to understand Convolutional Neural Networks (CNN) since machine learning based graph classification is build around the CNN principle.

2.4.2 Convolutional Neural Networks

CNN architectures make the explicit assumption that the inputs are images. Taking an input image and outputting a class is tasked as image classification. When a computer takes an image as input, it will see an array of pixel values. This array depends on the resolution and size of the image. Let's say we have a color image and its size is 480 x 480. The representative array will be 480 x 480 x 3. The number three refers to RGB values. Each of these numbers is given a value from 0 to 255 which describes the pixel intensity at that point. These numbers are the only inputs available to the computer. The idea is that you give the computer this array of numbers and it will output the probability of the image being a certain class. Using traditional neural networks for real-world image classification is impractical for the following reason: consider the same color image mentioned before. The number of input nodes would already be over 600.000 ($480 * 480 * 3$). This number would increase rapidly by adding hidden layers with multiple nodes. Let's say the first hidden layer has 20 nodes then the size of the matrix of input weights would be 12 million. If we increase the number of layers, the number increases more rapidly. Besides, vectorizing an image completely ignores the complex 2D spatial structure of the image [29].

A CNN consists of an input and an output layer, as well as multiple hidden layers. The hidden layers of a CNN typically consist of so-called convolutional layers, activation layers, pooling layers and fully connected layers. The different layers will be explained according to a simple example image of a bird with 32 x 32 pixels.

The convolutional layer

The first layer is always a convolutional layer. It is the core building block of a CNN. Imagine a spotlight shining over the top left of the example image. The spotlight covers a 5 x 5 pixel area. After shining over the top left area, the spotlight moves such that it slides across all the areas of the input image. In CNN this spotlight is called a filter or kernel. The area the filter covers is called the receptive field. The pixels of the filter are also an array of values in the same way as the computer sees the example image as an array of pixel values. Imagine again that the filter is covering the top left corner. It multiplies the values in the filter with the values of the pixels in the image it covers, called element wise multiplication. These multiplications are summed up into a single number. This process is repeated for every location of the example image by sliding the filter to the right by s units where s is called stride. After sliding the filter over all locations, the result is a 28 x 28 array of single numbers. This array is called the activation or feature map. The reason that the feature map is smaller than the input image is because there are a limited amount of locations a 5 x 5 filter can reach. Figure 2.8 visualizes the convolutional layer producing a feature map.

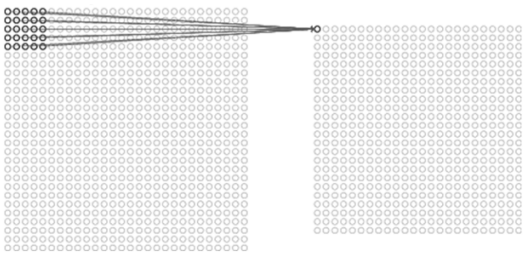


Figure 2.8: Visualization of a convolutional layer

The more filters, the greater the depth of the activation map, and the more information we have about the input volume. We could for example use 10 filters with a 5 x 5 pixel area with different pixel values. Convolutional networks exploit spatially local correlation by enforcing a local connectivity pattern between neurons of adjacent layers: each neuron is connected to the receptive field of the input image.

The activation and Pooling layers

After a convolutional layer it is conventional to apply a nonlinear layer with the purpose to insert non-linearity. The element wise multiplications and summations were just linear operations and a activation layer inserts non-linearity which helps to alleviate the vanishing gradient problem. The vanishing gradient problem would cause the lower layers of the network to train very slowly because the gradient decreases exponentially. The activation layer applies a function that changes all the negative activations to 0. This increases the nonlinear properties of the model.

After one or more activation and/or convolutional layers can be decided to apply a pooling layer. The function of pooling layers is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. Max-pooling is the most popular pooling layer. It takes a filter, normally of size 2 x 2, that moves over the input volume. It outputs the maximum number for the field it covers. Figure 2.9 shows how a max-pooling layer works.

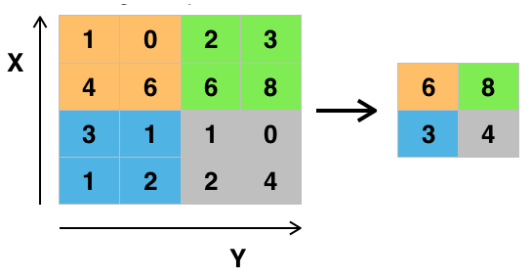


Figure 2.9: Example of a max-pooling layer

The pooling layer drastically reduces the spatial dimension of the input volume with two goals: 1. Computation cost are decreased because the amount of parameters is reduced by 75% and 2. It will control for overfitting.

Fully connected layers

The fully connected layer is the end of the network. The input volume is the output of the previous layer (activation or Pooling). The output is an N dimensional vector where N is the number of classes. For example of you want to classify between a bird, sunset, dog, cat or chicken, N would be 5. Each number in the vector represents the probability of a certain class. The fully connected layer determines which features most correlate to a particular class. In the example image of a bird, it will have high values in the feature maps that represent high level features like wings. In the example the resulting vector could be [0.75, 0, 0.05, 0, 0.2]. Meaning that there is a 75% change that the image represents a bird. Figure 2.10 shows the full CNN.

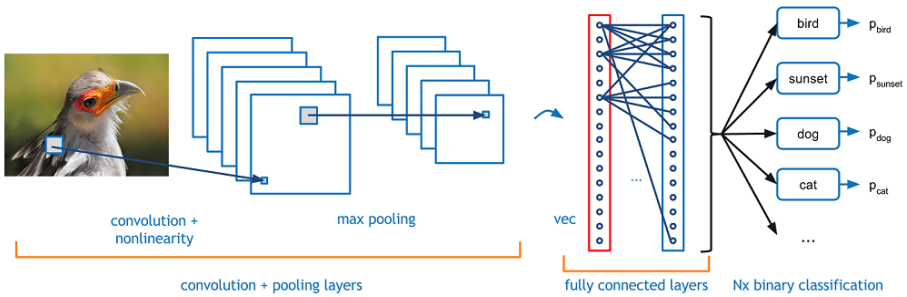


Figure 2.10: Convolutional Neural Network with convolutional, pooling and fully connected layers

The CNN architecture served as inspiration for another type of neural network, called Graph Convolutional Neural Networks. These neural network architectures accept graphs of arbitrary structure in a similar way as a CNN takes images as input. The next chapter is dedicated to these type of neural networks.

2.4.3 Graph Convolutional Neural Networks

A CNN can traverse a pixel sequence (top-left to bottom-right) generating receptive fields for each of the pixels. The sequence is implicit due to the spatial order of the pixels. For graph structured data an implicit ordering is missing. Therefore, two problems have to be solved: (1) Determining the node sequences for which neighbourhood graphs are created and (2) computing a unique mapping from a graph representation into a feature vector of fixed length for every graph in the dataset.

To the best of our knowledge, the first study conducted on Graph Convolutional Neural Networks was by Niepert et al [22]. Their approach, called PATCHY-SAN, addresses these two problems for arbitrary graphs. They proposed a framework for learning convolutional neural networks for arbitrary graphs. These graphs may be undirected, directed, and with node and edge attributes. They presented a general approach to extract local connected regions from graphs, like image-based convolutional networks operating on local connected regions from images. This approach applies three steps to each graph: node sequence selection, neighborhood assembly and graph normalization. These steps are visualized in Figure 2.11.

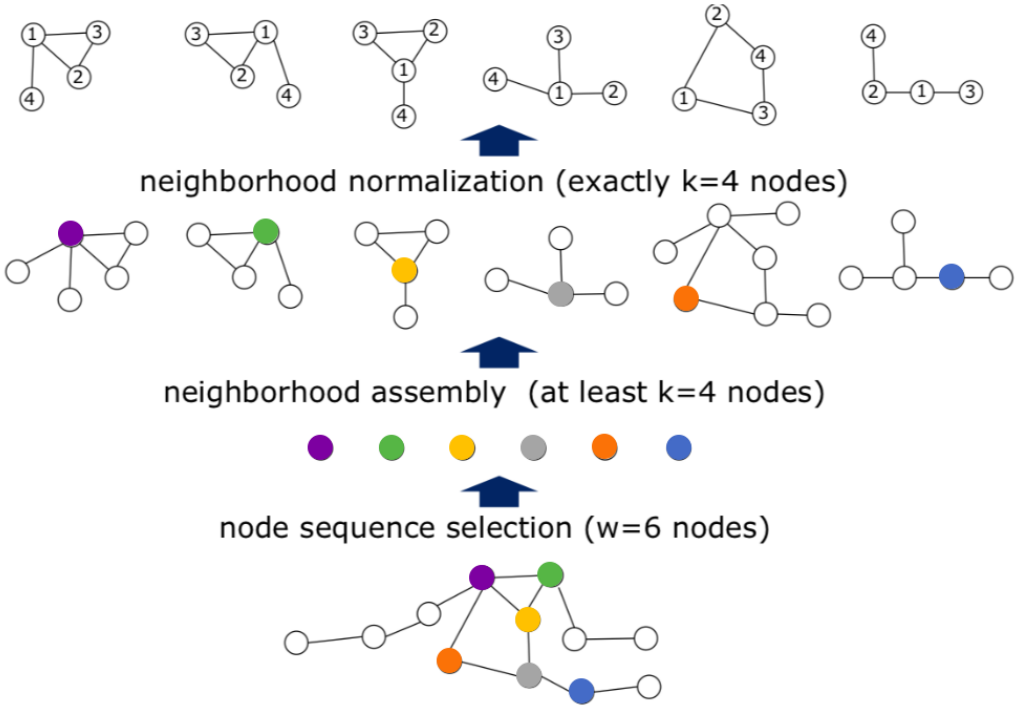


Figure 2.11: An illustration of the PATCHY-SAN architecture proposed

Node sequence selection is the process of identifying a sequence of nodes for which receptive fields are created. First, the nodes of the input graph are sorted with respect to a given graph labelling. Niepert et al. haven't specified which graph labelling can be used. Second, the resulting node sequence is traversed given stride s. A receptive field is constructed for each visited node. To

be able to construct a receptive field, a local neighbourhood has to be assembled for the input node. During the neighbourhood assembly step a fixed-size neighbourhood for each node in the selected sequence is assembled. The graph normalization step imposes an order on the nodes of the neighbourhood graph so as to map from the unordered graph space to a vector space with a linear order. This step isn't necessary for our shunting graphs, because their order is already determined. The last step, convolutional architecture, learns neighbourhood representations with convolutional neural networks from the resulting sequence of patches.

In contrast to graph kernels, mentioned in Subsection 2.3.1 , PATCHY-SAN learns substructures from graph data instead of predefined substructures beforehand. Graph kernels have a training complexity at least quadratic in the number of graphs [28], which is restrictive for large-scale problems. PATCHY-SAN scales linearly with the number of graphs. Experiments show that the approach is competitive with state of the art graph kernels such as the shortest-path kernel [6], the random walk kernel [31], the graphlet count kernel [27] and the Weisfeiler-Lehman subtree kernel [28]. PATCHY-SAN is between 2 and 8 times more efficient than the most efficient graph kernel. It is highly competitive on graph data and expected is the performance advantage to be much more pronounced for data sets with a large number of graphs.

Kipf and Welling [17] presented a scalable approach for semi-supervised learning on graph-structured data that is based on an efficient variant of convolutional neural networks which operate directly on graphs. They based their convolutional architecture on a localized first-order approximation of spectral graph convolutions. The model scales linearly in the number of graph edges and learns hidden layer representations that encode both local graph structure and features of nodes. They demonstrated that the approach outperforms related methods by a significant margin. The approach by Kipf and Welling [17] and Atwood and Townsley [3] achieve state-of-the-art results on node classification. When dealing with large graphs, such as those that arise in the context of online social networks, a subset of nodes may be labeled. These labels can indicate demographic values, interest or other characteristics of the nodes. A core problem is to use this information to extend the labeling so that all nodes are assigned a label. This is referred to as node classification. The approaches of Kipf and Welling and Atwood and Townsley perform relatively worse on graph classification tasks.

The most relevant study for this research project is conducted by Zhang et al. [33]. They proposed a novel neural network architecture accepting graphs of arbitrary structure. Given a dataset containing graphs in the form of (G, y) where G is a graph and y is its class, they aimed to develop neural networks that read the graphs directly and learn a graph classification function. Their architecture addressed two main challenges: 1. how to extract useful features characterizing the rich information encoded in graph classification and 2. how to sequentially read a graph in a meaningful and consistent order. They designed a Deep Graph Convolutional Neural Network (DGCNN). Their approach differs from the approach by Niepert et al. [22] because the word *deep* comes in. DGCNN is not a deterministic graph compression algorithm. It *learns* how to represent graphs after every training iteration. The feature vector changes because DGCNN learns a better representation with the training data. DGCNN has three sequential stages: (1) graph convolution layers extract local substructure features from nodes and define a consistent node ordering, (2) a SortPooling layer sorts the node features under the previously defined order and unifies input sizes and (3) traditional convolutional and dense layers read the sorted graph representations and make predictions. Figure 2.12 shows the overall structure of DGCNN.

To address the first challenge, they designed a localized graph convolution model consisting of four steps. First, a linear transformation is applied to the node information matrix. Matrix X denotes the graph's node information with each row representing a node. Columns in X are called *feature channels*. By multiplying X with W, where W is a matrix of trainable graph convolution parameters, the feature channels c are mapped to c' channels in the next layer. Second, node information is propagated to neighbouring nodes as well as the node itself by multiplying the outcome of the first step with the adjacency matrix of the graph. Third, normalizes each row

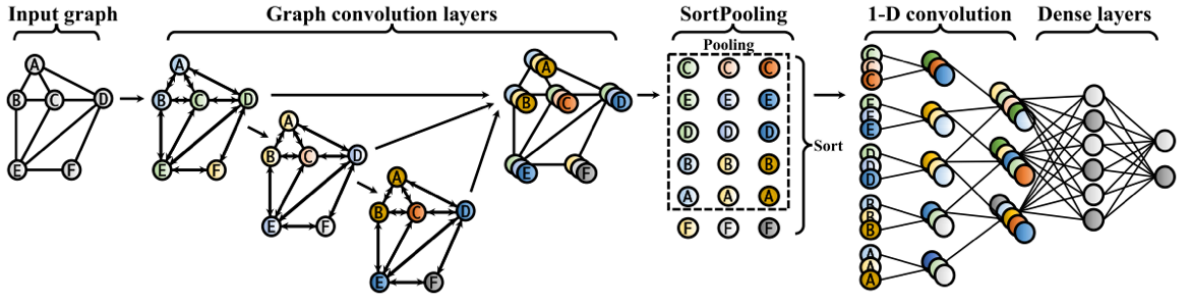


Figure 2.12: The overall structure of DGCNN. An input graph of arbitrary structure is first passed through multiple graph convolution layers where node information is propagated between neighbors. Then the vertex features are sorted and pooled with a SortPooling layer, and passed to traditional CNN structures to learn a predictive model. Features are visualized as colors.

in order to keep a fixed feature scale after graph convolution. The last step applies a point-wise nonlinear activation function f and outputs the graph convolution results. Their graph convolution model mimics the behaviour of a popular kernel, the WL subtree kernel, explaining its graph-level classification performance.

To address the second challenge, they designed a pooling layer, called SortPooling which sorts graph nodes in a consistent order so that traditional neural networks can be trained on the graphs. As mentioned before, shunting graphs are already sorted in consisting order by ordering them on ascending start time of the nodes. The other function of SortPooling is to unify the sizes of the output tensors (pooling layer). The intention is to unify graph sizes, making graphs with different numbers of nodes unify their sizes to k.

Experiments on benchmark graph classification datasets demonstrate that the proposed architecture achieves highly competitive performance with state-of-the-art graph kernels and other graph neural network methods.

The neural network architecture of Zhang et al. [33] seems like the most relevant study because it takes adjacency matrices, feature matrices and its class label of arbitrary graphs as input and shows highly competitive results on graph classification.

2.5 Limitations in literature and final conclusions

No literature is found on graph classification methods supporting in solving optimization problems. All methods designed as of today, are applied on graph data within other fields such as bioinformatics and social networks. This research project bridges the gap between theory and practice, and more specifically the practice towards the possibility of incorporating a graph classification method in a real operational context; train unit shunting.

Scientifically, this research project closes a part of the gap that is currently existing in literature of machine learning in operations research. Directly using the graph representation as input overcomes the need of extensive domain knowledge in solving an optimization problem.

Lastly, the research provides practical benefits for NS through its contribution at improving the local search heuristic. It does this by making the heuristic faster through predicting feasibility of initial solutions and making it more efficient if it is able to guide it to more promising areas.

Chapter 3

Data generation and preparation

This chapter involves the parts data understanding and data preparation of the CRISP-DM cycle, none of the sub research questions will be answered here. First, it elaborates on how the data is generated with the tools available at NS. Next, a descriptive analysis is performed to create a summary of the data to yield a useful understanding of the data. Then, the data is prepared for modeling. Data preparation is a time-consuming, but important, part of the research project.

3.1 Data generation

The local search heuristic is part of a larger algorithm consisting of several components. Figure 3.1 visualizes the components. It starts with the instance generator. Subsequently, the initial solution generator constructs an initial solution based on output from the instance generator. Finally, the local search heuristic can be applied with the initial solution as starting point.

Figure 3.1: Sequential components algorithm

The instance generator is a parameterizable program, developed by NS, which derives instances for the train unit shunting problem automatically. Instances drawn from the instance generator can be specified according to a set of input parameters. Each individual service site has a set of standard input parameters. These parameters are based on the day-to-day schedule at that service site. Parameters can be changed to test different scenarios at a service site. The most important parameters are:

1. The number of train units
2. The different train unit types and subtypes
3. Probability distributions of arrivals per train unit type
4. The set of service tasks including duration

The instances generated for this research project are based on one of the service sites operated by NS called 'the Kleine Binckhorst', which is situated near The Hague Central Station. The Kleine Binckhorst is a service site of medium size, and consists mostly of free tracks. An overview of this service site is provided in Figure 3.2. Dai [9] found during her research that the heuristic relatively easily solved instances with less than 21 train units. While the majority of instances with more than 21 train units were not solved by the heuristic. 21 train units provides a balanced dataset and is the hardest to correctly classify as either feasible or infeasible. As for the rest of the parameters, the standard ones are used. Meaning that there are two different types of train units arriving at the service site. Both train types have two corresponding subtypes. Regarding the set of service tasks, five different tasks can be performed at the service site: internal cleaning, external cleanings (soap and oxalic) and technical check-ups (small and large).

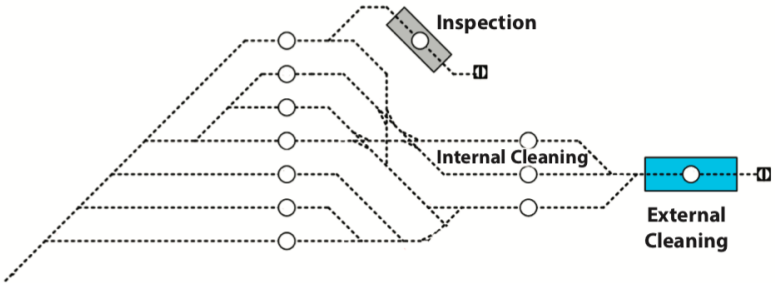


Figure 3.2: Schematic illustration of the kleine Binckhorst, with specific tracks for inspection and cleaning activities. Image adjusted from www.sporenplan.nl

The output of the instance generator is a set of arriving trains (AT), a set of departing trains (DT) and a set of service tasks for each train unit that has to be performed. For both (AT) and (DT), train composition, train units and arrival/departure time are specified. The set of service tasks contains a list of service tasks for each train unit that has to be done in the time that the train unit is present on the service site.

The output of the instance generator serves as input for the initial solution generator. Construction of the initial solution is described in Subsection 2.2.1. The initial solution serves as starting point for the local search heuristic. The maximum runtime to find a feasible solution is set to 300 seconds. Only if the heuristic is able to find a feasible solution within the maximum runtime, the output is a shunt schedule. If no feasible solution is found, there is no output. To collect the data necessary to carry out this research project, the algorithm is such that it outputs the initial solution and feasibility label. The output is delivered as JavaScript Object Notation (JSON). JSON is a lightweight data-interchange format which is easy to read and write for humans and easy to parse and generate for computers. Figure 3.3 shows the format of JSON.

```
{
    "key": "value",
    "key": "value",
    "key": [
        {
            "key": "value",
            "key": "value"
        },
        {
            "key": "value",
            "key": "value"
        }
    ]
}
```

```
{
    "name": "John",
    "age": 30,
    "cars": [
        {
            "brand": "Tesla",
            "type": "Model S"
        },
        {
            "brand": "Ford",
            "type": "Fiesta"
        }
    ]
}
```

Figure 3.3: JSON file format

This format has been chosen because the local search heuristic has been written in C Sharp (C#) programming language and this research project will be done in Python 3.6. JSON files can be easily generated in C# and parsed in Python. JSON files contain key/value pairs. Figure 3.3 shows that a key is the name of a person and John is the value belonging to that key. Keys can also contain key/value pairs, like with the cars john owns. JSON files in this research project have the same basic format, although the structure is more complex due to the amount of information an initial solution contains. JSON files in this research project contains the keys: node information, precedence constraints, feasible, number of iterations and runtime. Figure 3.4 shows a small part of an initial solution.


```
{
    "node_information": [
        {
            "node_id": 1,
            "node_type": "arrival",
            "track": "gateway_track",
            "train_units": [9409, 9408],
            "start_time": 12:00:00,
            "duration": 00:00:00,
        },
        {
            "node_id": 2,
            "node_type": "movement",
            "path": ["gateway_track", "track_3"],
            "train_units": [9409, 9408],
            "start_time": 12:00:00,
            "duration": 00:00:00,
        },
        {
            "...": "..."
        },
    ],
    "precendence_constraints": [(1,2), (2,3), ...],
    "feasible": "value", True
    "number_of_iterations": 343,
    "runtime": 124
}
```


Figure 3.4: JSON file initial solution

Every node represents an activity with data associated to it. Figure 3.4 shows an arrival and movement node. An arrival node contains the key 'track', while a movement node contains the key 'path'. The values are specific for every node. Precedence constraints contains a list of tuples. Each tuple states a precedence relation between two nodes. The movement in the example (node_id = 2) can only happen if the arrival (node_id = 1) has taken place. The key 'feasible' is a Boolean. False means the local search heuristic could not find a feasible solution within the maximum runtime. True means local search was able to find a feasible solution. This will also be the label y_i that belongs to each graph $G_i \in D$ in this research project, where $i = 1, ..., N$ and $G_i = (V_i, E_i)$. Number of iterations states the number of iterations the heuristic applied from the initial solution until a feasible solution was found or until maximum runtime was reached. Runtime is the time in seconds needed to find a feasible solution or, if no feasible solution was found, runtime is 300 seconds.

The largest dataset in the studies on graph classification contains 10.000 graphs [5]. For this research project also that amount of graphs is generated by running the algorithm, as visualized figure 3.1, for 10.000 times. Every time a new instance is drawn from the instance generator with the standard settings for service site the Kleine Binckhorst with 21 train units. The next section will be about descriptive analytics, providing a deeper look at the generated data to yield useful information.

3.2 Data understanding: descriptive analytics

The most important findings in terms of descriptive analytics are described in this section. From this point on a distinction will be made between feasible and infeasible instances. Feasible instances are initial solutions solvable by the local search heuristic within the maximum runtime of 300 seconds. Infeasible instances are the ones the heuristic could not solve. Note that not solvable by the local search heuristic does not mean it is definitely unsolvable. With an infinite amount of time, most initial solution would probably be solved the heuristic. For time purposes, 300 seconds will be set as maximum runtime.

Figure 3.5 visualizes the distribution of iterations for both feasible and infeasible solutions. Regarding feasible solutions, the minimum and maximum number of iterations in local search are 108 and 2599 respectively with an average of 733 iterations. The minimum and maximum number of iterations for infeasible solutions are 1057 and 2939 with an average of 1962 iterations. Clearly, the number of iterations for infeasible solutions are much higher because local search ran for the maximum time of 300 seconds and was not able to find a feasible solution.

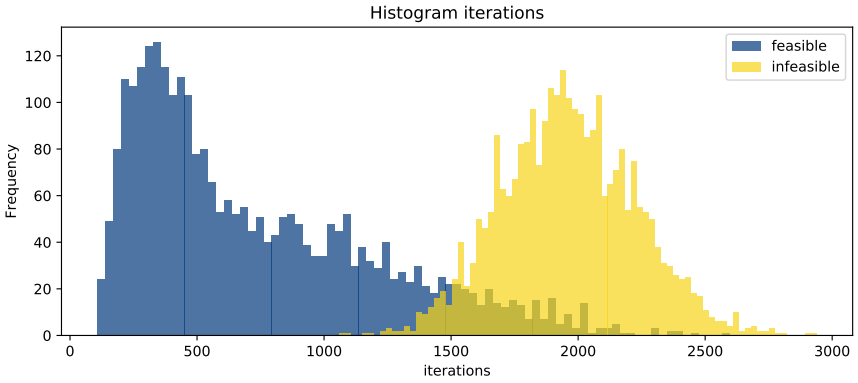


Figure 3.5: Distribution of iterations for feasible and infeasible solutions

The runtime per iteration is not fixed, since the runtime for infeasible instances is always 300 seconds and the number of iterations differs a lot. Figure 3.6 shows a scatterplot with the number of iterations on the x-axis and runtime on the y-axis. The runtime of feasible instances increases as the number of iterations increases. The spread in the beginning is small, meaning that the time per iteration is quite similar. As the runtime increases, the spread becomes larger.

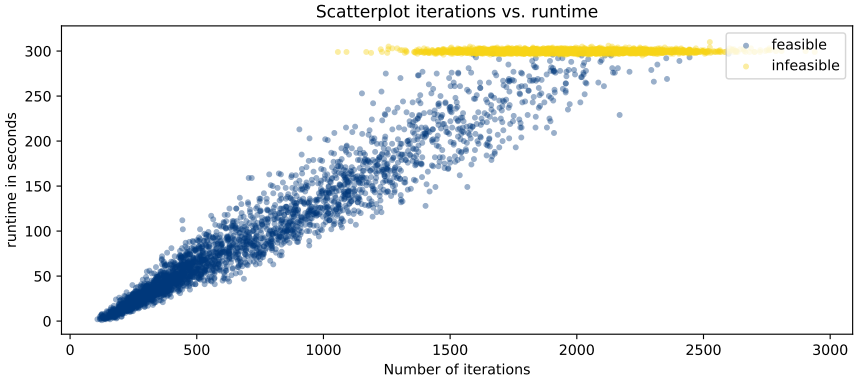


Figure 3.6: Scatterplot iterations versus runtime

Figure 3.7 shows a histogram of the runtime for all feasible instances. Infeasible instances are omitted for clarity because their runtime is always around 300 seconds. Considering feasible instances, the minimum runtime is 1 second, while the maximum runtime is 300 seconds. The average runtime is 96 seconds. ±80% of all feasible instances has been found within 150 seconds.

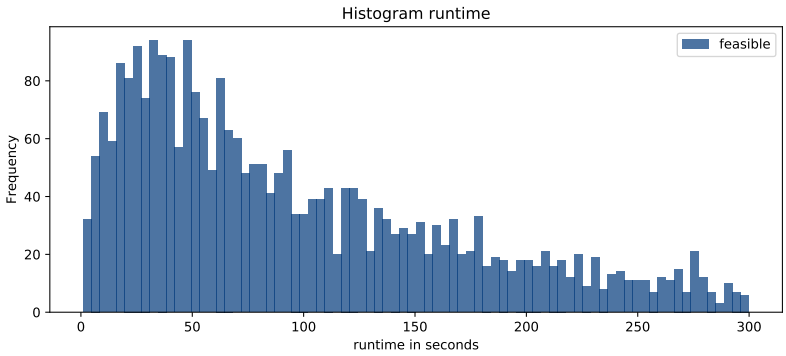


Figure 3.7: Histogram of the runtime (in seconds) for feasible solutions

Figures 3.8 shows two box plots. Box plots display variation in instances of a statistical population without making any assumptions of the underlying statistical distribution. These plots visualize the differences in amount of nodes for node types M (movements) and Sm (saw moves) between feasible and infeasible instances. As mentioned in Section 2.2, a shunt plan with minimal moves is preferred by the planners at NS. More movements require more resources which makes it more difficult to find a feasible shunt plan. The amount of movement and saw move nodes is due to the choices made when creating an initial solution. Meaning that the amount of movement and saw move nodes indicate the 'badness' of an initial solution. The box plots show that the minimum and maximum number of nodes for feasible instances are always equal or higher than for infeasible instances. This indicates that initial solutions for infeasible instances have consistently more movement and saw move nodes than initial solutions for feasible instances. This could be an indication why local search can not find a feasible solution in limited time for those initial solutions.

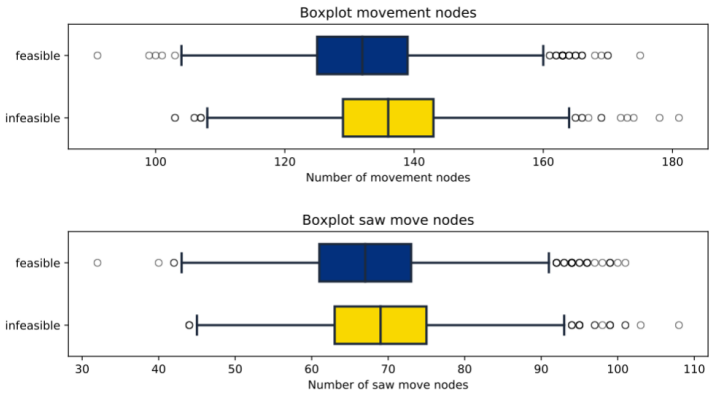


Figure 3.8: Boxplot showing the difference in movement and saw move nodes between feasible and infeasible instances

The difference in the number of nodes could also be an indication that graph classification will provide satisfactory results. These methods are build upon appending neighboring node labels to original node labels. If noteworthy differences exist in nodes between feasible and infeasible instances, the resulting feature vectors will be distinguishable.

3.3 Data preparation

Three data preparation tasks have been applied: class balancing, data construction and data formatting. As the data was originally designed for state representation during local search, it needs to be processed to be ready for modeling. In this section knowledge of previous chapters and findings in literature are used to prepare the data in such a way that its quality is as high as possible for the modeling phase.

3.3.1 Class balancing

Class imbalance occurs when one class represents the majority of the data points. For example, a classification task could be to diagnose 100 patients with a disease present in 20% of the general population. If a model simply labels every patient as negative (no disease), the accuracy would be 80%. That looks impressive, but no hospital would like to use this model. This is an imbalanced classification problem: the positive class, patients having the disease, is greatly outnumbered by the negative class. The same problem applies to this research project. The negative class, initial solutions that turned out to be infeasible, is outnumbered by the positive class. 2.795 are infeasible, while 7.205 are feasible.

Commonly used methods dealing with class imbalance can be divided into two categories: undersampling and oversampling. Undersampling is taking a subset of instances from the majority class to create a smaller set giving a more balanced dataset regarding the minority class. Oversampling is aimed at increasing the instances of the minority class giving a more balanced dataset regarding the majority class. Oversampling can be done by randomly duplicating existing instances or by creating synthetic instances based on existing instances. Synthetic oversampling techniques, like SMOTE [7], can not be applied on graph structured data. SMOTE can be broken down into three steps. Randomly pick a point from the minority class and compute the k-nearest neighbors. For example, compute its 5-nearest neighbors. For each neighbor, compute the vector connecting the chosen point to its neighbor and add a new point somewhere along that line. This is easily done for 2-dimensional data or data with continuous features, but not to synthetically oversample the minority class of infeasible instances. The risk of undersampling is loss of information due to removing potentially important instances. Whereas random oversampling increases the possibility of overfitting. Besides the risk of overfitting, timing of oversampling is also critical as it can effect the generalization ability of a model. A model will be trained on a subset of the data and the performance is validated on the remaining subset, the test set. Since, one of the primary goals of model validation is to estimate how it will perform on unseen data, the moment of oversampling is critical. By oversampling before splitting the dataset in a training and test set, information is leaked from the test set into the training of the model. Instances could end up in both the training and test sets. A complex enough model will be able to predict those instances perfectly, unfairly increasing accuracy. Figure 3.9 below visualizes the incorrect and correct way of combining oversampling and dataset splitting. In the example, the minority class consists of two instances. Left: incorrect, oversampling before splitting, ending up with an instance in both training and test set. Right: correct, splitting before oversampling, no leakage of information from training to test set and validating performance on unseen data is still valid.

To find the best method dealing with class imbalance, three datasets have been created:

1. Only undersampling: 2,795 : 2,795 (feasible : infeasible)
2. Only oversampling: 7,205 : 7,205
3. Both under- and oversampling: 5,000 : 5,000 (arbitrarily chosen)

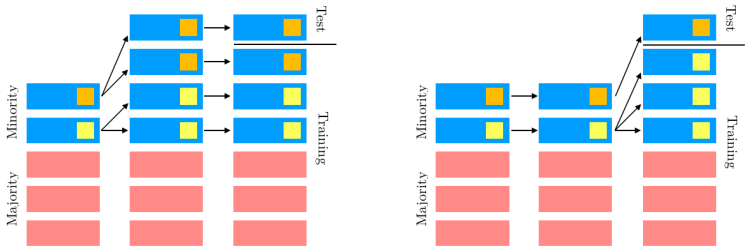


Figure 3.9: Left: incorrect oversampling, oversampling done before splitting. Right: correct oversampling, splitting done before oversampling

During the modeling phase (which will be described in Chapter 4), all datasets will be tested. The dataset with the best results being used for further analysis. Others will be disregarded.

3.3.2 Data construction

This task includes constructive data preparation, which will be the determination on the level of detail of the node labels. In Section 2.3 the Weisfeiler-Lehman subtree kernel is described. It works by appending node labels of neighbouring nodes to the original node labels. The appended labels are sorted alphabetically and compressed in to new, short labels. At the end of an iteration, the counts of the original node labels and the counts of the compressed node labels are represented as a feature vector. Neural networks are trained on these feature vectors. The original node labels define how many new node labels will be created after appending neighbouring node labels. The length of the feature vector depends on the amount of different node labels in the initial solution. Figure 3.10 visualizes how the length of the feature vector changes if node labels differ for the same graphs. In the left graph, originally, all nodes have the same node label. The right graph originally contains three different node labels. The appended and compressed labels after one iteration of the Weisfeiler-Lehman subtree kernel are visualized below both graphs. The feature vectors of both graphs contain the counts of the compressed node labels after one iteration. As can be seen, the length of the feature vector gets bigger when the level of detail (variety of node labels in the original graph) increases.

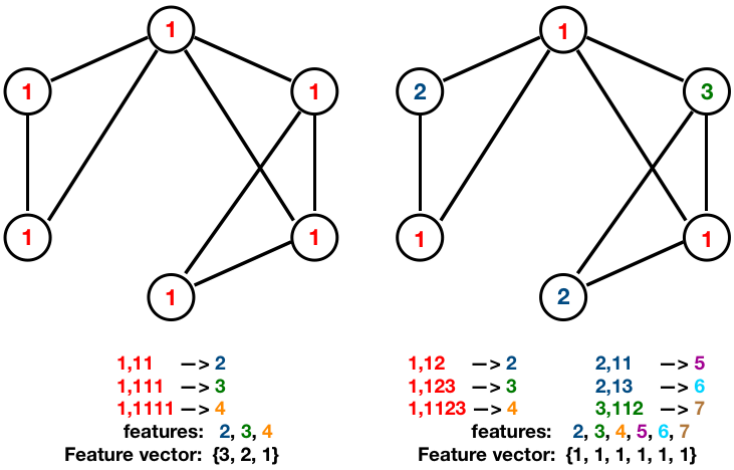


Figure 3.10: Different feature vectors for different amounts of node labels

Too many node labels results in very dissimilar feature vectors. In both cases, neural networks

may not be able to distinguish between feasible and infeasible instances. The best amount or original node labels needs to be chosen. Based on the data associated to the nodes (Table 2.4) the level of detail can be specified in three ways:

1. Regular labels
2. Regular labels + service tasks specified
3. Regular labels + service tasks and parking locations specified

Regular labels are the labels of the 8 node types described earlier in Table 2.4. This level of detail would results in the smallest feature vectors. One step further is to specify the service task as a node type. On the Kleine Binckhorst five different service tasks can be executed: internal cleaning, soap external cleaning, oxalic external cleaning, technical checkup A and technical checkup B. Either one would replace the regular service task node (S) resulting in 12 different node labels. The last one, specifying both parking locations and service tasks would result in 24 different node labels because the Kleine Binckhorst has 13 different tracks.

During the modeling phase, all levels of detail will be tested. The level of details providing the best results will be used for further analysis. The other will be disregarded.

3.3.3 Data formatting

Data often comes in formats other than the ones most convenient for modeling. Also in this research project, data formatting is necessary. As mentioned in Section 1.2, creating an algorithm for graph classification from scratch would be too unrealistic considering the time available. An existing algorithm will be adapted to fit this research project. Nevertheless, the existing algorithm is able to use arbitrary graphs as input, although they have to meet a certain format. The initial solution written as JSON files have to be rewritten to this format.

The algorithm accepts graph structured data in the format of NetworkX. NetworkX (version 2.1) is a Python library for studying graphs and networks. It provides data structures for graphs along with graph algorithms, generators and drawing tools. Graphs itself, as well as nodes in a graph, can have labels. Figure 3.11 shows a NetworkX graph drawing for the example activity graph given in Figure 2.4.

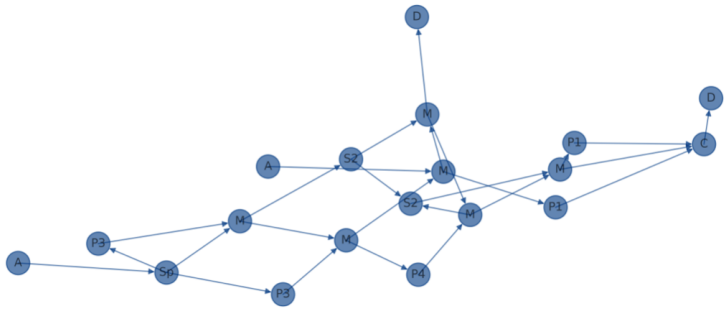


Figure 3.11: NetworkX drawing of the example graph in Figure 2.4

To conclude, 9 datasets have been created as a result of three different ways of balancing the dataset multiplied by three levels of detail. The next chapter will elaborate on the modeling part of this research project using a modified version of the Deep Graph Convolutional Neural Network (DGCNN) developed by Zhang et al. [33].

Chapter 4

DGCNN modeling and results

During the data preparation phase a variety of datasets have been created that serve as input for DGCNN. In this chapter is first explained how DGCNN is modified to this research project. This answers the fifth sub research question. Thereafter, the experimental setup is presented. Next, DGCNN classification results for each dataset are presented and the best performing dataset is optimized using hyperparameter tuning. Lastly, an in-depth analysis is done to compare the classification results of DGCNN with the results Dai [9] obtained with her domain knowledge based model. This answers the sub research questions how graph classification performs compared to domain knowledge based models. The DGCNN implementation is obtained from Github [34].

4.1 Deep Graph Convolutional Neural Network

DGCNN is not a deterministic graph compression algorithm like the deep topology classifier by Bonner et al. [5]. Deterministic graph compression algorithms transform graphs into fixed feature vectors. Meaning the feature vectors are the same every time the graph compression algorithm is applied. On the contrary, DGCNN *learns* how to represent graphs after every epoch. Learning how to represent graphs is achieved by using a matrix of trainable graph convolution parameters. This matrix is applied to the node information matrix. Each row in the node information matrix represents a node in a graph while the columns represent the label information. A linear feature transformation is applied to the node information matrix by multiplying it with the graph convolution parameters. These parameters are trained to become better in representing graphs by backpropagating loss.

A novelty of DGCNN is a pooling layer called *Sortpooling*. The general function of a pooling layer is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network. DGCNN combines this function with a sorting function. In image classification, pixels are naturally sorted with some spatial order. In text classification, dictionary order can be used to sort words. In graphs, sorting nodes is not as obvious as images or text. DGCNN sorts the nodes according to their structural roles within the graph [22]. The DGCNN algorithm used in this research project is a modified version of the algorithm by Zhang et al. [33]. In this research project nodes represent activities which can only take place if a precedent activity has happened. Nodes in shunt graphs are naturally ordered based on starting time in ascending order. Remember that activity graphs are directed graphs. Therefore, sorting nodes according to their structural roles should not be necessary. And yet, tests have been carried out to check whether performance would be better with or without Sortpooling. Results show that performance is better without Sortpooling. For this reason, DGCNN is adapted that the sequential order of nodes will be used as ordering. The spatial reduction function of the pooling algorithm will still be

used before feeding the set of input features into traditional 1-D convolutional and dense layers. In addition to sorting nodes and spatially reduce the representation, the third function of this layer is to unify the sizes of the output feature vectors. This is done by setting hyperparameter k to an arbitrary number of nodes. The size of output vectors are unified by deleting the last $n - k$ rows if $n > k$, or adding $k - n$ zero rows if $n < k$. Where n is the number of nodes in a graph.

Figure 4.1 visualizes how DGCNN transforms a graph into a feature vector. This should give an impression what the features look like and give a better understanding how a neural network is trained on these features.

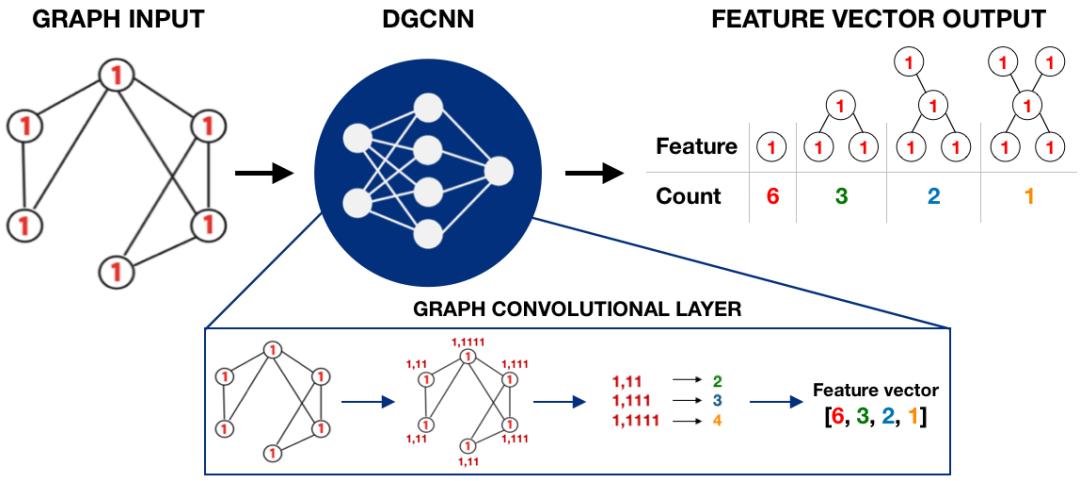


Figure 4.1: Visualization input and output DGCNN

The small example from Section 3.3.2 is used to visualize what the values in the feature vector actually represent. For example, the second value in the feature vector means that the original graph contains three times the subtree where a node with label one is connected to two other nodes with label one. Shunt graph used in this research project are transformed into features with over 5000 values. Meaning that over 5000 different substrees exist in all different graphs.

4.1.1 Experiment setup

DGCNN is implemented in both Torch and PyTorch. The PyTorch implementation is used in this research project together with Python (version 3.6.4). The following packages are necessary to run DGCNN:

- PyTorch (version 0.4.0)
- NetworkX (version 2.1)
- NumPy (version 1.14)
- SciPy (version 1.1.0)

DGCNN has several hyperparameters that can be tuned. The default set of hyperparameters, chosen by Zhang et al. [33], will be applied on the datasets described in Chapter 3. The default set commits the following hyperparameters:

- k: unifying nodes in graph *default: 0.7*
- r: learning rate *default:* 1×10^{-5}
- h: number of convolution layers *default: 2*
- l: number of nodes in convolution layers *default: 32*
- e: number of training epochs *default: 100*
- b: batch size *default: 50*

Hyperparameter k between 0 and 1 indicates the percentile of graphs having less nodes than a given percentage of all graphs. An example to clarify, $k = 0.7$ means the number of nodes is set to a number such that 70% of the graphs have less than that number of nodes. k closer to zero means the nodes in all graphs are unified towards the graph with the lowest number of nodes, while k closer to one means the nodes in all graphs are unified towards the graph with the highest number of nodes. The default number of training epochs is 100. However, initial tests have been done to check whether the best performance is found within this number of epochs. There has been decided to increase the number of epochs to 120. Every time a new epoch begins, training data is randomly shuffled and processed in batches of 100 graphs to enable faster learning. Training was done on an 1,7 GHz Intel Core i7 MacBook Air. The DGCNN implementation is not parallelized, thus only 1 CPU core is used.

4.1.2 Classification results

DGCNN is applied on all nine datasets to find the best combination of methods dealing with class imbalance and the level of detail of node labels. Table 4.1 shows the classification performance of DGCNN on all individual datasets. Performance increases as the node labels get more detailed. It is beneficial to include information about specific service tasks and parking locations in the node labels. The three datasets with the highest level of detail are highlighted in Table 4.1. Next, the results show that undersampling is the best of the methods dealing with class imbalance. The runtime for undersampled datasets is also significantly lower than when (a combination with) oversampling is used since the undersampled dataset is smaller. This research project will continue with the undersampled dataset with 24 node labels as it achieves the highest accuracy with the lowest runtime.

DGCNN RESULTS	**Undersampling**			**Oversampling**			**Undersampling and Oversampling**		
# node labels	**8**	**12**	**24**	**8**	**12**	**24**	**8**	**12**	**24**
Accuracy (%)	59,89	60,80	62,10	60,15	60,85	61,28	60,15	60,19	60,89
Standard deviation (%)	±1,28	±0,92	±0,88	±0,47	±0,74	±0,16	±0,56	±1,06	±0,50
runtime (h)	3,6	3,6	3,6	13,1	13,1	13,1	10,4	10,4	10,4

Table 4.1: Accuracy, standard deviation and runtime DGCNN on test sets of 9 datasets.

The performance of all datasets is validated by using cross validation. Figure 4.2 shows the cross validation results of the best performing dataset. The solid dark blue line indicates the test average accuracy. The dashed yellow line shows the development in loss as training proceeds. The dashed light blue line indicates the development of the training accuracy. This line is plotted because it looks like the accuracy is still increasing as 120 epochs have been reached. However, the model already starts to overfit as training and test accuracy diverge.

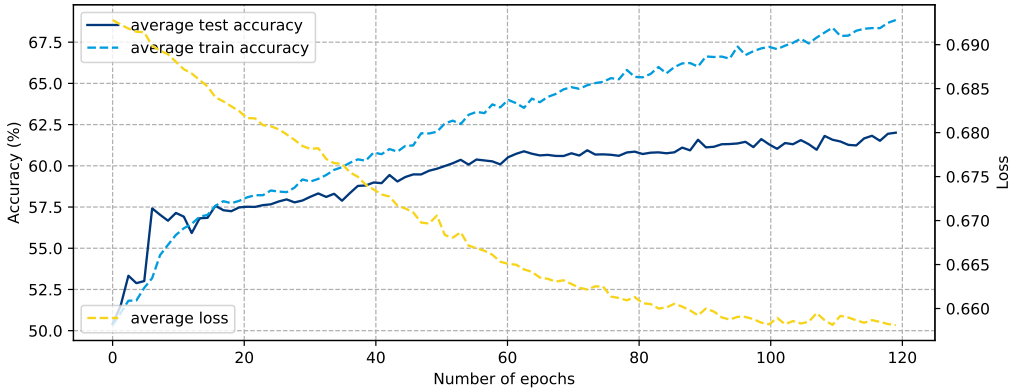


Figure 4.2: Performance best performing dataset: undersampling + 24 node labels

From this point on, the best performing dataset will be used. The other eight datasets will disregarded. The classification results above were obtained by applying the default set of hyperparameters. Now, hyperparameters can be tuned to choose a set of optimal hyperparameters. The next subsection will elaborate on this.

4.1.3 Results after hyperparameter tuning

Tuning is done by using *grid search*, an exhaustive search through a manually specified subset of hyperparameters. Grid search is again validated by using cross validation. The hyperparameters that are tuned using grid search with their corresponding subset are as follows (default parameters in **bold**):

- k: unifying nodes in graph *subset: [0.6,* ***0.7****, 0.8]*
- h: number of convolution layers *subset: [1,* ***2****, 3]*
- l: number of nodes in convolution layers *subset: [16,* ***32****, 64]*

The hyperparameters $k = 0.6$, $h = 3$ and $l = 64$ give the highest accuracy and are saved as hyperparameters for the final classification model. Additional test have been carried out to test whether decreasing or increasing the learning rate would result in an increase in performance. Decreasing enabled faster learning but decreased performance. Increasing lead to a significant increase in computation time while performance remained the same. Table 4.2 shows the confusion matrix, a visualization of the performance of the final classification model. Each column of the matrix represents the instances in a predicted class while each row represents the instances in an actual class. Each cell counts the number of instances that corresponds to the row and column value. Correctly predicted classes are true negatives (top left cell) and true positives (bottom right cell). Incorrectly predicted classes are false negatives (bottom left cell) and false positives (top right cell).

After hyperparameter tuning, the final classification model of DGCNN is able to predict feasibility of an initial solution with 65,1% certainty. The next section will compare the performance of DGCNN to the method Dai [9] proposed during her research. An in-depth analysis is performed to compare classification results of DGCNN and Dai's domain knowledge based model.

CONFUSION MATRIX DGCNN		Predicted labels 0	Predicted labels 1	right wrong
Actual labels	0	372 33,3%	185 16,5%	67% 33%
	1	205 18,3%	356 31,9%	63% 37%
	right	64%	66%	65,1%
	wrong	36%	34%	34,9%

Table 4.2: Confusion matrix final classification model DGCNN

4.2 Comparison domain knowledge based classifiers

The method Dai [9] proposed extracts around 40 features from the initial solution. This method relies on heavy feature engineering and extensive domain knowledge. The feature set includes information about trains, tracks and duration - i.e. service time, time between arrival, time interval between arrival and service, and many more. For each feature three metrics are calculated: average, standard deviation and variance. This is done to create a feature vector of fixed length for every graph in the dataset. Dai used Linear Discriminant Analysis (LDA) as classification method [15]. She used a combination of over- and undersampling using SMOTE and Edited Nearest Neighbours to solve class imbalance. Her balanced dataset consisted of 9.900 instances. She achieved a classification accuracy of 65,52%. A slightly better performance than DGCNN achieved. Note that Dai's dataset is different from the one used in this research project. However, the same parameters are used to generate instances. That means that individual instances are different in both datasets, but overall distribution is the same.

A comparison of only accuracy would be unsatisfactory and inadequate. Therefore, an in-depth analysis of classification results is performed. To be able to do this, a script has been written to extract the features Dai mentions in her research. In addition to the three metrics Dai calculates, four other metrics are added: skewness, kurtosis and minimum and maximum value [5]. Skewness is a measure of asymmetry of the probability distribution of a variable about its mean. Kurtosis is a measure of whether the data are heavy-tailed or light-tailed relative to a normal distribution. A feature selection method, called baseline variance approach, has been applied to reduce dimensionality of the feature set [25]. The resulting subset contains 127 features. Dai showed that LDA gave the best performance. Even though, four classification are applied on the new feature set, considering that the feature set is slightly different from the one Dai used. The four classification methods are Linear Discriminant Analysis (LDA), Support Vector Machine (SVM), Multi-Layer Perceptron (MLP) and Random Forest (RDF). Table 4.3 shows the the results. LDA gives the best classification performance. More important, accuracy increased to 66,35% with this slightly extended set of domain knowledge features. Moreover, these classifiers are trained on the dataset with 5.592 instances while Dai used a dataset with 9900 instances. All classifiers give a better accuracy, despite they are trained on less instances. Note that this result is achieved on the same dataset as DGCNN is trained and tested on.

DOMAIN KNOWLEDGE FEATURE SET	LDA	SVM	MLP	RDF	DGCNN
Accuracy (%)	66,35	66,12	65,99	65,65	65,12
Standard deviation (%)	±2,17	±1,61	±1,74	±1,83	±1,25

Table 4.3: Accuracy and standard deviation classifiers on undersampled dataset with domain knowledge features

Compared to DGCNN, all classifiers on domain knowledge features, show a better accuracy. Once more, this performance is the result of heavy feature engineering assuming extensive domain knowledge. Whereas, on the contrary, DGCNN takes initial solutions directly as input. Notwithstanding the better performance, but given the fact that DGCNN does not rely on heavy feature engineering can be stated that DGCNN is comparable to domain knowledge based models.

While their overall performance is comparable, prediction results on individual instances are not that similar. Table 4.4 shows a matrix representing the percentage of matching predictions. i.e. the percentage of predicted labels by DGCNN that matches labels predicted by domain knowledge classifiers.

	LDA	**SVM**	**MLP**	**RDF**	**DGCNN**
LDA	1	0,826	0,896	0,921	0,702
SVM		1	0,837	0,794	0,716
MLP			1	0,857	0,701
RDF				1	0,721
DGCNN					1

Table 4.4: Predication similarity matrix between domain knowledge classifiers and DGCNN

The matrix can be divided in two parts. The middle part shows the matching predictions between all domain knowledge based models. The predicted labels of the domain knowledge based models are quite similar. On the contrary, the right part of the matrix, the matching labels between domain knowledge based models and DGCNN, are not that similar. For example, the predicted labels of LDA and DGCNN only match 70,2%. Table 4.5 shows more into detail to what extent the predictions between LDA and DGCNN match.

Matching predictions		**DGCNN**	
		0	**1**
LDA	**0**	35,2%	13,4%
	1	16,4%	35,0%

Table 4.5: Matching predictions LDA and DGCNN

This means that it is possible that LDA performs better on one part of the instances, while DGCNN performs better on another part of the instances. For two reasons it would be beneficial to find out on which part of the instances each model performs better:

1. Getting insights into which features of an initial solution causes that either domain knowledge based models or DGCNN performs better;
2. Begin able to create a model with a higher performance predicting feasibility by combining the best of both models.

Note that from this point, not all individual domain knowledge classifiers are analyzed separately. They are combined into one classifier by averaging their prediction probabilities. A dimension reduction technique has been applied to transform high dimensional data into two dimensions. The next section will elaborate on this technique and the results it gave.

4.3 Finding patterns by reducing feature dimension

The technique is called t-distributed stochastic neighbour embedding, t-SNE [30]. t-SNE has become widespread in the field of machine learning, since it has an almost magical ability to

create compelling two-dimensional maps from data with hundreds or even thousands of dimensions. Specifically, it models each high-dimensional object by a two-dimensional point in such a way that similar objects are modeled by nearby points and dissimilar objects are modeled by distant points with high probability. t-SNE is helpful to see patterns in the data. Clearly separable clusters indicate good classification can be achieved, while mixed clusters indicate the opposite.

4.3.1 Show usefulness t-SNE visualization

To show that t-SNE is able to show patterns, the technique is applied on both datasets to show differences between instances that are easy and hard to predict by both models. It is expected that t-SNE is able to show separable clusters for the instances that get a high probability on either being feasible or infeasible. Figure 4.3 shows four scatterplots where t-SNE is applied on the domain knowledge feature set. The original dataset with 127 features is reduced into 2 features. The top two plots show the instances that are classified as either feasible or infeasible with a high probability by the domain knowledge based model. The probability is chosen to be $P(s) \geq 0.8$. These instances are *easily* predictable by the domain knowledge based model since the model is certain about its predictions. The two plots on the bottom show the instances that are hard to predict by the domain knowledge based model. The classification probability is $0.45 \leq P(s) \leq 0.55$, which means that the model was uncertain about its prediction. The green dots represent feasible instances. The red dots represent infeasible instances. The top plots show separable clusters, while the instances in the bottom plot are mixed. Separable clusters mean a good classification performance can be achieved. Unfortunately, the second plot shows good separable clusters, while all those instances are predicted incorrectly. On the other hand, the top left plot shows correctly predicted instances. More importantly, from all the easy instances, the majority is predicted correctly.

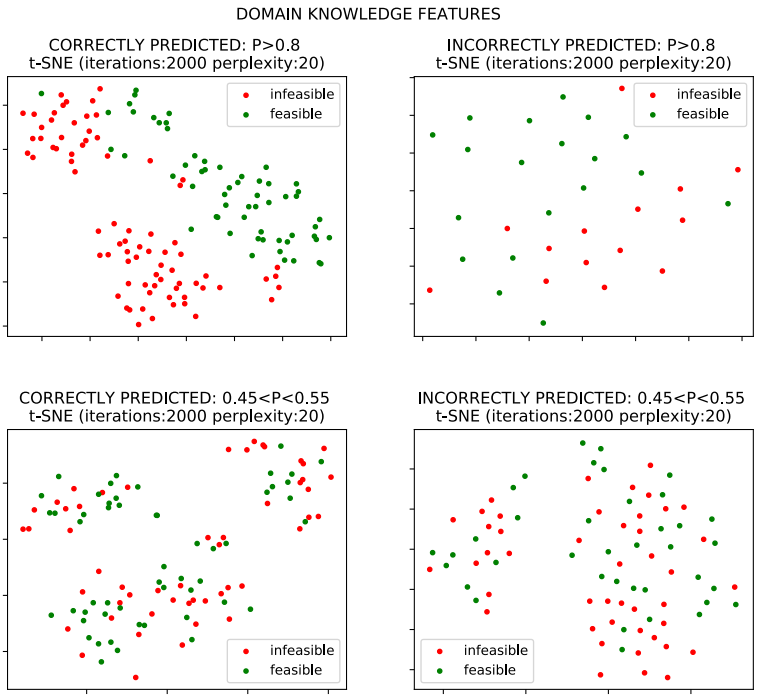


Figure 4.3: t-SNE dimension reduction domain knowledge features and classification performance shown as (in)separable clusters

Figure 4.4 shows four scatterplots where t-SNE is applied on the DGCNN features. The number of features DGCNN extracts from a graph is 5,344. T-SNE reduces this into 2 features. Again, the top two plots show the instances that are easily predictable by DGCNN. The two plots on the bottom show the instances that are hard to predict by DGCNN. Here, also the top two plots show clearly separable clusters. The instances of the two classes in the plots on the bottom are mixed.

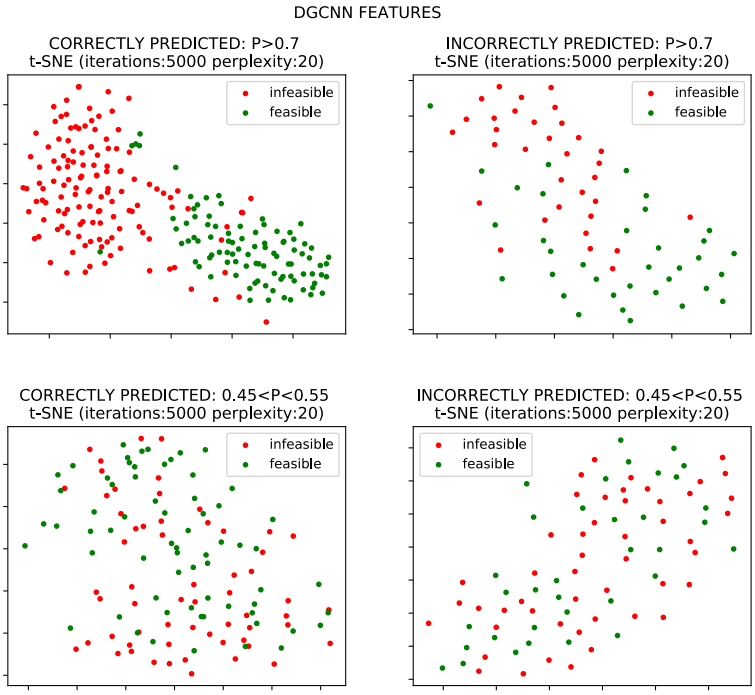


Figure 4.4: t-SNE dimension reduction DGCNN features and classification performance shown as (in)separable clusters

Figures 4.3 and 4.4 show that t-SNE is able to show patterns between the instances that are easy and hard to predict by both the domain knowledge based model and DGCNN individually. Therefore, in the next subsection the same technique is applied to find out if one model performs better on one part of the instances, while the other model performs better on another part of the instances.

4.3.2 Find patterns between domain knowledge model and DGCNN

A distinction is made between instances that are correctly classified by just either the domain knowledge based model or DGCNN. Green dots represent instances correctly classified by the domain knowledge based model. Red dots represents instances correctly classified by DGCNN. The idea is that if clearly separable can be observed a separate classifier can be trained that is able to predict which of the two models is probably correct. Plots have been made for both feasible and infeasible cases to get plots with just two different colors. In Figure 4.5 t-SNE is applied on the domain knowledge feature set. in Figure 4.6 t-SNE is applied on the DGCNN feature set. The clusters are completely mixed. t-SNE has also been applied on both feature sets combined in Figure 4.7. Again, the green and red instances are mixed. This means it is not possible to make a distinction on which parts of the instances the models perform better.

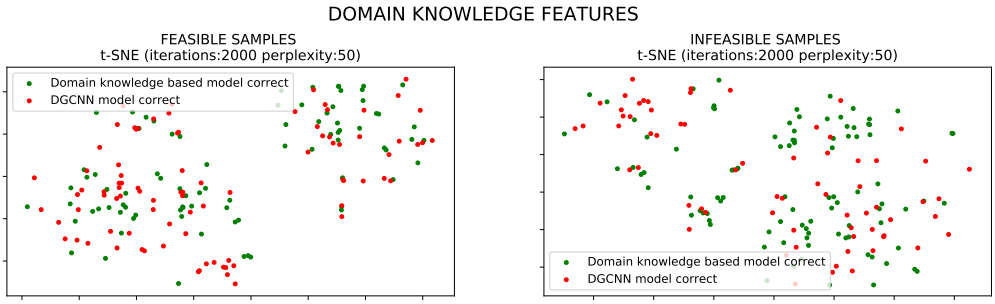


Figure 4.5: t-SNE dimension reduction on domain knowledge feature set for correctly predicted instances by only one of the models

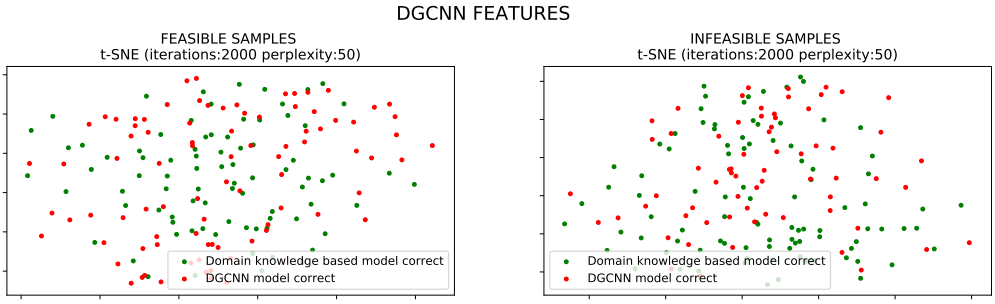


Figure 4.6: t-SNE dimension reduction on DGCNN feature set for correctly predicted instances by only one of the models

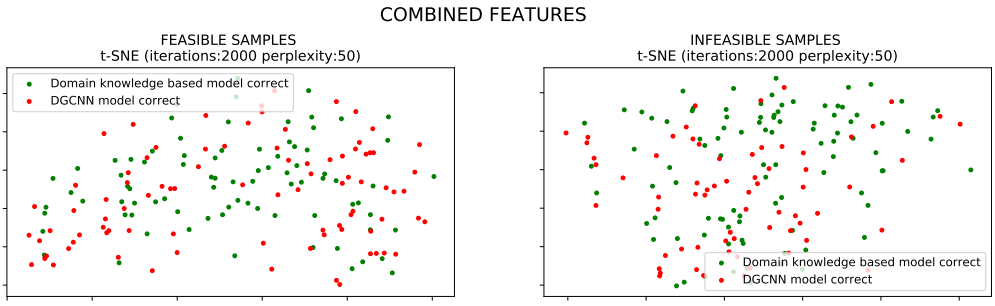


Figure 4.7: t-SNE dimension reduction on combined feature set for correctly predicted instances by only one of the models

4.3.3 Find patterns between local search and DGCNN

In addition, the same technique is applied to find out if patterns exist between (1) the difficult instances for local search and (2) the difficult instances for DGCNN. To be more specific, if the instances are plotted that are difficult to solve by local search and the instances incorrectly predicted by DGCNN with high probability, is it possible to observe patterns?

It is ambiguous how to distinguish between instances of an optimization problem that are easy and hard to solve by local search. But in principle, instances that can be solved within a short runtime are considered easy. While instances taking a long runtime are considered as more difficult. Note that only feasible instances are taken into account, since infeasible instances always take the maximum runtime. Decision boundaries in runtime for easy and hard instances are first set at *distant* values. Easy instances have a runtime less than 10 seconds, while hard instances have a runtime above 280 seconds. The idea is that if no patterns can be observed for these distant values, boundaries closer together also do not show any patterns.

Figure 4.8 shows t-SNE on three feature sets to find patterns between difficult instances for both local search and DGCNN. The green dots indicate difficult instances for DGCNN (incorrectly classified with $P(s) \geq 0.7$). The red dots indicate difficult instances for local search (runtime ¿ 280 seconds). There is quite some overlap between green and red dots. This could indicate that there's a pattern between difficult instances for local search and DGCNN. However, the clusters are not dense and since dissimilar objects are modeled by distant points, it does not seem that instances difficult to solve for local search are also difficult to predict by DGCNN.

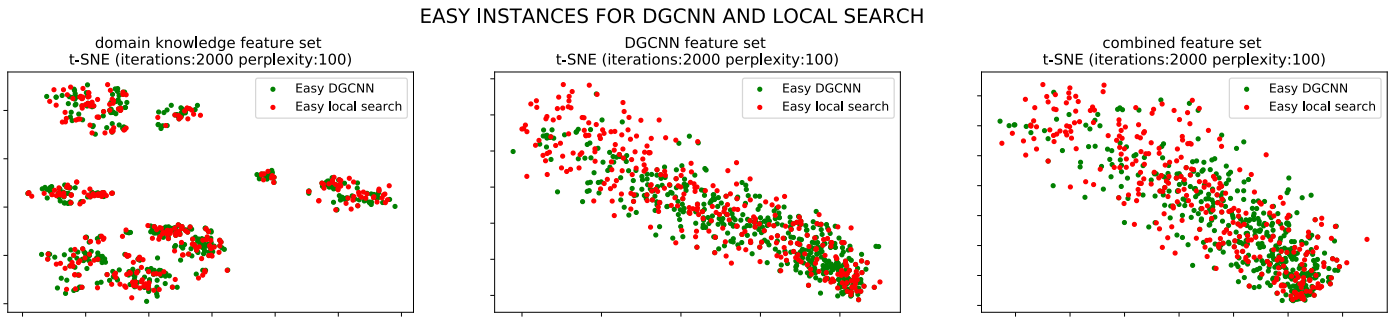


Figure 4.8: t-SNE on difficult instances DGCNN and local search

4.3.4 Conclusion finding patterns with t-SNE

So, unfortunately, this technique is not able to show why one classifier performs better than the other classifiers on a part of the instances. That being the case, it would still be beneficial to combine both classifiers to achieve a better performance. With this in mind, another technique has been applied. Not to discover patterns, but to increase classification accuracy by combining both classifiers directly. This is done by applying an ensemble learning technique. The next section will elaborate on this technique.

Chapter 5

Ensemble learning

Ensemble learning is a machine learning technique where multiple learners (classifiers) are trained to solve the same problem [35]. An ensemble contains two or more base learners. Base learners are generated from training data by a base learning algorithm. The base learning algorithms in this research project are LDA, SVM, MLP, SVM and DGCNN. An ensemble is constructed in two steps. First, the base learners are produced. Secondly, the base learners are combined according to a certain combination scheme - i.e. majority voting for classification. Many ensemble methods exist, out of which Boosting, Bagging and Stacking are the most commonly used. Boosting decreases bias, while bagging decreases variance. Stacking generally improves predictions and will therefore be used in this research project [32].

5.1 Stacking

Stacking is an ensemble learning technique to combine multiple base learners via a meta learner. Figure 5.1 visualizes the ensemble learning architecture in this research project. The ensemble consists of two base learning algorithms. Both base learners output the probability that an initial solutions will become feasible. The output of the first base learner, the domain knowledge model, is an average of the probabilities of the four domain knowledge classifiers. The second base learner, the DGCNN model, directly outputs the feasibility probability.

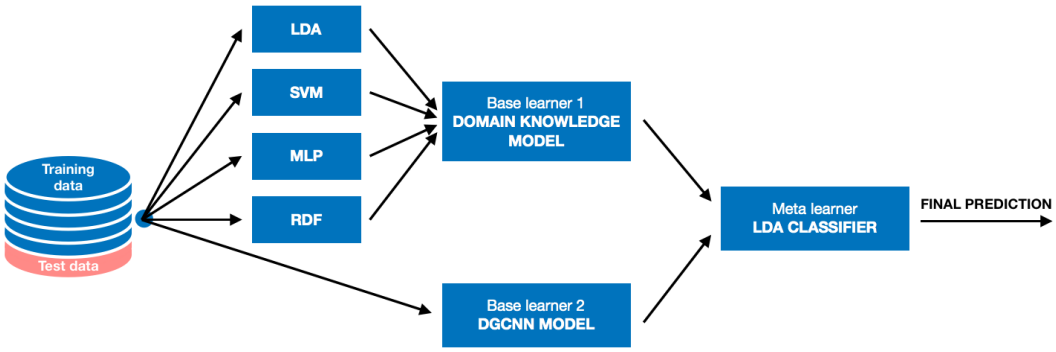


Figure 5.1: Ensemble learning architecture

The two base learners are trained on the complete training set. Cross validation is performed on each of these learners and the cross validated predicted values are collected from both base

learners. These values indicate the probability that an initial solution can be solved by local search. So, for each graph the feature vector contains two probabilities: (1) average probability by all domain knowledge based models and (2) probability predicted by DGCNN. The label remains the feasibility label. The predicted values form a new matrix. The meta learner is trained on this matrix. The meta learner can be any type of classifier. Since four classifiers have already been set-up and applied on the domain knowledge feature set, these classifiers will also used as meta learner. To find the best set of hyperparameters for each classifier, grid search has been included in the set-up. The next section will elaborate on the classification results of the four classifiers.

5.2 Classification results

The classification results from the meta learners are showed in Table 5.1. LDA gives the best accuracy. The corresponding hyperparameters are setting *solver* to *least squares* and *shrinkage* to *None*. Note that the performance of RDF decreases quite a lot. This is because the probability matrix contains two columns, while RDF performs better on high-dimensional data.

META LEARNER RESULTS	**Base learner 1**	**Base learner 2**	**Meta learners**			
			LDA	**SVM**	**MLP**	**RDF**
Accuracy (%)	66,35	65,12	67,70	66,36	67,08	61,53
Standard deviation (%)	2,17	1,25	1,89	2,31	2,11	5,26

Table 5.1: Comparison accuracy and standard deviation base learners vs. meta learner classifiers on output base learners

Accuracy has increased from 66,35% to 67,7% when stacking is used as ensemble learning method to combine the domain knowledge based model and DGCNN model via an LDA meta learner. This is the highest accuracy that has been achieved in this research project. Table 5.2 shows the confusion matrix.

CONFUSION MATRIX META-LEARNER		**Predicted labels**		
		0	**1**	right wrong
Actual labels	**0**	376 33,6%	181 16,2%	68% 32%
	1	180 16,1%	381 34,1%	68% 32%
	right wrong	68% 32%	68% 32%	67,7% 32,3%

Table 5.2: Confusion matrix ensemble learner

The main research question is how graph classification can contribute to the existing heuristic solution at NS. Contribution comes in two ways: (1) predicting whether local search can find a feasible solution based on an initial solution and (2) guiding local search by predicting the best operator to apply during local search. Section 5.3 will elaborate on the first contribution.

5.3 Implementation value NS

Being able to predict feasibility of an initial solution before applying local search leads to a decrease in computation time when determining the capacity of a service site. Due to difficulty integrating

the ensemble learning method into the existing heuristic solution directly, the real effect can not be tested. However, since all runtimes are known for the test instances in Table 5.2, theoretical effects can be tested. The effect will be measured as the difference in runtime between the current situation and the new situation. The current situation is the runtime without using the ensemble. Whereas the new situation is the situation in which the ensemble will be used. Table 5.3 shows the runtime in the current situation according to the four quadrants in Table 5.2 and averages of feasible and infeasible instances.

Quadrant	Type	Instances	Runtime (sec)
TN	*True Negatives*	376	112.069
FP	*False Positives*	181	54.825
FN	*False Negatives*	180	21.391
TP	*True Positives*	381	32.571

Average runtime feasible instances:	96,19
Average runtime infeasible instances:	299,63
Average runtime all instances:	197,55

Table 5.3: Runtime per quadrant and average runtimes

The total runtime in the current situation is 220856 seconds, roughly 61 hours. This runtime is fixed, as the local search heuristic simply tries to find a feasible solution for each instance. In the new situation the ensemble classifies an instance as either feasible or infeasible. Only when an instance is classified as feasible, the local search heuristic is going to try to find a feasible solution. If classified as infeasible, a new instance is drawn from the instance generator. This instance will again be classified as feasible or infeasible. This process continues until all instances have been classified as feasible. Figure 5.2 shows a Markov Chain to visualize this process. The probabilities are obtained from the four probabilities in Figure 5.2 in the red and green cells.

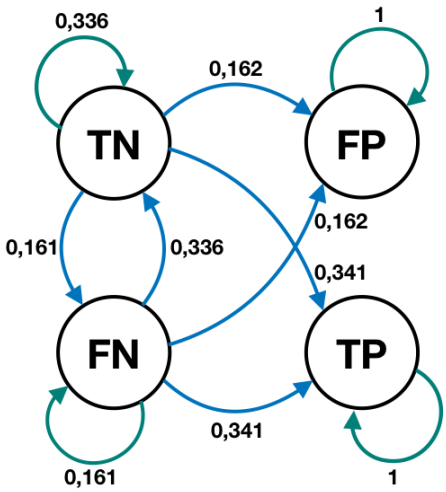


Figure 5.2: Markov Chain transfer probabilities

Figure 5.2 shows that if an instance is classified as feasible, it will never leave that state. Note that being classified as feasible can either be correct (true) or incorrect (false). If an instance is classified as infeasible, a new instance is drawn. This new instance can be transferred to any other state according to the probabilities. Since no new instances will be generated for instances classified as FP or TP, those runtimes remain the same in the new situation. The runtimes for TN and FN will change in the new situation. Thus, the next paragraphs will elaborate on the change

in runtime.

First, instances classified as TN. In the current situation, these instances account for almost 51% of the runtime. Because they are correctly classified as infeasible, most gains can be achieved for these instances. Change in runtime can be best explained when thinking in iterations - i.e. TN contains 376 instances in the beginning. After one iteration it contains 186 $(376 * 0,336 + 180 * 0,336)$ instances. 126 $(376 * 0,336)$ instances are again classified as TN and 60 $(180 * 0,336)$ instances from FN are classified as TN. The total number of instances in each iteration can be calculated with Equation 5.1.

$$E(S_{TN}^{i}) = P(TN) * S_{TN}^{i-1} + P(TN) * S_{FN}^{i-1} \tag{5.1}$$

where

S_x^i = the number of instances in x in iteration i
$E(x)$ = the expected number of instances in x in iteration i
$P(x)$ = the probability of transferring to x

After every iteration, 16,2% of the instances transfers from TN to FP and 34,1% transfers from TN to TP. Instances in FP are infeasible and are multiplied by the average runtime of 300 seconds. Instances in TP are feasible and are multiplied by the average runtime of 96 seconds. Table 5.4 shows the new runtime of instances that had first been classified as TN.

Iteration	Samples TN	To FP	To TP	New runtime (FP)	New runtime (TP)
0	376	-	-	-	-
1	186	61	129	18277	12409
2	92	31	64	9289	6156
3	45	15	32	4494	3078
4	22	8	16	2397	1539
5	10	4	8	1199	770
6	5	2	4	599	385
7	2	1	2	300	192
				36555	**24528**
Total runtime True Negatives new situation:					**61083**

Table 5.4: Runtime True Negatives in new situation

The total runtime for TN in the current situation is 112.069 seconds. The total runtime decreases to 60.803 seconds in the new situation. A decrease of 45,8%. This is a logical outcome since all instances originally needed the maximum runtime. In the new situation, for many of those instances a new solvable instance will be generated.

Regarding the instances in FN, the expectation is that the total runtime will be higher in the new situation compared to the current situation. This is because the instances are actually feasible, but incorrectly classified as infeasible. Therefore, new instances will be generated and some of those will turn out to be infeasible. As a result, a longer runtime is expected. The total number of instances in FN in each iteration can be calculated with Equation 5.2.

$$E(S_{FN}^{i}) = P(FN) * S_{FN}^{i-1} + P(FN) * S_{TN}^{i-1} \tag{5.2}$$

where

S_x^i = the number of instances in x in iteration i
$E(x)$ = the expected number of instances in x in iteration i
$P(x)$ = the probability of transferring to x

After every iteration, 16,2% of the instances transfers from FN to FP and 34,1% transfers from TN to TP. Instances in FP are infeasible and are multiplied by the average runtime of 300 seconds. Instances in TP are feasible and are multiplied by the average runtime of 96 seconds. Table 5.5 shows the new runtime of instances that had first been classified as FN.

Iteration	**Samples FN**	**To FP**	**TO TP**	**Runtime (FP)**	**Runtime (TP)**
0	180	-	-	-	-
1	89	30	62	8989	5964
2	44	15	31	4494	2982
3	21	8	16	2397	1539
4	10	4	8	1199	770
5	5	2	4	599	385
6	2	1	2	300	192
				17978	**11831**
Total runtime False Negatives new situation:					**29809**

Table 5.5: Runtime False Negatives in new situation

The runtime for FN in the current situation is 21.391 seconds. As can be seen in Table 5.5 the total runtime increases to 29.809 seconds in the new situation. This result is as expected.

While the runtime for FN instances increased with 39,3%, tot total runtime of all instances decreased with 42.568 seconds to a total of 178.288 seconds, roughly 50 hours. This decrease in runtime will save 19,2% when determining the capacity of a service site. With 35 sites in the Netherlands and many different situations that have to be tested, this has quite an impact. This is one of ways to contribute to the local search heuristic. The other contribution comes in the form of guiding local search by predicting the best search operator to apply. Chapter 6 will elaborate on this contribution.

Chapter 6

Guided local search

The local search heuristic attempts to find a feasible shunt plan by applying search operators in iterations to move through the search space. The heuristic uses 11 search operators to move through the search space. In every iteration, the search operators are shuffled in random order. This will be the order in which the operators will be evaluated. Starting with the first search operator, the heuristic evaluates the set of candidate solutions that can be reached through that search operator. A candidate solution is immediately accepted as new solution for the next iteration if it is an improvement over the current solution. If the candidate solution is worse, it is selected with a certain probability depending on the difference in solution quality and the progress of the search process. This probabilistic technique is called simulated annealing. DGCNN can replace simulated annealing by predicting in which order search operators should be evaluated. Figure 6.1 shows one iteration for both simulated annealing and DGCNN within the local search heuristic.

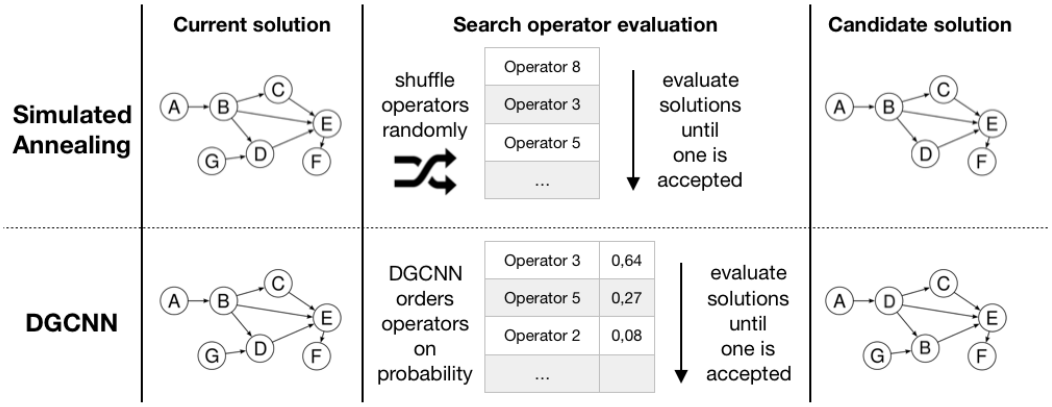


Figure 6.1: From random local search to guided local search

The DGCNN algorithm [33], can do this task without any modifications. As with predicting feasibility, DGCNN will be trained on graphs and corresponding labels, the difference is that the dataset not only contains graphs from initial solutions. It contains graphs from all solutions from initial solution to final solution. Furthermore, 11 different labels are defined instead of just two labels. The label for a particular solution is the search operator that has been applied to that solution to get to the next iteration. The output of DGCNN will be a vector of probabilities for each search operator. These probabilities determine the order in which search operators are evaluated. The operator with the highest probability will be evaluated first. Using DGCNN could be beneficial for two reasons:

1. Finding improvements faster;
2. Less randomness in search process.

As for the first, if DGCNN is able to accurately predict the order, the local search heuristic is more likely to find an improvement in the first search operator it evaluates. Regarding the second item, imagine that simulated annealing is used to find an improvement for a current solution. If an improvement has been found, the solution is reverted to the current solution and the process is repeated. Most likely, all improvements result in a different solution each time. On the contrary, DGCNN will always output the same order for that solution. Therefore, the search process contains less randomness. It could lead to more consistent shunt plans. Note that the randomness meant here is not randomness introduced to escape local optima during local search. The fundamental property of simulated annealing of accepting worse solutions with a certain probability still holds. Just the way of ordering the search operators changes by using DGCNN. The next section provides a descriptive analysis to gain more insights. Thereafter, results of applying DGCNN are described.

6.1 Descriptive analysis search operators

First, a short description on how the data is generated. The local search heuristic is slightly adapted such that it outputs all solutions from initial solution to final solution in a JSON file. The key/value pair about feasibility is changed into one representing the search operator. Instances will be generated for service site the Kleine Binckhorst with standard parameters and 21 train units. The local search heuristic is set up to run 1500 times and every time a new instance is drawn from the instance generator. This amount of runs minimally leads to 150.000 graphs since the least amount of iterations was over 100 in Chapter 3. The local search heuristic solved 859 out of 1500 instances. The remaining instances were not solvable within the maximum runtime of 300 seconds. There has been decided to train the DGCNN model only on feasible instances. Infeasible instances are disregarded. The idea is that the heuristic did not apply the correct search operators when it tried to solve instances that were infeasible in the end. Therefore, DGCNN should not be trained on instances in which wrong decisions have been made.

Regarding the feasible instances, over 600.000 search operators have been applied on those 859 instances. On average, 740 operators have been applied with a minimum amount of 122 operators and a maximum amount of 2527 operators. Figure 6.2 shows a plot that has been made to check whether there is a pattern between the number of iterations and the times a search operator has been applied.

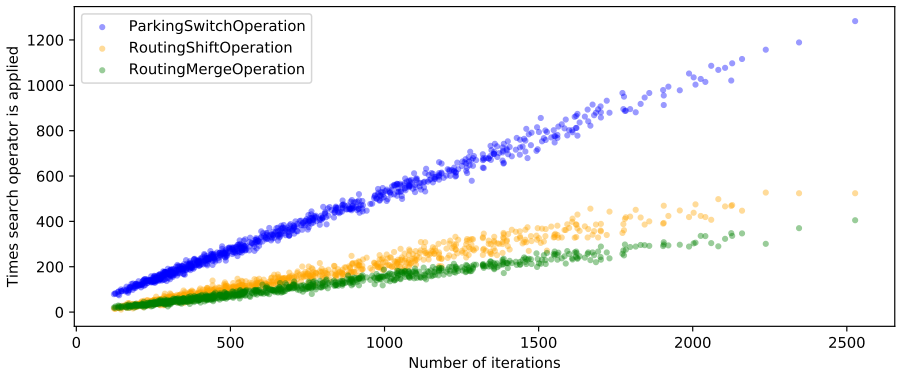


Figure 6.2: Scatter plot three biggest operators against number of iterations

The number of times an operator is applied increases linearly as the number of iterations increases. No unusual patterns can be observed. Box plots have been made to find out if there are major differences in the application of search operators between feasible instances with a low and high number of iterations. No major differences were observed. This could indicate that the major cause affecting the number of iterations is the initial solution and not the applied search operators. The box plots are shown in Appendix D. Figure 6.3 shows the distribution of search operators that have been applied in all feasible instances. Out of 11 operators, three are responsible for almost 90% of all applied operators.

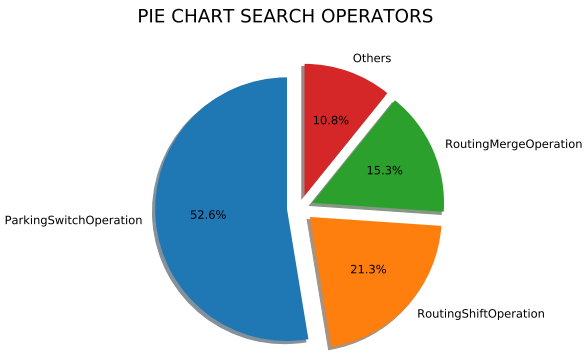


Figure 6.3: Pie chart search operators feasible instances

This distribution indicates that it could be beneficial to train multiple classifiers instead of just one. The basic approach would be to train one classifier directly on all 11 search operators. The other approach would be to train multiple, sequential classifiers. The first predicts whether one of the three largest operators should be applied or one the eight others. Dependent on the output of the first classifier, either the second or third classifier will be applied. If the first classifier predicts that one of the three largest operators should be applied, the second classifier will be used to predict which of the three. If the first classifier predicts otherwise, the third classifier will be used to predict which of the other eight operators will be applied. The idea is that the second approach yields better results if the first classifier performs well. The next section will elaborate on the results of both approaches.

6.2 DGCNN to determine operator order

First, a balanced dataset is created. The least applied search operator has been applied just over 1100 times in all 859 runs. To create a balance dataset, all search operators are undersampled to 1100 samples. Undersampling is done randomly, but in such a way that all graphs in the dataset are unique. Consequently, the dataset contains 12.100 unique graphs. The performance is validated by applying cross validation. Training the model on this amount of graphs took ± 25 hours. Figure 6.4 shows the performance of training DGCNN directly on 11 search operators.

The development of loss has also been plotted since loss starts increasing after around 145 epochs, this indicates that the model starts overfitting. The accuracy at that point is around 26,9%. This percentage means that in 26,9% of the instances, DGCNN places the search operator that the heuristic applied on top of the list. Meaning that if DGCNN would be implemented, the heuristic only needs to evaluate one search operator in 26,9% of the instances to find an improvement over the current solution. Even though this percentage does not seem very high, consider the fact that immediately picking the *best* search operator by randomly shuffling the order results in a probability of 9,1%. Also note that 26,9% is just a lower bound since DGCNN is trained on

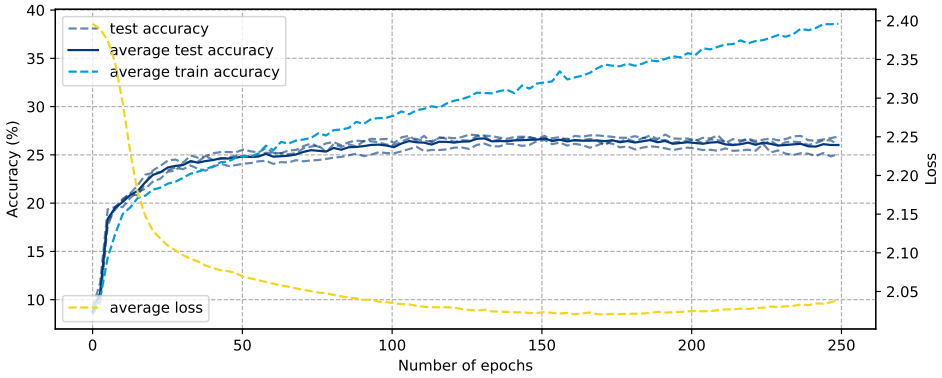


Figure 6.4: Performance basic approach

search operators that are not necessarily the best. Because the heuristic accepts the first candidate solution that is an improvement over the current solution, it could be that there were better search operators to apply. The chosen operator could even be one that worsened the current solution. 26,9% accuracy is achieved by comparing the operator that got the highest probability by DGCNN with the operator that the heuristic choose. When also looking at the operator with the second highest probability given by DGCNN, the accuracy would increase to 41,5%. While this approach already gives quite good results, the other approach has also been tried.

Again, a balanced dataset is created. 4000 graphs are randomly sampled from the three largest search operators. Another 4000 graphs are randomly sampled from the other eight search operators. As a result, the dataset contains 8000 unique graphs. Figure 6.5 shows the performance of the first classifier.

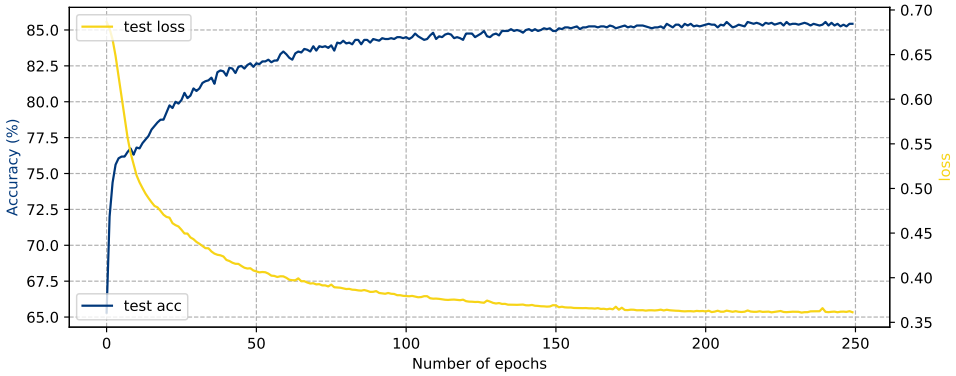


Figure 6.5: Performance first classifier

The first classier shows a good performance with an classification accuracy of 86,5%. Clearly, DGCNN is able to find patterns in the data such that it can accurately distinguish between applying one of the three largest search operators and one of the eight others. Figure 6.6 shows the performance of both the second and third classifier. Note that the number of classes are three and eight respectively. Meaning that randomly predicting the correct classes would be 33,3% and 12,5%.

The second classifier is trained on a balanced dataset with 12.000 randomly sampled, unique graphs. The third classifier is trained on a balanced dataset with 8000 randomly sampled, unique

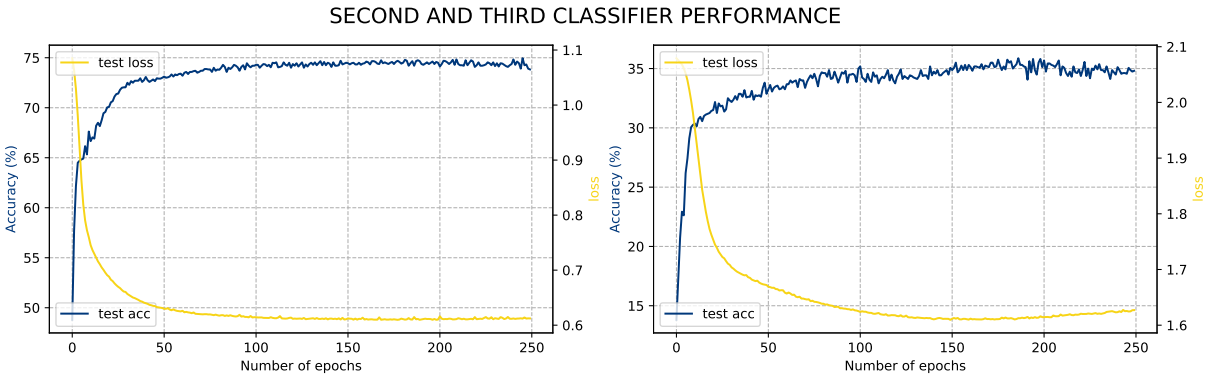


Figure 6.6: Performance second and third classifier

graphs. The performance of the second classifier before overfitting is 75,0%, while the performance of the third classifier is 35,9%. These percentages have to be multiplied by the accuracy of the first classifier to be able to compare this approach with the basic approach. The average accuracy of the second and third classifier is 48,0% $((0,865 * 0,75 + 0,865 * 0,359)/2)$.

The performance of both approaches seems quite fair, especially the results of the second approach. Unfortunately, the real effects of using DGCNN to determine the order of search operators could not be tested. Also, there is no data available about search operators that the local search heuristic evaluated before applying an operator. This is a limitation of this research project and will be discussed in the next chapter. The next chapter will describe conclusions, limitations, recommendations and future research.

Chapter 7

Conclusion and recommendations

First in this chapter, the overall research conclusion is presented. Subsequently, recommendations and limitations are given. Finally, important aspects for future research by NS based on the findings in this research are stated.

7.1 Overall research conclusions

The main research question and its corresponding sub research questions have been discussed in Section 1.3. The first four sub research questions were answered by reviewing the state-of-the-art literature and looking into the necessary background to understand the business situation. Adapting a graph classification method and its performance has been described in Chapters 4, 5 and 6. The main research question still needs to be answered.

"How can machine learning based graph classification support the local search heuristic at NS?"

During this research project, a machine learning based graph classification model, called DGCNN, has been modified to perform classification tasks using arbitrary shunt graphs as input. Using DGCNN overcomes the need of a model using feature engineering and extensive domain knowledge. To compare DGCNN with a domain knowledge based model, both have been used to predict feasibility of an initial solution. DGCNN achieves an accuracy of 65,1%, while the domain knowledge based model achieves an accuracy of 66,35%. Although the second model performs slightly higher, the fact that DGCNN does not rely on feature engineering makes both models comparable.

As DGCNN has proven to achieve a comparable performance, the effectiveness of the model is demonstrated by contributing to the local search heuristic at NS. The first contribution is predicting feasibility of an initial solution before applying the heuristic. Stacking, an ensemble learning technique, is used to combine DGCNN and the domain knowledge based model directly. The output of both models is used as input for a LDA classifier. The ability to predict feasibility increased to 67,7% when stacking is used as ensemble learning technique.

One of the tasks currently at hand at NS is to determine if the capacity of the existing service sites is sufficient to handle the growing amount of train units in the upcoming years. The local search heuristic will be used to carry out this task. By using the ensemble learning technique, the total computation time can reduced with 19%. With 35 service sites in the Netherlands and many different situations that have to be tested, this has quite an impact.

The second contribution comes in the form of guiding the local search heuristic. Using DGCNN to determine the order in which the heuristic evaluates search operators theoretically leads to finding improvements faster en reduces randomness in the search process. Unfortunately, due to the available data, these expectations can not be tested. However, results in Chapter 6 showed that DGCNN is able to accurately predict the search operator that improves a solution with 48%. Due to the data available, this is just a lower bound. In Section 7.2 will be explained how this accuracy can be improved.

In conclusion, DGCNN does not give a very high accuracy when predicting feasibility or predicting search operators. However, this research project proves that graph classification is able to contribute to the existing local search heuristic without the need of extensive domain knowledge.

7.2 Recommendations and limitations

The first recommendation is about labeling feasibility of initial solutions in a more robust way. Currently, the local search heuristic is applied once on every initial solution in order to solve it within 300 seconds. If local search fails, the initial solution is immediately labeled as infeasible. Since there is a lot of randomness in the local search process, it would be beneficial to apply local search multiple times on every initial solution. For example, apply local search five times on every initial solution. If local search is able to find a feasible solution more than 2 times, the initial solution is labeled as feasible. Doing it this way, it is expected that the available data contains less noise.

The majority of the time was put into efforts to generate and prepare the data, modify an existing graph classification method and achieve comparable results to a domain knowledge based model. Therefore, the second contribution in this research project has not been extensively studied. Even so, the brief analysis that has been performed shows already good results. It seems promising to continue working on guiding the local search heuristic, but then a limitation of this research has to be eliminated.

The limitation in this research project regarding the search operators is that the heuristic does not evaluate all search operators before applying one. Currently, it applies the first operator that finds a candidate solution that is an improvement over the current solution. Sometimes it even worsened the current solution because a worse solution was chosen with a certain probability. In this way, a graph classification model is trained on contaminated data. This limitation can be eliminated by adapting the heuristic that it evaluates all search operators such that the best operator can be applied. Obviously, only to generate the necessary data for research purposes. This can not be done in business situation as the computation time would become to big.

7.3 Future research

If the heuristic can be adapted to overcome the limitation, guiding the local search heuristic can even be taken a step further. Monte Carlo Tree Search is a heuristic search algorithm for decision processes. It builds a *statistics tree* that gathers information about search operators in order to guide them towards a feasible solution. This is different from the second contribution in this research project. That contribution only looks one step ahead by focusing on improving the current solution. A *statistics tree* is able to look further ahead by not only selecting the search operator that improves the current solution. The choice also incorporates which operator has the most positive effect on finding a feasible solution in the end. The *statistics tree* is build upon back propagation after finding a feasible solution or when it runs into the maximum runtime.

Bibliography

[1] Charu Aggarwal and Karthik Subbian. Evolutionary Network Analysis: A Survey. *ACM Comput. Surv.*, 47(1):10:1—-10:36, may 2014. 15

[2] Leman Akoglu, Hanghang Tong, and Danai Koutra. Graph Based Anomaly Detection and Description: A Survey. *Data Min. Knowl. Discov.*, 29(3):626–688, may 2015. 15

[3] James Atwood and Don Towsley. Search-Convolutional Neural Networks. *CoRR*, abs/1511.02136, 2015. 20

[4] László Babai, D Yu. Grigoryev, and David M Mount. Isomorphism of Graphs with Bounded Eigenvalue Multiplicity. In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 310–324, New York, NY, USA, 1982. ACM. 13

[5] Stephen Bonner, John Brennan, Georgios Theodoropoulos, Ibad Kureshi, and Andrew Stephen McGough. Deep topology classification: A new approach for massive graph classification. *Proceedings - 2016 IEEE International Conference on Big Data, Big Data 2016*, pages 3290–3297, 2016. 2, 15, 24, 30, 34

[6] Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-Path Kernels on Graphs. In *Proceedings of the Fifth IEEE International Conference on Data Mining*, ICDM '05, pages 74–81, Washington, DC, USA, 2005. IEEE Computer Society. 20

[7] Kevin W. Bowyer, Nitesh V. Chawla, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: synthetic minority over-sampling technique. *CoRR*, abs/1106.1813, 2011. 27

[8] V. Černý. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985. 9

[9] Lu Dai. A machine learning approach for optimisation in railway planning. Master's thesis, Delft University of Technology, March 2018. iii, iii, iv, 2, 12, 23, 30, 33, 34

[10] Pieter Jan Fioole, Leo Kroon, Gábor Maróti, and Alexander Schrijver. A rolling stock circulation model for combining and splitting of passenger trains. *European Journal of Operational Research*, 174(2):1281–1297, 2006. 57

[11] Richard Freling, Ramon M. Lentink, Leo G. Kroon, and Dennis Huisman. Shunting of Passenger Train Units in a Railway Station. *Transportation Science*, 39(2):261–272, 2005. 8, 56

[12] Michael R Garey and David S Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990. 6, 13

[13] Jørgen Thorlund Haahr, Richard M. Lusby, and Joris Camiel Wagenaar. Optimization methods for the Train Unit Shunting Problem. *European Journal of Operational Research*, 262(3):981–995, 2017. 8, 57

[14] John E. Hopcroft and Richard M. Karp. AN ALGORITHM FOR MAXIMUM MATCHINGS IN BIPARTITE GRAPHS. *Annual Symposium on Switching and Automata Theory*, 2(4):225–231, 1973. 10, 11

[15] Iuliana F. Iatan. The fisher's linear discriminant. In Christian Borgelt, Gil González-Rodríguez, Wolfgang Trutschnig, María Asunción Lubiano, María Ángeles Gil, Przemysław Grzegorzewski, and Olgierd Hryniewicz, editors, *Combining Soft Computing and Statistical Methods in Data Analysis*, pages 345–352, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. 34

[16] Per Munk Jacobsen and David Pisinger. Train shunting at a workshop area. *Flexible Services and Manufacturing Journal*, 23(2):156–180, 2011. 8, 57

[17] Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *CoRR*, abs/1609.02907, 2016. 12, 20

[18] S Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of statistical physics*, 34:975–986, 1984. 9

[19] L. G. Kroon, R. M. Lentink, and a. Schrijver. Shunting of Passenger Train Units: An Integrated Approach. *Transportation Science*, 42(December 2014):436–449, 2008. 8, 56

[20] Ramon M. Lentink, Pieter-Jan Fioole, Leo G. Kroon, and Cor Van 't Woudt. Applying Operations Research Techniques to Planning of Train Shunting. In *Planning in Intelligent Systems: Aspects, Motivations, and Methods*, pages 415–437. Wiley-Blackwell, 2006. 8, 56

[21] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard S Zemel. Gated Graph Sequence Neural Networks. *CoRR*, abs/1511.05493, 2015. 15

[22] Mathias Niepert, Mohamed Ahmed, and Konstantin Kutzkov. Learning convolutional neural networks for graphs. *CoRR*, abs/1605.05273, 2016. 3, 12, 19, 20, 30

[23] Dutch Railways (NS). Ns annual report 2017, 2017. iii, 1

[24] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. *World Wide Web Internet And Web Information Systems*, 54(1999-66):1–17, 1998. 15

[25] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011. 34

[26] Roel W. van den Broek. Train Shunting and Service Scheduling : an integrated local search approach. Master's thesis, Utrecht University, 2017. iii, 2, 3, 8, 9, 10, 12, 57, 58

[27] Nino Shervashidze, S V N Vishwanathan, Tobias H Petri, Kurt Mehlhorn, and Karsten M Borgwardt. Efficient Graphlet Kernels for Large Graph Comparison. *Proceedings of the Twelfth International Conference on Artificial Intelligence and Statistics*, 5:488–495, 2009. 20

[28] Nino Shervashidze NINOSHERVASHIDZE, Pascal Schweitzer PASCAL, Erik Jan van Leeuwen EJVANLEEUWEN, Kurt Mehlhorn MEHLHORN, and Karsten M Borgwardt KARSTENBORGWARDT. Weisfeiler-Lehman Graph Kernels. *Journal of Machine Learning Research*, 12:2539–2561, 2011. 13, 14, 15, 20

[29] Suraj Srinivas, Ravi K. Sarvadevabhatla, Konda R. Mopuri, Nikita Prabhu, Srinivas S.S. Kruthiventi, and R. Venkatesh Babu. *An Introduction to Deep Convolutional Neural Nets for Computer Vision*. Elsevier Inc., 1 edition, 2017. 17

[30] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, 2008. 35

[31] S.V.N. Vishwanathan, Nicol Schraudolph, Risi Kondor, and K.M. Borgwardt. Graph Kenrels. *Journal of Machine Learning Research*, 11:1201–1242, 2010. 20

[32] David H. Wolpert. Stacked generalization. *Neural Networks*, 5:241–259, 1992. 40

[33] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. An end-to-end deep learning architecture for graph classification. In *AAAI*, pages 4438–4445, 2018. iii, iv, 2, 12, 20, 21, 29, 30, 31, 45

[34] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. Pytorch dgcnn. `https://github.com/muhanzhang/pytorch_DGCNN`, 2018. 30

[35] Zhi-Hua Zhou. *Ensemble Methods: Foundations and Algorithms*. Chapman & Hall/CRC, 1st edition, 2012. 40

Appendix A

Train units and subtypes

Appendix B

Train Unit Shunting Problem literature

The Train Unit Shunting Problem (TUSP) in passenger railway stations was introduced by Freling et al. in 2005 [11]. Their solution approach decomposes the problem in two steps. The first step is matching of arriving and departing trains where the objective is to minimize the shunting effort by keeping train units together as much as possible. This problem is formulated as a Mixed Integer Problem (MIP) and solved with CPLEX. The second step, the track assignment problem, determines how to park the assigned matchings on the tracks at the shunting station. The station capacity should be never exceeded and parking should be done in such a way that movements are conflict free. The second sub problem is modeled as a set partitioning problem with side constraints and solved using a heuristic column generation procedure. This heuristic uses dynamic programming for generating the columns. The approach was tested on one train station in the Netherlands. Results were found within 20-60 minutes of computation time. Note that processing power in 2005 was a fraction of the processing power nowadays and solutions can be found a lot faster. Nevertheless, this approach doesn't integrate the sub problems and will not find an optimal solution. Not to mention that service and maintenance activities are not taken into account.

Lentink et al. [20] extended the work of Freling et al. [11] proposing a four-step approach to solve the TUSP. The matching problem remains unchanged while the track assignment problem is decomposed into more practical problems, such as cleaning and maintenance of units. First a matching is determined using the algorithm proposed by Freling et al. A graph representation of the shunting station is presented in their study, which is used to estimate the routing costs from and to each shunt track. These estimates are used in the third step, the parking subproblem, to improve the column generation approach proposed by Freling et al. Finally, the actual routes are computed using the graph representation and the track occupation resulting from the previous step. Trains can move simultaneously and the entire path of a train movement is reserved for the duration of the move. The routes are computed sequentially and the order of evaluation is improved by a local search approach that swaps the order of two movements. The problems are still solved independently. They have tested their model on small and big instances. The small instances are solved quickly, while the larger instances took at least 700 seconds of computation time.

Kroon et al. [19] also extended the work of Freling et al. [11] by proposing an integrated approach. Their model is able to solve the matching and parking sub problems in an integrated manner. Furthermore, the model incorporates complicating details from practice, such as trains composed of several train units and shunt tracks that can be approached from two sides. Testing the model

on a realistic case for a station in the Netherlands showed that the CPLEX wasn't able to find a feasible solution in a reasonable amount of time. They reduced the amount of constraints by grouping them in clique constraints. Unfortunately, even with the reduction in constraints, the computation time increases rapidly with larger problems, taking several hours to complete.

Jacobsen and Pisinger [16] proposed an integrated approach using three meta- heuristics: guided local search, guided fast local search and simulated annealing. Their results showed that local search approaches provide results close to shunting plans constructed by CPLEX, while taking only seconds of computation time. However, the largest instances contain no more than ten trains, with one maintenance task per train, which is not representative for real-world scenarios. Furthermore, the absence of routing and matching makes their approach not directly applicable to the scheduling problem for the shunting stations.

Haahr et al. [13] developed and benchmarked three different methods: a constraint programming formulation (CP), a column generation approach (CG) and a randomized greedy heuristic (RGH). Their CP formulation is inspired by the rolling stock composition model of Fioole et al. [10]. This formulation was originally used for Rolling Stock Scheduling. However, they used the idea of compositions and composition changes for the TUSP. A composition consists of a number of train units in a specific order. A composition change is the transition of one composition to another. For instance, if an arrival of a train unit takes place at a specific track, then a train unit is, in effect, coupled to the composition currently assigned to the track. CP directly proved to be ineffective when solving the full mathematical formulations. In the CG approach they decomposed the problem by track, and attempt to assign each track a set of possible matchings, called a matching pattern. A matching pattern is a subset of matchings that can be feasibly assigned to a given track over the planning horizon. In particular, it is a set of matchings that satisfies the LIFO requirements as well as the available track length restriction. A large number of possible matching patterns exist. Thus the approach relies on the dynamic generation of variables that represent promising matching patterns. It turned out that their CG approach was outperformed by all other methods. The last method, RGH, is a heuristic that greedily assigns arrivals and departures to and from tracks. The important aspects of this heuristic are the efficiency of the construction and the randomization of the greedy choice. Together these characteristics allow the heuristic method to try many different track assignments and extractions within a short time. This is true for small instances, but it proved to be inefficient for the more constrained instances.

Van den Broek [26] presented a simulated annealing algorithm to address the issue regarding matching, parking, routing and servicing train units at shunting stations. His integrated local search algorithm is able to create feasible shunt plans for large real-world scenarios. However, it only focuses on finding a feasible solution. No distinction is made between any solution. One of the downsides of this is that the resulting plans do not seem logical to human planners. The somewhat brute force search of the solution space seems not to incorporate much of the logical structures of shunting yards. Another downside of the generated plans is lack of robustness. The algorithm doesn't take uncertainty in arrival-, departure- and service times into account. This results in shunt plans becoming unfeasible due to arrival delays or unexpected arrivals of trains. As mentioned in the introduction, this project will focus on a machine learning approach to solve the TUSP. The approach of Van den Broek contains some useful elements when applying a machine learning approach. Therefore, we will describe the algorithm in the next section more in detail.

Appendix C

Pseudo code local search

Algorithm 1: local search algorithm Van den Broek (2016)

Input: set of Arriving Train units (AT), set of Departing Train units (DT), maximum_time

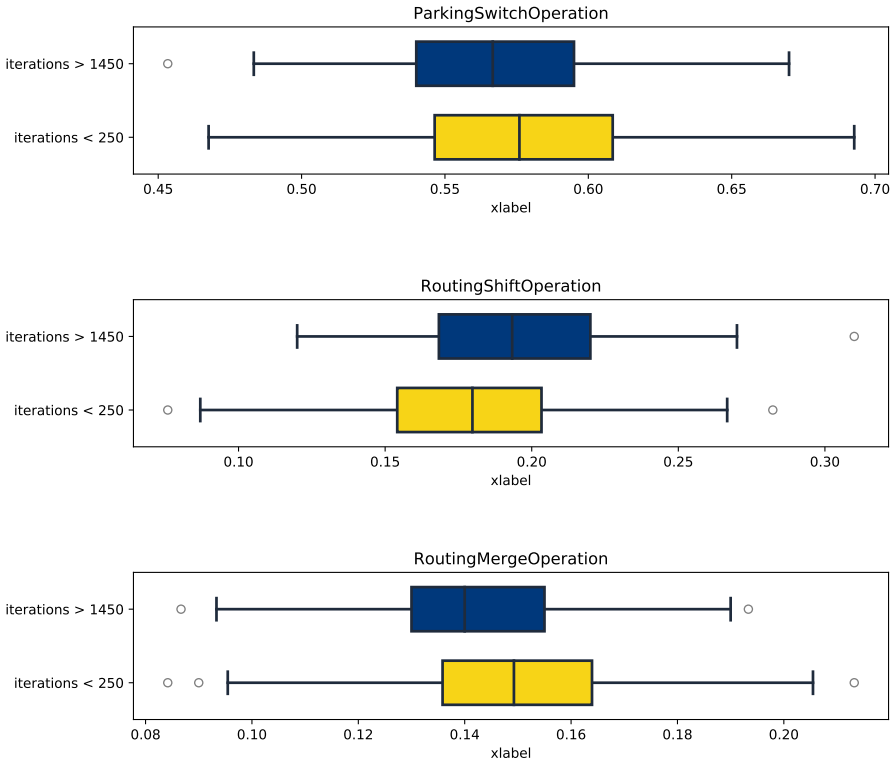
Output: feasible solution *or* no solution if maximum_time is reached

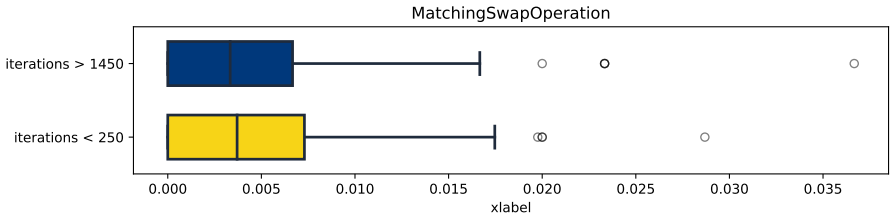
```
current_solution = create_initial_solution(AT, DT)
solution_status = check_feasibility(current_solution)                    not feasible
current_cost = calculate_cost(current_solution)                  given Cost(p) equation
While solution_status is not feasible and running_time is not maximum_time do:
|   shuffled_operator_list = randomly_order_operators()
|   For operator in shuffled_operator_list do:
|   |   new_solution = apply_operator(operator)
|   |   new_cost = calculate_cost(new_solution)
|   |   If new_cost > current_cost do:
|   |   |   current_solution = new_solution
|   |   |   current_cost = new_cost
|   |   |   solution_status = check_feasibility(current_solution)
|   |   |   Break for loop; Go back to while loop
|   End
End
```

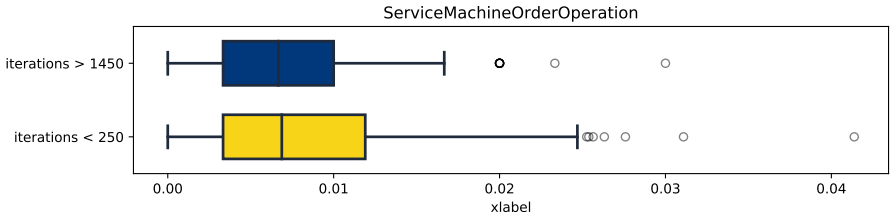
Figure C.1: Pseudo code local search heuristic Van den Broek [26]

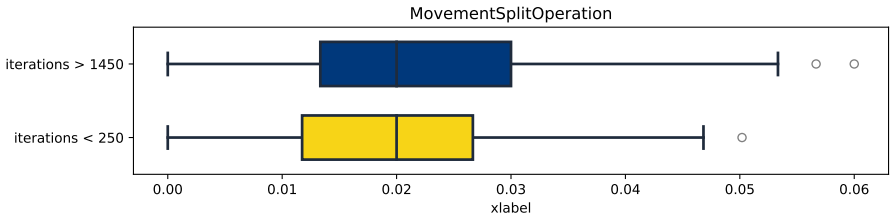
Appendix D

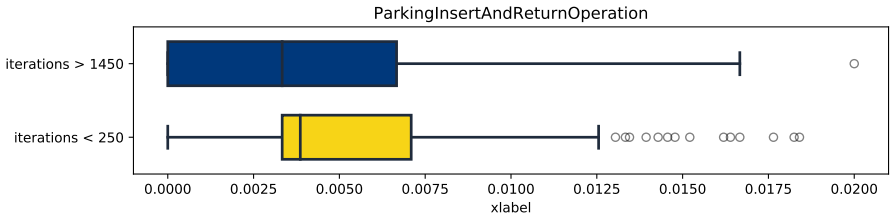
Box plots

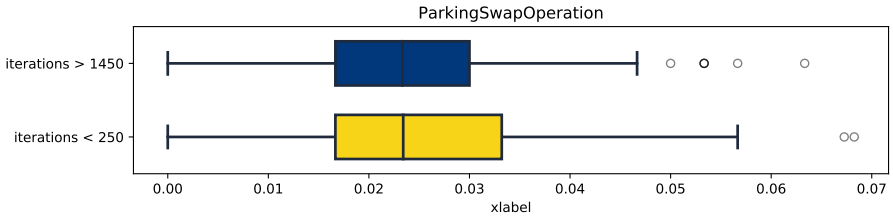


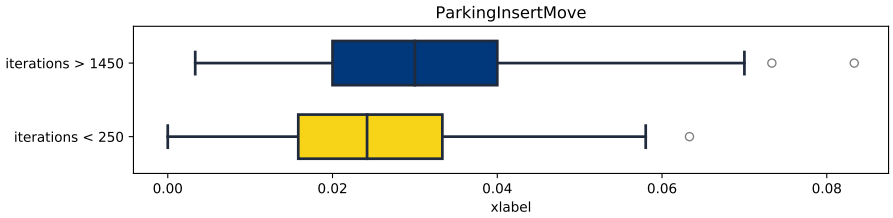

MatchingSwapOperation
iterations > 1450
iterations < 250
0.000
0.005
0.010
0.015
0.020
0.025
0.030
0.035
xlabel

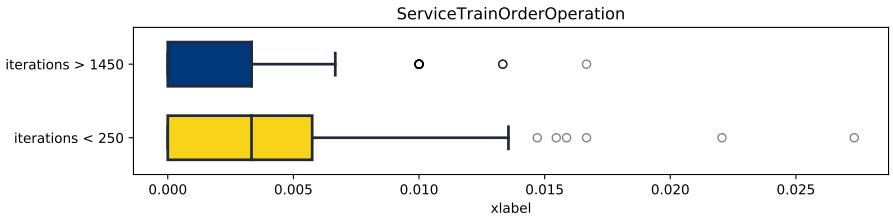

ServiceMachineOrderOperation
iterations > 1450
iterations < 250
0.00
0.01
0.02
0.03
0.04
xlabel

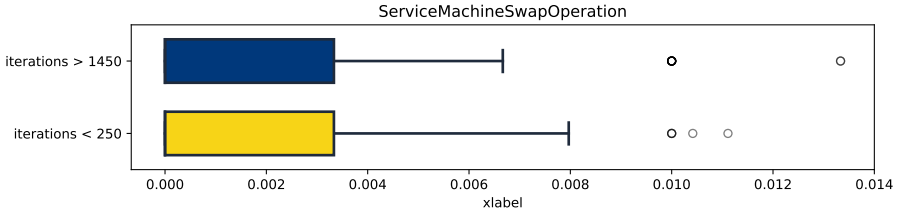

MovementSplitOperation
iterations > 1450
iterations < 250
0.00
0.01
0.02
0.03
0.04
0.05
0.06
xlabel


ParkingInsertAndReturnOperation
iterations > 1450
iterations < 250
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
0.0200
xlabel


ParkingSwapOperation
iterations > 1450
iterations < 250
0.00
0.01
0.02
0.03
0.04
0.05
0.06
0.07
xlabel


ParkingInsertMove
iterations > 1450
iterations < 250
0.00
0.02
0.04
0.06
0.08
xlabel


ServiceTrainOrderOperation
iterations > 1450
iterations < 250
0.000
0.005
0.010
0.015
0.020
0.025
xlabel


ServiceMachineSwapOperation
iterations > 1450
iterations < 250
0.000
0.002
0.004
0.006
0.008
0.010
0.012
0.014
xlabel