
Algorithms for Scheduling of Train Maintenance



Ronald Paulus Evers

Algorithms for Scheduling of Train Maintenance

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Ronald Paulus Evers,
born in Haarlem, The Netherlands



Algorithms Group
Department of Software Technology
Faculty EEMCS
Delft University of Technology
Delft, The Netherlands
www.eemcs.tudelft.nl



Maintenance Development
NedTrain
Utrecht, The Netherlands
www.nedtrain.nl

Cover page photograph copyright © Willem J. Steen
<http://www.flickr.com/photos/willemstreinen/4628997454>

Copyright © 2010 Ronald Paulus Evers

Algorithms for Scheduling of Train Maintenance

Author: Ronald Paulus Evers
Student id: 1099752
Email: `mail@ronaldevers.nl`

Abstract

In this thesis we will develop algorithms for the scheduling of train maintenance at NedTrain facilities. We present a detailed analysis of the maintenance process at NedTrain. The problem of scheduling train maintenance is formalized and expressed using linear inequalities. We show that the Simple Temporal Problem can be used to find schedules that satisfy the temporal constraints of the scheduling problem. However, when resource constraints are added, the STP is no longer sufficient. The problem can still be expressed using the Disjunctive Temporal Problem but this cannot be efficiently solved. Two algorithms are presented: a zero/one integer linear programming approach and an STP-based approach. In the STP-based approach we start with an STP that describes the temporal constraints. The STP is then iteratively updated until its earliest start time assignment corresponds to a valid solution to the scheduling problem. Experiments show that the ILP method is fast enough to produce optimal base schedules whereas the heuristic method is well suited to updating schedules with little disruption when changes have occurred.

Thesis committee:

| | |
|--------------------------------|--------------------------|
| Chair & university supervisor: | Prof. dr. C. Witteveen |
| Committee member: | Dr. T. B. Klos |
| Committee member: | Dr. S. O. Dulman |
| Company supervisor: | Ir. B. Huisman, NedTrain |

Preface

My time at Delft University of Technology has almost come to an end. I have spent nine wonderful years here in which I have learned a lot but have also had the opportunity to get involved in student organizations and see such places as China and the USA. Since March 2010 I have been working on my thesis project at NedTrain. It has been a pleasant environment to work in and one with many challenging problems. In some ways it is sad that I have to stop now because there are many things that could still be researched but this will always be true and I feel it is time to move on.

I would like to thank the following people for their support during my thesis project.

Michel Wilson for helpful suggestions and company during lunch breaks in Utrecht, Tomas Klos for getting me started, Cees Witteveen for sharp insights and for getting me introduced at NedTrain, Bob Huisman for fruitful discussions and a pleasant working environment at NedTrain headquarters at NS Station Utrecht, right on top of one of the biggest railway junctions in The Netherlands, Leon Planken for his excellent work on Simple Temporal Problems and for giving me his DTP codebase for reference and my girlfriend Eline for helping with editing the final text and putting up with me.

Ronald Evers
Delft, November 2010

Contents

| | |
|---|-----------|
| Preface | v |
| 1 Introduction | 1 |
| 1.1 NedTrain and NS Group | 1 |
| 1.2 Workload and the Scheduling Process | 3 |
| 1.3 Research Questions | 8 |
| 1.4 Overview | 9 |
| 2 Problem and Background | 11 |
| 2.1 Model and Representation | 11 |
| 2.2 Addressing the Temporal Constraints | 17 |
| 2.3 Addressing the Resource Constraints | 22 |
| 2.4 Conclusion | 24 |
| 3 Integer Programming Approach | 27 |
| 3.1 Model Variables | 28 |
| 3.2 Finding Valid Schedules | 29 |
| 3.3 Maximizing Throughput | 32 |
| 3.4 Experiments | 33 |
| 3.5 Conclusion | 36 |
| 4 Heuristic Approach | 39 |
| 4.1 Finding Valid Schedules | 40 |
| 4.2 Creating Robust Schedules | 44 |
| 4.3 Variable Capacity Resources | 48 |
| 4.4 Contributions | 49 |
| 4.5 Conclusions | 51 |
| 5 Experiments | 53 |
| 5.1 Problem Instances | 53 |
| 5.2 Metrics | 55 |
| 5.3 Results | 56 |
| 5.4 Conclusion | 60 |
| 6 Conclusions and Future Work | 63 |
| 6.1 Contributions | 63 |
| 6.2 Conclusions | 64 |
| 6.3 Future work | 66 |
| Bibliography | 71 |
| A Complexity Analysis | 73 |

| | |
|--|-----------|
| B Leidschendam Base Schedule | 77 |
| C CPLEX | 81 |
| C.1 CPLEX Problem Definition File Format | 81 |
| C.2 Interfacing with CPLEX | 82 |
| D Code listing | 85 |
| D.1 chaining.c | 85 |
| D.2 chaining.h | 88 |
| D.3 debug.c | 88 |
| D.4 debug.h | 88 |
| D.5 esta_plus.c | 89 |
| D.6 esta_plus.h | 94 |
| D.7 floydwarshall.c | 95 |
| D.8 floydwarshall.h | 95 |
| D.9 ifpc.c | 96 |
| D.10 ifpc.h | 98 |
| D.11 lex.l | 98 |
| D.12 list.h | 99 |
| D.13 list.h | 100 |
| D.14 main.c | 101 |
| D.15 salloc.c | 104 |
| D.16 salloc.h | 105 |
| D.17 stn.c | 105 |
| D.18 stn.h | 109 |
| D.19 timing.c | 109 |
| D.20 timing.h | 111 |
| D.21 tmisp.c | 111 |
| D.22 tmisp.h | 113 |
| D.23 token.c | 115 |
| D.24 token.h | 115 |

1

Introduction

In this thesis we will investigate the train maintenance scheduling process at NedTrain. We will first introduce NedTrain, then we will review the current train maintenance scheduling process at NedTrain and formulate our research questions. Finally, we will give an overview of the structure of the rest of the thesis.

1.1 NedTrain and NS Group

NedTrain is a subsidiary company of NS Group (NS). NS is the biggest Dutch railway operator, welcoming over a million people into its trains every day. NS is a state-owned holding company that consists of three branches: the passenger transportation branch, the stations development and exploitation branch and the infrastructure branch. Figure 1.1 shows an overview of the organization.

Strukton is a big railway contractor, operating not only in The Netherlands but also in many other European countries.¹ NS Poort runs the ex-

¹Strukton has recently been acquired by Oranjewoud N.V. and is no longer part of NS Group but is still listed as such on the NS Group website as of November 2010.

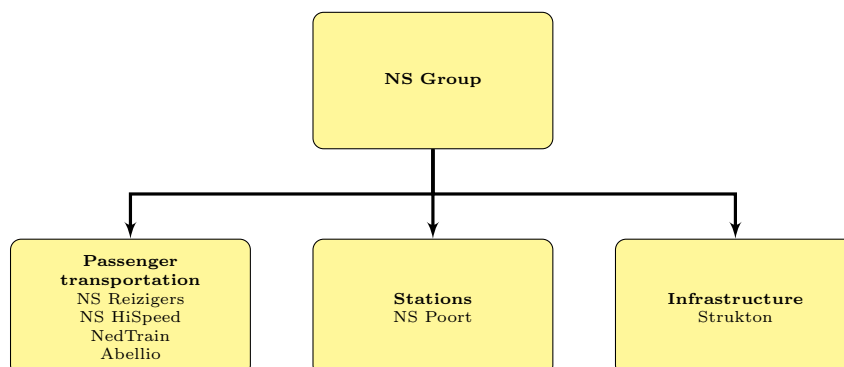


Figure 1.1: Organizational chart of NS Group.

exploitation of the train stations and NS Reizigers and NS HiSpeed run the exploitation of the trains. Abellio is a joint venture on the European market, transporting passengers in Great Britain, Germany and the Czech Republic.

Note that ProRail, which is responsible for the Dutch rail infrastructure is not part of NS Group but rather a separate state-owned company.

NedTrain is responsible for maintaining a high availability rate for NS Reizigers' and NS HiSpeed's fleet of trains consisting of roughly 3000 passenger carriages in total. To this end, all those trains undergo periodic maintenance at a NedTrain maintenance facility where NedTrain provides first-line service, technical maintenance and refurbishment.

First-line service: All carriages are cleaned at least once every day at one of thirty NedTrain service facilities throughout the country, small technical problems are solved if possible.

Technical maintenance: Once every couple of months—or after a certain mileage has been reached on a main part, see the following section for details—a train goes to one of four NedTrain depots for maintenance. Compare this to taking your car to the garage for a routine checkup.

Refurbishment: Because trains, nowadays, are expected to last several decades so once or twice in their lifetime they have to be refurbished to meet modern standards. This work is done in large overhaul projects mainly at the NedTrain facility in Haarlem.

In September 2010 there were approximately 250 carriages in NedTrain facilities for technical maintenance and in November 2010, with trees shedding their leaves and causing a lot of problems for the trains, this number even rose to 300—that is 10% of the entire fleet.

This thesis focuses on the second type of maintenance: the technical maintenance. From this point on, we will use the term “maintenance” solely in connection with this technical maintenance.

1.1.1 A Look Inside a NedTrain Depot

NedTrain has four depots for technical maintenance in The Netherlands: at Amsterdam, Leidschendam, Onnen and Maastricht. The Amsterdam location services mostly NS HiSpeed trains, the Leidschendam and Onnen location service VIRM (see Figure 1.2(a)) trains and Maastricht mostly services ICR carriages with 1600 and 1700 locomotives and the so-called Mat64 trains (see Figure 1.2(b)).

Figure 1.3 shows satellite imagery of the Leidschendam depot. In short, a depot consists of a shunting yard and a big hangar. The Leidschendam



(a) VIRM train series



(b) Mat64 train series

Figure 1.2: Train series

photos show the main hangar as well as the shunting yard to its north-east. Depots are operational around the clock; engineers work three eight-hour shifts starting at 7 am. The first shift is the biggest, the night shift the smallest. Each depot houses several lengths of specialized maintenance tracks laid out in parallel. The specialized types of track are (track types in Dutch):

- **putspoor:** elevated regular rail—or lowered floor, depending on your point of view—used for most maintenance, allows mechanics to work underneath the train, a maintenance facility typically has about four of these tracks—see Figure 1.4 for an example,
- **kuilwielbank:** separate section with machinery for re-profiling train “tires”, that is to say, the outer metal layer of the wheel,
- **aardwind:** specialized section with elevator and crane for replacing bogies (Dutch: *draaistellen*) and
- **ATB:** specialized rail that allows for testing of train safety systems (ATB is a Dutch abbreviation for “Automatische Trein Beïnvloeding”).

Due to safety considerations, only one train can move between the shunting yard and the hangar at any time. Such a movement can take up to half an hour because the driver has to walk to the train and the switching of power on the overhead lines has to happen in a safe and controlled manner.

1.2 Workload and the Scheduling Process

We will now describe how it is determined what work needs to be done and how corresponding schedules for train maintenance are created. NedTrain schedulers work with Excel and some planning tools like ProPlan to make

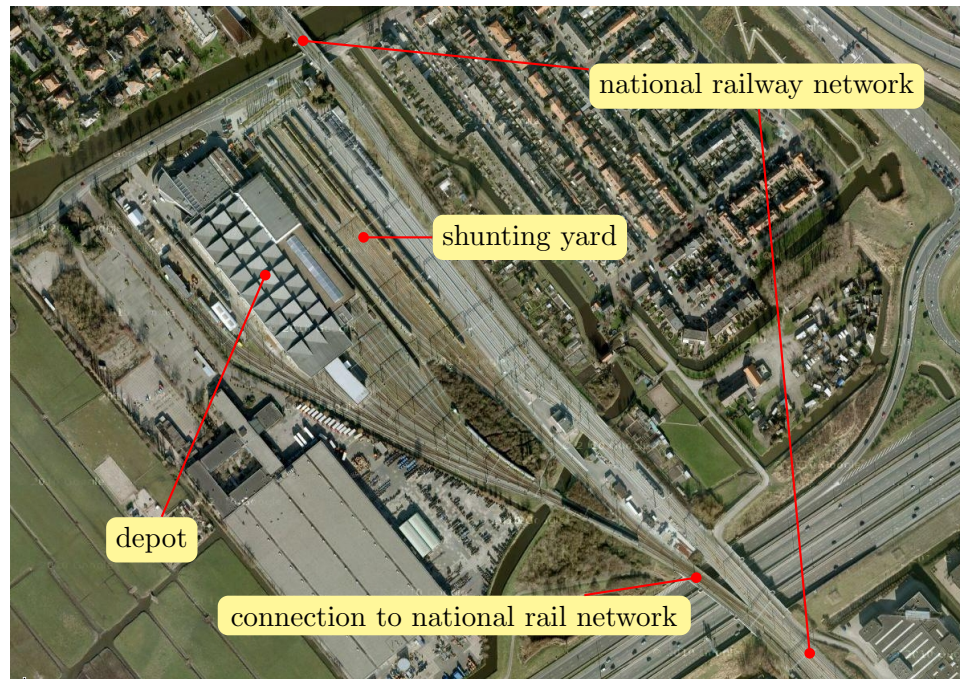


Figure 1.3: NedTrain maintenance facility in Leidschendam, image © 2010 Google Imagery and Aerodata International Surveys.



Figure 1.4: Putspoor with new Fyra train at NedTrain Watergraafsmeer.

schedules for one week at a time. The schedules start out very coarse and are refined as time goes by and more information becomes available. In this section we will follow the process from the high-level base schedule to a detailed production schedule and to dealing with changes and disruptions.

1.2.1 Base Schedule

NS Reizigers and HiSpeed operate their trains on a one-week-long timetable that repeats throughout the year—barring seasonal changes and minor irregularities caused for instance by national holidays. NS Reizigers and HiSpeed together with NedTrain have organized the timetables so that NedTrain handles—in principle—the same workload of trains every week. This enables NedTrain to make a one-week-long *base schedule* that, like the timetables, also repeats throughout the year. Such a schedule can be used to coordinate personnel requirements and inventory management.

The *base schedule* can be made far in advance and only has to be made roughly twice per year—once for wintertime and once for summertime because that is when the timetables change. A base schedule is constructed knowing only:

- when trains will arrive during the week,
- when they have to be ready,
- what type or series they are and
- which routine checkup they need.

So for example, ‘every week on Tuesday at 3 pm a train from the 9400-series will arrive for its type-3 checkup and it needs to be ready Friday at 7 am’. An example of such a weekly *base schedule* from the Leidschendam depot is shown in Appendix B. The important thing to note here is that at this level it is not known which specific train will come in for maintenance, only what series it is from, e.g. the 9400-series.

For every train series there are lists of standard maintenance tasks that must be performed on trains of the series when they come in for maintenance. Different lists are used in subsequent visits to the maintenance depot. So the first time ‘beurt 1’ is performed, then ‘beurt 2’ and so on. For the *base schedule*, however, the schedulers at NedTrain do not work on this level of detail but instead only schedule some big blocks. These blocks are basically placeholders that will be updated as more information becomes available. Judging from the Leidschendam base schedule, these blocks are usually:

- initial inspection,
- standard maintenance,

- repairs,
- ‘E-staat’ work (revisions that are carried out during normal maintenance),
- kuilwielbank (reprofiling tires),
- aardwind (replacing bogies),
- leftover work and
- cleaning.

The standard maintenance tasks are only one item in this longer list. Typically about 20% of the time is spent on inspections, 10% on planned maintenance, 20% on the planned exchange of main parts (see next section for details) and 50% on unplanned work based on the inspections.

In the next sections we will discuss how all these blocks are filled in. The blocks are usually processed in a predetermined order, that is: work usually starts with the initial inspection, immediately followed by the standard maintenance, repairs and ‘E-staat’ work. So activities can depend on other activities being finished before they can start. This can be true for the blocks in the base schedule but also on a more detailed level in the production schedules.

1.2.2 Detailed Production Schedule

The *base schedule* is used as a template to create a more detailed production schedule that can actually be used by a foreman and his workers. About two weeks before the beginning of a new week, it will become known which specific trains are coming in for maintenance. The base schedule can now be refined; for every train coming in for maintenance, the specific maintenance needed by that train can be scheduled. This includes the following.

- What standard maintenance is required: ‘beurt 1’, ‘beurt 2’ etc.
- Replacement of *main parts*: several *main parts* are defined for every train series, like the engine, the bogies etc. Each of these parts is replaced when its own life expectancy is reached. This can be a time limit or a specific mileage. By keeping track of and individually replacing every main part on every train, the main parts are used to their fullest potential without compromising fleet availability.
- So-called ‘E-staat’ work: refurbishments. Some of the refurbishments are done during the technical maintenance, just like some of the technical maintenance is performed during first-line service.

- Known malfunctions: for example, doors that will not open anymore, broken speakers etc.
- Vandalism: torn or broken interior, graffiti on the inside and outside etc.

These items are then scheduled in place of the big blocks in the *base schedule* from the previous section. At this point the schedule can be given to a foreman for execution and if no disruptions occur then this schedule would be final. In practice, unfortunately, many changes and disruptions can and often do occur.

1.2.3 Handling Changes and Disruptions

In the last two weeks before a given week as well as in the week itself during the execution of the work, several changes may occur and necessitate updating the production schedule. These include the following.

Different trains: NS Reizigers or HiSpeed sometimes decide they want to send a different train off for maintenance.² In such a case, all the train-specific maintenance must be rescheduled. NedTrain is normally informed when trains are swapped so that the schedule can be updated in time. But sometimes NedTrain is not informed in time and a different train than the one that was expected by NedTrain arrives causing a last-minute change of plans.

Extra trains: Sometimes trains run into problems while in service, necessitating an unplanned trip to a maintenance depot. In some cases trains even have to be taken out of service immediately and require maintenance before they can continue normal passenger service.

During the execution of the maintenance in the week itself, the following changes or disruptions may additionally occur and must be dealt with.

Extra activities: The initial inspection that each train is subjected to before work starts may bring additional issues to light that have to be addressed, for example unreported vandalism or unnoticed defects. Furthermore, engineers may find additional problems during their work on a train.

Delayed arrival: Although the timetables for the passenger services are very strict, sometimes delays occur. NedTrain is not immune to these delays and it happens that trains arrive later than planned (or earlier, or sometimes even not at all but these two are less of a problem than trains arriving later than planned).

²This can be caused for example by irregularities during the train service that causes trains to be in other locations than initially planned.

Resources unavailable: Delays during the execution of the work may be caused by personnel not being available due to sickness or vacations, by required parts not (yet) being available, by machines breaking down or being out of order and external factors such as power outages etc.

Leftover work: Due to delays adding up, work from one week may not be finished in time and may carry over into the following week. Note that it is also possible that trains are released when all work could not be finished in time. The work is then postponed until the next maintenance window for the train.

If any of the changes or disruptions listed above occur, the production schedule should be reevaluated to see if it is still valid and updated if necessary to accommodate the changes or extra work. Currently, NedTrain schedulers add a block labeled ‘restwerk’ to every train in the base schedule. This is basically a buffer of reserved time which is used to absorb potential extra work or delays during the maintenance process. When this buffer is not sufficient or when an extra train is added, a more thorough rescheduling must take place. Of course, simply increasing the buffer size lowers efficiency.

1.3 Research Questions

The previous section shows that there is much uncertainty in determining the exact workload and also in the availability of personnel and company assets like machines. Changes and disruptions in any of these factors can occur at any point in time. A big part of the workload is unknown until the initial inspection: NedTrain uses a philosophy of condition-based maintenance but the condition of the trains is partially unknown until after they are inspected at a NedTrain facility. At present, NedTrain depots keep high inventory levels of spare parts in order to handle this uncertainty.

This work is executed in the context of the “Applied Research & Development Program ‘Rolling Stock Life Cycle Logistics’ at NS/NedTrain”. This program aims to support NedTrain’s ambition to be a first-class European rolling stock maintenance company and is executed in cooperation with several universities. Together with Delft University of Technology, NedTrain is working on improving operational planning aspects. Two goals of the program are to gain more certainty about the condition of trains by processing the information from the thousands of sensors on modern trains and to develop and implement state-of-the-art operational planning and scheduling tools. Both of these efforts should lead to more predictable load levels and allow lower inventory levels to be kept, ultimately leading to a higher fleet availability at lower cost.

This thesis will focus on developing scheduling tools for use at NedTrain depots to create base and production schedules. Given the analysis presented in this chapter, we have come up with the following research questions:

- *How can valid one-week-long base schedules be generated and can optimal schedules be generated?*

Base schedules are generated based on the number of trains that is expected to arrive, the times at which they are expected to be delivered and picked up and their types. Optimality will be measured in terms of throughput.

- *How can a base schedule be transformed into a production schedule?*

Roughly two weeks before the start of a week, the identities of the trains coming in for maintenance become known. At this point a concrete production schedule should be constructed. This schedule will have more detail than the base schedule and we will have to investigate whether or not our methods can scale from the base schedule to the production schedule.

- *How can we update an existing production schedule with minimal change when faced with disruptions?*

We would like to be able to ask the system for an updated schedule given the new situation after a disruption has occurred. The system needs to be fast enough for it to be able to be used in this manner and it should also not generate completely different schedules because of small disruptions because this in itself is disruptive and may lead to chaos on the work floor and disgruntled engineers. We have set a time limit at 5 minutes for the updating of production schedules. This seems a reasonable term to us. If updating the schedule takes any longer, than the schedulers would only be waiting for the algorithm all the time.

These questions cover the entire scheduling process at NedTrain, from the very beginning at the base schedules, to dealing with changes in the production schedules. Creating and maintaining these schedules is a full-time job for NedTrain schedulers and we believe that there is much efficiency to be gained by using computerized scheduling algorithms to assist in their tasks. We will investigate such algorithms in this work in order to answer these questions.

1.4 Overview

In this introduction we have given a high-level view of NS Group and NedTrain and presented an analysis of the train maintenance scheduling process

at NedTrain, leading to the research question.

In the next chapter we will investigate the difficulty of the scheduling problem. We will present a model of the scheduling process at NedTrain, show the assumptions we make and introduce notation and a mathematical formulation of the problem as we see it. Then we will review solution methods from the literature. We find that we can use a so-called Simple Temporal Network to model the temporal restrictions but will have more difficulty in handling the resource constraints. We also present a complexity analysis of the scheduling problem in Appendix A.

In Chapters 3 and 4 we present a zero/one integer linear programming based approach and a heuristic method of finding schedules for train maintenance. The integer linear programming method is aimed at finding optimal schedules in terms of throughput. The heuristic method is aimed at finding valid schedules fast and repairing broken schedules. Ideally, we would like to compare our algorithms and the schedules they generate to real-life situations at NedTrain. Unfortunately, no data is available to perform this comparison. However, if the linear programming approach proves successful, it will give us optimal solutions and we can use these to determine how well the heuristic approach approximates the optimum. These and other experiments, aimed at answering the research questions can be found in Chapter 5.

Finally, we will summarize our contributions and present our conclusions and thoughts for future work in Chapter 6.

2

Problem and Background

In this chapter we dissect the scheduling problems faced by NedTrain schedulers. We will present a model of the scheduling process detailing how we conceptualize the process, specifying which parts we leave in and what we leave out. Next we will present a mathematical formulation based on the model. In Appendix A we show that the computational complexity of the specific scheduling problems we face is NP-complete.¹ Finally, we discuss some related literature, searching for prior work that we can reuse or extend upon.

2.1 Model and Representation

Like NedTrain schedulers, we focus on developing methods for creating base and production schedules for one week at a time and for one depot. Although the maintenance process at NedTrain is continuous in nature, it does not make sense to create production schedules more than one or two weeks into the future. There is not enough information available about the upcoming workload and even if it were available, many changes would likely occur as we have seen in the previous chapter.

2.1.1 Model

We make the following simplifying assumptions with regard to the maintenance process.

- NedTrain has no direct influence on the dates and times at which trains arrive at and depart from the maintenance facilities. These data are instead controlled mostly by the clients. Of course, if NedTrain can realize a structural improvement in throughput, the clients can start to pick up their trains earlier, increasing fleet availability.
- An estimate of the duration of an activity is known before the work starts. This is not always true in practice, but we need a duration to

¹More precisely, the decision variants are NP-complete. See Appendix A for more information on this subject.

use while scheduling. However, because we will develop methods of dealing with change, this assumption is not restrictive. If an activity takes longer than expected, we will use our methods to update the production schedule for the remaining work.

- We will not concern ourselves with inventory management of spare parts but instead assume that all parts are always available. In reality, trains are often waiting for engineers or parts to become available—see the NedTrain report “K3” [K307] for details—but dealing with this is outside the scope of this work.
- The amount of resources required by an activity, such as engineers or a timeslot on a specialized track, is constant throughout the entire duration of an activity. If an activity, for example, no longer requires 4 but can do with 3 engineers after the first 15 minutes, the activity should be split up into two activities with different resource requirements.
- We will not concern ourselves with the routing of trains on the shunting yard and assume that the shunting yard has an infinite capacity. Note that this is a challenge in reality: a train may be parked behind another one which would require first moving the other train before the blocked one can be moved. These actions can be time-consuming because the train drivers often have to cover long distances by foot.²
- Only one activity can be executed on any train at any time. This is a restrictive assumption but our methods require it. See 6.3.4 for more information.

The smallest unit of work we will schedule is a maintenance *activity*. An activity represents an amount of maintenance work that is to be performed on a train. Activities are performed by a given number of engineers and on one of the specialized types of track as reviewed in the previous chapter. The number of engineers and the specialized track required are constant during the entire duration of an activity.

2.1.2 Notation

To be able to precisely specify our problems and their solutions we need to conceptualize the model and introduce some notation.

Let

$$T = \{t_1, \dots, t_n\}$$

²Trains may not be driven in reverse due to safety regulations. Therefore, drivers routinely have to walk to the other end of a train if they need to turn it around and trains can be as long as 240 meters.

denote the set of all trains. Each train $t_i \in T$ has a set A_i of N_i activities associated with it and A is the set of all activities:

$$A_i = \{a_{i1}, \dots, a_{iN_i}\} \text{ and } A = \cup_i A_i.$$

Each activity can only be started when the train it corresponds to is physically present at the depot and each activity must be finished before NS picks up the corresponding train. To model these restrictions, we associate with each train t_i a *release date* and a *due date*. These terms are commonly used in scheduling literature and correspond in our case to the arrival and departure dates of the trains, respectively. We will use the terms interchangeably throughout this work. The release and due dates of train $t_i \in T$ are denoted respectively as

$$rd_i \in \mathbb{N} \text{ and } dd_i \in \mathbb{N}.$$

We have assumed that we know at least an expected duration for every activity at the time of scheduling and we write the duration of an activity a_{ij} as

$$d_{ij} \in \mathbb{N}.$$

In our model, activities require a constant number of engineers and they must be on a specific type of specialized rail throughout their entire duration. We call these constraints *resource requirements*. Resources are defined for different types of engineers and for the specialized rail sections in the depot. Their capacities are set equal to the total number of engineers or platforms in the depot. Obviously, the number of platforms is constant, but the number of engineers varies with time because the engineers work three eight-hour shifts around the clock. Some notation to go along with these notions will come in handy. Let

$$R = \{r_1, \dots, r_k\}$$

be the set of resources. Every resource $r_i \in R$ has a capacity

$$c_{it} \in \mathbb{N}$$

in time period t . The amount of the capacity of a resource r_i required for (and throughout) the execution of activity a_{ij} is written as

$$q_{ijk} \in \mathbb{N}.$$

Finally, activities may depend on other activities begin finished. The set P will contain all the precedence constraints:

$$P = \{p_1, \dots, p_l\}.$$

A precedence constraint $p_i = \langle a_{ij}, a_{vw} \rangle \in P$ is a tuple that signifies that activity a_{ij} must precede activity a_{vw} .

Now we have introduced all the notation we need for defining a model of the temporal and resource constraints that base and production schedules have to conform to.

2.1.3 Representation

Given the model we have described earlier in this section and our process description in the first chapter, we realize that the core of the problem faced by NedTrain schedulers is to find schedules for train maintenance in which none of the temporal constraints or resource constraints are violated, while optionally optimizing throughput as specified in the resource questions.

In the rest of this section we will formulate a model of the scheduling problems at NedTrain. We will focus on the generation of production schedules. From a scheduling-algorithmic standpoint the creation of base schedules and production schedules is quite similar. The difference is that in the base schedule there is less detail. But the model is the same.

Temporal constraints are constraints like “a train will arrive at 10:0am”, “activity a must be processed before activity b ” or “activity a takes 45 minutes”. Resource constraints are constraints like “activity a requires 3 mechanical engineers to complete” or “activity a must be performed on a *putspoor*”. We will now show that all the constraints faced at NedTrain can be modeled using linear inequalities. A linear inequality is an inequality with the left-hand side a linear combination of variables and the right-hand side a constant. The general form of a linear inequality is one of

$$\begin{aligned} a_1x_1 + \dots + a_nx_n &\leq c \\ \text{or} \\ a_1x_1 + \dots + a_nx_n &< c \end{aligned} \tag{2.1}$$

where each $a_i \in \mathbb{R}$ is a coefficient, each x_i a variable and $c \in \mathbb{R}$ a constant. Note that we will also use equalities and that an equality $a = b$ can be written as a conjunction of two inequalities $a \leq b \wedge b \leq a$.

Temporal Constraints

We will start with the temporal constraints because they are easiest to model. There are three things to constrain: the activity execution intervals and durations, the release and due dates and activity precedence relations.

Activities can only be executed when their corresponding train is at the depot. We have already introduced variables for train arrival and departure

so that the interval in which train t_i is at the depot is given by $[rd_i, dd_i]$. Now, let s_{ij} and e_{ij} denote the start and end of activity a_{ij} . Then for every train $t_i \in T$ and for every activity $a_{ij} \in A_i$ of that train t_i we add the following three equations to the system.

$$\begin{aligned} rd_i &\leq s_{ij} \\ s_{ij} + d_{ij} &= e_{ij} \\ e_{ij} &\leq dd_i \end{aligned} \tag{2.2}$$

Next, we must constrain the train release dates and due dates to take the correct values. We treat these values as constants so we add two equalities for every train $t_i \in T$ like shown in (2.3). Of course one should replace 0 and 100 with the actual release and due dates of every train t_i .

$$\begin{aligned} rd_i &= 0 \\ dd_i &= 100 \end{aligned} \tag{2.3}$$

Finally, activity precedence relations must be modeled. For every precedence relation $p = \langle a_{ij}, a_{vw} \rangle \in P$, which specifies that activity a_{ij} must be finished before activity a_{vw} can be started, add inequality (2.4).

$$e_{ij} \leq s_{vw} \tag{2.4}$$

Using the inequalities and equalities from equations (2.2), (2.3) and (2.4), it is possible to represent the temporal constraints of the NedTrain scheduling process. Solving the system of linear equations for a number of trains and activities, should produce a temporally valid schedule. Before moving on to the resource constraints, we will show how to describe temporal optimality as a function of the variables we have used so far.

A goal of optimal throughput, which can be described in layman's terms as finishing work on every train as soon as possible, translates to minimizing the following function (recall that n denotes the number of trains and N_i the number of activities for train t_i).

$$\sum_{i=1}^n \max_{j=1}^{N_i} e_{ij} \tag{2.5}$$

This is the sum of the completion times of the last activity to be executed for every train.

Resource Constraints

Using linear inequalities, as we did above for temporal constraints, we will attempt to model resource constraints. One might wonder whether or not

this is actually possible using such simple constructs as linear inequalities but when you think about it, all a resource constraint does is limit the number of activities that can simultaneously use a resource. So resources, by having limited capacity, force some ordering on activities that wish to use it. We will now give some examples of this idea before formalizing it.

Suppose there is a resource r_i , let us say it is a ‘kuilwielbank’, with constant capacity 1, denoted $c_{it} = 1$ for every time period t . Suppose also that there are two activities a_{ij} and a_{vw} that require its use. Then either activity a_{ij} must be performed before activity a_{vw} or vice versa. Such precedence constraints are denoted $a_{ij} \rightarrow a_{vw}$ or $a_{vw} \rightarrow a_{ij}$, with a right arrow, ‘ \rightarrow ’, denoting precedence.

As another example, suppose the ‘kuilwielbank’ from the previous example can now simultaneously handle two trains and another activity a_{yz} also requires its use in addition to activities a_{ij} and a_{vw} . Now there are six instead of just two options., they are:

$$\begin{aligned}
 a_{ij} &\rightarrow a_{op} \\
 a_{ij} &\rightarrow a_{vw} \\
 a_{op} &\rightarrow a_{ij} \\
 a_{op} &\rightarrow a_{vw} \\
 a_{vw} &\rightarrow a_{op} \\
 a_{vw} &\rightarrow a_{ij}
 \end{aligned} \tag{2.6}$$

In general, to synchronize the use of a resource r_i with capacity c_{it} by m activities, each requiring 1 unit of the resource’s capacity, it is necessary to post $\max(0, m - c_{it})$ constraints which can be done in at least $m!/c_{it}!$ distinct ways and each of these distinct ways describes a potential partial solution of the scheduling problem. If we want to capture all these solutions in our model, we have to add a constraint that is a big disjunction of all possible solutions. In the case of the six-options example given above, this would mean adding the disjunctive constraint give below. But for bigger problems with higher capacity resources and more activities, the constraint quickly grows in size.

$$a_{ij} \rightarrow a_{op} \vee a_{ij} \rightarrow a_{vw} \vee a_{op} \rightarrow a_{ij} \vee a_{op} \rightarrow a_{vw} \vee a_{vw} \rightarrow a_{op} \vee a_{vw} \rightarrow a_{ij}.$$

This concludes this section. We have shown that the scheduling problems faced at NedTrain can be expressed using only linear inequalities and a linear goal function to be optimized. However, a disjunction of exponentially many linear inequalities is required in order to model resource constraints.

In the following few sections we move on to exploring existing approaches in the literature for dealing with the types of constraints we introduced in

this section. We will discuss linear programming, the Simple Temporal Problem and the Disjunctive Temporal Problem. We will find that the temporal constraints pose no challenge; they can be solved in polynomial time. It is only when they are combined with resource constraints that the problem becomes hard as we shall see in our discussion of the Disjunctive Temporal Problem.

2.2 Addressing the Temporal Constraints

In this section we will review techniques for processing the temporal constraints that we have described in the representation in the previous section. We look at linear programming and at the Simple Temporal Problem.

2.2.1 Linear Programming

Linear programming is a group name for a number of optimization techniques that can be used to solve a linear program. In a linear program, the goal is to optimize a linear function while not violating a number of linear constraints. Linear programs can be efficiently solved, i.e. the amount of calculation needed to solve a linear program increases polynomially if the problem size increases.

The temporal constraints we described in Section 2.1.3 combined with the goal function in equation (2.5) are all linear and indeed describe a linear program. This means that, as far as temporal constraints go, we can always calculate optimal solutions. The downside of using a linear programming method, however, is that a small change in the input could lead to a large change in the output: it is not stable. To address this issue, we will look at another method, called the Simple Temporal Problem.

2.2.2 Simple Temporal Problem

We will now present the Simple Temporal Problem (STP). This is a formalism designed for dealing with the types of temporal constraints that we have seen earlier in this chapter. Because we will use the STP in the basis of our heuristic approach later in this thesis, we will give a quite detailed explanation of what it is and how it works. If the reader is already familiar with STPs, they can safely skip this section and continue with Section 2.2.3, in which we will show how we can apply the Simple Temporal Problem to the temporal constraints we have described earlier.

In this section we will first describe what a Simple Temporal Problem is, then we will show a convenient graph representation called the Simple Temporal Network, and finally we will show some solution methods.

The Simple Temporal Problem was introduced in 1989 by Dechter, Meiri and Pearl in [DMP89]. An STP $S = \langle X, C \rangle$ consists of a set of timepoints $X = \{x_1, \dots, x_n\}$ and a set of constraints C . A timepoint represents an event in the real world, like the arrival of a train at a depot, and takes its value from $\mathbb{Z}^* = \mathbb{N} \cup \{0\}$. A constraint $c_{i \rightarrow j} \in C$ has a weight $w_{i \rightarrow j} \in \mathbb{Z}$. The weight $w_{i \rightarrow j} \in \mathbb{Z}$ is an upper bound on the time difference between x_j and x_i . To be more precise, a constraint $c_{i \rightarrow j}$ imposes the inequality $x_j - x_i \leq w_{i \rightarrow j}$. Only one constraint may exist between every two timepoints and direction matters, so there can be only one constraint $c_{i \rightarrow j}$ and one constraint $c_{j \rightarrow i}$ for every $i \neq j$. If a constraint is not specified it is taken to have infinite weight.

A *solution* to an STP is a schedule that assigns a time to each timepoint $x_i \in X$ in such a way that no constraints are violated and an STP is said to be *consistent* if at least one solution exists.

At this point we can model constraints like “process activity a before activity b ” by adding a timepoint that marks the end of activity a , call it x_a , and a timepoint that marks the beginning of activity b , call it x_b , and introducing a constraint $c_{b \rightarrow a}$ with weight $w_{b \rightarrow a} = 0$. Indeed, this constraint forces the inequality $x_a - x_b \leq 0$ or $x_a \leq x_b$ so activity a is finished before activity b is started. We have, however, no means of referring to absolute points in time like 7 pm tomorrow. For this purpose, a special timepoint is often introduced, called the *temporal reference point* and labeled z . This timepoint represents a fixed, predetermined point in time and enables us to make references to absolute points in time by adding constraints from and to it.

Using what we have learnt so far, we can construct an STP instance that encodes the temporal constraints of a train maintenance scheduling problem. We simply add timepoints that signify the start and end of activities and the arrival and departure of trains and the necessary constraints as described in the model in the previous section. We will discuss this process in detail in Section 2.2.3.

Simple Temporal Network

The Simple Temporal Network (STN) is a graph representation of the STP where the timepoints serve as vertices and the constraints are represented by directed, weighted edges. An example is shown in Figure 2.1(a). A graph representation is convenient because it is easily depicted graphically and because it allows the use of many existing graph algorithms such as shortest-path algorithms.

If we closely examine the example STN from Figure 2.1(a) again, then the infinite weight $w_{3 \rightarrow 2} = \infty$ might strike the reader as odd. Indeed this value for $w_{3 \rightarrow 2}$, which imposes the inequality $x_2 - x_3 \leq \infty$, is not entirely accurate.

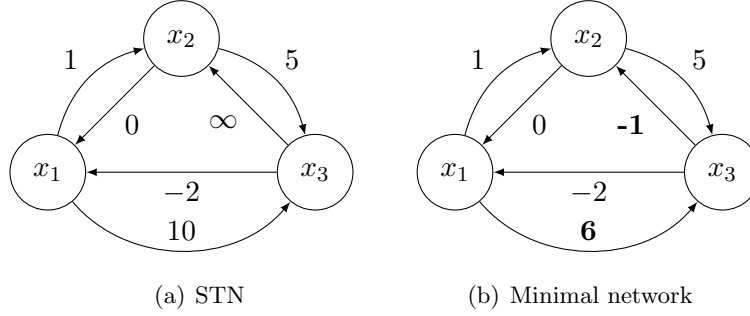


Figure 2.1: An example Simple Temporal Network (STN) and corresponding minimal network.

The constraints $c_{3 \rightarrow 1}$ and $c_{1 \rightarrow 2}$ impose the inequalities $x_1 - x_3 \leq -2$ and $x_2 - x_1 \leq 1$ respectively. If we take the sum of these inequalities we find that $x_2 - x_3 \leq -1$ or $w_{3 \rightarrow 2} = -1$ which is a much more tight bound than the previous value of infinity. In the same way we can also tighten $c_{1 \rightarrow 3}$ due to $c_{1 \rightarrow 2}$ and $c_{2 \rightarrow 3}$.

When every constraint $c_{i \rightarrow j}$ has a weight equal to the shortest path in the STN from i to j , the result is the *minimal network* of the STN. Figure 2.1(b) shows the minimal network of the example STN from Figure 2.1(a) that we discussed in this paragraph.

We will use the terms STP and STN interchangeably in the rest of this work because of their close relation.

Consistency

Dechter et al. [DMP89] show that an STN is consistent if and only if there are no negative cycles in the minimal network. Informally, the proof goes as follows.

Suppose that there is a negative cycle in the minimal network, from x_i to x_i . When we take the summation of the inequalities along the cycle, we will find that $x_i - x_i < 0$, meaning that x_i should occur before itself which is clearly inconsistent. On the other hand, if there are no negative cycles, all shortest-paths are well defined and the assignments $x_i = -w_{i \rightarrow 0} \forall x_i \in X$ and $x_i = w_{0 \rightarrow i} \forall x_i \in X$ are always possible.

The full and more formal version of the proof can be found in [DMP89, Theorem 3.1 and Corollary 3.2]. The two assignments or solutions given above that are always possible for a consistent STN are called the *earliest start time assignment* and the *latest start time assignment* respectively.

These assignments assign to each timepoint x_i the earliest possible time $-w_{i \rightarrow 0}$ and latest possible time $w_{0 \rightarrow i}$ respectively. This is of great convenience to us because we generally want to work with the earliest possible schedule to reduce makespan. We will use the notation $lb(x_i)$ and $ub(x_i)$ to refer to the value of timepoint x_i in the earliest and latest start time assignments respectively.

Algorithms

Because of its convenient graph representation, many algorithms—some of which were invented long before the STP—can be used for determining consistency or calculating the minimal network. We will mention two here that are of use to us. The first is the Floyd-Warshall all-pairs-shortest-paths algorithm [Flo62, War62] for calculating the minimal network, the second is the Incremental Full Path Consistency algorithm [Pla08a] for incrementally calculating the minimal network after updating a constraint.

Algorithm 1 Floyd-Warshall All-Pairs-Shortest-Path Algorithm

```

1: for  $k = 1$  to  $n$  do
2:   for  $i = 1$  to  $n$  do
3:     for  $j = 1$  to  $n$  do
4:        $w_{i \rightarrow j} = \min\{w_{i \rightarrow j}, w_{i \rightarrow k} + w_{k \rightarrow j}\}$ 
5:     end for
6:   end for
7: end for

```

The Floyd-Warshall algorithm is a simple algorithm, as can be seen in Algorithm 1. It has time complexity $O(n^3)$ with n the number of timepoints or vertices. It iteratively updates the shortest path from x_i to x_j by checking if the path through x_k is shorter for x_k from x_1 to x_n . When the algorithm is finished, all edges are labeled with the shortest distance from their source to their destination vertex if such a distance exists smaller than ∞ . This is why the algorithm is also called the all-pairs-shortest-paths algorithm: it calculates the shortest paths from all vertices to all other vertices. Note that the distances from vertices to themselves are taken to be 0 and if a negative cycle exists, some of the vertices will have a distance to themselves smaller than 0, allowing for easy inconsistency detection.

Note that the Δ STP algorithm due to Xu and Choueiry [XC03] has been empirically shown to outperform the Floyd-Warshall algorithm. We have not used it here because the Floyd-Warshall algorithm is simpler and works well enough for our purposes. See also [Pla08b, Section 3.3] for a discussion of Xu and Choueiry.

Often, we already have an STN that is minimal and need to update the temporal information, for instance because something has changed in

the real world process that is modeled by the STN. We could of course update the constraint and simply rerun the Floyd-Warshall algorithm. But since we already have a near-minimal network maybe we can recalculate the minimal network faster. This is indeed true. The Incremental Full Path Consistency (IFPC) algorithm can recalculate the minimal network when given a minimal network and a constraint to add with time complexity $O(n^2)$. Alternatively, if the network has become inconsistent due to the new constraint, the algorithm detects this. We will not list the entire algorithm here, but instead refer the reader to [Pla08a, Figure 4].

2.2.3 Applying the Simple Temporal Problem

We will now show how an STN representing the temporal information of a train maintenance scheduling problem is created. The temporal information consists of a set of trains with release dates, due dates and activities, the activities with durations. The construction of an STN $S = \langle X, C \rangle$ to encode that temporal information goes as follows.

We need to specify how the set of timepoints X is constructed and how the set of constraints C is constructed. The set of timepoints X is constructed as a union of three sets:

$$X = \{z\} \cup \{s_i, e_i | i \in \{1, \dots, n\}\} \cup \{s_{ij}, e_{ij} | j \in \{1, \dots, N_i\}, i \in \{1, \dots, n\}\}$$

The first set $\{z\}$ is simply the temporal reference point which we need to make references to absolute time. Unless otherwise specified, z will always have the value 0. The second set adds timepoints for the arrival and departure of trains. The third set adds start and end timepoints for all activities of all trains.

The constraint set C is a little more complex because we also have to define the value of the weights.

First, we add constraints to set the s_i timepoints to the release dates of the corresponding trains t_i . This requires adding constraints $c_{z \rightarrow s_i}$ and $c_{s_i \rightarrow z}$ with $w_{z \rightarrow s_i} = -w_{s_i \rightarrow z} = rd_i$ for every train $t_i \in T$. In the same way we add the constraints for the due date timepoints: $c_{z \rightarrow e_i}$ and $c_{e_i \rightarrow z}$ with $w_{z \rightarrow e_i} = -w_{e_i \rightarrow z} = dd_i$ for every train $t_i \in T$.

Activities have a fixed duration, so we constrain the allowed time difference between s_{ij} and e_{ij} to only allow the duration d_{ij} of activity a_{ij} by adding the constraints $c_{s_{ij} \rightarrow e_{ij}}$ and $c_{e_{ij} \rightarrow s_{ij}}$ with $w_{s_{ij} \rightarrow e_{ij}} = -w_{e_{ij} \rightarrow s_{ij}} = d_{ij}$ for every activity $a_{ij} \in A$.

Finally, we need to add precedence constraints so that all activities can only start after their train has arrived and must be finished before their train leaves. Furthermore, we need to add all the precedence constraints that may exist between activities. We have already seen that adding a precedence constraint such that timepoint x_a occurs before timepoint x_b is

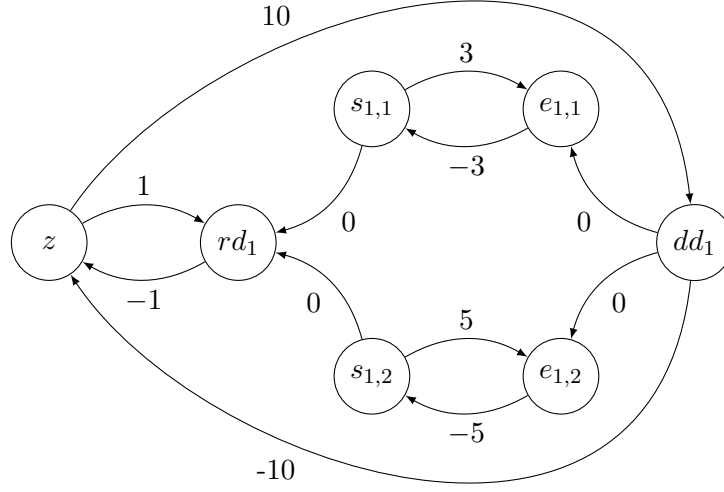


Figure 2.2: Example STN encoding temporal information.

done by adding a constraint $c_{x_b \rightarrow x_a}$ with weight $w_{x_b \rightarrow x_a} = 0$. Adding these constraints, we should end up with an STN that looks like the one drawn in Figure 2.2. In this example instance we have depicted one train with a release date of 1, a due date of 10 and two activities with durations 3 and 5.

This concludes our discussion of the Simple Temporal Problem for dealing with temporal constraints. We have seen that the STP can model the types of temporal constraints that we have to deal with in our problem, we have seen a graph representation called the Simple Temporal Network, polynomial-time algorithms for solving them and an example. We will now move on to the resource constraints.

2.3 Addressing the Resource Constraints

In our model we have seen that the resource constraints are disjunctive, they model the *choice* that must be made in ordering the activities on the resources. Unfortunately, these ‘either/or’-type situations cannot be modeled in a plain linear program or with the Simple Temporal Network. In this section we discuss extensions of both methods. Both extensions are known NP-complete problems. For the interested reader, we have included a complexity analysis of the train maintenance scheduling problem in Appendix A. The appendix includes a really short introduction to complexity theory.

2.3.1 Integer Linear Programming

In linear programming there is no way of adding the disjunctions we used to describe the resource constraints. Furthermore, even if there was a way to include all the disjunctions, there are exponentially many disjuncts in every disjunction so we can hardly call that an efficient encoding of the problem.

Nevertheless, there is a possibility to use *integer linear programming*. In integer linear programming the values of the decision variables are restricted to the set of integers, instead of real numbers. Integer linear programming is much harder than plain linear programming but there are very good solvers that can take on large problems.

To encode our problem as an integer linear program, we have to forget about the ‘old’ linear programming formulation that we have been discussing up to now and instead take a different approach. This approach involves a lot of counting and housekeeping: many constraints are used in such a way that at every time period t , no resource is overused and all temporal constraints are satisfied. Although we must add a lot of constraints to make this work, the amount is polynomial in the input size which is better than the exponential number of disjunctions we ran into when making the model.

In Chapter 3 we will explore an integer linear programming approach based on the work of Pritsker et al. [PWW69]. In the introduction to that chapter we will also more thoroughly introduce linear programming.

2.3.2 Disjunctive Temporal Problem

Besides linear programming, we have discussed the Simple Temporal Problem. Using this framework and the algorithms that we know for it, we can solve the temporal problem in polynomial time, just like with a linear program. However, just as with a linear program, we cannot model the disjunctions that make up the resource constraints.

In 2000, Koubarakis and Stergiou [SK00], introduced the Disjunctive Temporal Problem precisely to model these disjunctions we are faced with. However, the problem we already mentioned in our discussion of using a linear programming approach still remains in this approach: there are $n!$ ways to permute n activities and we would have to explicitly describe all of the permutations in disjunctions. Writing down an exponential amount of constraints takes an exponential amount of time. It should come as no surprise that the DTP is an NP-complete problem. We discard it as a solution method because we consider it too impractical to construct the instances because of the expected size problems when constructing $n!$ permutations for practical problem sizes.

2.3.3 Heuristic Approach

Having established that our problem is NP-complete,³ we now consider using a heuristic solution method in order to deal with the complexity of our problem.

If we initialize an STN with the temporal constraints for a set of trains requiring maintenance, then the STN can tell us if there is a solution to the temporal problem. Moreover, it describes precisely *all* the temporal solutions to the problem. It describes not just one, but all the possible execution schedules that are consistent with the temporal constraints. Now suppose that we add additional constraints to the STN so that all the temporally valid schedules that it describes are also consistent with the available resources and resource requirements. If we could pull that off, then we have a solution to the problem of maintenance scheduling.

The problem of satisfying the resource constraints involves sequencing the activities on the resources so that they are not overused. However, there are too much possible permutations to test them all and pick the optimal one. This is where we will seek the help of a heuristic. The heuristic should analyze the STN and figure out at what times (or timepoints rather) there is contention for the available resources. The heuristic can then alleviate some of the contention by selecting two of the contending activities and sequencing them. Luckily, such a heuristic has already been developed by Cesta, Oddi and Smith in [COS98].

2.4 Conclusion

Based on the analysis we have conducted we will attempt to solve the scheduling problem at NedTrain with an exact algorithm using integer linear programming based on the work of Pritsker et al. and a heuristic method based on the work of Cesta et al. based on STNs.

We have chosen the ILP method because it is directly applicable and because it can be used to calculate optimal solutions—even though that might take a long time. We can use these optimal solutions to determine the effectiveness of other methods. The reason we have chosen to use the work of Cesta et al. is that we believe that we can extend their methods so that they cover our problem and because it is a heuristic, incomplete algorithm. This last fact is important because the scheduling problem we face is NP-complete and using an incomplete method may be our only option if we want to keep runtime low.

Chapter 3 shows how we used the zero/one integer linear programming model of Pritsker et al. to calculate solutions using CPLEX, an optimization software suite. Chapter 4 describes the ESTA⁺ heuristic which we have

³See Appendix A for details.

developed as an extension to the ESTA heuristic from the literature.

3

Integer Programming Approach

In this chapter we will explore an integer linear programming approach to generating base schedules for train maintenance and possibly production schedules as well. An integer linear programming (ILP) approach can take a very long time to come up with a solution but the upside is that an ILP approach can always find an optimal solution when given enough time and the problem is feasible.

In linear programming the goal is to minimize or maximize a *goal function* over *decision variables*, given some inequalities that constrain the allowable values of the decision variables. The range of allowed values is also called the *feasible region*. The decision variables can represent, for example, times of events in the real world and as such, linear programming can be used for scheduling—which is what we will do in this chapter. When some or all decision variables can only take integer values we speak of *mixed-integer* or *integer (linear) programming* respectively. When the decision variables are further constrained to only take the values 0 or 1, we speak of *zero/one integer programming*.

Let us consider an example linear program of two variables x and y with $z = x + y$ as a goal function to be maximized and constraints $c_1 : x \leq 3$ and $c_2 : x + 2y \leq 6$. All variables are normally assumed to be positive. Figure 3.1 graphically depicts this problem. We can see that the point $(3, 1.5)$ is the optimal solution to this LP because no higher value for the goal function z is attainable in the feasible region. If x and y are restricted to integer values, we can see that z attains its highest value at $(2, 2)$ and $(3, 1)$. In these points $z = 4$. Note that the linear programming solution provides an upper bound for the integer linear programming problem. A proof of this fact is given in [Tij04, p. 91, Principe 1 (Dutch)].

Since at least the 1960s, research has been devoted to scheduling and scheduling with resource constraints using linear programming models. Although plain linear programming is in complexity class P (as shown by Khachiyan in 1979¹ and improved upon by Karmarkar in 1984 in [Kar84]),

¹Original text in Russian and hard to track down; see the references in the Karmarkar

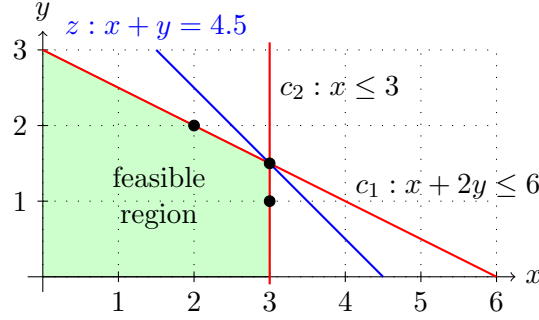


Figure 3.1: Example linear programming problem, the LP optimum is attained at $(3, 1.5)$, the ILP optimum at $(2, 2)$ and $(3, 1)$.

the variant we will use in this chapter is not. Because all our decision variables are binary variables which can only take the values 0 or 1, our problem is NP-complete. In fact, it is one of Karp’s famous 21 NP-complete problems which he wrote about in 1972 in his seminal work entitled “Reducibility among combinatorial problems” ([Kar72]). He shows that the zero/one integer programming problem can be reduced from the boolean satisfiability problem which is well-known to be NP-complete (see for instance [Heu08]).

In searching for a linear programming solution for the TMSP, we use the work of Pritsker et al. [PWW69]. They describe an easy-to-follow model which we can use directly. In the next section we will explain how we apply the model of Pritsker et al. by first explaining all the variables that are used and then the goal functions and constraints in Section 3.2.1. We use CPLEX as our integer programming optimization software suite because it is available to us at the university. Appendix C describes how we use CPLEX.

3.1 Model Variables

Table 3.1 gives an overview of the variables that we need for the ILP model in addition to the variables already introduced in the previous chapter. Note that the variables listed here come from [PWW69]. We have merely updated their description to reflect more clearly our application domain. There are a couple of things to note about these variables:

- The decision variables are x_{ijt} and x_{it} . Our goal function will be expressed in these variables and we will use an ILP solver to find values for these variables. The x_{ijt} variables indicate when activities

paper ([Kar84]) for more information.

| variable | description |
|-----------|--|
| x_{ijt} | 1 if activity j of train i was completed in period t , 0 otherwise |
| x_{it} | 1 if train i has been completed by period t , 0 otherwise |
| e_i | earliest possible completion period for train i |
| l_{ij} | earliest possible completion period for activity j of train i |
| u_{ij} | latest possible completion period for activity j of train i |

Table 3.1: Notation used in integer linear programming model. Note that x_{ijt} and x_{it} are the decision variables.

should finish. Subtracting the activity duration, we get a start time for every activity: a schedule.

- Most non-decision variables—let us call them support variables—take their values straight from the problem instance. Only l_{ij} , u_{ij} and e_i require some explanation.
 - The earliest possible completion period for any activity is attained when the activity is scheduled immediately after the train is released. The latest is attained when an activity is finished exactly on the due date of the train. Therefore we have defined $l_{ij} = rd_i + d_{ij}$ and $u_{ij} = dd_i$ for all activities $a_{ij} \in A_i$ for all trains $t_i \in T$.
 - The earliest possible completion period for any train is attained when processing starts immediately on the release date and all activities are processed with no breaks in between them. In this case, the completion period equals the release date of the train plus the duration of all its activities. Therefore we have defined $e_i = rd_i + \sum_{j=1}^{N_i} d_{ij}$ for all trains $t_i \in T$.

3.2 Finding Valid Schedules

Recall that a schedule is valid if it contains an activity start time for every activity so that no temporal constraints, resource constraints or precedence constraints are violated. Furthermore, note that finding valid schedules does not require the optimization of anything. Consequently, for finding valid schedules there is no (meaningful) goal function. Because a goal function is nonetheless required in a proper linear program—CPLEX will not accept a program without a goal function—we will add the following goal function:

$$\text{Maximize } z = 0. \tag{3.1}$$

Because this goal function z is constant it has no effect on the variables. Now we need to add the proper constraints such that only valid schedules are generated.

3.2.1 Constraints

To extract a schedule, we need the values of the x_{ijt} variables, which tell us when activities are finished and, by subtracting the activity duration, when activities should start. However, if we do not constrain the value of these variables, then we cannot be sure that they describe valid schedules. Here we will list the constraints that we must add in order to make sure that the values only take the values we want.

We will see that the number of constraints added is polynomially bounded by the total number of time periods, the number of trains, the number of activities and the number of resources. This is an improvement over the exponentially many disjunctions we encountered in Section 2.1.3.

Activity Completion Constraints

The x_{ijt} variables are constrained such that each activity a_{ij} is finished between l_{ij} and u_{ij} . If an activity is finished between these points, that means by definition of these variables that the activity has been executed in between the release and due dates of the corresponding train. Therefore it is valid as far as temporal constraints go. The resource constraints will be addressed later on. Because the x_{ijt} variables are binary variables that can only take the values 0 and 1, equation 3.2 specifies that for every activity there is exactly one period in which it is finished.

$$\sum_{t=l_{ij}}^{u_{ij}} x_{ijt} = 1 \quad \text{for } i \in \{1, \dots, n\}, j \in \{1, \dots, N_i\} \quad (3.2)$$

This will introduce a number of constraints equal to the total number of activities over all trains.

Precedence Constraints

For every precedence relation $pc = \langle a_{ij}, a_{vw} \rangle \in PC$, which specifies that activity a_{ij} must be finished before activity a_{vw} can be started, we added the equation $e_{ij} \leq s_{vw}$ to the model in Section 2.1.3. The counterpart in this model is almost as simple, only we have to figure out the start and end times of activities based on the x_{ijt} variables which denote when an activity is finished.

Recall from equation (3.2) that for every activity a_{ij} there is exactly one t for which $x_{ijt} = 1$ and $x_{ijt} = 0$ for all other values of t . If $x_{ijt} = 1$ then activity a_{ij} was finished in time period t . The finish time of an activity

a_{ij} is then given by $\sum_{t=l_{ij}}^{u_{ij}} tx_{ijt}$.² Observe that if an activity a_{ij} finishes at $t = 8$, i.e. $x_{ij8} = 1$, then it was started at $t = 8 - d_{ij}$ with d_{ij} the duration of activity a_{ij} .

Now, to specify that activity a_{ij} must precede activity a_{vw} we add a constraint

$$\sum_{t=l_{ij}}^{u_{ij}} tx_{ijt} \leq \left(\sum_{t=l_{vw}}^{u_{vw}} tx_{vwt} \right) - d_{vw}. \quad (3.3)$$

This specifies that the finish time of activity a_{ij} must be less than or equal to the start time of activity a_{vw} . Effectively, this adds the constraint $e_{ij} \leq s_{vw}$ which is what we set out to achieve.

At this point we have a model that is temporally correct. If we run this model, we get a temporally valid schedule—provided that it exists—just like we would if we had used an STN. We should now incorporate the resource constraints to make the ILP model useful.

Resource Constraints

Although the resource constraints are long and a lot of them are needed, they are fairly easy to understand. For every time period and every resource the used capacity must be less than or equal to the total capacity. Equation 3.4 shows the resource constraints.

$$\sum_{i=1}^n \sum_{j=1}^{N_i} \sum_{w=t}^{t+d_{ij}-1} r_{ijv} x_{ijw} \leq c_{vt} \quad \text{for } t \in \{\min_{i=1}^n(rd_i), \dots, \max_{i=1}^n(dd_i)\}, v \in \{1, 2, \dots, k\} \quad (3.4)$$

The difficulty here is that we need to calculate the total required capacity for every resource k on every time t . To make this sum we need two bits of information of every activity: whether or not it is running at time t and how much resources it requires. The latter is easy, that is just q_{ijk} —the required amount of a resource is equal over the entire duration of an activity. The former, however, is slightly harder. To figure out which activities are running at time t , we use the x_{ijt} variables and the activity duration variables d_{ij} .

An activity is running in time t if it was finished in the interval $[t, t + d_{ij} - 1]$. This is what the third summation does. The first two merely consider all activities of all trains.

²We do not have to sum over all t because $x_{ijt} = 0$ for $l_{ij} > t > u_{ij}$ by design.

At this point we have added all constraints needed for the model to produce valid schedules for train maintenance. In Section 3.4 we will run some experiments to see how well this method scales and look at the quality of the generated schedules. But first we will introduce a more meaningful goal function.

3.3 Maximizing Throughput

If instead of just finding valid schedules we would like to find optimal schedules for some measure of optimality, then we must add a meaningful goal function to the ILP encoding of the previous section.

Suppose we want to maximize throughput. Maximizing throughput means getting every train out of the door as quickly as possible or minimizing the combined completion times of all trains. Recall from Table 3.1 that the x_{it} variable is 0 for t smaller than the completion period of train t_i and 1 for t greater than or equal to the train completion period. Maximizing throughput therefore equals maximizing the sum of the x_{it} variables over all trains and time periods as shown in equation 3.5.

$$\text{Maximize } z = \sum_{i=1}^n \sum_{t=e_i}^{dd_i} x_{it} \quad (3.5)$$

Note that it is not necessary to take the sum over all time periods. By definition, $x_{it} = 0$ for $t < e_i$ and $x_{it} = 1$ for $t \geq dd_i$. It is between e_i and dd_i that the value of x_{it} can vary and we therefore only consider this interval when taking the sum. This optimization helps to keep the linear program smaller.

The linear program we have shown so far is unbounded because the x_{it} variables have not yet been constrained. We will constrain these now.

3.3.1 Train Completion Constraints

The train completion variables x_{it} can be deduced from the activity completion variables: a train is completed when all of its activities are completed. Therefore, the x_{it} variables depend on the x_{ijt} variables. The constraints can be hard to understand at first glance. We will first show what they look like in equation 3.6 and then explain them.

$$x_{it} \leq \frac{\sum_{j=1}^{N_i} \sum_{b=l_{ij}}^{t-1} x_{ijb}}{N_i} \quad \text{for } i \in \{1, \dots, n\}, t \in \{e_i, \dots, dd_i\} \quad (3.6)$$

Intuitively, what is happening here is that for every train t_i and time period t we are counting, in the numerator, the number of activities that have

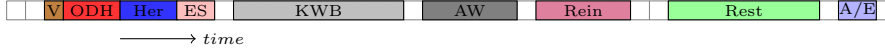


Figure 3.2: 7400-series train from Leidschendam base schedule.

been finished by time t and dividing that by the total number of activities, giving us a fraction, between 0 and 1, of activities completed. Because x_{it} is a binary variable, it will remain 0 until t is sufficiently large that the fraction of completed activities for train i no longer is less than 1 but instead equals 1, which indeed happens only when all activities have been completed.

3.4 Experiments

We have run some experiments to test whether or not the ILP approach is efficient enough to be used to create base schedules and production schedules. We have the CPLEX optimization suite at our disposal at the university. Appendix C gives a short overview of how to use CPLEX. The machine used has two quad-core Intel Xeon 3.06GHz processors and 16GB of RAM memory.

The main measures we use are throughput and runtime. Time is measured in seconds unless specified otherwise. Throughput is measured by taking the sum of the finish times of the last activity to finish for each train, just like the optimization function we used. This means that a lower outcome for the throughput is indicative of a faster and better schedule.

3.4.1 Performance on Base Schedules

To test if the approach is suitable to generating base schedules, we have taken the Leidschendam base schedule mentioned in the first chapter and converted it into an ILP instance. This schedule gives us a very concrete indication of the scale of base schedules.

Most of the trains in the Leidschendam base schedule start with an inspection (marked ‘V’ for ‘Voorinspectie’), then maintenance (marked ‘ODH’ for ‘Onderhoud’), repairs (marked ‘Her’ for ‘Herstellingen’) and so-called ‘E-staat’ (marked ‘ES’) work in that order. After these initial activities a train typically goes to the ‘kuilwielbank’ followed by the ‘aardwind’ and is cleaned and sometimes has some leftover work that still needs finishing. Figure 3.2 shows a train from the 7400-series from the base schedule.

To reflect the apparent standard way of working, we have added the precedences shown in Figure 3.3 to the ILP model for every train where they were applicable—not all trains in the base schedule have the same list of activities. Note that there can be only one activity running at any time on a single train. Therefore, the algorithm needs to choose how to order

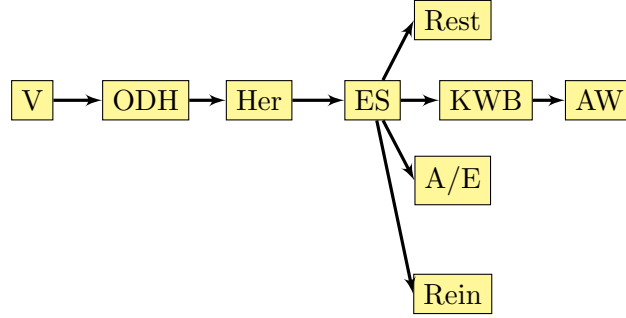


Figure 3.3: Precedence relations between activities in 7400-series train from Leidschendam base schedule.

| type | runtime | throughput |
|-------------|-------------|------------|
| actual | – | 978 |
| ilp optimal | 132 seconds | 710 |

Table 3.2: Comparison of actual Leidschendam base schedule and optimized version.

the tasks that are not already strictly ordered by the precedence constraints and synchronize the resource usage across all resources and all activities.

Taking the instance we deduced from the base schedule, we want to know how long it takes to calculate an optimal base schedule of the size of the Leidschendam base schedule and we want to compare it to the schedule that is used in practice. Table 3.2 shows the runtime and throughput that were obtained by running CPLEX on the Leidschendam base schedule instance using the linear program that we have described in this chapter, including goal function.

It took CPLEX slightly more than two minutes to calculate the optimal solution. Arguably, the instance was quite constrained, with all the precedence constraints that were added, possibly making it easier for CPLEX to find the optimum. Nevertheless, the result is better than we expected.

The throughput is 27% better than that of the actual base schedule. In fact, because we chose 1 hour to be the unit of time in this base schedule instance, we shaved off 268 hours in total, for an average of 10.7 hours per train. Part of the reason for the better throughput is that in the actual base schedule, schedulers have inserted a lot of slack space: buffers to deal with uncertainty. Our approach is intended to provide an efficient schedule while dealing with uncertainty as it comes along. We do not insert slack space everywhere but instead our methods can update the schedule to deal with the new situation whenever a change occurs: we only solve disruptions if they actually occur instead of anticipating them at all times.

3.4.2 Scalability

The result of the previous experiment is quite encouraging. We expected the calculation of an optimal base schedule to be more difficult but it was not really a challenge for CPLEX. Now we will feed it more activities to see how much it can handle.

The instances we will use are randomly generated with the following properties. Because of the temporal constraints imposed on the writing of this thesis, a runtime limit of one hour is included for every test performed. This means that after one hour, the ILP solver will return the best solution found so far and give up. The other properties are as follows, with $U[a, b]$ denoting the uniformly random selection of an integer from the set $\{a, \dots, b\}$:

- available resources are modeled after the Leidschendam depot: 6 ‘putsporen’, 2 ‘aardwinden’, 1 ‘kuilwielbank’, 12 mechanical engineers and 12 electrical engineers,
- 5–8 trains per instance,
- release dates all set to 0,
- due dates set to twice the duration of all the activities of a train,
- 5 activities per train,
- activity duration from $U[1, 8]$,
- activities require $U[1, 5]$ engineers, each one either an electrical or a mechanical engineers with equal probability and
- activities require either a ‘putspoor’, a ‘kuilwielbank’ or a ‘aardwind’ with 80%, 10% and 10% chance respectively.

Using random release dates would only lower concurrency and resource contention. Therefore we believe that setting the release dates to 0, allowing work on all trains to start immediately, is most interesting if we are to judge the strength of this method.

Figure 3.4 shows the time it took CPLEX to find the optimal solution to instances generated with the properties described above. For each number of trains, ten instances were generated. So the data points in the figure represent averages values over 10 instances. Note that we only took the average of the instances that were solved within the hour. The green line in the figure shows what percentage of the instances were in fact solved within the hour. It is clearly visible that once there is more resource contention, more trains, that the success rate sharply drops as solution time sharply increases.

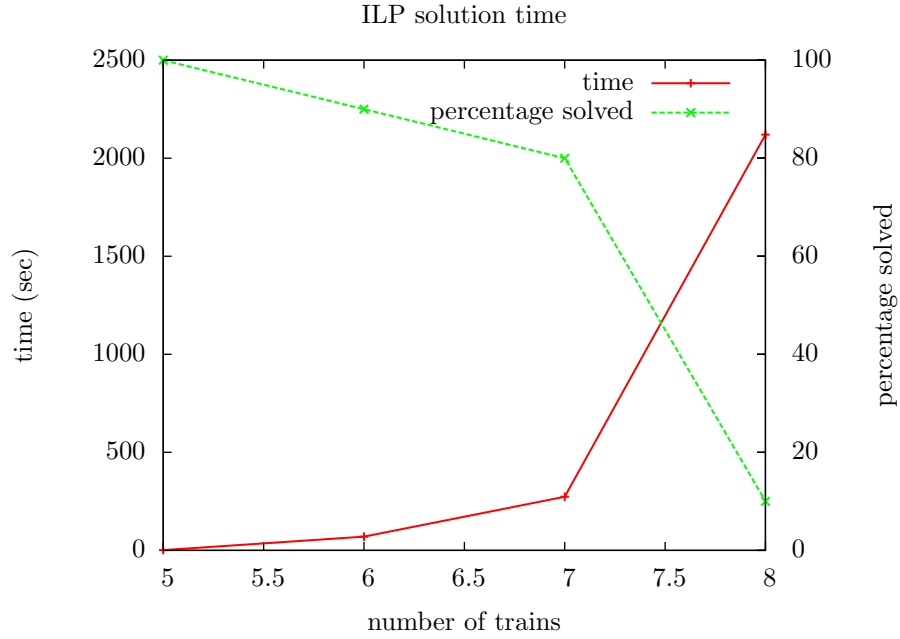


Figure 3.4: Solution time and success rate for different problem sizes.

3.5 Conclusion

In this chapter we have discussed a zero/one integer linear programming approach to creating schedules for train maintenance. We have shown a model from the literature that we can use to model our problem.

The method works well for creating base schedules. A 27% decrease in maintenance time was shown over an existing base schedule used in the Leidschendam NedTrain depot. Furthermore, this schedule was constructed in only two minutes. It appears the method is able to deal with the workload well.

For creating production schedules however, the method does not work well. Experiments have shown that the method takes too long to generate even small schedules—that is, with much less workload than in the base schedule—when there is any significant resource contention, such as for engineers which we have added in the scaling experiments. Note that we have kept the number of engineers relatively low when compared to the workloads to reflect the actual situation at NedTrain depots.³

Because of the NP-completeness of the problem we expected bad scaling with problem size and this is indeed the case. This is already evidenced by the size of the ILP input files that quickly grows from a few kilobytes to

³Internal NedTrain reports (the K3 report [K307]) suggest that much time is spent waiting for the availability of engineers.

multiple megabytes when the problem size increases. By using bigger time units—for example 15 minutes or 1 hour instead of individual minutes—and thereby decreasing the values of all time variables, the ILP instances become smaller and more manageable. However, the exponential size of the solution space quickly catches up with such linear optimizations and they come at the cost of decreasing precision with increasing time unit size. In fact, we already applied such an optimization for the base schedule by scheduling using whole hours only. For base schedules this is acceptable but for production schedules much higher resolution is needed.

In the next chapter we introduce a heuristic method for finding train maintenance schedules. Finally in Chapter 5 we will compare the methods introduced in this chapter and the heuristic method.

4

Heuristic Approach

In this chapter we will present a heuristic algorithm for finding schedules for train maintenance. This algorithm will be heavily based on the ESTA algorithm for the Multiple-Capacitated Metric Scheduling Problem from [COS98, COS00]. The motivation behind developing a heuristic approach is that the exact method that we explored in the previous chapter was unable to deliver results fast enough, at least for generating and updating production schedules.

So what is a heuristic? One way to look at it is to say that it is a rule-of-thumb or a best-practice. Human experts often use rules-of-thumb when working on complex problems. A heuristic mimics this behaviour. Using a heuristic can potentially speed up the solving process immensely at the sacrifice of some precision. The loss of precision comes from the fact that a heuristic method is usually not a *complete* method and therefore does not consider *all* possible solutions, like the ILP approach from the last chapter. This is also precisely why it can be much faster: using a heuristic we will search for valid solutions instead of optimal ones.

The heuristic used in this chapter is based on retaining temporal flexibility which means to exclude as few schedules as possible at each decision. The idea will become clearer in the coming sections.

The rest of this chapter is structured as follows. In Section 4.1 we will show the main algorithm for finding valid train maintenance schedules which we will call ESTA^+ . The algorithm produces a fixed start time for every activity. In Section 4.2 we will discuss what robustness is, why the ESTA^+ algorithm as well as the ILP approach from the previous chapter does not produce robust solutions and how we can achieve robustness. In Section 4.3 we will show a way to support resources with variable capacity, which is in our problem formulation, but is not directly supported by our algorithms in this chapter. Because the ESTA^+ algorithm is an extension of the ESTA algorithm from the literature, it is important that we make clear what our contributions are and what is an application of existing work. We have listed our contributions separately in Section 4.4. Finally, in Section 4.5 we will give some conclusions.

4.1 Finding Valid Schedules

In Chapter 2 we have made clear that there are two distinct types of constraints that we have to deal with to solve the TMSP: temporal constraints and resource constraints. We have also introduced the STN as a means of dealing with the temporal constraints. In this section we will introduce the $ESTA^+$ algorithm for finding valid schedules for the TMSP. This algorithm takes a TMSP instance, constructs an STN which encodes the temporal constraints of the instance and then proceeds to iteratively detect and resolve *resource peaks* by posting precedence constraints between activities until the earliest start time schedule of the STN is valid.

Using the STN representation of a TMSP and the algorithms introduced in Section 2.2.2 we can decide on consistency of the temporal information in the TMSP and derive a first schedule for a TMSP by taking the *earliest start time assignment* of the STN. Recall from Chapter 2 that the earliest and latest start time assignments are always valid for a consistent STN. We use the *earliest start time assignment* because it is preferable to finish work sooner instead of later. Note, however, that because we have so far totally ignored the resource constraints of the TMSP, the current earliest start time assignment is probably not a valid TMSP schedule.

In this section we will introduce the $ESTA^+$ algorithm for merging the temporal constraints with the resource constraints. The $ESTA^+$ algorithm is an extension of the ESTA algorithm by Cesta, Oddi and Smith [COS98, COS00]. It takes the earliest start time assignment of the TMSP's STN and examines it for resource constraint violations. When it finds a violation, it will add a precedence constraint in order to (at least partially) resolve the violation. Eventually, it will produce the TMSP's STN in a state where it is either inconsistent or the earliest start time assignment is a valid schedule for the TMSP. The problem with the ESTA algorithm is that it was designed for the Multi-Capacity Metric Scheduling Problem (MCMSP). This problem is similar to the TMSP but is more restricted. In the MCMSP

- there is a predetermined order among activities in each job—compare a job in MCMSP to a train in TMSP,
- activities can only require the use of exactly 1 unit of 1 resource.

We need to extend the ESTA algorithm so that it suits the TMSP. Note that we can force the above two limitations onto a TMSP by adding precedence constraints between all activities in a train for the first point and restricting the allowed resource requirements for the second point. Therefore, the MCJSSP is indeed a special case of the TMSP.

4.1.1 ESTA⁺ Algorithm Overview

We will now give a high-level overview of the ESTA⁺ algorithm and in the next three sections we will explain the inner workings.

The ESTA⁺ algorithm takes a TMSP instance with corresponding STN as input and gives an STN as output in which the earliest start time assignment constitutes a valid schedule if a solution to the TMSP was found. The algorithm works by iteratively detecting resource peaks in the earliest start time assignment of the STN, selecting a conflict from a peak and resolving the conflict by adding precedence constraints between the two activities in the conflict until no resource peaks remain in the earliest start time assignment. High-level pseudo-code of the ESTA⁺ algorithm is given in Algorithm 2.

Algorithm 2 ESTA⁺ Algorithm Overview

Input: TMSP \mathcal{T} with STN \mathcal{S} containing temporal constraints.

Output: Updated STN \mathcal{S} such that earliest start time assignment is a valid schedule or **false** if no schedule was found.

```

1: while true do
2:   peak  $\leftarrow$  FindPeak( $\mathcal{T}, \mathcal{S}$ )
3:   if no peak found then
4:     return  $\mathcal{S}$ 
5:   end if
6:   conflict  $\leftarrow$  SelectResolvableConflict( $\mathcal{T}, \mathcal{S}$ , peak)
7:   if no conflict found then
8:     return false
9:   end if
10:  ResolveConflict( $\mathcal{T}, \mathcal{S}$ , conflict)
11: end while

```

In the following three sections we will explain how the methods **FindPeak**, **SelectResolvableConflict** and **ResolveConflict** work as well as exactly what is a *resource peak* and a *resource conflict*.

4.1.2 ESTA⁺ Resource Peak Detection

From [COS98], the earliest start time demand $ESTD_{r_k}(t)$ for a resource r_k at time t is the sum of resource requirements of all activities that are scheduled, in the earliest start time assignment, such that their processing interval includes time t :

$$ESTD_{r_k}(t) = \sum_{a_{ij} \in A} P(t, a_{ij}) q_{ijk}.$$

Recall that q_{ijk} denotes the resource requirement of activity a_{ij} for resource r_k . The function $P(t, a_{ij})$ is a boolean function that takes on the

value 1 if and only if activity a_{ij} is running at time t :

$$P(t, a_{ij}) = \begin{cases} 1 & \text{if } lb(s_{ij}) \leq t \leq lb(e_{ij}), \\ 0 & \text{otherwise.} \end{cases}$$

The notation $lb(x)$ is used to denote the earliest possible value for a timepoint x in an STN: $lb(x) = -w_{x \rightarrow z}$ where z is the temporal reference point.

A *resource peak* is defined as a 3-tuple

$$\langle r_k, t, K = \{a_{ij} | a_{ij} \in A, P(t, a_{ij}) = 1, q_{ijk} > 0\} \rangle$$

for which the following condition holds:

$$ESTD_{r_k}(t) > c_{kt}.$$

The set K is the set of competing activities that require the use of resource r_k and are scheduled, in the earliest start time assignment, such that their processing interval includes time t . The *size* of a peak is defined to be the sum of the requirements, that is the earliest start time demand $ESTD_{r_k}(t)$.

We check for resource peaks only at those times t that correspond to the earliest possible assignments to activity start timepoints. This only works if resource capacity cannot change—especially not decrease—after an activity has started. Although in the problem statement in Chapter 2 we have defined resource capacity to be dynamic, we will assume that it is constant in the $ESTA^+$ algorithm. In Section 4.3 we present a simple fix that makes sure that the $ESTA^+$ algorithm still solves the entire problem as it was defined in Chapter 2.

When resource capacity is constant, then resource demand can only increase at the activity start timepoints. Consequently, these timepoints are the only timepoints at which we need to check for resource peaks. If there is a conflict at a time t that does not coincide with the start of any activity, then there must also be a conflict at at least one activity start timepoint.

The set of timepoints to check for each resource r_k can be further constrained to only those start timepoints of activities that themselves require the use of some non-zero amount of resource r_k , so $q_{ijk} > 0$. If an activity a_{ij} does not require the use of a resource, then there is no increase in demand for the resource at its start timepoint—at least not due to a_{ij} . Algorithm 3 gives the pseudo-code for the **FindPeak** procedure.

4.1.3 $ESTA^+$ Resource Conflict Selection

Given a resource peak, we now focus on resolving it. Two choices have to be made: which conflict to resolve and how to resolve that conflict. A *resource*

Algorithm 3 FindPeak Procedure

```

1: for  $r_k \in R$  do
2:   for  $a_{ij} \in A$  do
3:     if  $q_{ijk} > 0 \wedge ESTD_{r_k}(s_{ij}) > c_{ijk}$  then
4:        $K \leftarrow \{a_{vw} | a_{vw} \in A \wedge P(lb(s_{ij}), a_{vw}) = 1 \wedge q_{vwk} > 0\}$ 
5:       return  $\langle r_k, a_{ij}, K \rangle$ 
6:     end if
7:   end for
8: end for

```

conflict is simply a pair of activities $\langle a_{ij}, a_{vw} \rangle$ from the set of competing activities K in a resource peak. By posting a precedence constraint between the two activities in one of the conflicts the peak will become smaller and hopefully become resolved completely after some iterations.

For deciding which conflict to select the conflicts within a peak are classified into four categories from [COS98] as follows.

Type 1: $w_{e_{ij} \rightarrow s_{vw}} < 0 \wedge w_{e_{vw} \rightarrow s_{ij}} < 0$

Type 2: $w_{e_{ij} \rightarrow s_{vw}} < 0 \wedge w_{e_{vw} \rightarrow s_{ij}} \geq 0 \wedge w_{s_{ij} \rightarrow e_{vw}} > 0$

Type 3: $w_{e_{vw} \rightarrow s_{ij}} < 0 \wedge w_{e_{ij} \rightarrow s_{vw}} \geq 0 \wedge w_{s_{vw} \rightarrow e_{ij}} > 0$

Type 4: $w_{e_{ij} \rightarrow s_{vw}} \geq 0 \wedge w_{e_{vw} \rightarrow s_{ij}} \geq 0$

Type 1 conflicts are unsolvable because adding either one of the precedence constraints $a_{ij} \rightarrow a_{vw}$ or $a_{vw} \rightarrow a_{ij}$ will create a negative cycle, leaving the STN inconsistent. Note that a peak is unresolvable only when all of its conflicts are of type 1. Type 2 and 3 conflicts can only be resolved by adding $a_{ij} \rightarrow a_{vw}$ or $a_{vw} \rightarrow a_{ij}$ respectively and type 4 conflicts can be resolved either way.

We use a simplified version of the heuristic from [COS98] with the underlying idea to select the conflict that is closest to having its order forced, thus concentrating on areas where the problem is constrained the most. The heuristic selects the conflict with minimal ω_{res} where

$$\omega_{res}(a_{ij}, a_{vw}) = \begin{cases} w_{e_{ij} \rightarrow s_{vw}} + w_{e_{vw} \rightarrow s_{ij}} & \text{if all conflicts are type 4,} \\ \min\{w_{e_{ij} \rightarrow s_{vw}}, w_{e_{vw} \rightarrow s_{ij}}\} & \text{otherwise.} \end{cases}$$

4.1.4 ESTA⁺ Resource Conflict Resolution

Once a conflict $\langle a_{ij}, a_{vw} \rangle$ has been selected for resolving, we have to decide whether we will post the precedence constraint $a_{ij} \rightarrow a_{vw}$ or $a_{vw} \rightarrow a_{ij}$. In the case of a type 2 or type 3 conflict, there is no choice; the currently

existing constraints make one of the choices inconsistent. In either case, for any type conflict, we select the precedence constraint pc , where

$$pc = \begin{cases} a_{ij} \rightarrow a_{vw} & \text{if } w_{e_{ij} \rightarrow s_{vw}} > w_{e_{vw} \rightarrow s_{ij}} \\ a_{vw} \rightarrow a_{ij} & \text{otherwise.} \end{cases}$$

Using this simple condition, we maximize the number of valid start times with respect to each other for activities a_{ij} and a_{vw} or in other words, we retain the most flexibility or solutions.

When adding the selected precedence constraint, we need to update the temporal information in the STN to prepare it for the next iteration of the $ESTA^+$ algorithm. We use the Incremental Full Path Consistency (IFPC) algorithm by Planken [Pla08a] for adding the constraint, recalculating the minimal network and verifying whether or not the network is still consistent. The IFPC algorithm does this job in $O(n^2)$ by carefully examining what constraints need updating instead of simply recalculating all constraint weights like for instance the Floyd-Warshall algorithm mentioned in Chapter 2 does.

Note, finally, that the fact that a conflict is a type 2, 3 or 4 conflict only means that there exists an ordering of the two activities in the conflict that is consistent with the current constraints between the timepoints of these activities. However, it is still possible that adding the precedence constraint will make the STN inconsistent if the end of the activity that is delayed is pushed beyond the due date of the corresponding train. In that case the deadline for a train was exceeded. Luckily, the IFPC algorithm will notice this by detecting that the STN has in fact become inconsistent in such a case.

This concludes our description of the $ESTA^+$ algorithm for finding valid schedules for the TMSP. It takes a TMSP and an STN representing the temporal information as input and works by iteratively adding precedence constraints to the STN until the earliest start time assignment of the STN constitutes a valid TMSP schedule. In the next section we will discuss the robustness of the schedules generated by this algorithm.

4.2 Creating Robust Schedules

Robustness is generally considered to be a desirable property for a scheduling algorithm. We define it to be a measure of an algorithm's ability to absorb change. An important thing to realize is that a scheduling algorithm needs to calculate a schedule based on expectations of the future reality. As much as a scheduling algorithm would like to control reality, it does not and reality may have other plans. Indeed, "life is what happens to you while you are busy making other plans", as John Lennon famously wrote. Consequently, when a change occurs—e.g. an activity takes longer to complete than was planned—the algorithm should take this new information and come up with

a new solution that does not differ much from the original solution. The more an algorithm succeeds in this task, the more robust we will call it, or the solutions it generates. For the STN-based approach we will take a measure for robustness from [COS98]. This measure looks at the number of positions that different activities can take in time with respect to each other. See Section 5.2 for more information. For the ILP approach, we will first apply the Chaining Algorithm that will be introduced in this section and then apply the measure.

The ESTA^+ algorithm shown in the previous section generates an STN for which all schedules are consistent with the temporal constraints of the TMSP but only the earliest start time assignment is guaranteed to be consistent with the resource constraints. This is disappointing, because it does not allow us to take advantage of the innate ability of the STP or STN to absorb change. What we would like is to convert the earliest start time assignment into an STN in which all temporally consistent assignments are resource feasible without decreasing throughput in the earliest start time assignment. We can do this by removing all precedence constraints from the STN except those that were specified in the original problem definition and creating new precedence chains in a smart way.

From the earliest start time schedule we can deduce in what order activities are using all the different resources. All that is needed to make sure that no resource constraints are violated, is to make sure that each resource processes the activities requiring it in exactly the same order as in the earliest start time schedule. This is easily accomplished by throwing out all precedence constraints posted by the ESTA^+ algorithm and then adding new ones for each resource. These chains of precedences resemble what is called a ‘*werklijn*’ at NedTrain. On a related note, these chains also describe a partial order among activities such as describes by Policella in [Pol05]. He calls this ‘Partial Order Scheduling’ and presents it as a means of dealing with uncertainty in execution of schedules. If a delay occurs, there would be no immediate need to reschedule all activities because they simply shift a little bit in time where needed but the ordering among activities stays the same.

The Chaining algorithm from [COS98] performs the construction of the new chains of precedence constraints. The only catch is that it was designed for activities that can require 0 or 1 unit of only one resource’s capacity and in our setting resource requirements can be greater than 1 and more resources. We have to slightly adapt the algorithm to support this.

4.2.1 Chaining Algorithm

The Chaining algorithm [COS98] removes all precedence constraints posted by the ESTA^+ procedure and adds new chains of precedence constraints for every resource r_k , that force activities to take place in the same order as in

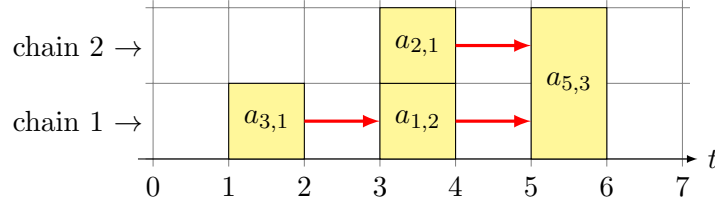


Figure 4.1: Cross section for resource r_k of example schedule with precedence constraints introduced by chaining algorithm marked red.

the earliest start time assignment. An example should make this clearer.

Suppose there are four activities, each with duration 1, that require the use of a resource r_k with capacity $c_k = 2$ and three of the activities require 1 unit of it and the fourth activity requires exclusive access. Figure 4.1 shows a graphical representation of this example. The horizontal position and width of the activity boxes correspond to the their fictitious earliest start time from the $ESTA^+$ algorithm and their duration. The vertical height corresponds to the amount they require, q_{ijk} . The vertical axis shows the total amount of resource r_k in use for every time t . The red arrows then show the precedence constraints that must be introduced in order to ensure that all STN assignments are valid with respect to the current resource. Note that a precedence constraint $a_{3,1} \rightarrow a_{2,1}$ is not needed because an earlier execution time for $a_{2,1}$ can never cause a conflict with $a_{3,1}$ with respect to this one resource.

A depiction like this is always possible for every resource when given a valid schedule because in a valid schedule all resource constraints are met by definition. Also, when the activities are chained in order of increasing start time, a greedy algorithm will always be able to chain all activities for a resource. Algorithm 4 shows such an algorithm for creating the chains. Once the chains have been created the last step is to actually remove all posted precedence constraints and add the new chains. This is done by adding a precedence constraint for every pair of consecutive activities in all chains. The number of constraints added this way is $|Q| - \sum_{r_i \in R} c_i$ with $|Q|$ the total number of resource requirements over all activities. $|Q|$ is bounded by the number of resources times the total number of activities minus 1. In this worst case all activities require the use of all resources and all resources have capacity 1. It is then necessary to add a number of precedence constraints equal to the number of activities minus 1 for every resource. The minus 1 is due to the fact that to $n - 1$ precedence constraints are required to sequence n activities. This is only a small polynomial in the size of the input. The Chaining algorithm generally runs fast and consumes a fraction of the time spent in the $ESTA^+$ algorithm.

Algorithm 4 Chaining Algorithm

```

1: for  $r_k \in R$  do
2:   chains  $\leftarrow \{\}$ 
3:   acts  $\leftarrow \{a_{ij} \mid j \in \{1, \dots, N_i\}, i \in \{1, \dots, n\}, q_{ijk} > 0\}$ 
4:   sorted  $\leftarrow \text{sort\_by\_earliest\_start\_time}(\text{acts})$ 
5:   for  $a_{ij} \in \text{sorted}, l \in \{1, \dots, q_{ijk}\}$  do
6:     // place  $a_{ij}$  in exactly  $q_{ijk}$  chains
7:     if  $|\text{chains}| < c_k$  then
8:       // put  $a_{ij}$  in a new chain
9:       newchain  $\leftarrow \{a_{ij}\}$ 
10:      chains  $\leftarrow \text{chains} \cup \text{newchain}$ 
11:    else
12:      // find an existing chain that can accept  $a_{ij}$ 
13:      for chain  $\in \text{chains}$  do
14:        if  $\text{lb}(\text{end timepoint of last activity in chain}) \leq \text{lb}(s_{ij})$  then
15:          chain  $\leftarrow \text{chain} \cup a_{ij}$ 
16:          break // chain found
17:        end if
18:      end for
19:    end if
20:  end for
21: end for

```

4.2.2 Using the Chaining Algorithm

Given a TMSP with STN generated by the $ESTA^+$ algorithm, now suppose a delay occurs on some activity. We then update the temporal information in the STN using the IFPC algorithm like in the $ESTA^+$ algorithm and immediately check for consistency. If the STN is still consistent, then the earliest start time assignment, which is now guaranteed to be a valid schedule, should again be chosen as a production schedule. If the STN has become inconsistent, then we need to start over and run the $ESTA^+$ algorithm from scratch with the updated information. If even that fails, then it is probably necessary to delay some deadlines or cancel some work before the $ESTA^+$ algorithm will find a solution.

4.3 Variable Capacity Resources

We have not discussed variable capacity resources in this chapter at all. The reason for this is that although it is possible to adapt the algorithms such that they directly support this, there is a much simpler solution that does not involve changing the algorithms.

The $ESTA^+$ algorithm only checks for resource constraint violations (resource peaks) at the start of every activity. It does not guarantee that the resource capacity does not change during an activity. Adapting the algorithms would involve not only checking the capacity at the beginning of an activity, but checking resource capacities at every resource capacity change as well. In practice there may only be three capacity changes, namely when the three different eight-hour shifts start at NedTrain depots. So it should not pose a big problem in terms of the amount of calculation needed.

However, a much simpler solution than updating the algorithms is possible. Consider the resource representing mechanical engineers and say that it has a capacity of 15, 10 and 5 respectively during the day, in the evening and in the night. Now we set the capacity to its maximum, which is 15, and add two dummy trains. The first dummy with a release date equal to the start of the second shift and a due date equal to the end of the second shift. The second dummy train the same but for the night shift. We then add one activity to each train, which lasts exactly eight hours and requires 5 and 10 units respectively of the mechanical engineers resource. Effectively, we have now blocked the capacity that is unavailable during the evening and night shift because the two dummy trains and activities can only be scheduled in one way: exactly overlapping the evening and night shift. The $ESTA^+$ algorithm will pick up on these extra activities and not schedule other activities during the same time if more than 10 or 5 engineers are required respectively. This approach of blocking excess capacity is simple and effective and we see no reason to take another approach.

4.4 Contributions

In this chapter we have discussed the ESTA^+ algorithm and the Chaining Algorithm. The algorithms are extensions of the ESTA and Chaining algorithms by Cesta, Oddi and Smith. Throughout the chapter, we have not been clear as to what constituted our contributions and what was taken straight from earlier work. We feel that this has made it possible for us to describe the algorithms more clearly. In this section we will highlight exactly what we changed to the algorithms so that we could use them.

We subdivide our contributions in extensions and optimizations. Some extensions or generalizations were needed because the original algorithms focus on a more strict set of problems than we do. Some optimizations were applied to make the algorithms run faster.

4.4.1 Extensions

The original ESTA algorithm expects activities within a single job to be totally ordered and non-overlapping. We have kept the non-overlapping constraint but removed the restriction that there must be a predetermined order among activities within each job. This does not require change to the ESTA algorithm, but merely influences the way the STN is constructed. To totally order activities within a job, one would add precedence constraints in the order desired. We make this optional instead of mandatory. Consequently, the start and end timepoints of activities become less constrained. There will be more resource peaks, but the algorithm has more freedom in choosing how to resolve them.

Secondly, the original algorithms assume that activities require only 0 or 1 unit of capacity of only 1 resource. At NedTrain, this is not the case. Activities may require multiple engineers plus other resources such as a specialized track. Allowing activities to require capacity of multiple resources instead of just one, necessitates a change to the ESTA algorithm. The ESTA algorithm checks for resource peaks at the start of every activity but only for the resource that each activity uses. In ESTA^+ we need to check for resource peaks on all resources that are used by an activity. This can be seen in Algorithm 3. The ESTA algorithm actually does support requirements larger than 1, although it is not stated as such. The Chaining Algorithm, which is only described on a high level, does not support requirements larger than 1. In our updated version, when creating the chains for a resource r_k , each activity a_{ij} is placed in exactly q_{ijk} chains for resource r_k instead of just one.

Furthermore, we have given a detailed description and implementation of the Chaining Algorithm. This was not previously available as Cesta et al. merely give a brief, high-level description. Although the listing in

Algorithm 4 is pseudo-code, we have also given a complete implementation in Appendix D, especially D.1.

Thirdly and finally, we have made it possible for resources to have a capacity that varies throughout time. See the previous section for details on this “extension”.

4.4.2 Optimizations

We have developed the following three optimizations to the original algorithms.

Faster peak collection: In the original ESTA algorithm all peaks are collected in each iteration of the algorithm. We only select the first peak we find. The motivation for this is that every peak *must* be resolved in order to find a valid schedule. We have run some small-scale experiments and they reveal that this change lowers runtime and, or because, it posts less precedence constraints before arriving at a valid schedule than the original ESTA procedure. See also Section 6.3.1.

Smarter peak resolving: In the original ESTA algorithm, conflict selection is done purely on the basis of temporal information. This choice is valid because in MCMSP activities may only require 1 unit of resource capacity. In our problem, however, not all activities are created equal: activities may require more than 1 unit of resource capacity. But then a different peak resolving strategy can be used to resolve peaks faster. This strategy is to move activities with larger requirements out of the way first to decrease peak size faster and reach a final schedule with less precedence constraints posted.

The size of the conflict set is quadratically proportional to size of the set of competing activities K in a peak: there are $\binom{n}{2} = \frac{n(n-1)}{2}$ conflicts for n competing activities. So if we remove some of the smaller activities from K we will quadratically reduce the number of conflicts that are considered for resolving and this should lead to an immediate decrease in runtime. Although we are not entirely familiar with the material, we believe this idea has some correspondence to the idea of Minimal Critical Sets mentioned in [COS00].

We have tested this idea on a small scale and unfortunately it appears that the procedure actually solves much less instances using this idea. Therefore we have not used this “optimization”.

Faster conflict selection heuristic: The calculation of ω_{res} differs from the original Cesta et al. version in the case where all conflicts are type 4. This version is simpler, ignoring a normalization factor. This leads to slightly less calculation and it performs equally well as the original in terms of instances solved.

The first and third optimizations were used, the second one was not adopted.

4.5 Conclusions

In the next chapter, we will experiment with the approaches introduced in this chapter, investigating their ability to generate base schedules and production schedules. In conclusion to this chapter we would like to stipulate an advantage the $ESTA^+$ approach has over the ILP approach from Chapter 3.

The $ESTA^+$ algorithm can be used to *repair* broken plans when a disruption has occurred by updating the problem instance with the new information and propagating that information to the STN. There is no need to construct a new STN. Indeed, the $ESTA^+$ algorithm takes an STN as its input and there is no reason that this cannot be the STN that it itself produced. We believe this to be a major advantage of using an STN based approach because it also opens the door to using local search techniques to improve any found schedule. More on this is written in Section 6.3.

5

Experiments

In the last two chapters we have shown two types of algorithms for generating schedules for train maintenance. At the end of Chapter 3 we have already shown some experimental results and suggested that the ILP approach is well suited to creating base schedules and probably not for creating production schedules. In this chapter we will test the ability of both algorithms to produce base schedules and production schedules under varying conditions. We will finish up with a recommendation as to how the developed techniques can be most effectively used to create schedules for train maintenance at NedTrain.

All experiments with the ESTA^+ algorithm were run on a dual-core Intel Core i3 M330 processor running at 2.13 GHz and 3 GB of RAM memory. The CPLEX experiments were run on a computer with two quad-core Intel Xeon E5345 processors running at 2.33 GHz and 16 GB of RAM memory. Although there is a big difference in the available memory and number of CPU cores, the results can still be compared because memory usage was far below the total available amount and both approaches do not take significant advantage of using multiple cores. The bottleneck in both approaches appears to be the speed of a single processor core. It is possible to speed up the ESTA^+ algorithm by using multiple cores through multi-threading. See Section 6.3 for more information.

5.1 Problem Instances

We have generated instances that look like base schedules and instances that resemble production schedules to the best of our knowledge. Each instance is modeled after the Leidschendam NedTrain location where the set of resources is roughly as follows—the engineers is an estimate that we have deduced from the workload in the Leidschendam base schedule.

- 6 putsporen,
- 2 aardwinden,
- 1 kuilwielbank,

- 12 mechanical engineers and
- 12 electrical engineers.

On any train only one activity can be executed at any time. To accomplish this, trains are regarded as unit-capacity resources with all activities requiring exclusive access.

In general, we have created 10 instances of every type that we mention. If we show figures displaying runtimes or solution quality, the data points represent an averaged value over 10 instances generated with identical parameters but different random seeds unless specified otherwise.

5.1.1 Base Schedules

For testing the ability to create base schedules we have taken the Leidschendam base schedule as an indication of problem size and rules of thumb. We created instances with 25 trains with uniformly and randomly chosen arrival times (release dates) in a one-week period. As in Chapter 3 we have chosen 1 hour as the unit of time. This seems appropriate because it appears to be used at NedTrain as well. All trains will have a set of standard activities consisting of: ‘voorinspectie’, ‘onderhoud’, ‘herstellen’, ‘E-staat’ and ‘reiniging’. Precedence constraints force these to be processed in the order in which they have been listed here. Furthermore, we will add additional activities ‘kuilwielbank’, ‘aardwind’, ‘restwerk’ and ‘A/E’ with a 75% chance. These additional activities will be constrained only to take place *after* the ‘E-staat’ work has been finished but we will enforce no order among the additional activities themselves. Figures 3.2 and 3.3 give impressions of how trains and their activities are laid out in these instances.

5.1.2 Production Schedules

To create production schedules, we will take the base schedules constructed by the experiments for generating base schedules and replace the standard blocks we mentioned above for ‘onderhoud’, ‘herstellen’ and ‘E-staat’ with more detailed lists of activities.

We have used the internal NedTrain document “Beurtinhouid ORM KCO” as an example to help us decide how many actual individual activities are performed in these blocks. As it turns out, there are roughly 120 distinct activities listed for a routine checkup (‘B1’ checkup) of ORM type trains (Figure 1.2(a)). In the Leidschendam base schedule the (V)ORM series 9400 trains spend 8 hours in ‘onderhoud’, ‘herstellen’ and ‘E-staat’. It is our understanding that these 120 activities are performed in this time, coming down to 4 minutes per activity.

So to generate production schedule instances for our algorithms to work on, we take the base schedules from the base schedule experiments and

replace the standard blocks with 120 activities of four minutes each. Each will require the train to be on a ‘putspoor’ and require 1, 2 or 3 engineers, chosen randomly and uniformly.

5.2 Metrics

We use three metrics for determining the quality of a schedule and the algorithm that produced it:

throughput: The throughput of a schedule is measured as the sum of train completion times, that is, the sum of the completion time of the last activity for every train. Note that a lower number is better because of the way we defined the measure, so a higher throughput means a lower measurement. Higher throughput is generally favorable, even in the case of NS and NedTrain where it does not directly improve efficiency because trains will not be picked up earlier than their due date. It can, however, set the stage for the long term where it will become possible to work with earlier due dates if throughput can be structurally decreased.

time to solution: This metric is especially important to determine the usefulness of a method. For production schedules, if it takes longer than 5 minutes to update a schedule, then the solution found will be of no use because a decision is needed faster. In the case of base schedules, more time is available because they only have to be generated roughly twice a year.

robustness: Because of the uncertain nature of the work that we have discussed already in the first chapter, we want our schedules to be able to handle disruptions. We quantify this ability, which we call robustness, by the number of positions in time activities can take with respect to each other. Adapted from [COS98], the robustness $RB(S)$ or a schedule S is defined to be:

$$RB(S) = \frac{\sum_{a_{ij}, a_{vw} \in A} w_{a_{ij} \rightarrow a_{vw}} + w_{a_{vw} \rightarrow a_{ij}}}{|A|^2}.$$

Note that the robustness is calculated on an STN representation of a schedule. In the case of the ILP method, which does not work with STNs, we will use the Chaining Algorithm from Section 4.2.1 to construct an STN from the fixed-time schedule that was calculated.

5.3 Results

5.3.1 Base Schedules

At the end of the chapter on the ILP method we have already shown some experiments. Here we will see if the results can be repeated over a multitude of randomly generated instances, as described in the Section 5.1. Furthermore, we will investigate how well the ESTA^+ method measures up to the ILP approach in terms of runtime and throughput.

For our first test we added precedence constraints like we described earlier to resemble what we think is the standard way of working at NedTrain. The results for 10 testsets averaged are shown in Table 5.1.

| method | runtime | throughput | approximation factor |
|-----------------|--------------|-------------|-------------------------|
| ILP | 50.6 seconds | 616.7 hours | 1 |
| ESTA^+ | 3.0 seconds | 619.7 hours | 1.005 |

Table 5.1: Solution time and throughput for base schedules with precedence constraints.

Both approaches run fast and the ESTA^+ algorithm is able to produce high quality solutions that are almost identical to the optimal solution with only a 3 hour difference in total.

For our second test, we leave out the precedence constraints giving the algorithms complete freedom in choosing the ordering of the activities. The ILP version of these tests have already been included at the end of Chapter 3. Figure 5.1 shows the same plot for the ESTA^+ method on the same testset.

The figure looks quite similar to the chart for the ILP method in Chapter 3. The difference is in the scale on the time axis which shows that the ESTA^+ method takes only a fraction of the time the ILP method uses.

To conclude this section we have plotted the throughput that the two methods achieved on the data sets. Figure 5.2 shows the plot for both methods. The approximation ratio (ratio between throughput of ESTA^+ and throughput of ILP) is roughly between 1.4 and 1.8. This tells us that the ESTA^+ method by itself does not perform very well at finding optimal schedules.

5.3.2 Production Schedules

In this section we will show experiments that are designed to find out if the ESTA^+ method can be effectively used to create and update production schedules, given a base schedule.

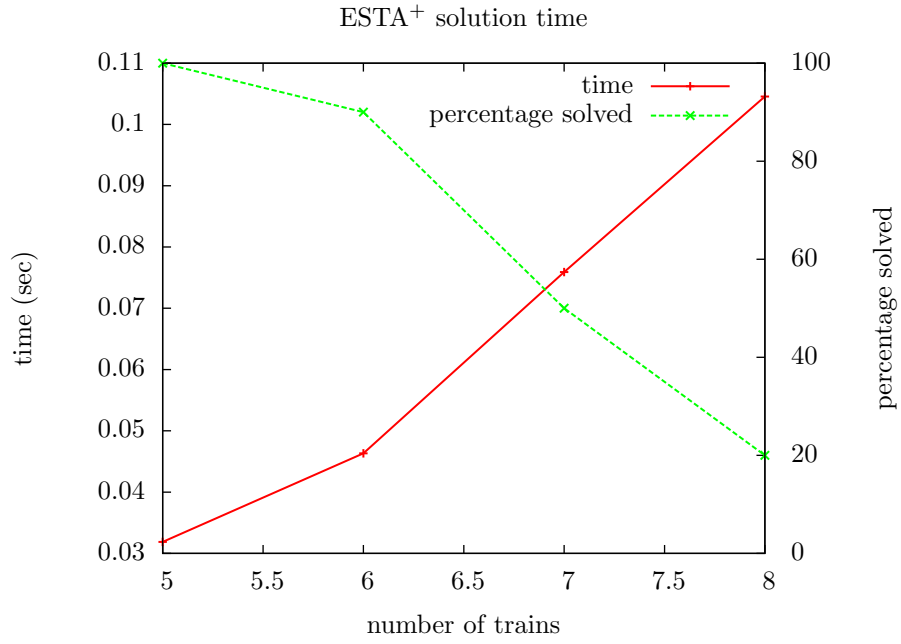


Figure 5.1: Solution time and success rate for different problem sizes using ESTA⁺ method.

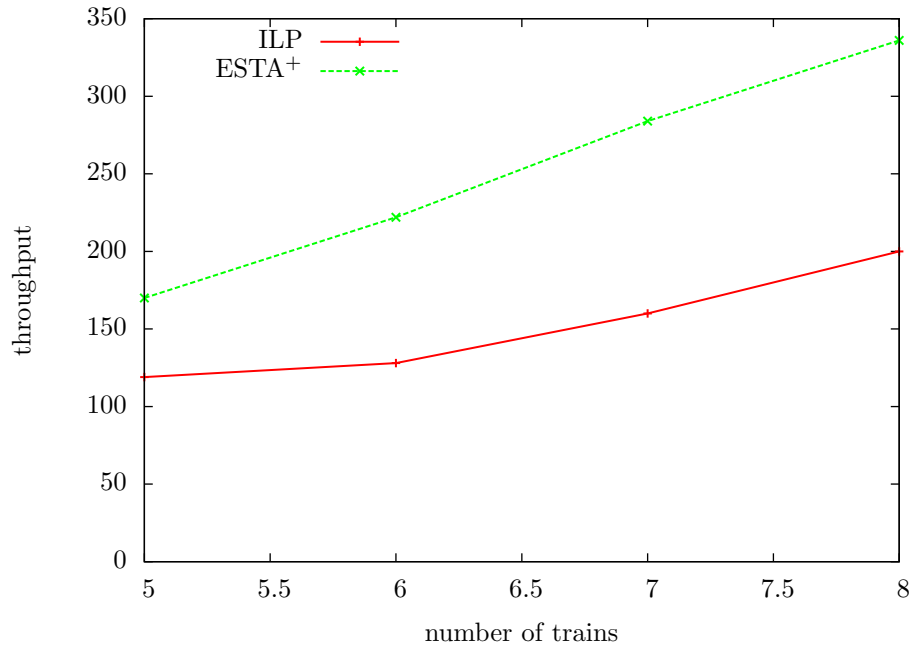


Figure 5.2: Solution throughput for different problem sizes.

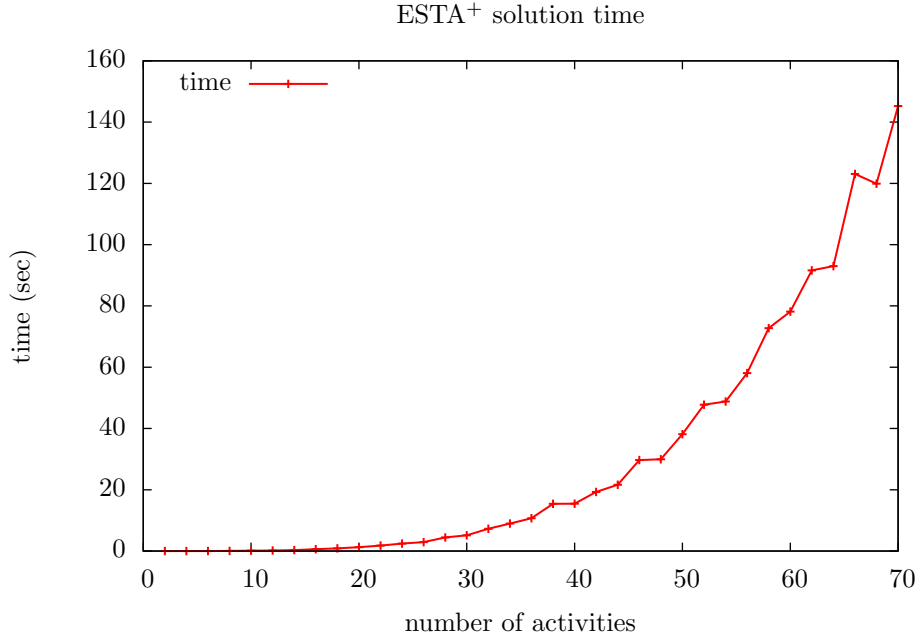


Figure 5.3: ESTA^+ runtime for increasing problem sizes.

We have not tested the ILP method for the simple reason that the input file size explodes when the time unit is decreased from one hour to a more detailed level like 10 minutes. We have tried to generate a few sets for the ILP method but only generating the input files already takes more than ten minutes and the file size exceeds 400 megabytes which is impractical to say the least and certainly useless for production schedules where we need a quick answer.

Figure 5.3 shows the runtime results of an additional testset we created to search for the limits of the ESTA^+ method in terms of instance size. We created instances with 4 trains and increasing number of activities per train with no precedence constraints added. The exponential nature of the problem, being NP-complete, is clearly visible. Once we reach the end of the graph we are at $4 * 70 = 280$ activities and the runtime has reached 2 min. We have stated that 5 minutes should be the time limit within which a result is needed. Extrapolating, it appears the limit of this method is somewhere around $4 * 75 = 300$ activities. Remember that in this case the algorithm is searching for satisfying schedules from the collection of roughly $400! \approx 6.4 * 10^{868}$ possible solutions. In other words, the search space is incredibly large.

NedTrain has descriptions of what maintenance must be performed on specific train types and these lists include more than 100 distinct actions for a single train. If we were to try to schedule at that level of detail, then

we would have to schedule 2500 activities for 25 trains per week. If we extrapolate Figure 5.3 it suggests that the ESTA^+ method is not capable of taking on problem instances of that size. We must then conclude that the approaches we have developed are both not suitable to do extremely detailed scheduling over the scope of an entire week.

The question then becomes, *what can we do with the developed approaches?* We think that it is sensible either to only do high-detail production schedules for a single day at a time or to stay on the level of the base schedules. The ESTA^+ algorithm can then take the base schedule produced by the ILP approach and be subsequently used to process changes and update the schedule.

5.3.3 Repairing Broken Production Schedules

To test the ability of the ESTA^+ method for the purpose of repairing broken production schedules, we will start with a valid production schedule, then introduce a disruption and observe the changes. The valid schedules that we start with shall be the base schedules constructed by the ILP method in the experiments we discussed earlier.

The following disruptions will be introduced:

- an activity is delayed,
- a resource is temporarily unavailable and
- an extra train is added at the last moment.

Now all we need is a way to measure the change to a schedule. We will measure the throughput before and after the change as an indication of the amount of change that occurred. This will tell us how much extra processing time was needed because of the change.

In the first experiment, we will delay a random activity for one hour and measure the average total delay incurred, that is, the increase in schedule throughput as we just discussed. Remember that we use the 10 instances from the section on base schedules above as starting points. We add precedence constraints between all the blocks in the base schedule to force them to take place in the order that was calculated by the ILP method. Table 5.2 shows the results for 1, 2 and 3 hours of delays introduced.

We notice very nice behaviour indeed. Because we are using an STN internally and all activities are linked by precedence constraints, the algorithm can handle small delays with almost no significant change of schedule at all. There are at most 9 trains simultaneously being worked on in the instances we use because there are not enough track resources to handle more trains. So disturbances in trains only propagate to so many other trains. In

| no. of 1-hour delays introduced | hours of delay incurred |
|---------------------------------|-------------------------|
| 1 | 2.1 |
| 2 | 2.3 |
| 3 | 3.5 |

Table 5.2: Effect of delays.

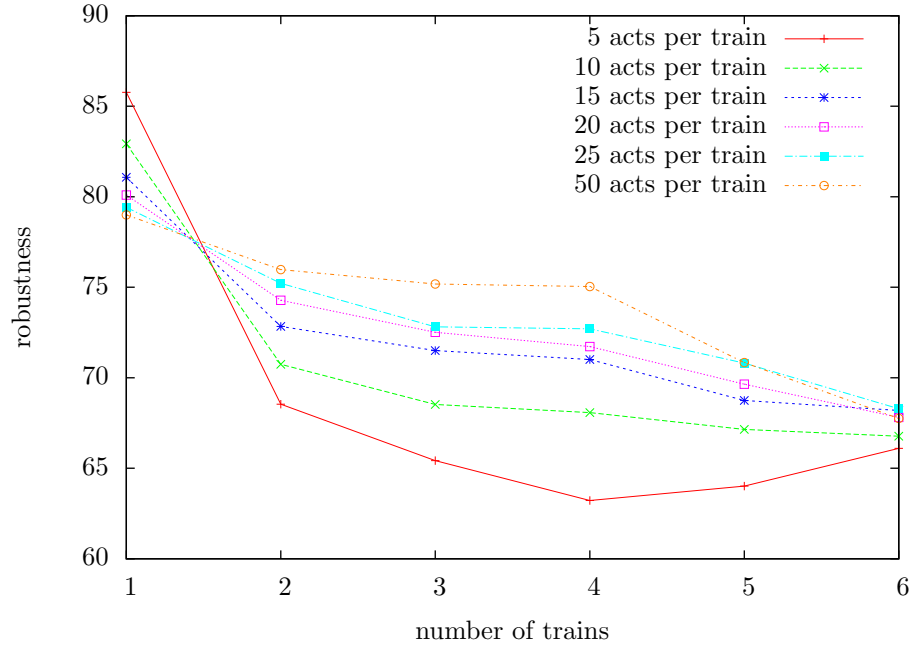


Figure 5.4: Robustness ESTA⁺ schedules for different problem sizes.

practice there is enough space in the base schedules to handle delays. We suspect that people are the real bottleneck resource at NedTrain and that not all tracks are always in use.

5.4 Conclusion

The ILP approach can handle the problem size of base schedules. This is good news, because it means we can generate optimal base schedules. For generating and maintaining production schedules the STN-based heuristic approach does well under specific circumstances. These circumstances are that it is necessary to help the heuristic by giving it a pretty good schedule to begin with and it should not be too fine-grained because then it start to take too long. It is important to strike a balance here. The ESTA⁺ algorithm is well equipped to deal with small changes but does not really

generate good schedules when starting from a blank slate. Fortunately, we do not have to start from a blank slate because we have the optimal base schedules to start from. Because of these base schedules much of the work is already sequenced and this makes it easier for the heuristic to update the schedule when a change occurs. The use of an STN to keep track of the temporal information makes this especially easy for the heuristic. The ILP method itself is not usable for this purpose because it has the potential of generating completely different solutions when a small change has occurred. This is in itself quite disruptive and should be avoided.

Given the results obtained here, we suggest generating base schedules using the exact ILP method and then using these as blueprints for the STNs representing the production schedules. The STNs and the ESTA^+ heuristic provide a quick and effective way to update the production schedules when changes occur without completely mixing up the schedule.

6

Conclusions and Future Work

This chapter gives an overview of the project’s contributions. After this overview, we will reflect on the results and draw some conclusions. Finally, some ideas for future work will be discussed.

6.1 Contributions

In this section we describe our contributions.

We have analyzed the maintenance scheduling process at NedTrain and realized that the problem of creating maintenance schedules can be modeled using linear inequalities. The problem consists of temporal and resource constraints. We have shown that the temporal constraints can be solved with the Simple Temporal Problem. However, when we take resource constraints into account, the Simple Temporal Problem is no longer sufficient. We then revert to the Disjunctive Temporal Problem. Unfortunately the DTP is not efficiently solvable. A complexity analysis, which we have included in the appendix, shows that the our problem is NP-complete, just like the DTP.

We have tried two solution methods, one is a zero/one integer linear programming approach that we found in the literature that could be directly applied to the scheduling problem at NedTrain. The ILP method is exact and is used to generate optimal base schedules. The second method is an incomplete heuristic approach. Although the STP itself cannot model the resource constraints we are faced with, we have shown that a resource constraint simply imposes ordering constraints. That is, if two activities a_{ij} and a_{vw} each want to use a resource with capacity 1, then either a_{ij} *before* a_{vw} or vice versa. These precedence relations *can* be modeled with the STP. Now all we need is a way to make ordering decisions among activities that are competing for the same resource. We have found a heuristic for this purpose in the literature and applied it.

However, the heuristic was not directly applicable to our problem. We had to generalize it so that it could work with resource requirements of more than 1 unit and with multiple resource requirements per activity, e.g. ‘need 3 engineers and a putspoor to complete task a ’, and we have relieved

the restriction that all activities within a train or job need to be totally ordered before the algorithm starts. This allows more freedom in creating schedules but also makes it more complex because there are much more possible schedules. We have also shown, thanks to a helpful suggestion by Michel Wilson, a way to handle resources with a capacity that is not constant but instead varies with time. Although the way we do this, by blocking excess or unavailable capacity with a dummy train and activity can be considered a ‘hack’, it is easy, requires no change to the algorithm and is effective. These contributions have already been listed in detail in Section 4.4.

Finally, we have created a visualization tool to review the schedules that the algorithms give as output. This has helped in understanding what the algorithms are good at and has also helped greatly during the occasional but almost inevitable debugging of the code.

6.2 Conclusions

The research questions we asked at the beginning of this work were as follows:

- *How can valid one-week-long base schedules be generated and can optimal schedules be generated?*
- *How can a base schedule be transformed into a production schedule?*
- *How can we update an existing production schedule with minimal change when faced with disruptions?*

We will now try to answer these questions based on the contents of this thesis.

First of all, *how can we generate base schedules?* and *can we generate optimal base schedules?* We have given a formalization of this problem in Section 2.1 and in Chapters 3 and 4 we have given two solution methods for creating train maintenance schedules, including base schedules.

Experiments have shown that the ILP method is well suited to the creation of one-week-long base schedules. It runs in a matter of minutes which is more than fast enough for the purpose because base schedules need only be generated a few times each year. The schedules it creates are optimal with respect to the goal function used, which in our case has been throughput.

The heuristic method from Chapter 4 has been shown not to be well suited to the creation of base schedules. It can generate schedules that approximate the optimum well but only if many precedence constraints are

already posted. Although it runs fast, it is not capable of producing high-quality schedules when it has to start from scratch.

Secondly, *how can a base schedule be transformed into a production schedule?* Our analysis of the maintenance process at NedTrain shows that the step from base schedule to production schedule increases the level of detail in the schedule. When the identities of the trains coming in for maintenance become known, the detailed production schedule can be constructed. There may, however, already be changes relative to the originally constructed base schedule. The recommended approach is to use the ILP method to generate a base schedule, then run the Chaining Algorithm to generate an appropriate STN and then use this STN as basis for the $ESTA^+$ method. At this point, the $ESTA^+$ method can be used to handle the changes. This brings us to the final research question.

Thirdly and finally, *how can we update an existing production schedule with minimal change when faced with disruptions?* Depending on how detailed the schedule needs to be, there can be a lot of activities in the production schedule. For the purpose of repairing broken schedules, the ILP method has two disadvantages. First of all, it is too slow, especially when the level of detail is increased. Secondly, it will always search for optimality and does not care about leaving the current schedule intact as much as possible. Even when the current schedule is used to initialize the ILP variables, it may still generate a completely different schedule.

This is where the heuristic method we developed shines. Although it does not perform well at the generation of new schedules from scratch, it is good at repairing existing schedules. An existing schedule is represented by an STN. When a disruption occurs, the information in the STN is updated. This may or may not lead to new conflicts being introduced which can then be dealt with simply by rerunning the $ESTA^+$ algorithm. Only this time, it will probably not have a lot of work to do because it started out with a nearly complete schedule. All it has to do is iron out the newly created cracks which usually is not hard. Because the Chaining Algorithm imposes a partial order among activities and this order is enforced through the STN, there will mostly only be propagating delays but no significant reordering of activities. This means that the schedule will mostly stay the same, thus not causing chaos in the workplace.

The $ESTA^+$ algorithm runs fast, never taking more than a few minutes for realistic problem sizes. Where the ILP approach quickly becomes useless with more activities added, the $ESTA^+$ approach scales quite nicely and can handle larger problem sizes.

6.3 Future work

In this section we give a list, in no particular order, of ideas for extending the current work that we collected during this project.

6.3.1 Smarter Heuristics

The heuristics used in the ESTA and ESTA⁺ algorithm are rather simple. This is part of the reason that the algorithm runs so fast. We believe that it is possible to improve the effectiveness of the ESTA⁺ algorithm by devising a slightly more sophisticated conflict selection heuristic. Of course a balance must be struck between sophistication and runtime.

6.3.2 Exploit Interchangeability of Trains

The trains in use by NS Group only come in a few dozen different configurations. For instance, most trains are one, two or three sets of three or four carriages in length. So, when two of the same trains are present in a maintenance depot and the one with the earliest deadline faces large delays in maintenance, it is sometimes possible to swap the two trains and instead deliver the second train first, thus making the first deadline and creating more time to deal with the problems on the first train.

6.3.3 Variable Resource Allocation

In our algorithms, activities require the same amount of resources throughout their duration. In reality, there is a minimum and maximum number of engineers that can efficiently work together on an activity and this number may be varied during an activity. Compare this idea to, for instance, mending a flat tire on a bike. Maybe if we would assign two people to this task it could be finished earlier than when we were to let one person do it on their own. However, when we add more than, say, 3 people, the extra investment does not pay off because there is no way to efficiently work together on the mending of one tire with more than 2 or 3 people.

When a foreman notices that some train is running behind on schedule, he might then assign extra men to the tasks of the train, dynamically influencing the durations of the activities, both on the train the men were working on and the train that was lagging in the first place. This is an additional and perhaps very difficult optimization layer but it could potentially help in keeping a schedule intact even in the face of slight delays.

6.3.4 Explicitly Assigning Resource Slots

Our approaches guarantee that all required resources are available for all activities, at least in a valid schedule. When an activity is listed as requiring

a specific type of wrench of which a total of 4 are available, then it does not matter which wrench is used. However, in the case of ‘putsporen’, for example, or engineers, of which multiple were available in our examples, at some point there will have to be an explicit assignment of an activity to one of the available ‘putsporen’ and in the same way exactly which engineers should work on which task should be decided at some point before work commences.

The Chaining Algorithm that we run guarantees that a specialized track of the required type is available by adding precedence constraints among activities. Each one of the chains created corresponds to a specific ‘putspoor’, if we stick to the examples of ‘putsporen’. The Chaining Algorithm in fact implicitly assigns activities to resource slots. This goes for any resource with a capacity greater than one.

In fact, our restriction that only one activity can take place on any one train at the same time, stems from the fact that our approaches are blind to this aspect. If we do not enforce a complete sequencing of all activities within a train, then it could very well happen that the algorithm tells us to work on replacing a bogie on the ‘aardwind’ while doing other work on a ‘putspoor’ on the same train. Of course this is physically impossible to the best of our knowledge.

An approach that takes the different nature of the resources into account and keeps track of and is aware of a train’s physical location could support scheduling of simultaneous work on one train leading to higher throughput.

6.3.5 Local Search Techniques

The $ESTA^+$ algorithm searches for valid algorithms for train maintenance. Although it tries to find schedules with maximal temporal flexibility or high robustness, it is not aimed at finding optimal schedules with respect to throughput. Cesta, Oddi and Smith have suggested a local search technique called Iterative Flattening [COS00] which tries to improve the schedule by examining neighbor solutions. It constructs a directed graph with all activities as nodes and the edges represent precedence constraints between the activities. The method of finding neighboring solutions is then to cut away some of the edges in this graph, which leads to the solution shrinking somewhat and causes some activities to now overlap. It is likely that this new solution is not feasible, new conflicts may have arisen. This new situation is then again used as input to the $ESTA^+$ algorithm. Because there are only a few conflicts, dependent on how many edges are removed, the algorithm should finish quickly, thus generating neighboring instances at little cost.

When a simple neighbor generation procedure has been established, we can break out all the nifty local search techniques that have been developed over the years such as tabu-lists to avoid redoing the same work and random restarts or working on more than one candidate solution in parallel.

Multi-threaded ESTA⁺ Algorithm

In the ESTA⁺ algorithm there are several phases that can be parallelized in order to speed up the algorithm, for example the **FindPeak** method and the code for classifying the conflicts into the four types as described in Section 4.1.3. These phases, which are each implemented using one or more **for**-loops, are currently run sequentially in a single thread. **For**-loops are generally easy to implement in a multi-threaded fashion. Since the code is written in C, one could use the OpenMP [OMP] library to do this. When the code is multi-threaded, this means that the work can be distributed over multiple CPU cores. Since most modern CPU's have multiple CPU cores and thus the ability to execute two or more threads in parallel, this approach may greatly speed up the ESTA⁺ procedure. It will then become possible to schedule with greater detail in the same amount of time as before.

6.3.6 Using ESTA⁺ to initialize ILP model

The previous point showed ways to improve ESTA⁺ solutions by using local search. Another way to go once a valid schedule has been found using ESTA⁺ is to use this schedule to initialize the variables of the ILP model. This might lead to much improved converging times in CPLEX and will perhaps make the generation of more detailed schedules possible in shorter time frames than currently.

6.3.7 ILP Optimizations

We have not taken account of precedence constraints when calculating the values of the u_{ij} and l_{ij} variables. These variables denote the upper and lower bound for the execution interval of activity a_{ij} . We have set their value to the due date of the corresponding train and the release date of the corresponding train respectively. However, when the problem precedence constraint are taken into account, it is possible to make these bounds more tight.

Interestingly, an easy way to calculate the actual lower bound l_{ij} for activity start and the upper bound u_{ij} for activity end is to use an STN. We simply initialize an STN with the same set of timepoints and constraints as we do for the ESTA⁺ algorithm, also adding the precedence constraints. The STN now contains all temporal information needed to deduce the bounds. They are as follows: $l_{ij} = -w_{s_{ij} \rightarrow z}$ and $u_{ij} = w_{z \rightarrow e_{ij}}$ where $w_{a \rightarrow b}$ denotes the weight on the edge from node a to node b in the STN.

This optimization would lead to a smaller problem definition, less terms in many constraints and hopefully in the end to a lower runtime.

6.3.8 The Human Touch

We can automate a lot of things, but in the end, there are real people at work in many of our factories, including NedTrain depots. The approaches presented in this work abstract away from this fact by considering engineers, and even trains, simply as a ‘resource’. This rather detached look on the situation may lead to undesirable schedules.

Take for instance a coffee-break. A break can be modeled easily by temporarily lowering the capacity of the engineer resource. The algorithms however, schedule activities in contiguous blocks. This means that activities will only be scheduled entirely before or entirely after coffee breaks whereas the best solution may be to start an hour before the break, then take the break and finish one hour after the break.

One simple solution to this problem is to implicitly model the situation by increasing activity duration for all activities that overlap with lunch hours et cetera.

Another example is the shunting of trains. We have already slightly touched upon this in Section 6.3.4. The algorithms do not realize that it takes time to drive a train from one workstation to another. Again there is a simple solution: impose a partial order among activities within a train using precedence constraints. The goal here is to cluster activities that require the train to be on the same type of track. In between the clusters a shunting activity can or should be added that requires (exclusive?) access to the shunting yard resource. This makes sure that there is always time to shunt the train but it does limit the number of solutions possible since more choices must be committed to before the algorithms can run.

Bibliography

- [BHvMW09] Armin Biere, Marijn J.H. Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, February 2009.
- [COS98] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Profile-based algorithms to solve multiple capacitated metric scheduling problems. In Reid Simmons, Manuela Veloso, and Stephen Smith, editors, *Proceedings of the Fourth International Conference on Artificial Intelligence Planning Systems*, pages 214–223, Menlo Park, CA, USA, 1998. The AAAI Press.
- [COS00] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. Iterative flattening: A scalable method for solving multi-capacity scheduling problems. In *Proceedings AAAI-00*, pages 742–747, Austin, TX, USA, 2000. AAAI Press / MIT Press.
- [DMP89] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. In Ronald J. Brachman, Hector J. Levesque, and Raymond Reiter, editors, *Proceedings of the First International Conference on Principles of Knowledge Representation and Reasoning*, pages 83–93, San Francisco, CA, USA, 1989. Morgan Kaufmann Publishers Inc.
- [Flo62] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, 1962.
- [Heu08] Marijn J.H. Heule. *SmArT solving: Tools and techniques for satisfiability solvers*. PhD thesis, TU Delft, 2008.
- [K307] Team K3. K3 strategieprogramma doorlooptijdverkorting. Technical report, NedTrain B.V., November 2007.
- [Kar72] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller and J. W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103, New York, NY, USA, 1972. Plenum Press.
- [Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. *Combinatorica*, 4:373–395, 1984.
- [KT06] Jon Kleinberg and Eva Tardos. *Algorithm Design*. Addison-Wesley, Boston, MA, USA, 2006.
- [OMP] The openmp api specification for parallel programming. <http://openmp.org>.

BIBLIOGRAPHY

- [Pla08a] Leon R. Planken. Incrementally solving the stp by enforcing partial path consistency. In Ruth Aylett and Ivan Petillot, editors, *Proceedings of the 27th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 87–94, Edinburgh, Scotland, December 2008.
- [Pla08b] Leon R. Planken. New algorithms for the simple temporal problem. Master’s thesis, Delft University of Technology, January 2008.
- [Pol05] Nicola Policella. Scheduling with uncertainty: A proactive approach using partial order schedules. *AI Communications*, 18(2):165–167, 2005.
- [PWW69] A. Alan B. Pritsker, Lawrence J. Watters, and Philip M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, September 1969.
- [SK00] Kostas Stergiou and Manolis Koubarakis. Backtracking algorithms for disjunctions of temporal constraints. *Journal of Artificial Intelligence*, 120:81–117, June 2000.
- [Tij04] Henk Tijms. *Operationele analyse*. Epsilon Uitgaven, Utrecht, The Netherlands, 2004.
- [War62] Stephen Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.
- [XC03] Lin Xu and Berthe Y. Choueiry. A new efficient algorithm for solving the simple temporal problem. *International Symposium on Temporal Representation and Reasoning*, pages 210–220, 2003.



Complexity Analysis

The problem of scheduling train maintenance can, although very tediously, be modeled as a DTP or using Integer Linear Programming. We suspect that our scheduling problem is NP-complete, like the DTP or the ILP. In this appendix we present an analysis of the time complexity of the problem of scheduling train maintenance as laid out in the mathematical model of Section 2.1.3. We will also explain the terms like NP-complete that we have already used a couple of times without explaining them.

In complexity theory, problems are classified into complexity classes. The complexity class of a problem is a measure of how difficult it is, expressed in the size of its input. In general it is said that problems that take an amount of time that is polynomially bounded in the size of the input—complexity class P—are efficiently solvable and problems that take an exponential amount of time in the size of the input—complexity class NP—are not. Another characterization of problems in NP is that they are problems for which a solution can be verified in polynomial time. We will use this fact later on in this section.

The hardest problems in any complexity class are called the *complete* problems for that class. Any problem in a complexity class can always be translated or *reduced* into a complete problem in the same class. By using reductions to complete problems, solution methods for complete problems can be used to solve all other problems in the same class. Because of this fact, the development of solvers for some popular complete problems attracts a lot of research attention and consequently these solvers have become quite powerful. Examples are solvers for the ILP problem or the boolean satisfiability problem (SAT), which is an NP-complete problem. See [BHvMW09] for an extensive treatment or [Heu08] for a more compact overview of SAT and SAT-solvers.

It is important to know the complexity class of a problem when designing a solver for it because one can choose between *complete* and *incomplete* solvers. A *complete solver* is a solver that considers all possible solutions to a problem, often using a branching and backtracking algorithm. A complete solver will always find an optimal solution (provided it exists) but it may

take a very long time to do so, especially if the problem is not in P . An *incomplete solver* does not consider all possible solutions to a problem but instead searches for acceptable solutions often using heuristics to guide itself through the solution space.

Given the above discussion on complexity theory and complete problems, we can now make a choice when taking on a new problem, for instance the problem of the scheduling of train maintenance. We can either reduce the problem to a well-known complete problem for which solvers are readily available, or we can create our own solver which can potentially exploit properties specific to our problem domain. Either way, we need to know the complexity of our problem to know what kind of worst-case performance we can expect.

Proposition 1. *The TMSP is NP-complete.*

The proof consists of two parts. In the first step we will prove that the problem of creating schedules for train maintenance is in NP by showing that a solution to the problem can be verified in polynomial time. In the second step we will show a reduction to a known NP -complete problem, proving NP -completeness of the problem of scheduling train maintenance. In the rest of this section we will abbreviate the problem of scheduling train maintenance as the TMSP: the Train Maintenance Scheduling Problem.

Proof. To verify a schedule we need to check that all activities are planned in between their train's release and due dates, that no activities overlap for every individual train, that no resource constraints are violated and that all precedence constraints are satisfied. All of these properties can be checked in polynomial time. Therefore the TMSP is in NP .

From [KT06, p. 493] it follows that scheduling with release dates and deadlines on one resource (SRDD) is NP -complete by a reduction from the subset-sum problem. We will use this result to prove that the TMSP is also NP -complete. SRDD is the problem of scheduling n activities with release dates and deadlines on one machine. The machine can handle only one activity at the same time and the objective is to minimize makespan.

Given an instance $I = \langle J = \{j_1, \dots, j_n\} \rangle$ of the SRDD, with J the set of activities, construct an instance of the TMSP as follows, using the notation we developed in Section 2.1.2.

- Set $R = \{r_1\}$ with capacity $c_{1t} = 1$ for every time period t ,
- for $i = \{1, \dots, n\}$
 - add a train t_i to T ,
 - set rd_i to the release date of j_i ,

-
- set dd_i to the due date of j_i ,
 - set $A_i = \{a_{i1}\}$,
 - set d_{i1} to the duration of j_i and
 - set $q_{i11} = 1$.

By the recipe above we can translate any instance of the SRDD into an instance of the TMSP. Because the SRDD is NP-complete and we can translate any SRDD instance into a TMSP instance, the TMSP has a worst-case complexity of NP-hard, which means NP-complete or harder. The “or harder” part follows from the fact that we can model a lot more with the TMSP than with the SRDD; the SRDD instances only use one resource while the TMSP allows multiple resources for example. These multi-resource instances may be harder to solve.

We have shown that the TMSP is in NP and is NP-hard. By definition, these two facts combined mean that the TMSP is indeed NP-complete. ■

Remark

All the complexity results given above are valid only for the decision problem variants of the problems given. A decision problem is a problem that must be answered by a ‘yes’ or a ‘no’. Because asking for a train schedule is more than asking for a simple ‘yes’ or ‘no’, the TMSP and the other problems do not actually fall into the NP-complete complexity class themselves. Searching for the fastest schedule is in fact an optimization problem. Luckily, a decision variant of an optimization problem is easily constructed simply by not asking what the optimal solution is, but by asking if a solution exists that is better than some bound. An example is to ask for a schedule that finishes before some time t .



Leidschendam Base Schedule

Please see the fold-out on the next page for the base schedule that is currently in use at the Leidschendam NedTrain depot. This schedule is a hand-crafted Excel file, made by schedulers in Leidschendam.



CPLEX is a powerful piece of software designed to solve optimization problems. Among others, it can solve integer linear programming problems, which is what we need. In this section we will briefly show exactly how we get CPLEX to solve the zero/one integer linear programming optimization problems we have specified earlier in this chapter. First we will show how CPLEX expects us to format our problems and then we will show how we let CPLEX solve them.

C.1 CPLEX Problem Definition File Format

The problem specification format is very natural in that it looks a lot like you would normally write down a linear programming problem. An example ILP is given in Figure C.1. This example shows all features we need to use.

A problem always starts with the declaration of the goal function: minimize or maximize some linear function over the problem variables. Next are the constraints in the *subject to* section, followed by the bounds. The defaults bounds, if no other values are specified are $0 \leq x \leq \infty$, for every variable x . Finally, all variables that are to be constrained to the set of integers should be listed in the *generals* section and variables that are to be further constrained to only take the values 0 or 1 should be listed in the *binaries* section. If a variables is not listed in the *generals* or *binaries* sections, then it is considered a normal real-valued variable. Every problem specification should end with the word *end*.

Note that the right-hand side of the constraints in the *subject to* section must be a constant. We must therefore rewrite equation 3.6 as

$$N_i x_{it} - \sum_{j=1}^{N_i} \sum_{q=l_{ij}}^{t-1} x_{ijq} \leq 0. \quad (\text{C.1})$$

```
maximize
    2x1 + 3x2
subject to
    2x1 + x2 <= 10
bounds
    2 < x1 < 5
    x2 < 10
generals
    x1 x2
binaries
    x2
end
```

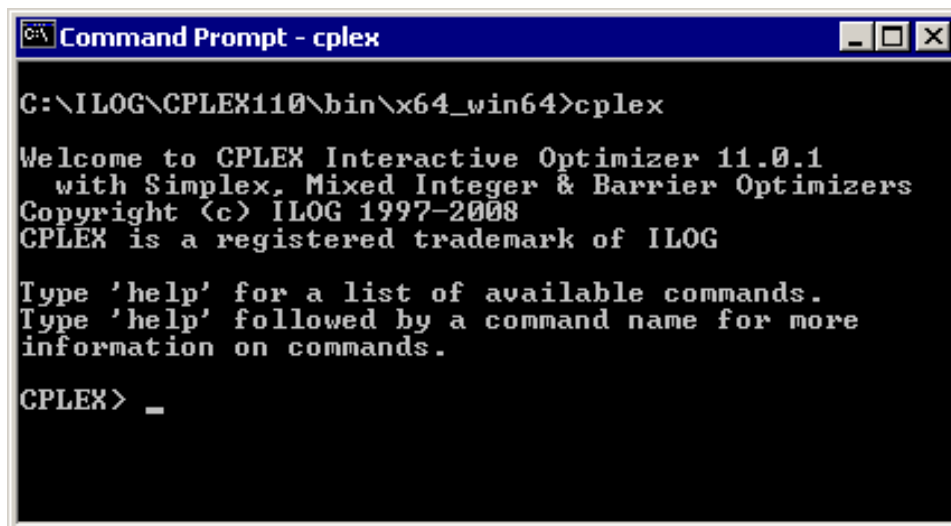
Figure C.1: Example integer linear program in CPLEX format

C.2 Interfacing with CPLEX

Although it is possible to call CPLEX solvers from within other programming languages such as C, C++ and Java¹ we will simply use the interactive optimizer. The interactive optimizer is nothing more than a black box with a prompt as shown in Figure C.2. We can key in commands, telling the solver what to do. This gets tedious really quickly and luckily the interactive optimizer also has the option of reading problem specifications from files.

To run many experiments, which we will do in the next chapter, we prepare files with problem specifications as described above and then write one other small file that contains a script telling CPLEX to read a problem from file, display some information, solve it, display some more information and exit. We can automate this process in order to run many experiments back-to-back by using input and output redirection by calling `plex < problem_file > output_file` from a script.

¹Please see the CPLEX Documentation for details. The documentation is distributed with the software.

A screenshot of a Windows Command Prompt window titled "Command Prompt - cplex". The window has a blue title bar with standard minimize, maximize, and close buttons. The command prompt shows the following text:

```
C:\ILOG\CPLEX110\bin\x64_win64>cplex  
Welcome to CPLEX Interactive Optimizer 11.0.1  
  with Simplex, Mixed Integer & Barrier Optimizers  
Copyright (c) ILOG 1997-2008  
CPLEX is a registered trademark of ILOG  
  
Type 'help' for a list of available commands.  
Type 'help' followed by a command name for more  
information on commands.  
  
CPLEX> _
```

Figure C.2: CPLEX Interactive Optimizer

D

Code listing

This appendix lists all the code that was developed by us to implement the ESTA⁺ algorithm. The following table explains what each file contains.

| Filename | Contents |
|-----------------|---|
| chaining.c | Chaining Algorithm |
| debug.c | Support functions for debugging |
| esta_plus.c | ESTA ⁺ algorithm |
| floydwarshall.c | Floyd-Warshall algorithm |
| grammar.y | Bison input file grammar definition |
| ifpc.c | Incremental Full-Path Consistency algorithm |
| list.c | General purpose 'list' type |
| main.c | Application entry point |
| s2h.c | HTML schedule generator |
| salloc.c | Support files for checking memory problems |
| stn.c | Functions for creating and updating STNs |
| timing.c | Functions for time measurements |
| tmisp.c | Functions for creating and updating TMSP structures for holding the problem definition |
| token.c | Support file for the Bison grammar |

D.1 chaining.c

```
1  #include <stdio.h>
   #include <string.h>

   #include "tmisp.h"
   #include "stn.h"
6  #include "list.h"
   #include "floydwarshall.h"
   #include "debug.h"

11 int leveling_constraints_after_chaining;
```

```
/* returns list of requirements, sorted by EST of the
   corresponding activity */
#define _R(i) ((requirement*)list_get(l, i))
#define _V(k) (EST(_R(k)->i, _R(k)->j))
16 void sort_requirements(int k) {
    int i, j;
    List* l = R(k)->requirements;

    for (i = 1; i < l->size; i++) {
21 // find place j for element i in sorted part 0 through i-1
        for (j = 0; j < i; j++) {
            if (_V(j) > _V(i))
                break;
        }
26 if (i == j) continue;
        // save element i
        requirement* req_i = _R(i);
        // shift j through i - 1 right once
        memmove(l->data + j + 1, l->data + j, (i - j) * sizeof(void
31 *));
        // insert element i at index j
        list_set(l, j, req_i);
    }
}
#undef _R
36 #undef _V

/* returns true if chaining succeeded, which it should if the
   stn
   * is consistent and the EST assignment valid, returns false
   otherwise
   */
41 int chaining_algorithm() {
    int i, j, k, r;

    if (!stn_consistent()) {
        debug("STN_inconsistent_at_start_of_chaining_algorithm.\n");
46 return 0;
    }

    List* all_chains = new_list();

51 for (k = 0; k < tmsp->n_resources; k++) {
    List* resource_chains = new_list();

    sort_requirements(k);

56 for (r = 0; r < R(k)->requirements->size; r++) {
        requirement* req = (requirement*) list_get(R(k)->
            requirements, r);

        // find 'demand' number of chains to place this activity
        in
```

```

61     for (i = 0; i < req->amount; i++) {
        if (resource_chains->size < C(k)) {
            // we can allocate this activity to a new chain
            List* chain = new_list();
            list_append(resource_chains, chain);
            list_append(all_chains, chain);
66         list_append(chain, req);
        } else {
            // find a chain that this activity has no overlap
            // with when considering earliest start times
            int chained = 0;
71         for (j = 0; j < resource_chains->size; j++) {
            List* chain = (List*) list_get(resource_chains, j);
            requirement* last_req = (requirement*) list_get(
                chain, chain->size - 1);

            if (EET(last_req->i, last_req->j) <= EST(req->i, req
76                ->j)) {
                list_append(chain, req);
                chained = 1;
                break;
            }
        }
        if (!chained) {
81         debug("ERROR! Could not chain activity. This means
            there is a conflict. That should not happen at
            this point!\n");
            return 0;
        }
    }
86 }
}
delete_list(resource_chains);
}

91 // we have create all chains, now reinitialize the STN and add
    all the chains
    stn_construct();
    leveling_constraints_after_chaining = 0;
    for (i = 0; i < all_chains->size; i++) {
        List* chain = (List*) list_get(all_chains, i);
96         for (j = 1; j < chain->size; j++) {
            requirement* r1 = (requirement*) list_get(chain, j - 1);
            requirement* r2 = (requirement*) list_get(chain, j);
            set_weight(AS(r2->i, r2->j), AE(r1->i, r1->j), 0);
            leveling_constraints_after_chaining++;
101        }
        delete_list(chain);
    }
    delete_list(all_chains);

106 if (!floyd_warshall()) {
    fprintf(stderr, "STN_INconsistent_after_chaining.\n");
    return 0;
}

```

```
    }  
111    if (!is_est_valid()) {  
        fprintf(stderr, "EST_not_valid_after_chaining.\n");  
        return 0;  
    }  
    return 1;  
116 }
```

D.2 chaining.h

```
#ifndef __CHAINING_H  
#define __CHAINING_H  
3  
  
extern int leveling_constraints_before_chaining;  
extern int leveling_constraints_after_chaining;  
  
int chaining_algorithm();  
8  
#endif
```

D.3 debug.c

```
1 #include <stdio.h>  
#include <stdarg.h>  
  
#include "debug.h"  
  
6 void debug(char* format, ...) {  
#ifdef DEBUG  
    va_list args;  
  
    va_start(args, format);  
11    vfprintf(stderr, format, args);  
    va_end(args);  
#endif  
}
```

D.4 debug.h

```
#ifndef __DEBUG_H  
#define __DEBUG_H  
  
void debug(char* format, ...);  
5  
#endif
```

D.5 esta_plus.c

```

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
4
#include "tmsp.h"
#include "stn.h"
#include "floydwarshall.h"
#include "ifpc.h"
9
#include "esta_plus.h"
#include "salloc.h"
#include "debug.h"

int leveling_constraints_before_chaining;
14
int get_conflict_type(conflict_t* c) {
    if (W(AE(c->i1, c->j1), AS(c->i2, c->j2)) < 0 && W(AE(c->i2, c->j2), AS(c->i1, c->j1)) < 0)
        return 1;
    if (W(AE(c->i1, c->j1), AS(c->i2, c->j2)) < 0 && W(AE(c->i2, c->j2), AS(c->i1, c->j1)) >= 0 && W(AS(c->i1, c->j1), AE(c->i2, c->j2)) > 0)
19        return 2;
    if (W(AE(c->i2, c->j2), AS(c->i1, c->j1)) < 0 && W(AE(c->i1, c->j1), AS(c->i2, c->j2)) >= 0 && W(AS(c->i2, c->j2), AE(c->i1, c->j1)) > 0)
        return 3;
    return 4;
}
24
conflict_t* new_conflict(requirement* req1, requirement* req2) {
    conflict_t* c = (conflict_t*) safe_malloc(sizeof(conflict_t));

    c->i1 = req1->i;
29    c->j1 = req1->j;
    c->i2 = req2->i;
    c->j2 = req2->j;

    c->type = get_conflict_type(c);
34
    return c;
}

/* puts all conflicts in the peak into the right conflict set t1
   to t4 */
39 void classify_conflicts(peak_t* peak, List* t1, List* t2, List*
    t3, List* t4) {
    int i, j;
    List* list = 0;

    for (i = 0; i < peak->competing_activities->size - 1; i++) {
44        for (j = i + 1; j < peak->competing_activities->size; j++) {

```

```
        requirement* req_i = (requirement*) list_get(peak->
            competing_activities , i);
        requirement* req_j = (requirement*) list_get(peak->
            competing_activities , j);

        conflict_t* c = new_conflict(req_i , req_j);
49
        switch (c->type) {
            case 1: list = t1; break;
            case 2: list = t2; break;
            case 3: list = t3; break;
54         case 4: list = t4; break;
        }

        list_append(list , c);
    }
59 }
}

double get_w_res(conflict_t* c , int div_S) {
64 //     if (div_S)/*{{{*/
//         return c.a1.endTimepoint.distance(c.a2.startTimepoint) +
//             c.a2.endTimepoint.distance(c.a1.startTimepoint);
//     else
//         return Math.min(
69 //             c.a1.endTimepoint.distance(c.a2.startTimepoint),
//             c.a2.endTimepoint.distance(c.a1.startTimepoint)
//         );/*}}}*/

    double S = 1;
74    if (div_S) {
        S = (double) (min(W(AE(c->i1 , c->j1) , AS(c->i2 , c->j2)) , W(
            AE(c->i2 , c->j2) , AS(c->i1 , c->j1))))
            /
            (max(W(AE(c->i1 , c->j1) , AS(c->i2 , c->j2)) , W(
                AE(c->i2 , c->j2) , AS(c->i1 , c->j1))));
    }
79    double sqrtS = sqrt(S);

    return min(
        W(AE(c->i1 , c->j1) , AS(c->i2 , c->j2)) / sqrtS ,
        W(AE(c->i2 , c->j2) , AS(c->i1 , c->j1)) / sqrtS
84    );
}

conflict_t* select_conflict(peak_t* peak) {
89    int i;

    if (peak->competing_activities->size == 1) {
        // this activity alone requires more than is available in
        // total
        return 0;
    }
}
```

```

94  List* type1_conflicts = new_list();
    List* type2_conflicts = new_list();
    List* type3_conflicts = new_list();
    List* type4_conflicts = new_list();
99  classify_conflicts(peak, type1_conflicts, type2_conflicts,
        type3_conflicts, type4_conflicts);

    if (type2_conflicts->size + type3_conflicts->size +
        type4_conflicts->size == 0) {
        // all conflicts are type 1, this peak is unresolvable
        return 0;
104 }

    /* select conflict with minimal w_res
    *
109  *  $w_{\{res\}}(a1, a2) = \min(d(e_{a1}, s_{a2})/S, d(e_{a2}, s_{a1})/S)$ 
    * if all conflicts are type 4, then  $S = \min/\max$ , else  $S = 1$ 
    */
    conflict_t* minimal_conflict = 0;
    double minimal_w_res = 0;
114  int div_S = (type2_conflicts->size + type3_conflicts->size ==
        0);

    for (i = 0; i < type2_conflicts->size; i++) {
        conflict_t* c = (conflict_t*) list_get(type2_conflicts, i);
        double w_res = get_w_res(c, div_S);
119  if (!minimal_conflict || w_res < minimal_w_res) {
            minimal_conflict = c;
            minimal_w_res = w_res;
        }
    }

124  for (i = 0; i < type3_conflicts->size; i++) {
        conflict_t* c = (conflict_t*) list_get(type3_conflicts, i);
        double w_res = get_w_res(c, div_S);
        if (!minimal_conflict || w_res < minimal_w_res) {
129  minimal_conflict = c;
            minimal_w_res = w_res;
        }
    }

    if (div_S) {
        /* if there are type 2 or 3 conflicts, then the type 4
           conflicts
134  * will not have minimal w_res so we need not check them
        */
        for (i = 0; i < type4_conflicts->size; i++) {
            conflict_t* c = (conflict_t*) list_get(type4_conflicts, i)
            ;
            double w_res = get_w_res(c, div_S);
139  if (!minimal_conflict || w_res < minimal_w_res) {
                minimal_conflict = c;
                minimal_w_res = w_res;
            }
        }
    }

```

```
144     }
    }

    delete_list(type1_conflicts);
    delete_list(type2_conflicts);
    delete_list(type3_conflicts);
149    delete_list(type4_conflicts);

    return minimal_conflict;
}

154 void select_precedence_constraint(conflict_t* c, precedence* p)
    {
        if (W(AE(c->i1, c->j1), AS(c->i2, c->j2)) > W(AE(c->i2, c->j2)
            , AS(c->i1, c->j1))) {
            p->i1 = c->i1; p->j1 = c->j1;
            p->i2 = c->i2; p->j2 = c->j2;
159        } else {
            p->i1 = c->i2; p->j1 = c->j2;
            p->i2 = c->i1; p->j2 = c->j1;
        }
    }
}

164 /* return list of activities that require the use of resource
   * r at the start of activity i, j
   */
List* get_competing_activities(int i, int j, int r) {
169     int k;
    int demand = 0;
    List* competing_activities = new_list();

    for (k = 0; k < R(r)->requirements->size; k++) {
174         requirement* req = (requirement*) list_get(R(r)->
            requirements, k);
        if (P_ik(i, j, req->i, req->j)) {
            list_append(competing_activities, req);
            demand += req->amount;
        }
179     }
    debug("Checked all activities, peak is %d, capacity = %d.\n",
        demand, R(r)->capacity);
    if (demand <= R(r)->capacity) {
        debug("Demand <= capacity.\n");
        delete_list(competing_activities);
184     return 0;
    }
    else {
        debug("Demand > capacity.\n");
        return competing_activities;
189    }
}

int find_peak(peak_t* peak) {
```

```

194  int i, j;
    for (i = 0; i < tmsp->n_resources; i++) {
        for (j = 0; j < R(i)->requirements->size; j++) {
            requirement* req = (requirement*) list_get(R(i)->
                requirements, j);
            debug("Looking_for_peak_at_start_of_%d/%d_on_resource_%d.\n",
                req->i, req->j, i);

199            List* competing_activities = get_competing_activities(req
                ->i, req->j, i);
            if (competing_activities) {
                peak->i = req->i;
                peak->j = req->j;
                peak->r = i;
204            peak->competing_activities = competing_activities;
                return 1;
            }
        }
    }
    debug("No_peak_found!\n");
209    return 0;
}

int esta_plus() {
214    /* Earliest Start Time Algorithm:
        *
        * 1. create STN
        * 2. loop
        * 3. refresh temporal information in stn
219    * 4. conflictSet = computeConflicts
        * 5. if no conflictSet { return EST schedule }
        * 6. if unsolvable conflictSet { return null }
        * 7. conflict = selectConflict
        * 8. pc = selectPrecedenceConstraint(conflict)
224    * 9. postConstraint(pc)
        * 10. end loop
        */

    peak_t* peak = (peak_t*) malloc(sizeof(peak_t));
229    precedence* p = (precedence*) malloc(sizeof(precedence));

    long iteration = 0;
    leveling_constraints_before_chaining = 0;

234    while(1) {
        debug("Iteration_%ld_starting.\n", iteration++);
        if (!find_peak(peak)) {
            // no peak found, done
            free(peak); free(p);
239            return 1;
        }
        debug("Peak_found,_selecting_conflict.\n");

        conflict_t* conflict = select_conflict(peak);

```

```
244     if (! conflict) {
        debug(" Unresolvable_peak_found!\n");
        free(peak); free(p);
        return 0; // peak is unresolvable
    }
249     debug(" Conflict %s/%s_selected.\n", A(conflict->i1, conflict
        ->j1)->name, A(conflict->i2, conflict->j2)->name);
        delete_list(peak->competing_activities);

        select_precedence_constraint(conflict, p);
        debug(" Precedence_constraint_selected %s->%s.\n", A(p->i1,
            p->j1)->name, A(p->i2, p->j2)->name);
254     if (! ifpc_add_edge(AS(p->i2, p->j2), AE(p->i1, p->j1), 0)) {
        debug(" STN_inconsistent!\n");
        free(peak); free(p);
        return 0;
    }
259     leveling_constraints_before_chaining++;
}

}
```

D.6 esta_plus.h

```
2  #ifndef _ESTA_PLUS_H
    #define _ESTA_PLUS_H

    #include "tmstp.h"
    #include "stn.h"

7  extern int leveling_constraints_before_chaining;

    typedef struct peak_t {
        int i; // train
        int j; // activity
12     int r; // resource
        List* competing_activities;
    } peak_t;

    typedef struct conflict_t {
17     int i1, j1;
        int i2, j2;
        int type;
    } conflict_t;

22 int esta_plus();

    /* Returns true iff a2's earliest possible run is running at a1.
        start.earliest. */
    #define P_ik(i1, j1, i2, j2) ((i1 == i2 && j1 == j2) || \
```



```

27                                     (EST(i2 ,j2) <= EST(i1 ,j1) && EST(i1
                                     ,j1) < EET(i2 ,j2)))
#define min(a,b)  (a > b ? b : a)
#define max(a,b)  (a > b ? a : b)

#endif

```

D.7 floydwarshall.c

```

#include "stn.h"

int floyd_warshall() {
    int i, j, k;
5
    // main algorithm
    for (k = 0; k < stn->n_vertices; k++) {
        for (i = 0; i < stn->n_vertices; i++) {
            if (EDGE(i,k)->infinite) { continue; }
10            for (j = 0; j < stn->n_vertices; j++) {
                if (EDGE(k,j)->infinite) { continue; }
                // update bound if tighter
                set_weight(i, j, EDGE(i,k)->weight + EDGE(k,j)->weight);
15            }
        }
    }

    // consistency check
    for (i = 0; i < stn->n_vertices; i++) {
20        if (EDGE(i,i)->weight < 0) {
            return 0;
        }
    }
    return 1;
25 }

```

D.8 floydwarshall.h

```

#ifndef _FLOYDWARSHALL_H
#define _FLOYDWARSHALL_H
4 #include "stn.h"

int floyd_warshall();

#endif

```

D.9 ifpc.c

```
2  #include <assert.h>
   #include <stdio.h>

   #include "tmstp.h"
   #include "stn.h"
   #include "salloc.h"
7  #include "debug.h"

   /**
    * Adds the edge e to the stn. Precondition: the stn is
    * consistent.
    * Postcondition: the stn is consistent. Returns 1 if the edge
    * could
12  * be added without making the stn inconsistent.
    * @param stn
    * @param e_ab
    * @return
    */
17 int ifpc_add_edge(int a, int b, int weight) {
    int i, j, k;

    if (!stn_consistent()) {
        debug("Error: stn was inconsistent at beginning of IFPC_
        algorithm.\n");
22  }

    // if the return-edge exists and adding e would induce a
    // negative cycle
    // then reject
    if (!EDGE(b,a)->infinite && EDGE(b,a)->weight + weight < 0) {
27  return 0;
    }

    // if the edge already exists and has lower weight, then do
    // nothing
    if (!EDGE(a,b)->infinite && EDGE(a,b)->weight < weight) {
32  return 1;
    }

    // ok we're good, now add (or replace) the edge
    set_weight(a, b, weight);
37

    /**
     * check if k -> a -> b is faster than k -> b for all vertices
     * k
     * in a's incoming set, build set I
     */
42 List* I = new_list();
    for (k = 0; k < stn->n_vertices; k++) {

        if (k == a || k == b || EDGE(k,a)->infinite) {
```

```

47     continue;
    }

    set_weight(k, b, EDGE(k,a)->weight + EDGE(a,b)->weight);

    int* ip = (int*) safe_malloc(sizeof(int));
52    list_append(I, ip);
    *ip = k;
}
/*
 * check if a -> b -> k is faster than a -> k for all vertices
      k
57 * in b's outgoing set, build set J
 */
List* J = new_list();
for (k = 0; k < stn->n_vertices; k++) {

62     if (k == a || k == b || EDGE(b,k)->infinite) {
         continue;
     }

    set_weight(a, k, EDGE(a,b)->weight + EDGE(b,k)->weight);

67     int* ip = (int*) safe_malloc(sizeof(int));
    list_append(J, ip);
    *ip = k;
}

72 /* finally, process sets I and J */
for (i = 0; i < I->size; i++) {
    int* v = (int*) list_get(I, i);
    for (j = 0; j < J->size; j++) {
77         int* w = (int*) list_get(J, j);
         if (*v == *w) {
             continue;
         }

82         assert(!EDGE(*v,a)->infinite);
         assert(!EDGE(*v,b)->infinite);
         assert(!EDGE(a,*w)->infinite);
         assert(!EDGE(b,*w)->infinite);

87         set_weight(*v, *w, EDGE(*v,a)->weight + EDGE(a,*w)->weight
        );
     }
}

// free memory
92 for (i = 0; i < I->size; i++) { free(list_get(I, i)); }
for (j = 0; j < J->size; j++) { free(list_get(J, j)); }
delete_list(I);
delete_list(J);

97 if (!stn_consistent()) {

```

```
        debug("Error: _stn_was_inconsistent_at_end_of_IFPC_algorithm\n");
    }
    return 1;
102 }
```

D.10 ifpc.h

```
3 #ifndef _IFPC_H
#define _IFPC_H

int ifpc_add_edge(int, int, int);

#endif
```

D.11 lex.l

```
/* Lexical analyzer that creates Token structs for all matched
   tokens
   and prints error messages for unmatched tokens. Tries to
   match as
   much of GNU 'as' AT&T syntax assembly as possible. */
4 %option nounput

%{

9 #include "token.h"
#include "salloc.h"
#include "grammar.tab.h" /* import token definitions from Yacc
   */

#define YY_NOINPUT
14 Token *new_token(int);

%}

19 TYPE [RTAQP]
STRING  \"[^\n"]*\ "
INT_1000 (0|([1-9][0-9]*))
WHITESPACE_ [\t\r_]
COMMENT_   #.*
24 %%

\n_1000000000{yyval.token=_new_token(T_LINE_SEP);_return_yyval.
token->type;_}
```

```

29 {TYPE} _yyval.token = _new_token(yytext[0]); _return _yyval
    .token->type; _}
{INT} _yyval.token = _new_token(T_INT); _return _yyval.token
    ->type; _}
{STRING} _yyval.token = _new_token(T_STRING); _return _yyval.
    token->type; _}

{WHITESPACE}+
34 {COMMENT}

    _yyval.token = _new_token(T_ERROR); _return _yyval.token->type; _}

%%

39 /*
    * creates Token struct, sets type, copies yytext
    * and sets context info (line and column number)
    */
44 Token *_new_token(int type) {
    Token *token = safe_malloc(sizeof(Token));
    token->type = type;
    token->text = safe_strdup(yytext);
    return token;
49 }

```

D.12 list.h

```

1 #ifndef LIST_H
#define LIST_H

typedef struct List
{
6     int size, allocated;
    void **data;
}
List;

11 List *_new_list(void);
    /* Create a new list and allocate memory for it */

void delete_list(List *);
    /* Deallocate a list.
16     * NOTE: This does not deallocate the items in the list
        * since the list is unaware of the type of data it
            contains.
        */

void list_grow(List *);
21 /* Double the list capacity.

```

```
        */
void list_append(List * list , void *data);
/* Set a data item in the list on position list->size.
26 */

void list_set(List * list , int index, void *data);
/* Set a data item in the list at the specified index.
31 */

void list_get(const List *, int index);
/* Return the 'index'th item in the list
   * First item has index 0.
36 */

#endif
```

D.13 list.h

```
#ifndef LIST_H
#define LIST_H
3
typedef struct List
{
    int size , allocated;
    void **data;
8 }
List;

List *new_list(void);
/* Create a new list and allocate memory for it */
13

void delete_list(List *);
/* Deallocate a list.
   * NOTE: This does not deallocate the items in the list
   * since the list is unaware of the type of data it
   * contains.
18 */

void list_grow(List *);
/* Double the list capacity.
23 */

void list_append(List * list , void *data);
/* Set a data item in the list on position list->size.
   */

28 void list_set(List * list , int index, void *data);
/* Set a data item in the list at the specified index.
   */
```

```

33 void *list_get(const List *, int index);
    /* Return the 'index'th item in the list
     * First item has index 0.
     */
#endif

```

D.14 main.c

```

#include <stdio.h>
#include <string.h>
3
#include "token.h"
#include "grammar.tab.h"

#include "tmstp.h"
8
#include "stn.h"
#include "esta_plus.h"
#include "chaining.h"
#include "salloc.h"
#include "timing.h"
13
#include "debug.h"

int solve() {
    debug("Constructing_STN.\n");
    timing_start("stn");
18
    int stn_consistent = stn_construct();
    timing_stop("stn");
    if (!stn_consistent) {
        debug("Problem_inconsistent.\n");
        return 0;
23
    }

    debug("Running_ESTA+_algorithm.\n");
    timing_start("esta+");
    if (esta_plus(tmstp, stn)) {
28
        timing_stop("esta+");
        if (is_est_valid()) {
            debug("Found_valid_schedule.\n");
            debug("Running_CHAINING_algorithm.\n");
            timing_start("chaining");
33
            if (!chaining_algorithm()) {
                timing_stop("chaining");
                debug("Chaining_failed!\n");
                return 0;
            } else {
38
                timing_stop("chaining");
                debug("Chaining_succeeded.\n");
                return 1;
            }
        } else {

```

```
43     debug("ESTA_succeeded_but_cannot_find_valid_schedule!\n");
        return 0;
    }
} else {
    timing_stop("esta+");
48     debug("Could_not_find_valid_schedule.\n");
        return 0;
    }
}

53 // calculate throughput as sum of EET of entire train
int throughput() {
    int i,j,tp = 0;

58     for (i = 0; i < tmsp->n_trains; i++) {
        if (N(i) > 0) {
            int latest_end_time = 0;
            int earliest_start_time = EST(i,0);
            for (j = 0; j < N(i); j++) {
63                 if (EST(i,j) < earliest_start_time) {
                    earliest_start_time = EST(i,j);
                }
                if (EET(i,j) > latest_end_time) {
68                     latest_end_time = EET(i,j);
                }
            }
            tp += latest_end_time - earliest_start_time;
        }
    }
73     return tp;
}

double robustness() {
78     return 0.0;
}

void print_one_line_summary() {
    fprintf(stderr, "%lf_", find_timing_info("parsing")->total);
    fprintf(stderr, "%lf_", find_or_create_timing_info("esta+")->
        total);
83     fprintf(stderr, "%lf_", find_or_create_timing_info("chaining")
        ->total);
    fprintf(stderr, "%d_", leveling_constraints_before_chaining);
    fprintf(stderr, "%d_", leveling_constraints_after_chaining);
    fprintf(stderr, "%lf_", robustness());
    fprintf(stderr, "%d\n", throughput());
88 }

int main(int argc, char *argv[]) {

93     timing_start("total");

    // scan-only option to test lexer
```



```

    if (argc == 2 && !strcmp("-s", argv[1])) {
        fprintf(stderr, "Lexing_only.\n\n");
        while (yylex()) { print_token(yylval.token); }
98     return 0;
    }

    debug("Parsing.\n");

103    // parser
    timing_start("parsing");
    yyin = stdin;
    if (yyparse() != 0) {
        fprintf(stderr, "Parsing_failed..Aborting!\n");
108    }
    timing_stop("parsing");

    // only one activity per train at any time
    int n_real_resources = tmsp->n_resources;
113    add_train_mutexes();

    // print error summary
    if (error_counter > 0) {
        fprintf(stderr, "\n%d_error%s\n\n", error_counter,
            error_counter == 1 ? "s" : "s");
118    return -1;
    }

    int solved = solve();
    int tpbefore = throughput();

123    int i = random() % tmsp->n_trains;
    // duplicate train i
    int newi = tmsp->n_trains;
    add_train(tmsp->n_trains, RD(i) / 2, RD(i) / 2 + (DD(i) - RD(i)
        )), "\newtrain\n");
128    add_resource(tmsp->n_resources, 1, safe_strdup(T(i)->name));
    int j;
    for (j = 0; j < N(i); j++) {
        add_activity(newi, j, D(i,j), A(i,j)->name);
        add_requirement(newi, j, n_real_resources, 1);
133        if (j > 0)
            add_precedence(newi, j - 1, newi, j);
        int k;
        for (k = 0; k < n_real_resources; k++) {
            if (Q(i,j,k) > 0) {
138                add_requirement(newi, j, k, Q(i,j,k));
            }
        }
    }

143    //fprintf(stderr, "Throughput before delay: %d\n", throughput
        ());
    solved = solve();

```

```
    //fprintf(stderr, "Throughput after delay: %d\n", throughput()
    );
    int tpafter = throughput();

148     int latest_end_time = 0;
    int earliest_start_time = EST(newi,0);
    int tp = 0;
    for (j = 0; j < N(newi); j++) {
153         if (EST(newi,j) < earliest_start_time) {
            earliest_start_time = EST(newi,j);
        }
        if (EET(newi,j) > latest_end_time) {
158             latest_end_time = EET(newi,j);
        }
    }
    tp += latest_end_time - earliest_start_time;

163     fprintf(stderr, "%d\n", throughput() - tp - tpbefore);

    timing_stop("total");

    fprintf(stderr, "Instance_%ssolved.\n", (solved ? "" : "not_"))
    );
168     print_one_line_summary();

    timing_print_summary();
    fprintf(stderr, "leveling_constraints_before_chaining: %d\n",
        leveling_constraints_before_chaining);
    fprintf(stderr, "leveling_constraints_after_chaining: %d\n",
        leveling_constraints_after_chaining);
173     fprintf(stderr, "robustness: %lf\n", robustness());
    fprintf(stderr, "throughput: %d\n", throughput());

    fprintf(stderr, "EST_schedule:\n");
    //print_est_schedule();
178
    return 0;
}
```

D.15 salloc.c

```
#include "salloc.h"

3  #include <stdio.h>
    #include <stdlib.h>
    #include <string.h>

8  static void *null_pointer_check(void *ptr) {
```

```
    if (ptr == 0) {  
        fprintf(stderr, "Out_of_memory\n");  
        exit(1);  
    }  
13     return ptr;  
}  
  
18 void *safe_malloc(size_t size) {  
    return null_pointer_check(malloc(size));  
}  
  
23 void *safe_realloc(void *ptr, size_t size) {  
    return null_pointer_check(realloc(ptr, size));  
}  
  
28 char *safe_strdup(const char *string) {  
    return strcpy(safe_malloc(strlen(string) + 1), string);  
}
```

D.16 salloc.h

```
#ifndef SALLOC_H  
#define SALLOC_H  
  
#include <stdlib.h>  
5  
  
extern void *safe_malloc(size_t size);  
    /* Does malloc(size) with null-pointer check.  
    * Deallocate with free().  
    */  
10  
  
extern void *safe_realloc(void *ptr, size_t size);  
    /* Does realloc(ptr, size) with null-pointer check.  
    * Deallocate with free().  
    */  
15  
  
extern char *safe_strdup(const char *string);  
    /* Returns a copy of string with null-pointer check.  
    * Deallocate with free().  
    */  
20  
#endif
```

D.17 stn.c

```
#include <stdio.h>
#include <stdlib.h>

4  #include "stn.h"
    #include "tmstp.h"
    #include "floydwarshall.h"
    #include "debug.h"

9  stn_t* stn;

    int stn_consistent() {
        int i;
        for (i = 0; i < stn->n_vertices; i++) {
14         if (EDGE(i,i)->weight < 0) {
                return 0;
            }
        }
        return 1;
19 }

    diedge_t* set_weight(int a, int b, int weight) {
        // get the edge
        diedge_t* e = EDGE(a,b);

24         // set weight
        if (e->infinite || e->weight > weight) {
            e->infinite = 0;
            e->weight = weight;
29         }

        return e;
    }

34 int add_new_vertex() {
        return stn->n_vertices++;
    }

    void stn_add_activity(int i, int j) {
39         // add timepoints
        AS(i,j) = add_new_vertex();
        AE(i,j) = add_new_vertex();

        // add constraints
44         set_weight(AS(i,j), TS(i), 0);
        set_weight(TE(i), AE(i,j), 0);

        set_weight(AS(i,j), AE(i,j), D(i,j));
        set_weight(AE(i,j), AS(i,j), -D(i,j));
49     }

    void stn_add_train(int i) {
        int j;
```

```

54 // start end vertices
   TS(i) = add_new_vertex();
   TE(i) = add_new_vertex();

   // 4 directed edges to and from z
59   set_weight(Z, TS(i), RD(i));
   set_weight(TS(i), Z, -RD(i));
   set_weight(Z, TE(i), DD(i));
   set_weight(TE(i), Z, -DD(i));

64 // all activities
   for (j = 0; j < N(i); j++) {
       stn_add_activity(i, j);
   }
}

69 int stn_construct() {
   int i, j;

   // create stn
74   stn = (stn_t*) malloc(sizeof(stn_t));
   stn->n_vertices = 0;
   stn->n_total_vertices = 1 + 2 * (tmstp->n_trains + tmstp->
       n_activities);
   stn->edges = (diedge_t*) calloc(stn->n_total_vertices * stn->
       n_total_vertices, sizeof(diedge_t));

79 // add temporal reference point
   add_new_vertex(stn);

   // initialize edges
   for (i = 0; i < stn->n_total_vertices; i++) {
84     for (j = 0; j < stn->n_total_vertices; j++) {
         if (i == j) {
             EDGE(i, j)->weight = 0;
         }
         EDGE(i, j)->infinite = (i != j);
89     }
   }

   // add all trains
   for (i = 0; i < tmstp->n_trains; i++) {
94     stn_add_train(i);
   }

   // and all precedences
   for (i = 0; i < tmstp->precedences->size; i++) {
99     int i1 = P(i)->i1;
     int j1 = P(i)->j1;
     int i2 = P(i)->i2;
     int j2 = P(i)->j2;

104     set_weight(AS(i2, j2), AE(i1, j1), 0);
   }

```

```
    // calculate minimal network
    return floyd_warshall();
109 }

int demand(int k, int t) {
    int i, j, d = 0;

114     for (i = 0; i < tmsp->n_trains; i++) {
        for (j = 0; j < N(i); j++) {
            if (EST(i, j) <= t && EET(i, j) > t) {
                d += Q(i, j, k);
            }
119     }
    }
    return d;
}

124 int is_est_valid() {
    int i, j, k;

    for (i = 0; i < tmsp->n_trains; i++) {
        for (j = 0; j < N(i); j++) {
129             if (EST(i, j) < RD(i) || EET(i, j) > DD(i) || EST(i, j) + D(i, j) != EET(i, j)) {
                debug("Temporal constraint violated, t:%d a:%d rd:%d s:%d d:%d e:%d dd:%d.\n", i, j, RD(i), EST(i, j), D(i, j), EET(i, j), DD(i));
                return 0;
            }
            for (k = 0; k < tmsp->n_resources; k++) {
134                 if (demand(k, EST(i, j)) > C(k)) {
                    debug("Resource constraint violated, demand of %d exceeds capacity of %d on resource %d.\n", demand(k, EST(i, j)), C(k), k);
                    return 0;
                }
            }
139     }
}

    for (i = 0; i < tmsp->precedences->size; i++) {
        precedence* p = list_get(tmsp->precedences, i);
144         if (EET(p->i1, p->j1) > EST(p->i2, p->j2)) {
            debug("Precedence constraint between %d/%d and %d/%d violated.\n", p->i1, p->j1, p->i2, p->j2);
            return 0;
        }
    }
149 }

return 1;
}

void print_est_schedule() {
```

```

154  int i, j;

    for (i = 0; i < tmsp->n_trains; i++) {
        for (j = 0; j < N(i); j++) {
            //printf("a:%d,%d s:%d e:%d t:%s a:%s\n", i, j, EST(i,j),
            EET(i,j), T(i)->name, A(i,j)->name);
159      printf("%d_%d_%d\n", i, j, EST(i,j));
        }
    }
}

```

D.18 stn.h

```

2  #ifndef __STN_H
    #define __STN_H

    #include "list.h"

    typedef struct diedge_t {
7      int infinite;
        int weight;
    } diedge_t;

    typedef struct stn_t {
12     int n_vertices;
        int n_total_vertices;
        diedge_t* edges;
    } stn_t;

17  extern stn_t* stn;

    int stn_construct();
    int stn_consistent();
    diedge_t* set_weight(int, int, int);
22  void print_est_schedule();
    int is_est_valid();

    #define Z      0
    #define EDGE(a,b) (stn->edges + a * stn->n_total_vertices + b)
27  #define W(a,b)   (EDGE(a,b)->weight)

    #endif

```

D.19 timing.c

```

#include <sys/time.h>
#include <string.h>
#include <stdio.h>

```

```
5  #include "timing.h"
   #include "list.h"
   #include "salloc.h"

   List* timings = 0;
10 struct timeval* tv_end = 0;

   timing_info* new_timing_info(char* name) {
       timing_info* ti = (timing_info*) safe_malloc(sizeof(
           timing_info));
       ti->name = safe_strdup(name);
15   ti->start = (struct timeval*) malloc(sizeof(struct timeval));
       ti->total = 0;
       return ti;
   }

20 void timing_init() {
       if (!timings) {
           timings = new_list();
       }
       if (!tv_end) {
25   tv_end = (struct timeval*) malloc(sizeof(struct timeval));
       }
   }

30 timing_info* find_timing_info(char* name) {
       int i;
       for (i = 0; i < timings->size; i++) {
           timing_info* ti = (timing_info*) list_get(timings, i);
           if (!strcmp(name, ti->name)) {
35   return ti;
           }
       }
       return 0;
   }

40 timing_info* find_or_create_timing_info(char* name) {
       timing_info* ti = find_timing_info(name);
       if (!ti) {
           ti = new_timing_info(name);
45   list_append(timings, ti);
       }
       return ti;
   }

50 void timing_start(char* name) {
       timing_init();
       timing_info* ti = find_or_create_timing_info(name);
       gettimeofday(ti->start, NULL);
   }

55 void timing_stop(char* name) {
       double seconds = 0;
```



```

gettimeofday(tv_end, NULL);
timing_info* ti = find_timing_info(name);
60
    int diff_sec = tv_end->tv_sec - ti->start->tv_sec;
    int diff_usec = tv_end->tv_usec - ti->start->tv_usec;

    seconds += diff_sec;
65    seconds += diff_usec / (double) 1000000;

    ti->total = ti->total + seconds;
}

70 void timing_print_summary() {
    int i;
    for (i = 0; timings && i < timings->size; i++) {
        timing_info* ti = ((timing_info*)list_get(timings, i));
        fprintf(stderr, "%s: %lf\n", ti->name, ti->total);
75    }
}

```

D.20 timing.h

```

#ifndef __TIMING_H
#define __TIMING_H
3
typedef struct timing_info {
    char* name;
    struct timeval* start;
    double total;
8 } timing_info;

timing_info* find_timing_info(char*);
timing_info* find_or_create_timing_info(char*);
void timing_start(char*);
13 void timing_stop(char*);
void timing_print_summary();

#endif

```

D.21 tmisp.c

```

#include <stdlib.h>
#include <stdio.h>
3
#include "salloc.h"
#include "tmisp.h"
#include "list.h"

8 tmisp_t* tmisp;

```

```
void tmsp_init() {
    if (!tmsp) {
13         tmsp = (tmsp_t*) malloc(sizeof(tmsp_t));

        tmsp->resources = new_list();
        tmsp->trains = new_list();
        tmsp->precedences = new_list();

18         tmsp->n_resources = 0;
        tmsp->n_trains = 0;
        tmsp->n_activities = 0;
    }
}

23 void add_resource(int i, int capacity, char* name) {
    tmsp_init();

    resource* r = (resource*) malloc(sizeof(resource));
28     r->capacity = capacity;
    r->name = safe_strdup(name);
    r->requirements = new_list();

    list_set(tmsp->resources, i, r);
33     tmsp->n_resources++;
}

void add_train(int i, int release_date, int due_date, char* name
) {
38     tmsp_init();

    train* t = (train*) malloc(sizeof(train));
    t->release_date = release_date;
    t->due_date = due_date;
    t->name = safe_strdup(name);
43     t->activities = new_list();
    t->n_activities = 0;

    list_set(tmsp->trains, i, t);
    tmsp->n_trains++;
48 }

void add_activity(int i, int j, int duration, char* name) {
    tmsp_init();

53     activity* a = (activity*) safe_malloc(sizeof(activity));
    a->duration = duration;
    a->name = safe_strdup(name);
    a->requirements = new_list();

58     list_set(T(i)->activities, j, a);
    tmsp->n_activities++;
    N(i)++;
}
```

```

63 void add_requirement(int i, int j, int k, int q) {
    requirement* req = (requirement*) safe_malloc(sizeof(
        requirement));

    req->i = i;
    req->j = j;
68    req->k = k;
    req->amount = q;

    list_set(A(i,j)->requirements, k, req); // save indexed by
        resource id in the activity
    list_append(R(k)->requirements, req); // save in 'normal'
        list in the resource
73 }

void add_precedence(int i1, int j1, int i2, int j2) {
    precedence* p = (precedence*) malloc(sizeof(precedence));

78    p->i1 = i1;
    p->j1 = j1;

    p->i2 = i2;
    p->j2 = j2;
83    list_append(tmsp->precedences, p);
}

88 void add_train_mutexes() {
    int i,j,r;

    r = tmsp->n_resources;

93    for (i = 0; i < tmsp->n_trains; i++, r++) {
        add_resource(r, 1, safe_strdup(T(i)->name));
        for (j = 0; j < N(i); j++) {
            add_requirement(i, j, r, 1);
        }
98    }
}

```

D.22 tmsp.h

```

1  #ifndef __TMSP_H
    #define __TMSP_H

    #include "list.h"
    #include "stn.h"
6  typedef struct tmsp_t {

```

```
    List* resources;
    List* trains;
    List* precedences;
11    int n_resources, n_trains, n_activities;
    } tmsp_t;

    typedef struct resource {
        int capacity;
16        char* name;
        List* requirements;
    } resource;

    typedef struct train {
21        int release_date;
        int due_date;
        List* activities;
        int n_activities;
        char* name;
26        int start_vertex;
        int end_vertex;
    } train;

    typedef struct activity {
31        int duration;
        char* name;
        List* requirements;
        int start_vertex;
        int end_vertex;
36    } activity;

    typedef struct requirement {
        int i; // train index
        int j; // activity index
41        int k; // resource index
        int amount;
    } requirement;

    typedef struct precedence {
46        int i1, j1, i2, j2;
    } precedence;

    void add_resource(int, int, char*);
    void add_train(int, int, int, char*);
51    void add_activity(int, int, int, char*);
    void add_requirement(int, int, int, int);
    void add_precedence(int, int, int, int);
    void add_train_mutexes();

56    extern tmsp_t* tmsp;

    // resources
#define R(i) ((resource*)list_get(tmsp->resources, i))
#define C(i) (R(i)->capacity)
61
```

```

// precedences
#define P(i)      ((precedence*)list_get(tmsp->precedences, i))

// treinen
66 #define T(i)      ((train*)list_get(tmsp->trains, i))
#define RD(i)      (T(i)->release_date)
#define DD(i)      (T(i)->due_date)
#define N(i)      (T(i)->n_activities)
#define TS(i)      (T(i)->start_vertex)
71 #define TE(i)      (T(i)->end_vertex)

// activiteiten
#define A(i,j)      ((activity*)list_get(T(i)->activities, j))
#define AS(i,j)      (A(i,j)->start_vertex)
76 #define AE(i,j)      (A(i,j)->end_vertex)
#define D(i,j)      (A(i,j)->duration)
#define EST(i,j)      (-W(AS(i,j), Z))
#define EET(i,j)      (-W(AE(i,j), Z))
#define REQ(i,j,k)    (A(i,j)->requirements->size <= k ? 0 : (
    requirement*)list_get(A(i,j)->requirements, k))
81 #define Q(i,j,k)    (REQ(i,j,k) == 0 ? 0 : REQ(i,j,k)->amount)

#endif

```

D.23 token.c

```

#include <stdio.h>
2 #include "token.h"

void print_token(Token *t) {
    fprintf(stderr, "type_%d --text_%s\n", t->type, t->text);
}

```

D.24 token.h

```

#ifndef __TOKEN_H
#define __TOKEN_H
3
typedef struct Token {
    int type;
    char *text;
} Token;
8

void print_token(Token *t);

int yylex(void);
int yyparse(void);
13
extern FILE *yyin;

```

```
extern int error_counter;
```

```
#endif
```