Centrum voor Wiskunde en Informatica

REPORT*RAPPORT*

PNA

Probability, Networks and Algorithms

Probability, Networks and Algorithms

Online capacity planning of repairs

J.W. van Kempen

REPORT PNA-E0608 JULY 2006

Centrum voor Wiskunde en Informatica (CWI) is the national research institute for Mathematics and Computer Science. It is sponsored by the Netherlands Organisation for Scientific Research (NWO).
CWI is a founding member of ERCIM, the European Research Consortium for Informatics and Mathematics.

CWI's research has a theme-oriented structure and is grouped into four clusters. Listed below are the names of the clusters and in parentheses their acronyms.

Probability, Networks and Algorithms (PNA)

Software Engineering (SEN)

Modelling, Analysis and Simulation (MAS)

Information Systems (INS)

P.O. Box 94079, 1090 GB Amsterdam (NL)
Kruislaan 413, 1098 SJ Amsterdam (NL)
Telephone +31 20 592 9333
Telefax +31 20 592 4199

ISSN 1386-3711

Online capacity planning of repairs

ABSTRACT
In this thesis we describe our research into a capacity planning problem that occurs in practice at a big group of car dealers, the Rüttchen group. A considerable part is about the construction of a model for this problem, showing and justifying the assumptions we make. Then it is shown that this model can be written as a Linear Program. We end by investigating related models, and show how we can solve them as either maximum flow problems, or with the help of a self-constructed algorithm based on finding augmenting paths with possibly additional cycles.

2000 Mathematics Subject Classification: 90B35 90B80 90C05
1998 ACM Computing Classification System: G.2.2
Keywords and Phrases: Scheduling; Modeling; Linear programming; Maximum flow
Note: This work is the final result of a Master's thesis at PNA1 under the supervision of K.I. Aardal.

Online capacity planning of repairs

Designing, solving, and implementing a planning model for Rüttchen car dealers

Jules van Kempen

May 2006

Supervisor at CWI: Prof. dr. ir. Karen Aardal
Supervisor at Utrecht University: Dr. Han Hoogeveen

Preface

This thesis is performed to complete the Master study Algorithmic Systems at Utrecht University. For my final project I was contracted by the Center for Mathematics and Computer Science (CWI) in Amsterdam, but mainly detached at Rüttchen Holding in Breda, a big car dealer with several franchises. Part of of this research has been funded by the Dutch BSIK/BRICKS project.

By coincidence, my application for a trainee place and the request from Rüttchen to design a planning system for their workplaces coincided at Karen Aardal's desk at CWI. I am very grateful to her that she coupled these two requests such that I could solve the Rüttchen planning problem as my final project. As a result of this somewhat unorthodox construction with multiple parties I enjoyed the support of two very good supervisors in the area of optimization. My supervisor at Utrecht was Han Hoogeveen, and my supervisor at CWI was Karen Aardal. Han Hoogeveen is an expert in the area of planning and scheduling. I have done work for him before, and I admire him very much for his outstanding insights. He helped me a lot in my search for solution strategies and in reviewing this thesis. Karen Aardal is specialized in the area of (integer) linear programming; she offered great help in finding the linear programming optimizer *QSopt* and in reviewing this thesis, also because of her excellent knowledge of the English language.

I would like to thank Guus Weisscher, my supervisor at Rüttchen, for explaining to me the way in which the workplaces operate. My thanks also go to Gosse Bakker from IBS. He helped me to implement the program, and also (very important!) made it possible for me to travel in reasonable time from Utrecht to Breda.

Furthermore, I want to thank the QSopt team for providing us with their linear programming optimizer, on which our final program is based. In particular, I would like to thank Monika Mevenkamp for giving a very quick response to my questions about QSopt.

Finally, I want to emphasize that I enjoyed the cooperation with both CWI and Rüttchen, and I am grateful for the opportunities given to me in this project. In my opinion, this project had an excellent mixture of both practical and theoretical problems.

Jules van Kempen
May 2006

Contents

Chapter 1

Introduction

1.1 Company background

Rüttchen is a big Mercedes car dealer in the Netherlands. Besides selling vehicles from Mercedes-Benz, they also sell Smart and Chrysler vehicles. They own 19 franchises, mainly in the southern part of the Netherlands. In 1994 they joined forces with their competitor EMA, also a Mercedes dealer, in the automatization of their Dealer Management System. The software development team of Rüttchen is responsible for maintenance, implementing changes, and the development of new modules. The result of this cooperation is ADIS (Auto Dealer Information System), which provides several functionalities for their administration. All franchises of EMA and Rüttchen make use of ADIS.

1.2 Problem background

Even though Mercedes cars are known for their high quality, they also need repairs and maintenance regularly. Most car dealers have their own garage to repair cars and perform maintenance tasks. When a customer notices a problem with his vehicle, he calls his dealer to ask for an appointment. The receptionist at the dealer's end then has to check when the workplace has capacity available to fix this problem and schedule the repair or maintenance task. For simplicity, we will from this point forward only speak of *repairs*, instead of *repairs and maintenance tasks*.

The car dealers of Rüttchen and EMA make a distinction between the responsibilities for planning and for executing the repairs. The receptionist needs to plan as much work as possible, but within the capacity of the workplace. The head of the workplace is responsible for assigning the planned repairs to mechanics who have to carry out these repairs. Thus the receptionist tells customers when they can bring their car and when the repair will be finished. At the beginning of each day the receptionist hands the

head of the workplace a list of repairs that have to be executed that day. The head of the workplace will then have to make up a schedule for these repairs and assign them to mechanics.

1.3 Current situation

In the current setting of Rüttchen and EMA the receptionists use a physical planning table that has the available mechanics listed vertically and the days listed horizontally. They can plan repairs by putting special cards in the cells of this table. See Figure 1.3 for a sample illustration of part of this table.

Figure 1.1: Part of a planning table at Rüttchen.

When a customer calls, the receptionists have to determine the amount and type of work needed for the repair, and have to answer whether there is still capacity left for the new repair in a particular time interval. Mainly due to the different types of work each mechanic is capable of executing, as the problem description later on will clarify, this task is far from easy. A good calculation of whether there is enough capacity includes shifting repairs in many possible ways, a task which no receptionist can be considered capable

of, especially when you take into account that the customer is waiting on the phone. Therefore, the quality of the planning they make is very poor, and the workplaces very often are forced to have substantial excess capacity. It also happens that they want to make up for this and have planned too many repairs in certain time intervals.

1.4 Project definition

Due to the shortcomings of the current planning method by receptionists, Rüttchen decided to change it. Their plan was to transform their manual system into an automated planning system, which can compute the capacity planning much better. To this end, Rüttchen is currently developing a big new module in ADIS for planning repairs and maintenance tasks in the workplace, which is called WPS (Workplace Planning System). Rüttchen has been developing ADIS in the language ILE RPG, but because of the graphical nature of a planning module they wanted to develop this module using new technology. They chose Java and the JPA-framework from IBS Nieuwegein. Because of lack of experience from Rüttchen in this matter, IBS was brought in to help with developing this system. A project group was formed to define the required functionalities of this system. In this project group all parties who have an interest in this system are represented. The project group determines what precisely needs to be incorporated in the system.

At first, constructing the WPS system went along well, but Rüttchen and IBS had difficulty in designing an algorithm that correctly computed the planning for them. They had already thought about possible algorithms, but they were not certain these were correct. Since the planning is really at the heart of their system, they decided to request help from an external party. This led them to contact the Centre for Mathematics and Computer Science (CWI). This thesis describes the research connected to this problem, and the design and construction of a computer program that implements the ideas from this research.

Chapter 2

The scope of this thesis

At the start of this project a lot of research had already been done by Rüttchen. Most of the requirements for the planning system were already identified and put on paper. What still had to be done was designing a model, which then had to be solved. The solution also had to be implemented and presented in a computer program, and this program then had to be incorporated into the WPS system, which is being constructed in parallel.

A large part of this thesis is about designing the model. This was a difficult task, as the instances of this model need to be solved very fast. A customer on the phone does not want to wait longer than a few seconds, while on the other hand, the model needs to correspond to reality as much as possible, which by nature makes it hard to solve. Therefore the algorithm for this model needs to answer each planning question that could be posed to it within a few seconds and the model needs to be as close to reality as possible.

Goal 1 *For each planning question posed, the planning algorithm must return an answer within a few seconds.*

Goal 2 *The model that is solved by the algorithm must be as close to reality as possible, while the algorithm satisfies Goal 1.*

More than one model and a corresponding algorithm was designed. Most of them appear in this thesis in Chapter 10, but of course only one was eventually implemented and used.

After designing this model and a fast algorithm for it, the planning system was to be implemented in a Java program. This program then had to be incorporated into the WPS system.

This thesis is organized as follows. We will start with discussing which planning questions Rüttchen wants to have computed. All these questions are based on a capacity planning problem, which we will describe in the

following chapter. We will create a model for this problem in Chapter 5, and in Chapter 6 we will show the precise algorithms that solve the planning questions. In the next chapter, we will discuss some additional practical problems we had to cope with while constructing the program, and we will very globally describe the program in Chapter 8. Then, we report on some experiments with this program to see whether computations are completed in time. In Chapter 10 we will discuss some models related to our capacity planning problem, and we will give algorithms to solve these models as well. We will end this thesis by giving suggestions for future research and a final conclusion.

Chapter 3

The required functionalities

In this chapter we will summarize the different functionalities the system needs. Since Rüttchen of course wants to meet the wishes of their customers as well as they can, the planning system needs to be able to answer whether, and how, these can be met. Receptionists desire additional functionalities of the system. We will summarize these wishes as well. From all these functionalities we can define what problems we need to solve.

When the receptionist has determined the amount and type of work needed for the new repair, he must plan the repair according to the customer's wishes. The system must provide functionalities for the following wishes and the corresponding scenarios to handle them:

Planning at a fixed interval: A customer calls with a request to plan his repair at a specific time interval and defines the date and time at which he will bring his car and the date and time at which he will pick it up. The system computes whether this request can be met. The answer is a simple *yes* or *no*. If this answer is *yes*, the repair can be stored by the receptionist at the specified time interval.

Finding a suitable day: A customer may have a time at which he can bring his car every day, and a time at which he wants to pick his car up the same day. Given these two points in time, the customer wants to know at which days the repair can be planned. For the next seven days the system needs to compute whether the repair can be planned at that day, within the given time interval. The answer is a list of the next seven days, with for each day a simple *yes* or *no* that indicates whether the repair can be planned at that day. The receptionist and the customer then have to agree to one of the days at which the repair can be planned. The receptionist then stores this repair at this chosen day at the specified time interval.

Planning as early as possible: Customers often want to get their car fixed as soon as possible. They can define a time and date at which

they can bring their car and the system will compute the earliest time at which the repair can be completed. If the customer agrees to this time at which he can pick up his vehicle, then the receptionist stores this repair, with the specified start time, and the computed end time. Customers often indicate that they will pick up their vehicle a little later than the time computed. If this is the case, then the receptionist will store the repair with this adjusted end time, as it leaves more room for shifting repairs, which makes all plans easier.

Planning a waiter job: Sometimes, for small repairs, customers want to wait at the garage while their vehicle is being repaired. If this is the case, the customer needs to specify a time and date at which he wants to come to the garage. The system then needs to compute the first time, after the specified time the customer brings his car, at which the repair can be started and then completely executed. Because the customer waits for his car, mechanics really MUST continuously work at this repair. This differs from *planning as early as possible*, in which we will allow preemption in our model. So, the system computes the earliest start time at which the repair can be continuously executed until completion, which implicitly defines an end time, which is shown to the receptionist as well. If the customer agrees to this time, then the receptionist stores the repair at the computed start time and end time.

Planning large repairs: Rüttchen has many transportation companies among their customers, and transportation companies often have large repairs that need to be executed. Typically, the work on these repairs spans several days. Transportation companies of course want to leave their vehicles at the workplace for as short time as possible. The customer specifies the date at which he can bring the vehicle to the workplace. Then, for the next seven days, the system needs to compute the earliest date and time the repair can be finished, given that the vehicle arrives at the start of that specific day. Then, from these answers, the customer indicates at which day he will bring his vehicle. Again, the customer may indicate that he will pick his vehicle up a little later, and if so, the end time is adjusted. The receptionist then stores the repair at the chosen start time and the corresponding end time.

Rüttchen wants to give receptionists the ability to force the planning of a repair, even if the system answers that this repair can not be planned. Furthermore, the system must provide a functionality for the receptionists to give them as much insight in the planning as reasonably possible. Therefore, the following additional functionalities are needed:

Forced planning: A forced planning actually is not a planning at all. The repair just needs to be planned at the defined dates and times, regardless of available capacity. This is needed for cars that break down on the road and need to be repaired immediately. Also, when one of the dealer's best customers calls with a specific request for a repair, the receptionist may force it in the planning so that it will be executed within the specified time interval. This often is feasible because dealers reserve capacity for these forced plannings, we will explain this in Chapter 4.

Insight into the planning: Up to here, the only information that the receptionist receives from the system are answers to his planning requests. In this way he will not be able to gather much insight into the planning, while this is most necessary. The receptionist needs to get accustomed with the new system and trust(!) it. To do so, he will need insight into the process. Furthermore, since he has been given the possibility of forcing a planning for some customers, he needs to be able to see whether this forcing will be practically feasible at all. There are different types of capacity at the workplace, and therefore, the receptionists want to know for each type:

- The possible capacity shortage
- How much capacity is still available
- How much capacity has already been assigned

Depending on how detailed information he requires, there should be three methods.

View of a week: For each day the total capacity shortage, the total available capacity, and the total assigned capacity is computed. We will call such a view of a day a *global view*.

View of a day: For each type of capacity, the shortage of capacity, and the amount of available capacity is computed.

Detailed view of a day: For each part of the day (morning, afternoon, evening) and for the total day, for each type of capacity, the shortage of capacity, and the amount of available capacity is computed. We will call such a detailed view of part of a day a *detailed view*.

If one looks more carefully at the possible problems the system needs to solve, one might see that some of the above problems are just a repeated application of other problems. To be precise, *Finding a suitable day* just consists of seven instances of *Planning at a fixed interval*, and likewise, *Planning large repairs* is nothing more than solving seven instances of *Planning*

as early as possible. Therefore, *Finding a suitable day*, and *Planning large repairs*, can be neglected as problems that need to be solved, as they are implicitly solved by solving other problems. Also, a forced planning is not a planning at all, and is not considered a problem that needs to be solved. Finally, the view of a day is a special case of a *detailed view*. Therefore, we have the following problems that need to be solved:

1. Planning at a fixed interval
2. Planning as early as possible
3. Planning a waiter job
4. Computing the *global view*
5. Computing the *detailed view*

In Chapter 6 we will show how we will solve these problems, and we will see there that these problems are all based on the problem of capacity planning that we will describe in the next chapter.

Chapter 4

The capacity planning problem

All the problems introduced in the previous chapter are based on the problem of deciding whether a set of repairs fits within the capacity. Therefore, we will describe this problem here and make a model for it in the next chapter. The capacity of a workplace is determined by the available mechanics, their qualifications and effectiveness, and their working schedule. To make a set of repairs fit together, there must exist an adequate assignment of capacity to all of them.

For this project, the model of the capacity planning that needed to be solved was designed with the help of the project group formed by Rüttchen. Since a model always is a simplification of the reality, the group had to decide which model was best, and therefore needed to be informed as much as possible about the consequences of the different assumptions that could be made. We have put a lot of work into this stage, from defining the possible simplifications to showing their consequences by presenting the project group with examples that capture their benefits and drawbacks. Regularly this resulted in long discussions about whether some assumption could or could not be made. But in the end the following simplified problem, which pleased all, was opted for.

We start this chapter by showing how capacity is provided, then we will show how we may assign capacity to our repairs, and discuss some additional constraints. In this chapter, we will extract the requirements for our model and show the assumptions we make. We will end with an evaluation of our simplified problem, justifying why we relax some existing constraints.

4.1 Available capacity

The available capacity of a workplace is determined by its mechanics. Each mechanic has a list of types of work which he is capable of doing, these

types of work are named qualifications. The qualifications are based on courses given by the importer of a car brand. Typically, there are around 10 qualifications for each brand. Examples of these qualifications are:

Example 1 *Possible qualifications for a certain car brand:*

- *APK*[1]
- *Maintenance Light (ML)*
- *Maintenance Heavy (MH)*
- *Repair (Rep)*
- *Service Light (SL)*
- *Alarm (Al)*

The project group made it very clear that they did not want a model that assigned mechanics to the repairs at the stage of the receptionist planning, because of the desired separation of responsibilities between the receptionist and the head of the workplace. The receptionist only has to check whether capacity is available, and the head of the workplace (who knows the specific qualities of his mechanics) makes a schedule, which assigns the mechanics to the repairs at specific time intervals. Furthermore, since the amount of work entered by the receptionist is very likely to differ a lot from the real amount in some cases, planning the individual mechanics is unnecessarily detailed. Therefore, the model only needs to check whether enough capacity is available without having to make an explicit assignment of repairs to individual mechanics.

As a result of this, the capacities provided by mechanics are aggregated into groups, defined by their collection of qualifications. The capacity of such a group is simply the total of the capacities the mechanics provide it with.

Assumption 1 *Mechanics with the same collection of qualifications may be grouped together and their capacities combined.*

The reason why we construct groups of mechanics in this way, is that we now can easily ensure that repairs can only be planned on mechanics that are capable of the type of work. This gives the following requirement:

Model Requirement 1 *The capacity of a group of mechanics can be assigned to a task only if the group is qualified for the type of work of that task.*

[1] APK is the compulsory annual checkup for a car in the Netherlands

Next to this list of qualifications, a mechanic also has a percentage that indicates his effectiveness at work. An experienced mechanic will do repairs faster than a trainee mechanic and therefore has a higher effectiveness percentage. If a mechanic for example has an effectiveness of 125%, this means that he will complete a repair of 5 hours work in only 4 hours. The typical effectiveness percentage is 100%, and nearly all mechanics have this effectiveness. There are some really experienced mechanics with slightly higher percentages, and there are trainee mechanics, whose effectiveness's are mostly 50%. This leads to:

Model Requirement 2 *Mechanic capacities must be multiplied by their mechanic effectiveness percentage before they can be added to their group.*

Finally, mechanics have working schedules, which define the time intervals in which they add capacity to the workplace.

As briefly mentioned in Chapter 3, the workplace needs to be able to deal with a forced planning. Cars do break down on the road occasionally, and you cannot plan these repairs in advance. However, if a car breaks down on the road, it will be repaired immediately. To be able to cope with these repairs, the dealers reserve a sizable part of their capacity for them, which we will call the *buffer*. Typically, they reserve about 20% of their total available capacity. This buffer makes it possible to deal with cars that break down and need to be repaired immediately. A nice extra is that it also allows the receptionist to be able to feasibly force a repair for his best customers, even if our algorithm answers that there is not enough capacity available. Furthermore, it also tackles another very important problem, as there possibly is a large difference between the planned amount of work and the actual amount of work for some repairs. We will further explain the possible difference between these two amounts in the next section. We will show that there is no way to avoid the possible gap between these two amounts, but luckily we now have a large buffer, which in most cases will ensure that repairs for which too little capacity was reserved, can still be executed in time at the workplace.

Thus, in the new system a buffer percentage needs to be defined. The system may only use (100 - bufferperc) % from the total available capacity for planning regular repairs. This buffer must exist for all types of work and is to be equally divided over the day. There is only one buffer percentage for all qualifications each day. There may be different buffer percentages for different days, as cars break down more often at the start of the winter season for example. This leads to the following requirement:

Model Requirement 3 *The capacity that each group provides during any time interval must be decreased by the daily buffer percentage before it can be assigned.*

4.2 Repairs

The other thing, next to available capacity, that determines whether there is enough capacity present at the workplace, is of course the set of repairs that needs to be executed.

As said before, it must be possible to plan repairs at specific time intervals. Therefore, all repairs have a time at which the customer brings his car, which is called the *release date* of the repair. Likewise, all repairs have a time at which the customer picks up his vehicle, which is called the *deadline* of the repair. Our requirement will be:

Model Requirement 4 *We can only assign capacity to a repair from a time interval that is between its release date and deadline.*

To make a set of repairs fit within the given capacity, there must be some assignment of the available capacity to them. This means that there may not be any shortage of capacity at any time interval:

Model Requirement 5 *A set of repairs fits only if it causes no shortage of capacity in any interval.*

Each repair is built up by a list of subtasks. These subtasks define the type of work and the time required. Each subtask has a minimum and maximum number of mechanics that must, or may, work at this subtask simultaneously. Example 2 further illustrates how a repair can be built up.

Example 2 *A possible build-up of a repair:*

Type of work	Time needed (hours)	Min Mech.	Max Mech.
APK	*2*	*1*	*2*
Maintenance Light	*1.5*	*1*	*1*
Maintenance Light	*3*	*2*	*3*

The receptionists define these subtasks based on the information provided by the customer on the phone. They are acquainted with the possible repairs and are able to estimate the amount and type of work and the number of mechanics that are needed. In most cases receptionists can define the work needed fairly exactly, but it also happens that they have to make a guess. For example, when the customer says that his engine makes an odd sound, there can be a number of causes for that. The actual cause can only be determined at the workplace once the car has arrived. Still, the receptionist has to determine the type and amount of work needed for it. Therefore, the actual amount of work can differ substantially from the amount planned by the receptionist. This is the gap between the planned amount of work and the actual amount of work on a repair we spoke about

in the previous section. One can see there is no way of avoiding this possible difference, and therefore, we are very glad to have the buffer to help us cope with this. Still, sometimes the buffer is just not sufficient to cope with this difference. If this is the case, the mechanics are asked to work overtime.

In reality, preemption of the repair is mostly not allowed, but because we solve a capacity planning problem and do not need to make a feasible schedule, we allow ourselves to use preemption of the repair, also because this will make it possible for us to solve the resulting model within the required time bound. Allowing preemption of a repair means that we may split up a repair and its subtasks into as many parts as we like, and distribute these parts over the time intervals in any way that fits best. Therefore:

Assumption 2 *Preemption of a repair is allowed.*

Preemption thus allows us to assign the capacity needed for a repair, for example, partly from the time interval 9:00 to 10:00, partly from the time interval 12:00 to 12:15, and partly from the time interval 16:00 to 18:30.

There is no fixed order in which the subtasks of a repair need to be executed.

Because we have grouped the individual mechanics, we may ignore the fact that a repair is often executed by the same mechanic(s). We will also not require that a repair is executed by the same group, as the project group indicated this was not desirable, since some subtasks may be executed by different mechanics. Therefore, we may assign the repair to different groups in any way that fits best:

Assumption 3 *Different parts of a repair may be executed by different groups.*

Not assigning the repairs to the individual mechanics makes it impossible to directly model some of the constraints on the repairs. These constraints are:

Constraint 1 *At any point in time, for any subtask, no fewer than the minimally required number of mechanics may work at this subtask.*

Constraint 2 *At any point in time, for any subtask, no more than the maximally allowed number of mechanics may work at this subtask. We will refer to this constraint as the* maximum number of mechanics constraint*.*

Especially, ignoring Constraint 2, the maximum number of mechanics, causes lots of trouble.

Without this constraint, we would be allowed to let 20 mechanics execute a 20 hour repair in 1 hour! This would of course be disastrous, since the repair can very likely not be executed by more than, for example, two mechanics, which means it cannot be completed in less than 10 hours. Therefore,

we can not make the assumption that the maximum number of mechanics constraint does not need to be satisfied. A big part of the long discussions about the desired simplifications with the project group was on this topic. They had to be convinced that ignoring this constraint would have large consequences. Also, once it was established that this constraint, at least in a weaker form, needed to be satisfied, there were even longer discussions about which requirement could replace it. First, the project group seemed pleased by only demanding that a repair could never be planned in an interval that is shorter than the minimum execution time for this repair, which is Model Requirement 6. Within this requirement, the constraint that subtasks of the same repair cannot be executed at the same time (Constraint 3) is also incorporated, albeit not directly, but in a weaker form.

Constraint 3 *Different subtasks of the same repair can never be executed at the same time.*

Model Requirement 6 *A repair can never be planned in a time interval that is shorter than the minimum time needed to complete this repair, based on the amount of work and the maximum number of mechanics of its subtasks.*

With Model Requirement 6 it is ensured that a repair, consisting of only one subtask of 20 hours work, at which, say, maximally 2 mechanics can work, needs an interval of at least 10 hours and adequate capacity within this interval. This would however still cause big problems, as it neglects the fact that all the capacity needed for the repair can still be sited within an interval of, say, 1 hour. Therefore, the following situation could occur, to which the algorithm would give an inadequate answer, see Example 3.

Example 3 *Suppose we would ignore the the maximum number of mechanics constraint, and instead only satisfy Model Requirement 6. The following situation could occur. We have 20 mechanics at our workplace, all of them are busy from 7:00 to 16:00, so there is no capacity available at the workplace from 7:00 to 16:00, but from 16:00 to 17:00 all mechanics are free. This can happen if all repairs must be completed at 16:00. Now suppose we get a new repair, consisting of one subtask that needs 20 hours of work, at which maximally 2 mechanics can work simultaneously. The algorithm will answer that it will fit between 7:00 and 17:00, since this interval satisfies Model Requirement 6, also, there is enough capacity available between 7:00 and 17:00, as we have 20 hours of capacity between 16:00 and 17:00. However, one can easily see that this repair cannot properly fit within this time interval.*

As is shown by Example 3, replacing the maximum number of mechanics constraint by Model Requirement 6 is inadequate.

Once the project group was convinced of this as well, which actually was mainly done by the very same example as described here by Example 3, a new and better requirement had to be formed. The previous requirement failed, because it still allowed the algorithm to assign too much capacity to a repair from a certain subinterval, as it only ensured that the algorithm could not let a repair take too much capacity from one big interval. Therefore the new requirement should ensure that the algorithm can never take too much capacity from any interval, however small. This will solve the problem we encountered in Example 3. More precisely, we will let our new requirement guarantee that any subtask at which maximally m mechanics may work simultaneously may take from any interval of length i at most $m \cdot i$ capacity. Or, in other words, we will require the following: say a certain subtask, with maximum number of mechanics m, reserves x capacity from an interval of length i, then $x/m \leq i$ must hold. Because repairs can consist of multiple subtask and subtasks can never be executed at the same time (Constraint 3), we have to ensure that all subtasks of a repair together may not take more capacity from any interval than the length of this interval allows. So, suppose we have a repair, consisting of 2 subtasks s_1 and s_2, where the maximum number of mechanics that may work on $s_1(s_2)$ is $m_1(m_2)$. We look at a certain interval of length i; let s_1 take x_1 capacity from this interval, and let s_2 take x_2. Then we must require that $x_1/m_1 + x_2/m_2 \leq i$. The general requirement is the following:

Model Requirement 7 *Consider a repair that consists of subtasks* $\{1, \ldots, n\}$ *with their corresponding maximum numbers of mechanics* $m_1, \ldots, m_n$*. Let* $x_1, \ldots, x_n$ *be the capacities reserved for subtasks* $\{1, \ldots, n\}$ *for any interval of length* i*. Then* $x_1/m_1 + \ldots + x_n/m_n \leq i$ *must hold. We will refer to this requirement as the* minimum execution time requirement *or simply* met*-requirement.*

One must note that in the *minimum execution time requirement* no notion of mechanic effectiveness is embedded. This is necessary, as we do not know which mechanics are still free to execute a repair. Therefore, we have to go with the typical effectiveness percentage of 100%.

It can easily be seen that this last requirement is stronger than Requirement 6. With this new *met*-requirement we have ensured that situations, like the one illustrated in Example 3, will be handled correctly. However, Requirement 7 is weaker than Constraints 2 and 3 combined, as we can have a situation as is described in Example 4.

Example 4 *Suppose we have a repair consisting of the following subtasks:*

Type of work	Time needed (hours)	Max Mech.
APK	*1*	*2*
Maintenance Light	*1*	*2*

We have two mechanics present at our workplace: one is only capable of APK, and the other one is only capable of Maintenance Light. We want to plan the repair between 9:00 and 10:00, at which both mechanics are present. With Model Requirement 7, the model would allow to let both mechanics work simultaneously at these two subtasks and complete them in 1 hour. But the Constraints 2 and 3 cause the repair to take at least two hours, as the two mechanics cannot work simultaneously, for they work on different subtasks of the same repair.

The reason why the *met*-requirement allows the infeasible plan from Example 4 is because the subtasks can be executed by 2 mechanics, while we only have one present. Our model cannot know this, as we only do capacity planning. Therefore, we will assume that the maximum number of allowed mechanics is always present, which would allow the plan.

The project group argued that situations like the one in Example 4 are very unlikely to happen, as there are always many mechanics of each qualification present at the workplace. So they judged Model Requirement 7 to be an adequate replacement for Constraints 2 and 3.

The constraint about the minimum number of mechanics (Constraint 1) will however be completely ignored. We do not directly see a good solution to how this constraint can be modeled, and moreover, the project group indicated that they did not mind losing it.

4.3 Additional constraints

Planning on a day that has already started: It often happens that customers want to get their car fixed at the same day they call. This case is special, as part of the repairs that were planned might already have been (partly) executed. Furthermore, the available capacity has decreased since mechanics have fewer working hours left. Still, the system has to compute the correct planning answers in this case, based on the updated information. We will however not make a special model requirement for this, as we will solve this problem by supplying the model with the correctly updated data.

Supplying substitute vehicles: Customers who leave their vehicle at the dealer for repair often want a substitute vehicle for the time their car is out for repair. If they do, they usually have additional demands about the type of this vehicle. For example, a customer who drives a luxurious E-type Mercedes, does not want a normal A-type Mercedes as a substitute. Therefore, the system must check whether there is a suitable substitute vehicle present at the interval of the scheduled repair, if the customers desires so. Again, we will not make a model

requirement for this, as the problem lies outside the scope of our capacity planning problem. We will let the WPS system answer this for us.

Resource constraints: We assume that there are no additional resource constraints like available bridges. Also, other constraints like the legal maximum of inspections per hour were shown to be no problem at Rüttchen.

4.4 Justification for our assumptions

When defining and selecting the requirements our model has to satisfy, we had the restriction that computations have to be performed within a few seconds for each question posed to the algorithm for the model (Goal 1). Secondly, we want our model to be as close to reality as possible (Goal 2). As will be shown in the next chapter, we can define a model that satisfies all our requirements, and that can be written as a Linear Programming (LP) problem. We will show in Chapter 9 that with this LP formulation we will be able to satisfy Goal 1.

With respect to Goal 2, the violation of constraints that have not been modeled is not considered harmful by the project group. Allowing preemption and allowing repairs to be executed by different groups do fit nicely within the idea of a capacity planning, as the project group desired. Moreover, grouping mechanics is an assumption they even wish to make. The only assumptions that could do real harm are the constraints concerning the number of mechanics that can simultaneously work at a repair. Of these, losing Constraint 1 is said to do little harm by the project group, as it practically never occurs there are too few mechanics available at the workplace. The only concern thus is Constraint 2, the maximum number of mechanics constraint, and Constraint 3. We argued that ignoring these constraints could cause problems, and therefore, we replace them by the *minimum execution time requirement* (Model Requirement 7). We have shown that this requirement is weaker than the two constraints, but only for a very few possible situations. Altogether, the project group is very pleased with how close the model resembled reality. Also, suppose we would require our model to resemble reality even more. This would probably cause the problem to become much harder, and we most likely would not be able to solve it to optimality within the time bound in practice. The only other option would then be to use a sub-optimal solution strategy. This is highly unwise, as this could cause inconsistency between answers of the system. Since receptionists must really be convinced of the correctness of the system, this is highly undesirable. Therefore, we think our model provides the best possible solution to satisfying Goal 2. One could argue that this simplified model could yield impossible schedules for the head of the workplace, which should of

course be avoided at all cost. Luckily, the buffer that is kept for reserving capacity will nearly always give the head of the workplace enough free space to make a feasible schedule for the planned repairs.

Now that we know all the requirements we need to model, we are ready to design a mathematical model, which we will do in the following chapter.

Summarizing, we will need to define a model that satisfies the following constraints:

- Model Requirement 1, about group qualifications.
- Model Requirement 2, about mechanic effectiveness.
- Model Requirement 3, about the buffer.
- Model Requirement 4, about release dates and deadlines.
- Model Requirement 5, there must be no shortage of capacity at *any* interval.
- Model Requirement 7, the *minimum execution time requirement*.

Chapter 5

Our model of the capacity planning problem

We have defined the requirements for our model of the capacity planning. In this chapter, we will design a mathematical model for the problem, which satisfies all the model requirements.

First, we will partition time into certain intervals, and explain how we will use these intervals to formally define our model. In this chapter we will also show how we will choose these intervals. Then we will give the full specification of our model as a Linear Programming (LP) problem, and show that all requirements are satisfied by this LP formulation.

5.1 Partitioning time into intervals

Because we want our model to cope with time intervals, the capacities should be divided among time intervals. Then, for each time interval, we will have aggregated groups, defined by their set of qualifications, that are present at this interval. We will refer to such a group g in a time interval i by an *intervalgroup* ig.

Definition 1 *An* intervalgroup *ig is an element for a group g of mechanics with the same set of qualifications, combined with an interval i during which these mechanics add capacity to the workplace. An* intervalgroup *has an amount of capacity that it provides.*

The *intervalgroups* will be the elements that provide capacity in our model. Note that we will denote a *intervalgroup* for an interval i and group g, by ig. Therefore, in the remainder of this thesis we can simply speak of the interval i or the group g, while we have only defined an *intervalgroup* ig.

We will now give an example that shows how these *intervalgroups* are constructed.

Example 5 *Consider for example that we zoom in at the time from 8:00 to 9:00. We have, for example, three intervals here, the first one from 8:00 to 8:15 (i_1), the second one from 8:15 to 8:40 (i_2), and the last one from 8:40 to 9:00 (i_3). We have three mechanics who work from 8:00 to 9:00. The following table shows these mechanics, their effectiveness, and the capacity (in minutes) they provide at the different intervals.*

Mechanic	Group	Eff. Perc.	Cap i_1	Cap i_2	Cap i_3
Peter	*APK & ML*	*100*	*15*	*25*	*20*
Steven	*APK & Rep*	*100*	*15*	*25*	*20*
John	*APK & ML*	*60*	*9*	*15*	*12*

Note that the values in the columns Cap i_1, Cap i_2, *and* Cap i_3 *are the result of simple calculations. The value for the capacity that Steven provides in i_1 is simply calculated by the length of this interval times Steven's effectiveness. Thus,* $15 \times 1.00 = 15$.

Because Peter and John have the same set of qualifications, their capacities will be aggregated into the APK & ML *group. Also, suppose the buffer is 25%. The resulting 6* intervalgroups *and their rounded capacities are then:*

Interval	Qualifications	Capacity
i_1	*APK & ML*	*18*
	APK & Rep	*11*
i_2	*APK & ML*	*30*
	APK & Rep	*19*
i_3	*APK & ML*	*24*
	APK & Rep	*15*

To give an example of the exact computation of the capacity of such an intervalgroup*, we will show this for the value of the capacity in interval i_1 for the group* APK & ML*. Peter adds* 15 *minutes of capacity, and John adds* 9 *minutes of capacity, so together they provide* 24 *minutes of capacity for the* intervalgroup i_1 APK & ML*. Before we can assign this capacity, we must subtract the buffer, so* $24 \times 0.75 = 18$ *minutes remain.*

So, in our model, time will be partitioned into intervals, each having its own *intervalgroups* as illustrated in Example 5. By construction of these *intervalgroups*, we have automatically satisfied Model Requirements 2 and 3.

Given the *intervalgroups* and their capacities we have to assign the capacity to repairs. We will define the time span of a repair as the time interval between its release date and deadline. We may only assign capacity from an *intervalgroup ig* to a repair r if its interval i lies fully within the time span of r. Therefore, we will avoid the situation of having any interval that lies partially within the time span of any repair, this will immediately

satisfy Model Requirement 4. How we will determine the intervals, will be explained later on in this chapter. We may only assign capacity from an *intervalgroup ig* to subtasks, if its group g has the qualification of the subtask in its collection (Model Requirement 1). The repairs together may not use more capacity than provided at any interval in time (Model Requirement 5). Finally, we may of course never assign more capacity to a single subtask than it requires.

Choosing the intervals

As we said before, we need to define how we will split up time into appropriate intervals. On the one hand we need to create these intervals such that we can enforce our requirements, and on the other hand we want to make as few intervals as possible, as this will affect the performance of our algorithm. The big problem here lies in the fact that Model Requirements 5 and 7 need to be satisfied for *all* possible intervals. But because we want to make as few intervals as possible, we will create them in such way that by satisfying these requirements for them, we automatically satisfy these requirements for all other possible intervals as well. In this subsection we will define how we will split up time into intervals. In Section 5.3 we will show that, if we let these intervals satisfy Model Requirements 5 and 7, then all possible intervals can satisfy them.

As we already briefly mentioned in this section, we want all of our intervals to always lie fully within the time span of a repair or fully outside this time span, but they may never partially lie within the time span of any repair (Property 1). Also, we do not want any interval to cross a start time or end time of the working schedule of any mechanic. In this way we guarantee that for any interval i, mechanics are always continuously working during i, or not working at all during i (Property 2).

Property 1 *For each repair r and every interval i, i either lies fully within or fully outside the time span of r.*

Property 2 *For each mechanic m and every interval i, mechanic m is either continuously working during i, or not working at all during i.*

To better explain how we achieve these properties, we introduce the definition of a *breakpoint*.

Definition 2 *A* breakpoint *is a point in time that marks the start of one interval and/or the end of another interval.*

We will define a set of breakpoints, order them chronologically, and create an interval between any two consecutive breakpoints: we then have partitioned time into intervals. Coming back to constructing the time intervals

in such a way that Properties 1 and 2 hold, we let the set of breakpoints be defined by the release dates and deadlines of the repairs, and also add each start and end time (including breaks) of the working schedule of any mechanic to this set. In this way an interval can never lie partially within the time span of a repair, as intervals can cross neither the release date nor the deadline of this repair, because these are breakpoints, which satisfies Property 1. Also, no mechanic can ever partially work during an interval, for every start and end time is a breakpoint as well, which satisfies Property 2. An illustration of this is shown in Example 6.

Example 6 *Suppose we have the following set of repairs:*

Repair	Release date	Deadline
r_1	*10:00*	*18:00*
r_2	*08:00*	*11:00*
r_3	*07:30*	*18:30*
r_4	*10:00*	*18:00*
r_5	*15:00*	*19:00*
r_6	*18:00*	*19:00*

These are the working schedules of the three available mechanics:

Mech.	Start	Break 1	Break 2	End
Peter	*07:00*	*10:00 to 10:15*	*12:00 to 12:30*	*17:00*
Steven	*07:00*	*10:00 to 10:15*	*12:00 to 12:30*	*17:00*
John	*13:00*	*15:30 to 15:45*	*18:00 to 18:30*	*22:00*

This set of repairs would give the following ordered set of breakpoints: 07:30, 08:00, 10:00, 11:00, 15:00, 18:00, 18:30 and 19:00. Then, adding all start and end times of the mechanic working schedules, we get: 07:00, 07:30, 08:00, 10:00, 10:15, 11:00, 12:00, 12:30, 13:00, 15:00, 15:30, 15:45, 17:00, 18:00, 18:30, 19:00, and 22:00. Which would then make the following time intervals: 07:00 to 07:30, 07:30 to 08:00, 08:00 to 10:00, 10:00 to 10:15, 10:15 to 11:00, 11:00 to 12:00, 12:00 to 12:30, 12:30 to 13:00, 13:00 to 15:00, 15:00 to 15:30, 15:30 to 15:45, 15:45 to 17:00, 17:00 to 18:00, 18:00 to 18:30, 18:30 to 19:00, and 19:00 to 22:00. It can easily be verified that all of these intervals either lie fully within or fully outside the time span of any of the repairs, which satisfies Property 1. In a similar way, it is easily seen that Property 2 is satisfied.

So, the breakpoints created in this way are sufficient to satisfy Properties 1 and 2. Furthermore, they are necessary, since otherwise there will be an interval crossing some start time or end time of a repair, violating Property

1, or there will be an interval crossing some start or end time of a mechanic, violating Property 2. Therefore, this construction is sufficient and necessary to satisfy Properties 1 and 2, and thus gives the minimum number of intervals that yield these properties. A similar use of this partitioning of the time into intervals for a capacity planning problem can be seen in [7].

In Section 5.3, we will prove that if Model Requirements 5 and 7 are satisfied for the intervals as constructed above, then they can be satisfied for *all* possible intervals.

5.2 The overall model as a Linear Program

Given the intervals for our model, the resulting *intervalgroups*, and the repairs, we will formulate our model by a linear program in this section. We will start with some notation. We denote the set of repairs by R, the set of subtasks from a repair $r \in R$ by J_r, and the union of all subtasks from all repairs by J. We denote the set of intervals by I, the set of *intervalgroups* that an interval $i \in I$ provides by IG_i, and the union of all *intervalgroups* by IG. Let C_{ig} be the amount of capacity that *intervalgroup* ig provides, let S_j be the amount of work needed on subtask j, let m_j be the maximum number of mechanics that may simultaneously work on subtask j, and let L_i be the length of interval i. For $i \in I$ and $r \in R$, we say that $i \prec r$, if and only if i lies fully within the time span of r.

We will create variables that indicate how much capacity is assigned from a specific *intervalgroup* to a specific subtask. We make a variable x_{jig} if subtask j may take capacity from *intervalgroup* ig, that is, if g has the qualification for j in its list, and i lies fully within the time span of j. This x_{jig} variable can have any real positive value.

In our LP model we will enforce the following constraints:

- For any *intervalgroup*, no more than the provided amount of capacity may be assigned from it (Capacity constraint).
- For any subtask, no more than the required amount of work may be assigned to it (Subtask constraint).
- No repair may take more capacity from any interval than Model Requirement 7 allows.

We want to maximize the total amount of work that can be planned. This all leads to the following LP formulation of the overall model:

$$\max \quad \sum_{j \in J} \sum_{ig \in IG} x_{jig}$$

subject to

$$\sum_{j \in J} x_{jig} \leq C_{ig}, \text{ for each } ig \in IG;$$

$$\sum_{ig \in IG} x_{jig} \leq S_j, \text{ for each } j \in J;$$

$$\sum_{j \in J_r} \left(\sum_{ig \in IG_i} x_{jig} \right) / m_j \leq L_i, \text{ for each } r \in R, i \in I, \text{ with } i \prec r;$$

$$x_{jig} \geq 0, \text{ for each } j \in J, ig \in IG.$$

Note that the formula $\sum_{j \in J_r} (\sum_{ig \in IG_i} x_{jig})/m_j \leq L_i$ is a direct formulation of the *minimum execution time requirement*. The *met*-requirement states that $x_1/m_1 + \ldots + x_n/m_n \leq L_i$ must hold. In our LP model, x_j is $\sum_{ig \in IG_i} x_{jig}$, which directly gives the formula.

Due to the construction of our intervals, every solution of this LP model can be further refined into a feasible schedule. In the next section we will show how we can do this, and show that this schedule satisfies all our model requirements.

The answer to the LP formulation of the problem is the maximum amount of work that can be feasibly assigned to capacity, while we wanted to answer whether a set of repairs fits within the given capacity. This latter question can easily be answered if we have the answer to our LP formulation of the problem. If the maximum amount of capacity that can be assigned to repairs is equal to the amount of work that needs to be done, the set of repairs obviously fits, while, if the maximum capacity that can be assigned to this set is smaller than the capacity needed, then there is obviously too little capacity for this set and it does not fit. Also, the total amount of capacity that is missing can easily be calculated from this answer, which we will use later on to determine whether a new repair fits.

5.3 Satisfying all model requirements

Our LP model from the previous chapter does NOT necessarily satisfy all requirements. The problem lies in the fact that Model Requirements 5 and 7 need to be satisfied for *all* possible intervals, while we have only ensured that it is satisfied for the intervals we have created. We do not know whether a feasible solution to our model guarantees that Model Requirements 5 and 7 can be satisfied for *all* intervals. One might remember the situation from Example 3, in which there was a sub interval that caused trouble.

Therefore, in this chapter we will show that any solution to our LP model can be further refined into a feasible schedule, which does satisfy all requirements.

To form our schedule, we will look at all the intervals of our model, and for each interval we will create a feasible schedule. Because our intervals partition time, their corresponding schedules may straightforwardly be *glued* together to form our resulting feasible schedule. Note that in our schedule, we allow a group to execute multiple repairs at the same time.

So, we will show how we will form the schedule for an arbitrary interval i. We know exactly how much capacity is assigned from any *intervalgroup* $ig \in IG_i$ to any subtask j, as this is the value of the variable x_{jig} in our LP solution. If $x_{jig} > 0$, then, in our schedule for this interval, we will let ig execute x_{jig}/L_i work at every moment in time in this interval. Thus, we evenly spread out this amount of work over this interval. This is feasible since our intervals satisfy Properties 1 and 2. We will spread out all work that is assigned over the intervals in this way. Then we will glue together all schedules created in this way, which gives our resulting schedule s.

We now only still need to prove that s satisfies all model requirements. We will summarize all requirements, and show that they are satisfied by our schedule s.

- Model Requirement 1, about group qualifications, is satisfied by the construction of the variables.
- Model Requirement 2, about mechanic effectiveness, is satisfied by the pre-computation of the available capacities.
- Model Requirement 3, about the buffer, is also satisfied by the pre-computation of the available capacities.
- Model Requirement 4, about release dates and deadlines, is satisfied by the construction of the variables.
- Model Requirement 5, which states that no shortage of capacity is allowed at *any* interval. This is one of the two requirements our LP problem could not satisfy completely. It can be seen that our schedule s does not require more capacity than is provided in *any* interval.
- Model Requirement 7, the *minimum execution time requirement*. This is the other requirement our LP problem could not satisfy completely. Again, it can be seen that our resulting schedule s does satisfy this requirement.

With this, we have shown that our schedule s satisfies all model requirements, which is sufficient for the correctness of our LP formulation.

In the next chapter we will find and use the solution to our LP model in order to correctly compute the answer to the planning questions from Chapter 3.

Chapter 6

Algorithms for the planning questions

At the end of Chapter 3 we have defined the planning questions that our system needs to answer. In this chapter we will show how we will compute the correct answers to the four main questions. All the algorithms we will use to solve these problems, are based on solving the Linear Programming (LP) problem of the previous chapter. Therefore, we will first give a short introduction into Linear Programming, and argue why we will solve our LP problems with the use of an existing software package. After this, we will show how we will build up the model for a particular problem instance, as this is also time consuming. Then we are ready to show how we will compute the correct answers to the different planning questions. We will end by an analysis of the runtime for each of these questions.

6.1 Introduction to Linear Programming

Linear Programming, or LP, problems are problems in which we optimize a linear objective function, while satisfying a set of linear (in)equality constraints. LP problems are of great importance because they appear very often in practical problems in the field of operations research. In Section 5.2 we already showed an example of an LP problem. Because LP problems have linear constraints, all feasible solutions with regard to these constraints are situated within a polyhedron, which is a convex set. Furthermore, we also have a linear objective function, which makes each local optimum in this polyhedron a global optimum as well, due to the convexity of the polyhedron. Moreover, this optimum will be achieved at the boundary of this polyhedron, or even stronger, at an extreme point of this polyhedron, if it is a bounded polyhedron that is. Therefore, these LP problems allow a solution strategy that *walks* over the edges of the polyhedron, towards the optimum. This is the basis of the most famous method for solving these

LP problems, the *simplex method*. This method was published in 1949 by George B. Dantzig, who is also regarded as the founder of LP problems. Many of the current LP optimizers are still based on the principles of this method, which runs in exponential time in the worst case. LP problems are in the complexity class P, so there exist polynomial time algorithms for them, like some of the *interior point methods*. The fact that most LP optimizers use algorithms that run in exponential time in the worst case to solve LP problems, might seem strange. However, for practical reasons and sometimes strictly for performance reasons, the simplex method is still the most commonly used algorithm to solve LP problems.

In general, LP problems of reasonable size, like our LP formulation of the capacity planning problem, are easily solved by existing software packages. For further information on linear programming we refer the reader to [8] or [3].

6.2 Building an LP problem instance

In order to solve a particular problem instance by inserting its LP formulation into an existing LP optimizer, we must of course first create the variables and constraints from the input we are provided with by the WPS system. This is not straightforward, as the input we work with are a set of repairs and chunks of capacity. The time intervals have to be determined, the capacity divided over them and the variables and constraints created.

The input we get from the WPS system is the available capacity, a list of the repairs, and the buffer value. WPS passes capacity to us in chunks; for each mechanic, his capacity is divided over time intervals, where the bounds of these intervals are the start and end times of their breaks. Also, the effectiveness percentage of the mechanic is already incorporated before this capacity is passed. We will denote this list of intervals that contain capacity by IC. See Example 7 for further illustration. The repairs are passed to our system in a list containing all planned repairs. For each repair, we have the subtasks it consists of, and the release date and deadline. The subtasks contain the information about the type of work, the amount of work, and the maximum number of mechanics that may execute it simultaneously.

Example 7 *Let there be two mechanics; for simplicity we will neglect both the buffer and effectiveness of mechanics. The first mechanic has the qualifications APK & SL. He works from 8:00 to 17:00 and has 3 breaks, the first from 10:00 to 10:15, the second from 12:00 to 12:30 and the last from 14:30 to 14:45. Then his capacity is divided over 4 intervals, the first from 8:00 to 10:00, the second from 10:15 to 12:00, the third from 12:30 to 14:30 and finally from 14:45 to 17:00. The second mechanic has qualifications SL & ML. He works from 13:00 to 22:00 and has 2 breaks, the first from 16:30 to 17:30 and the last from 20:00 to 20:15. His capacity is divided over the*

intervals 13:00 to 16:30, 17:30 to 20:00 and 20:15 to 22:00. Therefore, the resulting list of intervals and their capacities (in minutes) is:

Start	End	Qualifications	Capacity
08:00	*10:00*	*APK & SL*	*120*
10:15	*12:00*	*APK & SL*	*105*
12:30	*14:30*	*APK & SL*	*120*
14:45	*17:00*	*APK & SL*	*135*
13:00	*16:30*	*SL & ML*	*210*
17:30	*20:00*	*SL & ML*	*150*
20:15	*22:00*	*SL & ML*	*105*

This list, which we denoted by IC, is the input of capacities we get from the WPS system.

Given this input, we first have to determine the time intervals that our model will use, which will be defined by breakpoints. Each release date and deadline of any repair will lead to a breakpoint, and each start and end time in the working schedule of any mechanic will produce one too. To eliminate duplicates and order the breakpoints, we simply use a sorted set implementation. Given the breakpoints, we can now very easily and quickly define the intervals I, as was described in Section 5.1.

Once we have the intervals I for our model, we have to divide the input list of intervals with capacity IC over them. Because, by construction, for all elements in IC, the mechanic that provides the capacity is continuously working during the interval, we may proportionally divide this capacity over the appropriate intervals in I. For each such element in IC we simply determine the intervals of I over which it should be divided and divide it over these intervals proportionally to their lengths. When capacity for a new group is passed to an interval, a new *intervalgroup* is created with the proper amount of capacity. If the group already exists within the interval, the amount of capacity is simply added to this *intervalgroup*.

We identify the variables for the model by looking at the repairs, one at a time. For each repair, we have to determine the time intervals that lie fully within its time span, and for each such interval we will create variables for that repair. Thus for each combination of a repair r and an interval i that is found in this way, we will have to create variables. We do so by looking at the subtasks from r. For each subtask j from r, we determine the *intervalgroups* IG_i, that are qualified for the type of work of j. For each $ig \in IG_i$ and this j we define one variable x_{jig}, which can take on any real positive value.

Once we have created the variables, only the constraints remain. If we construct a variable for a subtask j and a *intervalgroup* ig, we will pass a pointer to this variable to both j and ig. After creation of all the variables

we can then very easily create the constraints regarding subtasks and capacity. The constraints regarding Model Requirement 7 need to be handled a little more subtly, as we do not want to compute the feasible combinations of repairs and intervals again. As we already have computed these combinations during the creation of variables, we will create these constraints at that same time. In this way the entire model is built.

6.3 Planning at a fixed interval

Up till here we have only described a model for checking whether a list of repairs fits within the given capacity, and, if not, how much capacity we are short of. However, the question that needs to be answered is whether we can feasibly plan an extra repair. Computing whether there is too little capacity for the list of repairs, including the new repair, and solely base our answer on that, is inadequate. This is because the planning without the new repair may already have a shortage of capacity, due to forced repairs. This should not mean that new repairs can never fit. We can easily overcome this problem by solving the model twice. First, we compute the capacity shortage without the new repair in the list of repairs, then we compute the capacity shortage with this new repair included in the list. If the total capacity shortage did not increase by inserting the new repair, it is concluded that it fits.

For the computation of the initial capacity shortage, the exact LP formulation as described in Section 5.2 is used. We will build the model as was described in the previous chapter and solve it by an LP optimizer. When we want to compute the capacity shortage with the new repair added, we will not rebuild the entire model, as almost all of it remains the same. We will simply change it a little by adding the new repair. Since the new repair has a fixed interval, we may need to create extra breakpoints for the release date and deadline of this repair. If this defines a new breakpoint, an interval as was used by the previous model needs to be split, and we have to update this. How this is precisely done is described later on in this section. After this is done, the model is ready to have the new repair inserted to it. Building the variables for this new repair is done in the same way as was done for a repair in Section 6.2. The subtask constraints and the constraints regarding Model Requirement 7 are easily dealt with, as they have to be newly created. To update the capacity constraints, each capacity has a pointer to its constraint and the variables for a new repair are added to this constraint. When this is done, the LP optimizer is called again to compute the updated LP, which gives the capacity shortage. The answer to the question whether this new repair fits is *yes*, if and only if the capacity shortage has not increased by adding the new repair.

Remember, in Chapter 3, we said that the problem of *finding a suitable day* would be solved by applying this method seven times. Of course, we can

do this calculation a lot faster, as we will only build the initial LP problem once, and then reuse it seven times. Therefore, we only need one standard computation, and six *simple* computations, as we only need to delete and insert one repair for every of these six computations.

Splitting an interval

If we have to insert a new breakpoint, we need to split the interval in which it lies. We will create two new intervals and delete the old one. The capacities of the *intervalgroups* of the old interval will be divided over the two new intervals, proportionally to their lengths. For the two new intervals, variables are created to replace the variables in the old interval. Specifically, for each variable x_{jig}, we will create two new variables $x_{jig'}$ and $x_{ig''}$, where ig' and ig'' are the *intervalgroups* of the new intervals that were created from *intervalgroup* ig in the old interval, and j is some subtask. When creating these new variables, we must ensure that they end up in the proper constraints. Both the capacity constraints and the constraints regarding Model Requirement 7 are easily dealt with, for they have to be newly created. However, the subtask constraints already exist and need to be updated. Therefore, subtasks have a pointer to their constraint and new variables created by a split are added to this constraint.

6.4 Planning as early as possible

If we want to plan a repair as early as possible, we are given a release date for this repair and have to compute the earliest deadline with which the repair can be feasibly added to the existing planning. Again, we will start by computing the initial shortage of capacity for the existing repairs, without the new repair inserted. Then we will fix some deadline for the new repair and see whether the new repair can be planned, as was done in the previous section, and search for the earliest deadline in this way. However, we will choose the deadlines carefully at first, and we will more subtly check whether the new repair fits for a chosen deadline.

We will start with building the model exactly as was described in Section 6.2, but with an additional breakpoint for the release date of the new repair, and let our LP optimizer solve it. We will determine the first deadline, based on the *minimum execution time requirement* and the release date, and we will find the corresponding interval. Then we will set our first deadline to the end of this interval, and add the new repair to the planning with this deadline, as was done in the previous section. Because we do not want to introduce new breakpoints at first, we will always let our deadline be an existing breakpoint. To find the earliest interval in which the repair can be completed, we will do the following:

1. Compute the capacity shortage using our LP optimizer.
2. If there is not more capacity shortage than initially, we have found the earliest possible interval, return this interval i.
3. Say there is x more capacity shortage. Set a new deadline, defined by the previous deadline, and the time still minimally needed, which is computed using the *met*-requirement applied to x.
4. Find the interval in which this new deadline lies, and shift the new deadline to the end of this interval.
5. Given this new deadline, find the new intervals I_{new} from which the new repair may take capacity.
6. Create variables for the new repair and the intervals from I_{new}.
7. Go back to 1.

In Step 3 of this algorithm, we say that we will compute the additional time minimally needed to complete the repair by applying the *met*-requirement to x. By this we mean that we will look for $\max_{j \in J_r} m_j$ for the new repair r, and divide x by this number to compute this time value t. This is the time minimally needed to do x more work on the new repair. As we still need to assign x units of extra work to repairs in order to have a feasible plan and can only do so by increasing the deadline of r, we must minimally increase this deadline by t. Therefore, we may shift the deadline forward by t.

If we have completed this computation, we have determined the interval i in which our best possible deadline lies. However, we want to compute the exact deadline and therefore need to fix this deadline within this interval i. We will need to split i up properly, as Property 1 must remain satisfied for the new repair. We will apply an algorithm, which is nearly the same as the previous. The only difference is that we will leave out Step 4 and adjust the notion of *interval* to simply *time*. Also, we will set our first deadline to the start of i, the best possible deadline we can hope for. Because we set the deadline back, we must delete the variables that allow the new repair to take capacity from i. The algorithm is the following:

1. Compute the capacity shortage using our LP optimizer.
2. If there is not more capacity shortage than initially, we have found the earliest possible deadline, return this deadline d.
3. Say there is x more capacity shortage. Set a new deadline, defined by the previous deadline, and the time still minimally needed, which is computed using the *met*-requirement applied to x.

4. Given this new deadline, split the current interval at this new deadline, as was described in Section 6.3.

5. Create variables for the new repair and the first interval that was created by the split.

6. Go back to 1.

Once we have completed this computation, we have the smallest deadline d, such that the repair can be feasibly added to the planning at the interval specified by the given release date and the deadline d.

Remember, in Chapter 3, we said that the problem of *planning large repairs* would be solved by applying this method seven times. Of course, we apply the same trick as we did with *finding a suitable day*. So, again, we only build the initial model once, but we add a new repair, compute the earliest feasible deadline, and remove the repair again, seven times.

6.5 Planning a waiter job

When we want to plan a waiter job, we are provided with a first possible release date and have to compute the earliest interval of fixed length at which the repair can be planned. We will name such an interval of fixed length a *block of time*. Remember that this repair must be executed as fast as possible. To do so, we will require that it can only be planned within a block of time of a certain length. If we will ensure that this block is not longer than the minimum execution time and the plan is feasible, we will have ensured that the repair can be executed as fast as possible, which was our intention. As the project group wanted to have some slack, we will divide the minimum execution time by (100 - buffer)%, and let this be the length of our time block. Now we will search for the earliest such block in which the repair can be feasibly added to the planning. We will let the first time block we try start at the first possible release date. We will allow ourselves to shift the time block by 1 hour, since otherwise the problem will become too complex.

We will first build the initial model as described in Chapter 6.2, but this time we will add a breakpoint for all points in time at which our time block might start or end. We compute the first deadline d, which gives us our first time block. Then, we add the new repair in this time block to the planning, in the same way as any new repair. Then we will shift our time block by 1 hour, until it either fits in that block or we reach the end of the day. The algorithm for shifting the time block is the following:

1. Compute the capacity that is short using our LP optimizer.

2. If there is not more capacity short than initially, we have found the earliest time block, return this time block i.

3. Shift the time block by 1 hour
4. If the new deadline lies outside the time span of the current day, return that no plan is feasible
5. Create variables for the new repair and the new intervals
6. Delete the variables of the new repair and the removed intervals
7. Go back to 1.

At the end of this computation, we either have the earliest interval at which we can plan the waiter job, or we know it cannot be planned that day.

6.6 Insight into the planning

A receptionist must be able to get more insight into the planning than the simple answers to his planning questions provide. What a receptionist would like to know the most is how much capacity of each qualification is still available to use for planning repairs. Furthermore, he would like to know how much shortage of capacity already exists for each qualification. He also wants to know how much work of each qualification has already been planned. Therefore, in Chapter 3, we have said that we would give the receptionists three additional methods to gain a deeper insight into the planning. These are:

1. View of a week
2. View of a day
3. Detailed view of a day

To view a week, the project group indicated that we only need to calculate the total amounts of capacity shortage, available capacity, and assigned capacity as a whole for each day, thus neglecting qualifications. So, we just add up all work of repairs that need to be done, and we calculate the total initial capacity. From these two numbers we straightforwardly compute the desired values.

However, to view a day, either detailed or not, we need to compute how much shortage of capacity we have and how much is available, for each individual qualification in a given time interval. Computing the amount of shortage or how much is available for a particular qualification is a difficult task. First of all, we need to define when capacity from a particular qualification is still available, and how much is available. This is not straightforward, as we cannot just solve the standard LP formulation of our problem and look at how much capacity of this qualification is still unassigned. This is inadequate, because we might be able to feasibly shift repairs away from groups

that are capable of this qualification, thus freeing additional capacity of this qualification. Therefore, we will define the amount of capacity that is still available for a certain qualification, as the maximum amount of capacity of this qualification that can be made free, which is equal to the amount of work of this qualification that can still be feasibly planned. See Example 8 for further illustration.

Example 8 *We will use a very simplified example to show how we will compute available capacity. We have two mechanics: mechanic m_1 is capable of APK & ML, and mechanic m_2 is capable of ML & SL. Both provide 4 hours of capacity. We have planned two repairs. Repair r_1 needs 2 hours of APK, and repair r_2 needs 2 hours of ML. A feasible assignment of capacity to repairs would be to let m_1 execute r_1, and let m_2 execute r_2. If we look at the available capacity of SL, we see in our feasible assignment that m_2 still has 2 hours unassigned. However, we can free more capacity by shifting repair r_2 to m_1, completely freeing m_2, thus giving a total of 4 hours available capacity for SL, which is the relevant number, as it indicates how much work of this qualification can still be feasibly planned.*

The amount of capacity shortage for a particular qualification faces similar problems. How do we define this? We could just look at all the subtasks that require this qualification, and sum up the amount of capacity that these still need in our LP solution. But again this neglects the fact that we might feasibly shift repairs in order the increase this shortage. Therefore, we will define the capacity shortage for a certain qualification, as the extra shortage of capacity that subtasks with this qualification cause. Example 9 further illustrates this.

Example 9 *Like Example 8, we will use a very simplified example. We have the same two mechanics, mechanic m_1 is capable of APK and ML, and mechanic m_2 is capable of ML and SL. Both provide 4 hours of capacity. But this time we have planned two other repairs. Repair r_1 needs 4 hours of APK and repair r_2 needs 5 hours of ML. A best possible assignment of capacity to repairs would be to let m_1 execute r_1, and let m_2 execute 4 hours of r_2. So, we are 1 hour of work on ML short. Now, we will look at how much capacity of APK is short. In our assignment, there is no shortage of capacity for APK. But we could let m_1 execute only 3 hours of APK and the remaining 1 hour of ML. This assignment is just as good as the previous, yet, this time, we are 1 hour of work on APK short. This is the relevant number, as it indicates that without the APK repair, we would have 1 hour shortage less.*

Capacities still available

We have defined the amount of capacity of a certain qualification that is still available as the maximum amount of capacity of this qualification that

can be made free. This is equal to the amount of work of this qualification that can still be feasibly planned. To compute this for a given time interval i and a certain qualification q, we will make a dummy subtask that requires qualification q. Before we add it to the planning, we will let our LP optimizer compute the maximum amount of work that can be assigned, say this is x_{excl}. Then we will add the dummy subtask to the planning in i, but without enforcing the *minimum execution time requirement* or the subtask constraint on this subtask, such that it can take an unlimited amount of capacity. Then we will compute the maximum amount of work that can be assigned including this repair, say this is x_{incl}. The additional amount of work that our LP optimizer could assign, by completely filling the possible free spaces in qualification q during i, is $x_{incl} - x_{excl}$, which is the available capacity of q during i.

Because the amounts of available capacities must always be computed for all qualifications, and possibly for different time intervals, we only build the whole model once and reuse it by slightly adjusting it.

Shortage of capacities

As we argued, we will define the capacity shortage for a certain qualification, as the extra amount of shortage that is caused by work of that qualification. To compute this value for a given time interval i and a certain qualification q, we will first solve our standard LP problem, but without the subtasks requiring qualification q that are present in i. If a subtask lies fully within i, we just remove it from the problem. However, a subtask j can also partially lie within i. We cannot just remove it, as (part of) the shortage it causes may very well lie outside i. Therefore, we compute how much work we can maximally assign to j within i, based on the *minimum execution time requirement*. This amount is removed from the amount of work needed on the subtask, and has no more influence on the shortage of capacity. Furthermore, we will remove the variables of these subtasks that lie inside the time interval we are investigating. Note that a repair may consist of several subtasks with qualification q. If this is the case, we of course compute how much capacity we can maximally assign to these subtasks together.

Once we have removed (parts of) the subtasks with qualification q that lie in i, we will let our LP optimizer compute the shortage of capacity, say this is s_{exl}. This value indicates how much shortage of capacity is already being caused by the other subtasks. In order to compute how much extra shortage the removed subtasks cause, we simply restore them and recompute the shortage of capacity, say this is s_{incl}. Then the capacity shortage caused by this qualification in this interval is $s_{incl} - s_{excl}$.

Because the shortage of capacities must always be computed for all qualifications, and possibly for different time intervals, we only build the whole model once and reuse it by slightly adjusting it.

Chapter 7

Additional practical problems

Next to the theoretical aspect of the capacity planning problem, there are some additional practical problems that we need to deal with. These are discussed in this chapter. We will start with discussing why we recompute the solution to the initial model every time for a new planning question. Then, we will discuss how we can make our problem instances smaller by shortening the planning horizon. We will then discuss a problem that may arise as receptionists can plan in parallel. Also, the system needs to run on a rather old operating system, the System i5 from IBM. This system does not allow the use of many LP optimizers, therefore, we have used an alpha version of QSopt. We will end this chapter by discussing the problems we encountered while using this alpha version.

7.1 Storing the previous solution

All our computations start with building the current instance of our model and solving it from scratch. We do this over and over for every new planning question. We already argued in our performance analysis that this uses a major part of our total computation time. Since our planning changes very little over time, and we recompute it every time we plan a new repair, we would achieve a much better runtime if we would reuse the solution to the previous planning question. This would however mean that we need to store the previous instance of our model, the solution found by our LP optimizer, and the internal state of the LP optimizer. This would cost a lot of space in the database storage of Rüttchen, which was undesirable to Rüttchen, and we cannot store and reuse previous solutions. Luckily, the current method of building and solving the instance from scratch, performs well enough.

7.2 Shortening the planning horizon

Our planning horizon is in general very large, since there can be repairs that are planned over, say, 30 days. Repairs that can be planned over multiple days, or more specifically, whose release date and deadline are not on the same day, will be named *multiple day repairs*. If we have a multiple day repair that spans 30 days, then our instance of the capacity planning problem also spans at least 30 days. This would cause our LP model to be big, and thus our runtime to be large. While, in most cases, only considering a single day is adequate, which would of course make our LP model a lot smaller. Therefore, we will at first try to make a plan only for the day at interest, and insert additional days into our model if necessary. But making a plan for only one day gives problems with multiple day repairs. We do not know how much we need to plan on such a repair at a single day. Therefore, for each of these repairs, we need to make an explicit distribution of the work over the days at which a vehicle is present at the workplace. So, if we store a new multiple day repair, we examine the solution to our LP model, and extract this distribution of work on the repair from this solution. In this way, we are certain that the distribution is feasible. Then, we will store this distribution together with the repair in the database of Rüttchen. If we now examine a particular day, we can straightforwardly determine how much work on multiple day repairs we need to plan.

However, we might need to change this distribution later on, in order to feasibly plan a new repair. Therefore, shifting multiple day repairs must be possible. We will only try to shift them if it is necessary and possibly useful. We will use the following techniques for the different algorithms for our planning questions:

Planning at a fixed interval: We will start with retrieving all days between the release date and deadline of the new repair. Then, if the plan is feasible, we are obviously done. But if the plan is not possible, we will check whether removing the work on multiple day repairs would help. If so, shifting them might help, and we insert all the additional days to which this work might be shifted into our model, and we recompute the plan. After this, we may insert additional days again, as we might have picked up new multiple day repairs.

Planning as soon as possible: For this method, we need to optimize. To truly optimize, we should allow all possible shifts of multiple day repairs in advance. This is not too smart however, as our LP model will then become very big again. Therefore, we will retrieve the day of interest, and only allow multiple day repairs that have work planned at this day, to shift. Thus, we will retrieve the days within the time span of these repairs as well.

Planing a waiter job: This technique is exactly similar to *Planning as soon as possible.*

Insight into the planning: For this method, we will completely neglect the possible shifting of multiple day repairs. This is done, since the computations already take a lot of time for a single day.

One should note that, when we shift multiple day repairs to make a new repair fit, we thus have altered their distributions, and their previous distributions are not feasible anymore. Therefore, we should update the distributions. In order to avoid recomputing distributions, we will try to plan work on multiple day repairs as much on *free* days as possible. In general, days that lie further into the future have much less work planned, and need to be retrieved much less often in order to compute the planning questions. Therefore, we would like to plan work on multiple day repairs as far into the future as possible. If we do this, we will only need to shift them occasionally. Moreover, if a multiple day repair does not have work planned at days which are used often by the computation of planning questions, then we do not need to retrieve additional days to shift this repair. Therefore, every time before we compute distributions, we will let our LP optimizer shift the work planned on these repairs as far into the future as possible, while still having a feasible planning.

One should be aware that shifting work on multiple day repairs towards their deadline does not mean that they cannot be executed at earlier days. The list of the repairs the head of the workplace receives includes the repairs on all vehicles that are present at the workplace that day. Therefore, he can start work on multiple day repairs any time he wants if he has spare capacity. In order to maintain consistency in the planning, the head of the workplace is informed how much work is assigned to the multiple day repairs at the current day by the planning system. The head of the workplace should try to do at least this amount of work on these repairs, such that the plans of future days cannot become infeasible.

Example 10 *Suppose we have a repair that consists of the following two subtasks:*

Qualification	Time needed (hours)	Max Mech.
APK	*8*	*1*
ML	*8*	*1*

This repair is planned between Monday and Friday, a feasible distribution of the work on this repair over the days might be:

Qualification	Monday	Tuesday	Wednesday	Thursday	Friday
APK	*3*	*1*	*4*	-	-
ML	-	*1,5*	*2*	*4*	*0,5*

It is likely that Monday is more busy than Friday, and it is likely that there are more planning requests for Monday than for Friday. Therefore, we will shift the work to the back as much as possible, which could give the following feasible distribution:

Qualification	Monday	Tuesday	Wednesday	Thursday	Friday
APK	-	-	-	*3*	*5*
ML	-	-	-	*5*	*3*

One can see from this latter distribution, that this new repair can be neglected when planning on either Monday, Tuesday or Wednesday.

7.3 Parallel storage

There exists some time between the moment that the receptionist gets the answer to his planning question and the moment at which he gives the command to store the repair. As there can be multiple receptionists that plan repairs in parallel, there is a slim chance of two planning scenarios crossing. This crossing could lead to infeasible plans. An example of this is the following:

Example 11 *At the following points in time, the following events could occur:*

12:00:05 *Receptionist Peter asks whether he can plan repair* r_1*, which needs 4 hours of APK, on Monday, the algorithm answers* yes.

12:00:47 *Receptionist John asks whether he can plan repair* r_2*, which needs 3 hours of APK, on Monday, the algorithm answers* yes.

12:01:09 *Peter stores* r_1 *at Monday, which is still feasible, but uses all the free capacity on APK.*

12:01:23 *John stores* r_2 *at Monday, which is no longer feasible, as* r_1 *has already taken all the remaining capacity on APK.*

To overcome this problem, we need to lock the database for the days we are investigating. We do not want to lock it from the moment the receptionist gets an answer, until the moment that he stores the repair, as this can take far too much time. Therefore, we will only lock the database when a receptionist wants to store a repair. Since there is no function in WPS that tells us whether the database has changed since our original computation, we need to recompute our plan before the final storage. Luckily, we know which days we need to investigate in advance, as we pass this information from the original planning computation. Therefore, at the command of a

storage, we get and lock the planning of these days, and recompute whether planning the repair is feasible. If it is feasible, we store it in the database, if not, we return that we could not store it. After this check and the possible storage, the database is unlocked again. This way, we guarantee that no infeasible plans can be stored. In Example 11 this would mean that John cannot store his repair, and needs to compute another planning. The customer on the phone will not really like this, but it is far better than storing infeasible plans. Moreover, a situation like Example 11 will seldom occur in practice.

Our method to avoid possibly infeasible plans, will mean that we recompute each planning again before we store it. This nearly doubles the computation time, but we need to do this to avoid infeasible plans. Furthermore, recomputing the plan before storage will not take long, as we already know which days we need to retrieve in advance, and the planning question is of the simplest form, that of *planning at a fixed interval*. Therefore, the time needed to recompute the plan is likely to be very small, making the time that the database is locked small.

7.4 The i5 System from IBM

The machine that our program needs to run on is the rather old i5 System from IBM. The main reason why Rüttchen uses this machine, is because ADIS is built in ILE RPG for this machine. Since the new WPS system needs to interact with ADIS, and our system needs to interact with WPS, our new system will need to run on the i5 as well. To this end, Rüttchen uses a compiler that compiles Java code, such that it can run on the i5. Therefore, all of the code we need to write can be written in Java, without causing any problems. Problems do arise from the fact that we need to use an LP optimizer. There are plenty of optimizers written in C code, but they are no good to us. We need to have an optimizer that we can use on the i5 system. Normally we could use the commercial, but expensive, optimizer CPLEX from ILOG. However, there does not yet exist a CPLEX version that is compatible with the i5. Luckily, there does exist a Java version of an academic code, named *QSopt*, which the authors of this code, David Applegate, William Cook, Sanjeeb Dash, and Monika Mevenkamp, kindly made available to us. Therefore, we use this program to solve our LP models. This code performs really well, but we did encounter some problems as it is still an alpha version. We will discuss these problem in greater detail in the next section.

Rüttchen could not indicate which performance results we could expect from the i5 System, and testing on the final production machine has not yet started at the moment this thesis is written. Therefore, we do not really know what the final runtime will be for our different algorithms. We have

tested them on a regular Windows desktop, the results of which are given in Chapter 9. We do not know how these results will compare to the results for the final i5 System.

7.5 QSopt

The QSopt linear programming solver has been developed by David Applegate, William Cook, Sanjeeb Dash, and Monika Mevenkamp, and can be used for free for research or educational purposes. The QSopt program and documentation can be downloaded from [2]. The alpha Java version of QSopt has been semi automatically generated from an older version of the C code. We are very pleased with this optimizer, but we did encounter some problems, which we will discuss in this section. The first small problems arise from the fact that the code is semi automatically generated, and the result is that all objects and methods have straightforward names like *a*, *aa*, *b*, etcetera. Therefore, while debugging, we had no idea what was happening inside QSopt. Exceptions from within QSopt, which by the way are only thrown when we pass it incorrect data, contain some text to indicate what went wrong, but it still is difficult to get to the real reason for the exception and to correct our program. Furthermore, the documentation for the Java version is limited, which has cost us some extra time in order to appropriately use the program. But, all the problems caused by the lack of extensive documentation are very minor, and with a little more time, were all solved easily.

Bigger problems are caused by the fact that the alpha Java version misses some functions to slightly change our LP model. This is mainly due to the fact that it is generated from an older C version of the program, which did not have these functionalities implemented in it yet. The current C version of QSopt does implement all these required functionalities, but they are of course of no use to us. We will summarize the functionalities that we cannot use, explain why we cannot use them, and show how we worked around them.

Deleting variables: This function is actually available in our Java alpha version, but we cannot use it. To better understand why we cannot delete variables, we should first explain how we can refer to existing variables. The only way to refer to existing variables is by an integer index, which probably points to the position this variable has within the QSopt data structure. Now, QSopt does provide a functionality for deleting variables, but when we do so, the integer indices of the other variables seem to have shifted. Therefore, we cannot use this method, and have to work around it. We do so very simply by changing the upper bounds on these variables to 0. Note that the same problem

occurs when deleting constraints, but we can easily work around this as well.

Changing the right hand side values for constraints: This function simply does not exist within our version of QSopt, while it does in the C version. Luckily, we can work around it. We calculate the difference between our desired right hand side value and the current right hand side value, say this is d. Then, we introduce a new variable in the left hand side of this constraint, that has a fixed value of $-d$. It can easily be verified that this does the trick.

Changing the objective coefficients of variables: Again, this function simply does not exist within our version of QSopt. And unfortunately, we do not see any straightforward way to work around it. This causes a problem, as we want to use it to shift our multiple day repairs to the back, which could have been computed efficiently by changing the objective coefficients. Therefore, we need to design some method to work around it. We have chosen to replace the variables at interest by new variables with the proper objective coefficients. This will cost some additional time, but we see no better way to work around this problem. Luckily, this is the only method we need this function for.

Overall, we are very pleased with QSopt, since we really need a LP optimizer and have no good alternatives. Furthermore, even though it is an alpha version, it performs very well and it never once erred in our tests. The only problem lies in the fact that it misses some functionalities, most of these can be worked around easily, without causing any harm, but missing the functionality to change the objective coefficients is inconvenient.

Chapter 8

The program

The final intention of this project was to build a Java program that computes the planning questions for Rüttchen. Therefore, we have constructed a Java program that provides these desired functionalities. The way these computations are performed is based on the theory provided in this thesis. In this chapter we will very globally discuss our program. The way it is built up, and the way the computations are performed.

We need to provide functions to the WPS system that:

- Answer the planning questions as described in Chapter 3
- Store a repair
- Give insight into the planning

We implemented all these functions in our program and allow the WPS system to call them. WPS can access these functions at the class *PlanningSystem*, this will also be the only class that WPS has access to. What happens behind it is of no interest to the caller of this class.

Not only do we need to provide the WPS system with functions, but we require functions from the WPS system as well. In particular we need to be able to:

- Get the capacities
- Get the list of repairs
- Get the buffer values
- Let WPS reserve substitute vehicles
- Store repairs
- Lock the database

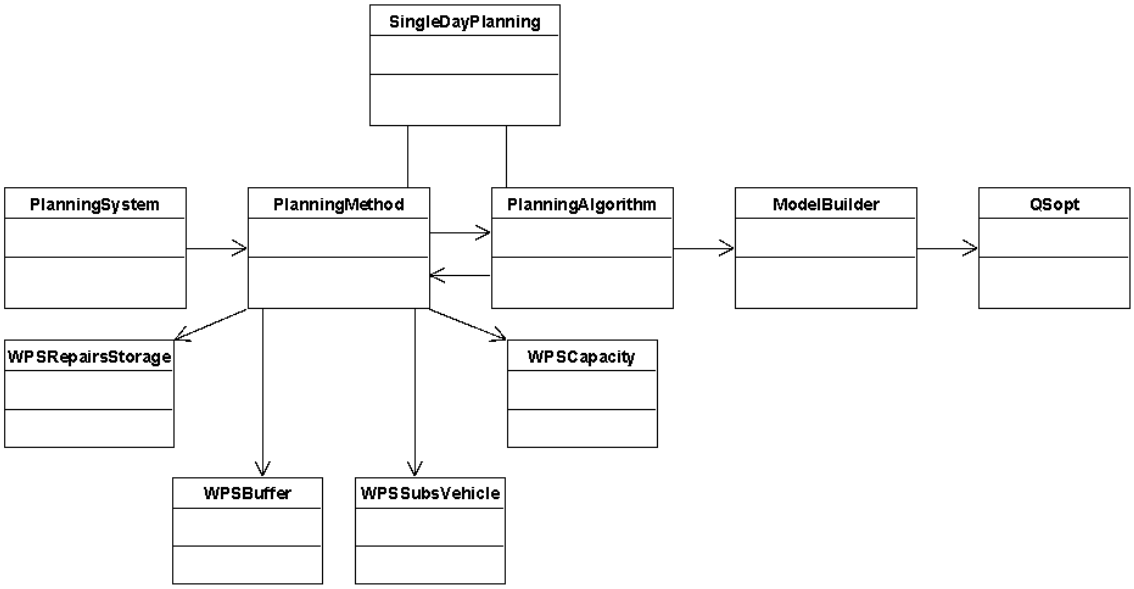

Figure 8.1: A very simplified UML of the program

Locking the database is done by the same class that stores and gets the repairs, as we lock the repairs in order to lock the planning; this is the class *WPSRepairsStorage*.

The very simplified overall form of the program in Unified Modelling Language can be seen in Figure 8.1.

As we said, all planning questions enter our program through the PlanningSystem class, after which the request is processed and computed by our program. In general, a planning question is processed by our classes in the following order:

1. The WPS system calls one of our planning functions in PlanningSystem.

2. PlanningSystem passes this request to one of the main planning functions in PlanningMethod.

3. In PlanningMethod all data from WPS about the required plannings are collected.

4. These data are combined into SingleDayPlanning objects; these objects represent the planning of a single day. At construction, these objects already split up the day into the appropriate intervals and divide the available capacities over these intervals.

5. These SingleDayPlannings are passed to PlanningAlgorithm, and PlanningAlgorithm is asked to compute the plan.

6. PlanningAlgorithm has all the data it needs and makes the LP model in QSopt. It does not do this directly, but uses a ModelBuilder to build the LP model more easily.

7. PlanningAlgorithm asks QSopt, by the ModelBuilder, to optimize the model.

8. PlanningAlgorithm performs the rest of the computations that are needed with the help of the ModelBuilder and QSopt.

9. PlanningAlgorithm forms an answer and passes this back to PlanningMethod.

10. Possibly, PlanningMethod needs to check for a substitute vehicle and/or store the repair by classes from WPS.

11. PlanningMethod passes the answer to the planning question back to the PlanningSystem, which returns the final answer to the WPS system.

We have given a very brief overview of our program. In reality, our program is much bigger and more complex. It uses a range of extensive classes in order to shorten its computation time and to be correct. To give an idea of the size of this program, it contains 25 classes, with a total of around 6500 lines of code. It has cost a considerable amount of time to construct this entire program, as it not only implements the main ideas from this thesis, but has to cope with all possible details in order to be completely correct. However, we do not want to waste our time with discussing these details in depth. As the main ideas behind this program are already discussed in this thesis, we will end the discussion on our program here.

Chapter 9

Computational results

We have given our theory on how our computations should be performed, and want to investigate the runtime in practice. Therefore, we will test our program on real instances in order to see how well it performs. At the end of this chapter we will be able to conclude whether our program satisfies Goal 1. We will start this chapter by defining the problem instances on which we will test our program. Then, we will give the results of these tests, and finally draw conclusions from these results.

9.1 Problem instances

Most of all, we would like to test real instances of the planning problems that occur at the workplaces of Rüttchen. However, we are only given the planning of one day at one of their workplaces. We will first set up tests for this instance, as it gives the best indication to what we can expect of our runtime in the final setting at Rüttchen. But because this is the only instance, we will also create our own instances with some randomness, and test them. For each planning instance we will test the following planning questions:

- Planning at a fixed interval (where the interval is 9:00 to 17:00)
- Planning as soon as possible (where the release date is 9:00)
- Planning a waiter job (where the release date is 9:30)
- View of a day

For the planning questions that need to insert a new repair, we will randomly generate one. We will let the number of subtasks be 1, 2, or 3, with respectively 50%, 30%, and 20% chance. For each subtask, the chance that a certain qualification is selected for it, is proportional to the amount of capacity of that qualification that is present at the workplace. The amount

of work (in minutes) needed on each subtask is drawn from the uniform distribution between 30 and 180. Finally, the maximum number of mechanics that may simultaneously work at a subtask is 1, 2, or 3, with respectively 60%, 30%, and 10% chance.

The real instance that we are given contains the capacities of 11 mechanics, which operate by two schedules: a morning shift and an evening shift. Together they provide 10 different sets of qualifications, and there are 11 different qualifications. Furthermore, there are 22 repairs planned, together they contain 36 subtasks. Experiment 1 will test this original problem. Then, in order to make our problem more complex, we will change about half of the release dates and deadlines of the repairs in order to cause more breakpoints; this will be Experiment 2.

As we said, we will generate the next instances ourselves. We want these instances to resemble reality as much as possible. In reality, at each workplace, mechanics can only have 2 different working schedules, the morning shift or the evening shift. Furthermore, the real instance that we have of the capacity present at the workplace is a good resemblance to the capacity that is present at any workplace in reality. Therefore, we will reuse these capacities for all our testing instances; we only might double or triple them for some instances in order to have enough capacity present at the workplace. All the repairs that we will put into the planning will be randomly generated in exactly the same way as we do when we randomly generate a new repair for our planning questions. However, for these repairs, we have to fix release dates and deadlines as well. We will draw the release date r_j for a repair j from the uniform distribution between 8:00 and 13:00. In order to make a feasible initial plan, we must fix an appropriate, but still partly random, deadline. Since the earliest possible feasible deadline is $r_j + met_j$, where met_j is the minimum execution time needed for j, we will draw our deadline from the uniform distribution between $r_j + met_j$ and $r_j + met_j + 360$ (in minutes). In Experiment 3 we will use the capacities from our real instance and randomly generate 20 repairs, where we will round the release dates and deadlines to 1 hour. Then, we will do the same experiment, but we will round the release dates and deadlines to 30 minutes; this will be Experiment 4. We will double the capacities and generate twice as many repairs for Experiments 5 and 6, which use the same rounding routines as Experiments 3 and 4. Finally, we will test the limits of our program, by Experiments 7 and 8, for which we will triple our capacities and use 60 repairs; again we use the same rounding routines as before.

9.2 Results

We would like to have tested our problem instances on the final System i5 production machine at Rüttchen, but this system is not yet fully operational.

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	12	12	0.3	0.3
asap	12	12	0.5	0.7
wait	21	21	0.7	1.3
view	12	12	1.6	1.6

Table 9.1: Results for Experiment 1

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	22	22	0.5	0.5
asap	22	22	0.8	1.3
wait	27	27	0.9	1.0
view	22	22	5.0	5.0

Table 9.2: Results for Experiment 2

Therefore, we have tested our instances on a regular Windows desktop, a Dell Optiplex GX270 P4 3,0 Ghz with 512 MB RAM. We compiled our program with the Java compiler (Version 1.4.2_0.5).

For all experiments, we will run 20 instances for every planning question, and will show for each planning question:

- The average number of breakpoints the instances caused (*avg b*)
- The maximum number of breakpoints an instance caused (*max b*)
- The average runtime in seconds (*avg rt*)
- The maximum runtime in seconds (*max rt*)

In our resulting table we will denote the tested planning question in the column *pq*. In this column, *int* denotes *Planning at a fixed interval*, *asap* denotes *Planning as soon as possible*, *wait* denotes *Planning a waiter job*, and finally *view* denotes *View of a day*. The results of the various experiments are shown in Tables 9.1 to 9.8:

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	20	21	0.3	0.4
asap	20	21	0.3	0.6
wait	28	30	1.1	1.6
view	20	21	2.5	4.3

Table 9.3: Results for Experiment 3

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	28	30	1.0	1.3
asap	27	30	1.3	1.9
wait	28	30	1.3	2.1
view	26	29	7.5	13.5

Table 9.4: Results for Experiment 4

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	20	21	0.9	1.4
asap	20	21	1.2	2.5
wait	29	31	3.5	5.0
view	20	21	7.8	11.1

Table 9.5: Results for Experiment 5

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	28	30	3.3	4.3
asap	26	30	3.0	5.2
wait	30	31	3.6	4.7
view	29	30	22.3	33.3

Table 9.6: Results for Experiment 6

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	20	21	1.7	2.7
asap	21	21	2.0	3.9
wait	30	31	6.1	8.8
view	21	21	14.5	21.8

Table 9.7: Results for Experiment 7

pq	*avg b*	*max b*	*avg rt*	*max rt*
int	30	31	4.6	8.3
asap	30	30	4.9	8.0
wait	30	31	6.8	8.7
view	29	30	56.3	75.7

Table 9.8: Results for Experiment 8

9.3 Conclusions

If we globally look at the results from the previous section, we can see that, in general, all planning questions are computed in time to satisfy Goal 1. Only if the number of breakpoints is large, our program has some difficulties to complete its computations within the required time bound, see Experiments 9.6, and especially 9.8.

For only one real problem instance, we can demonstrate that Goal 1 is satisfied, as this is the only real instance we have. For this instance we can see that our system easily satisfies this goal, as the system answers in much less than 1 second on average, and in about 1 second in the worst case. However, this is an instance of the planning for a workplace of regular size. We want to know whether our system satisfies Goal 1 for workplaces of larger size too. Since we do not have real instances of these workplaces, we can only try to create instances that resemble them as much as possible. If we look at the biggest possible workplace, we have a maximum of about 60 repairs each day, and about three times the number of mechanics. Most repairs at these workplaces are on company vehicles, which mostly have standard release dates and deadlines, thus causing few breakpoints. Therefore, these instances are most likely to correspond to Experiment 7, for which we can see in Table 9.7 that all computations are performed within a few seconds. Another type of a difficult planning is a workplace that has fewer repairs per day (about 40), but all the planned repairs are personal vehicles, which, in general, have many different release dates and deadlines. These instances are most likely to correspond to Experiment 6. In Table 9.6 we can see that all computations are performed within the required time bound as well.

We can thus conclude that experiments indicate that our system does satisfy Goal 1 for all planning questions at (probably) all workplaces. The number of breakpoints seems to have the largest influence on the computing times.

One must however note that computing the problem of *planning a waiter job* can take considerably more time than the other computations. It is completed well within 10 seconds for the most difficult instances, but it still takes rather long. This is due to the fact that it can cause the model to have many breakpoints.

Chapter 10

Related models and solution strategies

In this chapter we will discuss problems that are related to our capacity planning problem. We will first show how we will solve the capacity planning problem if we do not need to satisfy the *minimum execution time requirement* (Requirement 7). Then we will look at our capacity planning problem, but with the property that we have only one subtask per repair, or equivalently, that we can execute subtasks in parallel. We will show that we can model both these problems as standard maximum flow problems. Then we will look at our capacity problem again, but with the additional property that all subtasks of a repair require the same qualification. This problem can also be written as some sort of flow problem, but with additional constraints. We will design our own algorithm that computes the optimum for this problem. Finally, we will look at the problem that the workplace chef is facing, that of making a feasible schedule, while satisfying additional requirements. We will argue how this problem can be solved with the use of the theory provided in this thesis.

10.1 Leaving out the minimum execution time requirement

At the start of this project, we only had to satisfy Model Requirement 6, which was replaced by the stronger *minimum execution time requirement* later on. In this section we show how we will solve the problem, if we do not need to satisfy the *met*-requirement.

In this case we would use the same strategy to partition time into intervals, which gives nearly the same LP formulation as the capacity planning problem discussed in this thesis, but we can omit the *met*-requirement. We will use the same notation as was used in Section 5.2, which would give the

following model:

$$\max \quad \sum_{j \in J} \sum_{ig \in IG} x_{jig}$$

subject to

$$\sum_{j \in J} x_{jig} \leq C_{ig}, \text{ for each } ig \in IG;$$

$$\sum_{ig \in IG} x_{jig} \leq S_j, \text{ for each } j \in J;$$

$$x_{jig} \geq 0, \text{ for each } j \in J, ig \in IG.$$

This model has much fewer constraints than our model, so it would be much easier for an LP optimizer to solve it. But more importantly, we can straightforwardly formulate it as a maximum flow problem. We will create two sets of vertices. The first set V_1 has one vertex for every subtask, and the second set V_2 will have one vertex for every capacity. Then we will create a source s, and make arcs from s to every vertex $j \in V_1$, with a maximum capacity equal to S_j. Then, for each variable in our LP model, we will create a corresponding arc in our flow problem, with an unlimited maximum capacity. Finally, we will create a sink t, and make an arc from every vertex $c \in V_2$ to t, with a maximum capacity equal to the amount of capacity provided by the capacity corresponding to c. This completes the maximum flow problem we need to solve. The flow model of our problem is graphically shown in Figure 10.1. There are very efficient software packages available to solve maximum flow problems. These solvers can solve instances with over 10.000 vertices and 30.000 arcs within a second, see [4]. Our flow network is bipartite and unbalanced, since the number of subtasks is likely to be a lot smaller than the number of *intervalgroups*. This allows us to use an even faster algorithm, see [1]. Therefore, we would not have had any problem with solving this model for the instances we can encounter at Rüttchen.

10.2 Only one subtask per repair

We will consider our capacity planning problem again, but this time we have that each repair consists of only one subtask. This problem is equivalent to our capacity planning problem without the requirement that subtasks cannot be executed in parallel. For this latter problem, we can simply make a repair for every subtask, which gives us the former problem.

In addition to the problem of the previous section, we now do need to satisfy the *minimum execution time requirement*, but only for repairs with

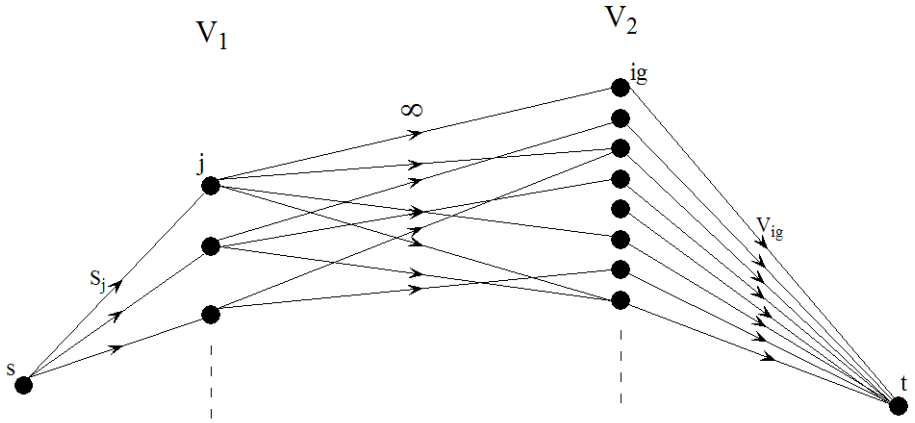

Figure 10.1: The flow model of our problem without the minimum execution time requirement

one subtask. This makes the problem a lot easier than our capacity planning model, as we do not need to require that multiple subtasks *together* cannot take more capacity from an interval than the *met*-requirement allows. In addition to our previous flow model, we only need to ensure that a single subtask cannot take more capacity from any interval than the *met*-requirement permits. Therefore, we will take the flow model from the previous chapter as our basis. We will create an additional vertex for every combination of a subtask j and an interval i, let this be v_{ji}, and make an arc from j to every v_{ji} that has a maximum capacity that ensures the *met*-requirement. More specifically, an arc from a subtask at which maximally m mechanics can work simultaneously, to an interval of length i, has a maximum capacity of $m \cdot i$. Next, we will delete all old arcs from any subtask j to any *intervalgroup* ig, and replace it by an arc from v_{ji} to this ig. This completes the new model. Every arc from any subtask to any capacity in the previous model is replaced by a path of 2 arcs in the new model, thus allowing flow to travel in the same directions. Furthermore, we have ensured that the *met*-requirement is satisfied by adding new vertices between subtasks and intervals. A graphical representation of this model is given in Figure 10.2.

10.3 Only one qualification per repair

In this section we make the problem of the previous section resemble the capacity planning problem of this thesis even a little more, by allowing multiple subtasks per repair, with the additional property that all subtasks of the same repair require the same qualification, but still can have different numbers of maximum mechanics. To solve this new problem, we will reuse

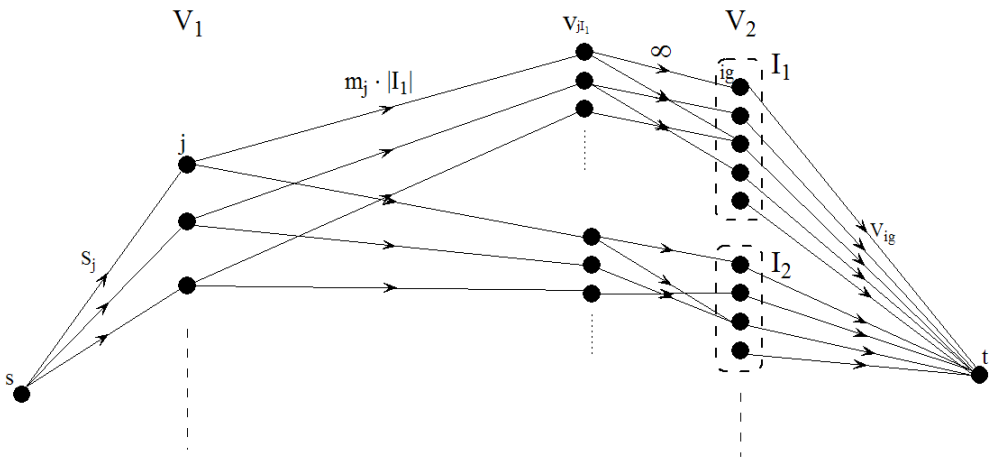

Figure 10.2: The flow model of our problem with the additional property that repairs consist of only one subtask. In this figure, m_j denotes the maximum number of mechanics that can simultaneously work at subtask j, and $|I_1|$ is the length of interval I_1.

the basis of our flow model of the previous section. Additional to the requirements of the previous problem, we now require that subtasks of the same repair *together* may not use more capacity than allowed by the *minimum execution time requirement*. Unfortunately, we do not see how we can incorporate this requirement in a standard maximum flow formulation. We will use the graph from the previous section, but will not run an existing maximum flow solver on it; instead we will develop our own algorithm to compute the optimal flow that also satisfies our additional requirements. We will do so by finding augmenting *c-paths* and sending flow over them.

Definition 3 *A c-path is a path with possibly additional cycles.*

Our algorithm thus needs to look for augmenting c-paths in our graph. We will base our algorithm on the standard technique to find a single augmenting path in the residual graph as was already used by Ford and Fulkerson [6]: We keep track of all vertices to which we can send additional flow from s; these vertices will be named *reachable*. Recursively, we will try to send flow one arc further from the vertices that could already be reached, possibly making new vertices reachable. We have found an augmenting path if t becomes reachable. For further information on finding augmenting paths in this way, we refer the reader to [5].

For regular flow problems, it is adequate to search only for augmenting paths, and not for the more general augmenting c-paths. This is due to the nature of the problem, if there does not exist an augmenting path, we can

apply the *maximum flow minimum cut theorem* from [6] to prove that the flow is optimal. However, in our problem, it might very well occur that there is no augmenting path, but the flow is not optimal. Consider the following example instance of our flow model:

Example 12 *We want to send additional flow from the reachable subtask vertex j_1 over the arc α_1 to the vertex $v_{j_1 i}$. But we are already sending 15 units of flow from j_2 over the arc α_2 to the vertex $v_{j_2 i}$. Furthermore, the interval corresponding to i is 15 minutes long, j_1 and j_2 are subtasks of the same repair, and the maximum number of mechanics for j_1 and j_2 are respectively 3 and 1. So, the flow over arc α_2 prevents that additional flow can be send over arc α_1. We can work around this problem, by sending some of the flow over α_2 back to j_2, and then send it back to s. Then we are allowed to augment over α_1, making $v_{j_1 i}$ reachable. If we can reach t from $v_{j_1 i}$, then we have found a flow improvement, while no simple augmenting path exists. Note that this flow can be improved, since we can send 15 units of flow over α_1 to t, and only need to send 5 capacity back over α_2 to s. Thus we would achieve a flow improvement of 10.*

Therefore, we do not necessarily have a cut with capacity 0 in our residual graph if there is no simple augmenting path. However, in our algorithm we will either find an augmenting c-path or prove that there is a cut with capacity 0.

In order to apply the max-flow min-cut theorem to our problem we must slightly alter the definition of the *capacity* of a cut. The difference in our definition of the capacity of cut lies in the fact that we will look at the *net* capacity of a cut (S, T), since we can sometimes send additional flow from a vertex in S to a vertex in T, but only at the cost of sending some other flow back from another vertex in T to some vertex in S. In our example, we can send 15 units of flow from $s_1 \in S$ to $t_1 \in T$, by sending 5 units of flow back from $t_2 \in T$ to $s_2 \in S$; therefore this cut (S, T) has a net capacity of at least 10.

Definition 4 *The net capacity of a cut (S, T) is the maximum net amount of flow that can feasibly be sent from S to T.*

It can easily be seen that, if there exists a cut in the residual network with net capacity 0, then our flow is optimal, similar to the original max-flow min-cut theorem.

In the remainder of this section, we will allow ourselves to denote the subtask corresponding to a vertex j, as simply j. Furthermore, we will denote the maximum number of mechanics who can work simultaneously at a subtask j by m_j. We will refer to the vertices that ensure the *met*-requirement, that are in the middle column of Figure 10.2, by *met*-vertices.

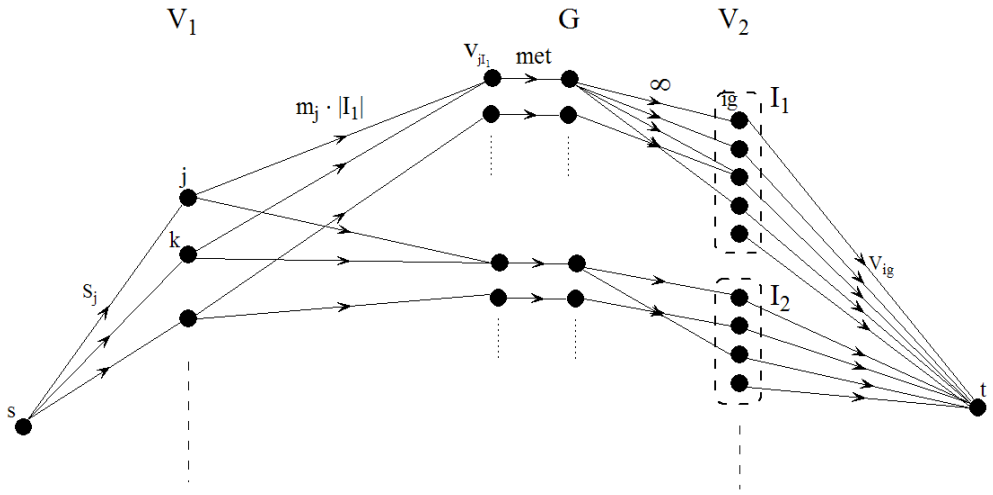

Figure 10.3: The *met*-vertices are doubled. Vertices j and k belong to subtasks of the same repair, and thus are connected to the same *met*-vertices.

Finally, for simplicity of our proof, we will assume without loss of generality, that all *met*-vertices are doubled, such that we have a single arc that indicates how much flow is going through a *met*-vertex. We will call the newly generated exit vertex a *gateway*-vertex, and the new arc a *gateway*-arc. Our new graph is shown in Figure 10.3.

Note that for each interval, the *met*-vertices of subtasks of the same repair will be contracted, so we only have maximally one *met*-vertex per combination of a repair and a interval. This is feasible since all subtasks of the same repair require the same qualification. We will make use of this property in the remainder of this section.

In our algorithm we will search for an augmenting c-path in our residual graph, and if we do not find one, we will prove that the flow is optimal. We apply the strategy of finding a single augmenting path as long as possible, but we will avoid violating the *met*-requirement. If we cannot send flow from a reachable vertex v_1 to another vertex v_2, due only to the *met*-requirement, we will give v_2 the state *special*. We will continue finding reachable vertices, until we cannot find any new reachable vertices in the standard way. If we have no *special* vertices at this point, we can straightforwardly apply the original max-flow min-cut theorem, and we have the optimal flow. So we assume that there is a vertex that has the state *special*. By construction, we know that this is a *met*-vertex. Let this vertex be v_{ri}, corresponding to repair r and interval i. As v_{ri} is *special* we know that a subtask vertex $j \in J_r$ is reachable, we denote the arc from j to v_{ri} by α. Furthermore, we know that we cannot augment over α, due only to the *met*-requirement. Therefore, we know that there exists an arc, which we denote by β, from a

subtask vertex $k \in J_r$ to v_{ri}, which has flow running over it, and this flow prevents us to directly augment over α. We will try to find a *special* vertex v_{ri}, for which there exists such a j and k with $m_k > m_j$. If we find such a v_{ri}, we will show by Lemma 1 that we can reach the gateway-vertex behind v_{ri} with the help of a cycle, and we will continue finding new reachable vertices. If there does not exist such a j and k for any *special* vertex, then we can apply our own version of the max-flow min-cut theorem to prove that we have the optimal flow, as will be proven in Lemma 2.

Thus, we will try to find an augmenting c-path in the following way:

1. Find all reachable vertices, but avoid violating *met*-requirements. If we are halted by a *met*-requirement, we will give the corresponding *met*-vertex the state *special*.

2. If t is reachable, then we have an augmenting c-path.

3. If we have a *special* vertex v_{ri}, for which a vertex $j \in J_r$ is reachable, and we have another vertex $k \in J_r$, with $m_j > m_k$, and we can decrease the flow from k to v_{ri}, then the gateway-vertex behind v_{ri} is reachable with the help of a cycle by Lemma 1, and we go back to Step 1.

4. If there does not exist such a *special* vertex, then our flow is optimal by Lemma 2.

Lemma 1 *If we have a* special *vertex v_{ri} for which there exist a vertex $j \in J_r$ that is reachable, and we have another vertex $k \in J_r$, with $m_j > m_k$, and we can decrease the flow over the arc from k to v_{ri}, then the gateway-vertex behind v_{ri} is reachable with the help of a cycle.*

Proof:
We know that j is reachable, so there can be only two cases:

1. j is reachable by s

2. j is reachable by some $v_{ri'}$

Case 1: We will augment flow as shown in Figure 10.4. As $m_j > m_k$, we have that $1 - m_k/m_j > 0$, and this flow is augmenting and reaches the *gateway*-vertex. Furthermore, it is feasible for the *met*-requirement $x_1/m_1 + \ldots + x_n/m_n \leq i$. At the left hand side, x_j/m_j increases by x/m_j, while x_k/m_k decreases by $\frac{x \cdot m_k/m_j}{m_k}$, which is x/m_j as well. Therefore, this *met*-requirement is still satisfied.

Case 2: We will augment flow as shown in Figure 10.5. Note that $v_{ri'}$ was reachable, so we can send flow to it coming from s. Again, as $m_j > m_k$, this flow is indeed augmenting and reaches the gateway. Furthermore, it is feasible, by the same arguments as we used for Case 1.

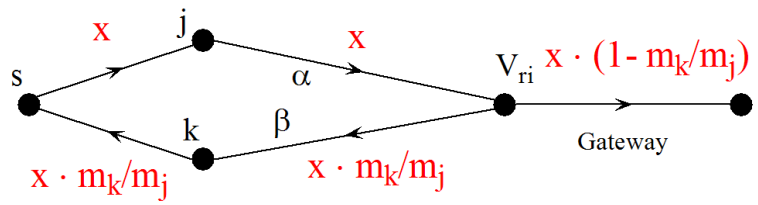

Figure 10.4: The augmenting c-path for Case 1 in the proof of Lemma 1. In red are the amounts of flow we will send over the arcs.

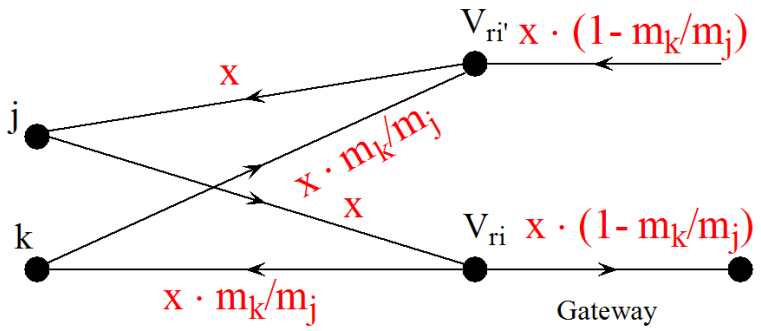

Figure 10.5: The augmenting c-path for Case 2 in the proof of Lemma 1. In red are the amounts of flow we will send over the arcs.

■

Lemma 2 *If we cannot reach new vertices, and we have no* special *vertex v_{ri}, for which there exist a vertex $j \in J_r$ that is reachable and another vertex $k \in J_r$ with $m_j > m_k$ and we can decrease the flow over the arc from k to v_{ri}, then there is a cut with net capacity 0 in our residual network, and our flow is optimal.*

Proof:
We will look at all *special* vertices. For a *special* vertex v_{ri}, we denote the subtask vertices from r that are already reachable by $J \subset J_r$. By assumption of this lemma, we cannot directly reach the gateway behind v_{ri} from these vertices in J, but we might be able to reach some new subtask vertices $K \subseteq J_r - J$ by some special vertex; we will call the vertices in K *special2* vertices. Then, from these *special2* vertices, we might reach some of the gateways behind other *special* vertices. Note that these gateways are the only new vertices that we can hope to reach from the vertices in K, since the vertices in K belong to the same repair as the vertices in J, and thus are connected to the same *met*-vertices. We will however show that the cut, in which the set S is formed by the vertices which are already reachable, including the *special* and the *special2* vertices, has a net capacity of 0.

Clearly, we can only hope to send flow from S to T, by the gateways behind the *special* vertices. Also, we can only hope to do so by a *special2* vertex. Therefore, we only have to prove that sending flow to a gateway by a *special2* vertex can never have a positive effect on the *net* amount of flow from S to T.

We can only reach an arbitrary *special2* vertex k by a vertex j and a special vertex v_{ri}, for which $k, j \in J_r$ and $m_j \leq m_k$. To send x units of flow from to v_{ri} to k, we can only send $x \cdot m_j/m_k$ units of flow from j to v_{ri}. As $m_j \leq m_k$, we minimally need to import $(1 - m_j/m_k) \cdot x$ units of flow from the gateway behind v_{ri}, which is flow from T to S. Now we will show that we can not hope to send enough of these x units of flow from k to T, to get a positive net capacity for this cut.

Sending flow from k to a gateway behind an arbitrary $v_{ri'}$ is halted by a *met*-requirement. Therefore, in order to send flow from k to the gateway, we must send some flow back from $v_{ri'}$ to some $l \in J_r$ with $m_l < m_k$. If we would send x units of flow over the arc from k to $v_{ri'}$, we would need to send at least $x \cdot m_l/m_k$ back over the arc from l to $v_{ri'}$. Thus we could then maximally send $(1 - m_l/m_k) \cdot x$ units of flow from k to T.

In Figure 10.6 this structure is illustrated. If we now look at the net amount of flow from S to T, we send $(1 - m_l/m_k) \cdot x$ units of flow from S to T, and we send $(1 - m_j/m_k) \cdot x$ units of flow from T to S. By assumption $m_l \geq m_j$, so we have that $(1 - m_l/m_k) \leq (1 - m_j/m_k)$, which means that our net amount of capacity can never increase in this way. As this was our

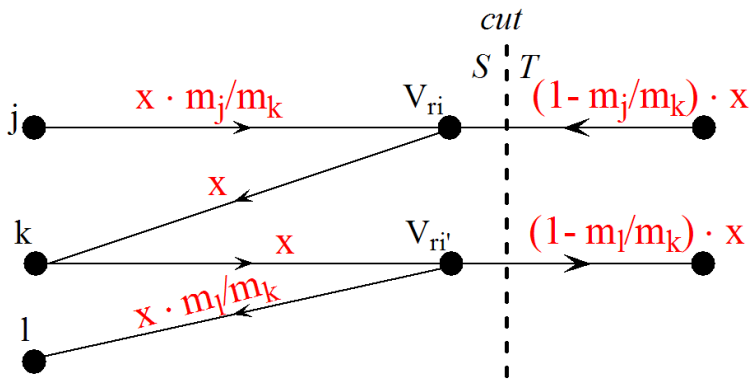

Figure 10.6: Illustration of the structure from Lemma 2. In red are the amounts of flow that can be sent.

only hope to send capacity from S to T, we can conclude that no additional flow can be send from S to T, thus the net capacity of this cut in the residual network is 0, and our flow is optimal.

Note that we ignored the fact that we can send flow to a *special2* vertex from multiple other subtask- and *met*-vertices, however, this would not alter our argument, therefore, we have neglected this possibility in our proof. ■

10.4 Adding one requirement

If a mechanic may work at a repair only when he is qualified for the types of work of all the subtasks on that repair, we can model our capacity planning problem by a flow problem, equivalent to that of the previous section. This is due to the fact that a mechanic is either qualified for all subtasks of a single repair or not qualified for any subtask of that single repair. This would allow us to use maximally one *met*-vertex for every interval and every repair. Our algorithm from the previous section uses exactly this property to find augmenting structures and to prove that the net capacity of a cut is 0 if we cannot find such an augmenting structure. Therefore, we can straightforwardly apply the solution strategy from the previous section to this problem. Note that we can not apply this strategy to our general capacity planning problem, as subtasks of the same repair may need to use separate *met*-vertices for every interval.

10.5 Planning on mechanics without preemption

The final problem that we will look at is the problem that the head of the workplace has to face. At the beginning of every day, he has to make a feasible schedule for a given set of repairs, based on the mechanics that are

present. In addition to our model requirements, we will not be allowed to use preemption, and we must satisfy the constraints about the maximum and minimum number of mechanics. Most likely, there are even more requirements in addition to these.

These new requirements change the problem in such a way, that we cannot possibly hope to apply our previous strategies in a direct manner. Also, this new problem is very likely to be hard to solve effectively, both in theory and in practice. A very simplified version of this scheduling problem, scheduling repairs of only one subtask with release dates and deadlines on identical mechanics without allowing preemption, is already NP-hard, as one can make an easy reduction from the *partitioning problem*. As the problem for the head of the workplace will be far more complex than this simplification, we are likely to have a hard time solving it effectively.

We can however use our capacity planning model to check whether there can be a schedule for a particular instance at all. Then, we could employ some sort of branch-and-cut strategy to compute a feasible schedule, which is quite common for these problems. In this branch-and-cut procedure, we cut off a branch if the solution to our capacity planning problem is already infeasible. Furthermore, we can base our branching choices on the schedule that is found implicitly by solving the LP model of this capacity planning problem; see the transformation as was described in Section 5.3.

However, it is very hard to say how well this branch-and-cut strategy will work, as we have no real idea of the quality of our cutting criterium, or the quality of these branching strategies. However, if one desires to compute a feasible schedule for the head of the workplace, it is a method that deserves attention.

Chapter 11

Future research

During this project, we have tried to solve our capacity planning problem as a flow problem. We tried to design our own method to find augmenting structures if they exist, just like the algorithm we designed for a related model in Section 10.3. A lot of effort was put into this; however, without any conclusive result. We would have liked to model our problem as a flow problem, because it would have made the use of a LP optimizer obsolete. However, since an LP solver was made readily available to us, there was no urgent need to model the problem as a flow problem. Still, one could probably argue that it would be more elegant mathematically. Also, there is a good chance it would speed up the computations. Therefore, as a future research question we still wish to look into this modeling issue.

Another possible area for future research is the more practical problem that the head of the workplace faces. In the current setting he has to create a schedule, based on the individual mechanics, to execute all repairs. We could assist the head of the workplace by giving him a proposal for this schedule, which he then can use as a basis to design his own schedule. Note that this schedule would only be a suggestion to the head of the workplace, and he himself will make the final schedule. This is necessary as he knows the specific qualities and conditions of his mechanics, he knows far more about them than we could ever hope to incorporate into a model. Still, giving him a proposal will probably make it a lot easier for him to design a feasible schedule, especially if the buffer is small. At the end of the previous chapter, we have already argued that solving this problem effectively will probably be very difficult. We also have given a suggestion as to how this problem might be tackled. Looking further into how to effectively solve this problem is a nice area for future research.

In this thesis, we have defined the requirements that our model of the capacity planning problem needs to satisfy. We could however add or ad-

just requirements in order to let our model resemble the real problem even more. This will probably make it a lot harder to effectively solve the resulting model. We think it would be a nice area for future research to see what models would resemble reality even better and investigate how one can effectively solve them.

Chapter 12

Conclusion

In this thesis we have described our research into the capacity planning problem at Rüttchen. We have discussed the process of designing a model for this problem, where we had the goals that:

1. Computations had to be completed within a few seconds
2. The model had to incorporate as many real constraints as possible

We have designed and justified a model for this problem. We can formulate our model as a linear programming problem, which we can solve very effectively. All the possible planning questions that Rüttchen wanted to compute were shown to be effectively computable by various algorithms, which all use this linear programming formulation of the capacity planning problem as their basis. As the final result for Rüttchen, we have implemented all our ideas in an extensive computer program. Currently, Rüttchen is testing the program, and so far they are very pleased with the results. It is the intention of both Rüttchen and EMA to start the deployment of the WPS system, with this program included, in all their franchises at the end of this year.

Bibliography

[1] R.K. Ahuja, J.B. Orlin, C. Stein, and R.E. Tarjan. Improved algorithms for bipartite network flow. *SIAM J. Comput.*, 23(5):906–933, October 1994.

[2] David Applegate, William Cook, Sanjeeb Dash, and Monika Mevenkamp. Qsopt linear programming solver. Website. `http://www2.isye.gatech.edu/~wcook/qsopt/`.

[3] M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali. *Linear programming and network flows*. John Wiley & Sons, Inc., second edition, 1990.

[4] B.V. Cherkassky and A.V. Goldberg. On implementing the push-relabel method for the maximum flow problem. *Algorithmica*, 19:390–410, 1997.

[5] J. Edmonds and R.M. Karp. Theoretical improvements in algorithmic efficiency for network flow problems. *ACM*, 19(2):248–264, April 1972.

[6] L.R. Ford and D.R. Fulkerson. *Flows in networks*. Princeton U. Press, 1962.

[7] L.G. Kroon. *Job scheduling and capacity planning in aircraft maintenance*. PhD thesis, Erasmus University Rotterdam, 1990.

[8] Alexander Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer-Verlag, Berlin, 2003.