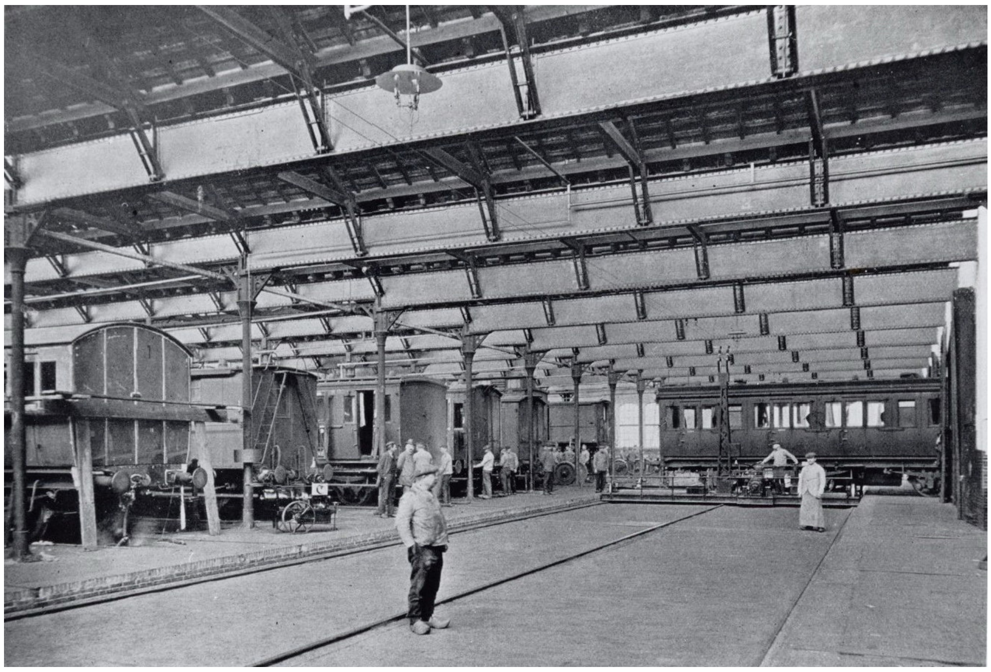
Improving PCP algorithms using flexibility metrics

Creating flexible schedules for Technical Maintenance



Jonathan Staats

Improving PCP algorithms using flexibility metrics

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jonathan Staats
born in Cork, Ireland

Algorithmics Group
Department of Software and Computer Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Maintenance Development
Fleet Services, NedTrain
Utrecht, the Netherlands
www.nedtrain.nl

Improving PCP algorithms using flexibility metrics

Author: Jonathan Staats
Student id: 4049969
Email: j.j.staats@student.tudelft.nl

Abstract

In this work we improve on existing scheduling techniques suitable for scheduling problems at the train maintenance provider, NedTrain. Scheduling problems that require flexible solutions can be modeled using variations of the Resource Constraint Project Scheduling Problem (RCPSP), which can be solved using Precedence Constraint Posting (PCP). Earlier work already showed that PCP is able to create fixed-time schedules for NedTrain. In this work we investigate two variations of PCP for creating flexible schedules. The envelope approach, where the problem instance is directly reduced to a flexible solution and the 'solve and robustify' approach, where a fixed inflexible schedule is created and then relaxed to create a flexible schedule. Using state of the art flexibility metrics and Interval Schedules we show that the envelope approach, which has been considered the weaker approach in recent literature, can be greatly improved. By using Interval Schedules to remove temporal interaction between tasks we were able to estimate resource violations much more accurately, improving the flexibility of the resulting schedules.

Preface

The work before you is the result of my final project as a master student. I started my studies at the Delft University of Technology in 2009, in which time I have had the opportunity to develop many new skills. The new skills allowed me to help NedTrain to better understand their scheduling challenges and to produce this thesis.

However I did not do this alone. If not for the support of my friends and family and without the guidance of the member of the algorithms group I would never have been able to finish this work. First I would like to thank Bob Huisman for giving me the opportunity to work for NedTrain and for providing me with an interesting and challenging thesis project. I want to thank Michel Wilson for helping me with the many dilemmas I encountered and for reminding me to be clear and concise in my work. Finally I want to thank Cees Witteveen for knowing which questions to ask whenever I was unsure about my work or when I felt I was at a dead end.

I would also like to thank colleagues at NedTrain and students at the university. They helped me with many small problems by asking questions or by just letting me explain something. I also want to thank my father and sister for proof reading my work. Finally I would like to thank the thesis committee for taking the time to read this thesis.

Jonathan Staats
Delft, the Netherlands
March 18, 2014

Contents

List of Figures

Chapter 1

Introduction

In this chapter we will introduce NedTrain and the challenges they face with respect to maintenance scheduling. We will also have a quick look at scheduling in general. Finally, we will formulate what NedTrain wants to learn with this thesis work.

1.1 NedTrain

NedTrain is a subsidiary company of the NS Group (*Nederlandse Spoorwegen*), see Figure 1.1. The NS Group also has the controlling shares of NS Reizigers, the biggest passenger railway operator in the Netherlands. With over 3000 passenger carriages it ensures that over 1 million passengers get to where they need to go each day. NedTrain is tasked with keeping a high availability of the rolling stock of NS Reizigers.

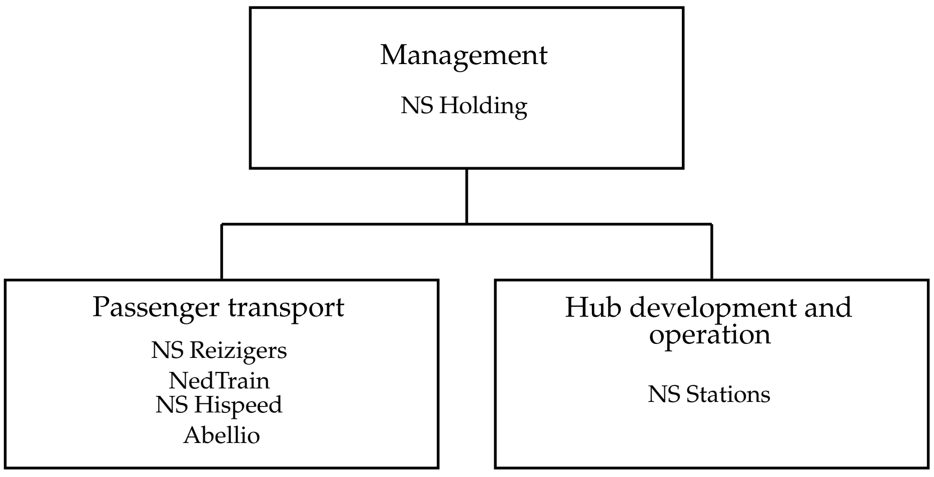


Figure 1.1: Organization chart NS.

NedTrain performs the maintenance, repair and refurbishment of the complete rolling stock (locomotives, railroad cars, coaches, and wagons) of NS. It also does work for other smaller railway operators like Arriva. NedTrain has depots in locations all over the Netherlands for these tasks. The tasks of NedTrain are divided into three main categories:

First-line service This consists of cleaning the railway cars and fixing small technical problems. There are about 30 facilities across the country able to provide these services.

Technical maintenance Every couple of months, after a certain amount of mileage, all train material receives a checkup and maintenance. Parts may also be replaced or tuned up. This can be done at one of four NedTrain maintenance depots.

Refurbishment Once or twice in the life time of a train it is sent to the NedTrain overhaul facility in Haarlem where it is refurbished to bring it up to modern standards. For example, in 2010 the DD-AR passenger train series received an overhaul and was renamed to DDZ, as shown in Figure 1.2.

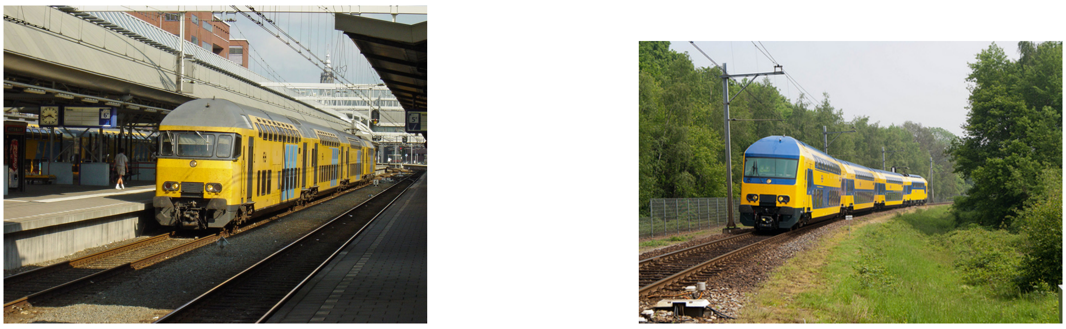

(a) The old DD-AR. (b) The refurbished DDZ.

Figure 1.2: Refurbishment of the DD-AR train series.

It is the wish of NedTrain to compete on the open European market for rolling stock maintenance. To be able to do this, NedTrain needs to continue to improve its internal processes, which will reduce costs and improve the quality of their services. Reducing costs can be done by fine-tuning supply chains and increasing the utilization of resources.

In this work, we will focus on *technical* maintenance and more precisely the scheduling process of technical maintenance. In the following sections we will discuss the scheduling process at NedTrain and the particular scheduling challenges faced by NedTrain.

1.2 Maintenance Scheduling

For the technical maintenance NedTrain keeps a detailed schedule to manage resources and deadlines. In this section we will look at how NedTrain currently constructs schedules for technical maintenance.

NS Reizigers (NSR) has about a dozen different train series. Each series has its own set of recurring check-ups. A check-up is a set of predefined tasks, consisting of inspection tasks, maintenance tasks and cleaning tasks. Which check-up is done is dependent on the mileage of the train or the amount of time since its last check-up. At any one time NedTrain performs check-ups on as many as 250 carriages in all maintenance facilities.

Not only check-up tasks are scheduled, when a train enters a depot for technical maintenance, unexpected repairs and small refurbishments are also done. The scheduling process for technical maintenance is complex with all the different tasks, all requiring different resources. NedTrain has an extended scheduling process to create the schedules which manage and control the flow of work and resources in the maintenance depot.

The scheduling process itself is divided into two main parts. The first part is the construction of a base schedule and the second part is the construction of a *week schedule*. The base schedule is constructed twice a year, once for the summer season and once for the winter season. The base schedule is generally repeated every week with only a few variations to account for some national holidays or other special occasions. For example during Queens Day (or Kings Day), NS Reizigers will need extra trains and so NedTrain will generally have less trains in the depot.

When constructing the base schedule only a few things are known: the train type, when it will arrive at the depot and when it will depart the depot. With only the train type information, many resources can already be allocated because all the train series have a predefined set of checkup tasks.

The base schedule can also be viewed as a kind of planning because it consists of placeholder tasks with very little detail. Using the base scheduling the detailed scheduling process refines the base schedule by replacing the placeholders with more fine-grained tasks. This way the base schedule evolves as more details become known.

The more detailed week schedule starts to become final about two weeks beforehand. At this point the train unit becomes known and a large portion of the tasks can be scheduled. For example, these tasks may consist of the following:

- The tasks that belong to the standard planned check-up. For example, replacing air conditioning filters or removing waste from the on-board toilets.

- The replacement of *main parts*. Each train series has a set of well defined main parts which are generally replaced based on the age or mileage of the part, to ensure high availability of the train. Examples of a main parts are bogies (wheel set of a train) or engines.

- Small refurbishments and repairs that do not require much work or cannot wait until a big overhaul in Haarlem. For example, adding warning stickers or removing vandalism.

Many tasks in the detailed week schedule can not be done at the same time and are partially ordered using precedence constraints. Example 1 shows an example of a detailed week scheduling problem with one resource type.

Example 1 (A scheduling problem for a maintenance project at NedTrain.)**.** *A train of series 5100 is planned to arrive at 9:00 on the 25th of March and must be ready to leave at 10:00 the next day. So the train is available to NedTrain for 25 hours. It needs to* *receive a type 2 check-up. This consists of 8 tasks,* $(t_1, \ldots, t_8)$ *with the following durations in hours:* $(2,5,5,3,4,13,2,3)$*. The tasks have the following execution order:* $t_1 \prec t_2$ *and* $t_3 \prec t_4 \prec t_5$ *and* $t_6 \prec t_7 \prec t_8$*. All tasks must start after 9:00 on the 25th of march and must finish before 10:00 on the 26th. However some tasks also require the uses of a maintenance track (*"putspoor" *in Dutch) of which there are* 2 *in the depot. The following tasks:* t_2, t_3, t_6, t_8 *require one maintenance track, while* t_5 *requires 2. This example is shown as a graph in Figure 1.3.*

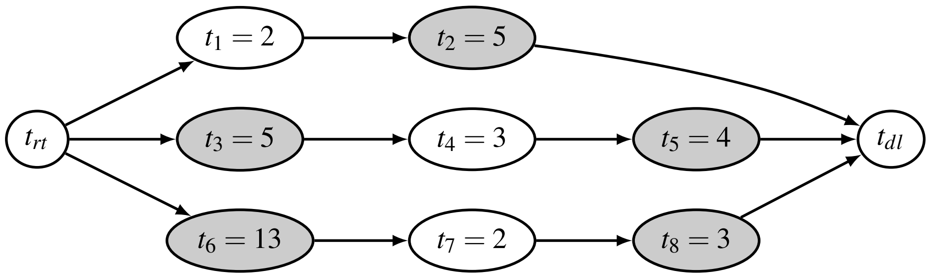


Figure 1.3: Graph representation of Example 1. The node labels are the task names and their duration in hours. The shaded tasks require the *maintenance track*.

For each train in the depot there is an extra task added to the end of the schedule, which is labeled *remainder work*. The remainder work placeholder is to add slack to the schedule so some delays may be handled without violating deadlines. Unexpected tasks may also be scheduled within this time slot.

The schedules are created by employees, with the aid of Excel and some simple planning tools. These employees often have many years of experience and are also often specialized in a smaller subproblem. This kind of system of scheduling is often hard to change and can be error-prone with many hidden processes. Making changes to the schedule requires a lot of work and so any disruptions in the schedule are solved by moving tasks to the remainder work place-holder.

1.3 NedTrain scheduling challenges

The projects at NedTrain have a set of specific properties that have great impact on the scheduling process. In this section we will highlight the complications that arise from the uncertainty in the world of maintenance and the hard deadlines imposed on NedTrain by clients and third party organizations.

The arrival and departure times of trains in a depot of NedTrain is very strict. There are two main factors that contribute to the strict time frames. First, NSR maintains a complex passenger schedule to transport 1 million passengers each day. It is important for NSR to keep this schedule stable so passengers will not be continually confronted with changing traveling times. As a consequence NSR requires trains to be available at very strict time slots, which forces NedTrain to have completed all work before a train is needed to pick up passengers. The second factor lays in the national railway network. The Netherlands has a complex and very busy railway network, which is managed by the state company ProRail. ProRail divides the space on the network between transport operators and is also responsible for safety. A train can only enter the national railway network once it has been assigned a time slot by ProRail.

The second challenge faced by schedule makers at NedTrain is the inherent uncertainty that comes with the execution of maintenance projects. Projects at NedTrain are very unpredictable because many tasks only become known after inspection and the duration of tasks are heavily dependent on the condition of a train. This uncertainty is very hazardous for the strict deadlines of the maintenance projects at NedTrain.

Adding to the above challenges is the fact that the tasks within maintenance projects also have resource and precedence constraints. An example of a resource constraint is the limited number of *maintenances tracks* at a NedTrain depot, see Figure 1.4. Tasks requiring this track should not be scheduled all at the same time. Another limited resource could be a specialized group of maintenance crew. Precedence constraints are also very common in the maintenance projects at NedTrain, for example a task may require preparation tasks to be completed first for accessibility or safety reasons.

Figure 1.4: The VIRM on maintenances tracks in Onnen.

Currently, NedTrain and NSR are also working on a project that require NedTrain to make major improvements to their scheduling process. To reduce costs and improve revenue NSR wants to deploy more trains during rush hour. To avoid investing in extra trains NSR wants NedTrain to apply technical maintenance for shorter periods to ensure trains are available during rush hours. Current technical maintenance projects will have to be cut up into smaller projects, which will reduce the efficiency of the remainder work placeholder mentioned in Section 1.2. To compensate for this and to reduce the chance a deadline is not met, schedules need to be more flexible and robust.

It is in the interest of NedTrain to improve the scheduling process of technical maintenance. NedTrain has three main goals. First is help NSR to increase fleet availability by reducing the period of time a train is undergoing maintenance. Currently the remainder work place holders produce much unneeded slack in the schedules and should be reduced without increasing the chance of deadline violations. The second goal of NedTrain is to increase resource utilization so resource overhead costs can be reduced. Finally NedTrain needs to continue to guarantee clients and other third parties that deadlines are met.

Providing schedule makers with better software tools is a good idea because computers are able to search for a schedule much faster than a person. This will help to improve the scheduling process. The software should however not only provide the schedule makers with a powerful solver but also be able to provide them with more insight into the scheduling problem itself, in other words the software needs to be fast and interactive so it can not only produce high quality schedules but also schedules that are well understood.

1.4 Improving scheduling

In this section we will look at how to improve the scheduling process at NedTrain such that costs and deadline violations can be reduced. Computers can be a great aid to schedulers to help them create cost efficient schedules in less time. To better understand how the technical maintenance schedules can be improved we will first have to look at what the function is of a schedule and how it can be used by NedTrain.

A schedule can be seen as having two main tasks. From a management point of view it enables us to make predictions on the development and status of a project. This allows for the needed resources to be made available and enables realistic deadlines to be set. The other task of a schedule is to guide maintenance teams on the shop-floor on what tasks should be done and when, to ensure that the deadlines are met and resources are not over-allocated.

To be able to make predictions and set deadlines one does not need a very detailed schedule. The exact duration or ordering of tasks can be estimated when making predictions about realistic deadlines and the minimum required resources. The schedule for the shop-floor however, does require a greater amount of detail. There needs to be a clear list of tasks with starting times in order to ensure a good flow of resources in the workplace and to ensure that the final deadline is met. The amount of detail required by the shop-floor depends on the nature of the work. A very detailed schedule can allow for very tight deadlines but will also become invalid very quickly if there is uncertainty on the shop-floor.

It is possible to create a schedule that does not become invalid too quickly while still having enough detail to inform the shop-floor of which tasks need to be done and when. To be able to create such a schedule one needs to make use of *human recovery*. Human recovery was defined by Wiers [32], which means that a schedule leaves some space in the schedule to allow people on the shop-floor to directly recover from unexpected events. By combining the presence and absence of uncertainty and human recovery, Wiers described four types of shop floor. The four types are shown in Table 1.1. The first column defines two shops with no uncertainty and the second column defines two shops with high uncertainty. The two rows show two shops with no human recovery and with human recovery.

A shop with no human recovery and no uncertainty is called a *smooth shop*. The smooth shop is ideal for optimization as it allows tasks to be scheduled very close to each other. The

Table 1.1: Shop types and autonomy [32].

	No uncertainty	Uncertainty
No human recovery	**Smooth shop** optimize	**Stress shop** support reactive scheduling
Human recovery	**Social shop** schedule as advice	**Sociotechnical shop** shop schedule as suggestion

smooth shop approach works very well in the automobile industry because robots do much of the work in a controlled environment and there is little to no execution uncertainty. The only uncertainty the automobile industry has to worry about is order uncertainty. In other words how many cars of a certain color are needed and what extra features need to be added, but once the production process starts there is no uncertainty.

However, maintenance is inherently uncertain in its execution. The duration of tasks are heavily dependent on the condition of the rolling stock. NedTrain can only choose schedules with no human recovery (*stress shop*) or a schedule that does allow human recovery (*sociotechnical shop*). A stress shop requires a fast and efficient rescheduling process which can be made possible using the right scheduling software, however it will probably have a negative impact on maintenance teams at NedTrain. Maintenance teams will become weary of continually going back to the planner for a new schedule. Also if the schedule is continuously changed it looses a lot of its planning and predicting properties which are required to set realistic deadlines.

The sociotechnical shop allows for a maintenance team to solve a limited amount of delays themselves and it will bring more stability on the shop floor if the schedules are constructed correctly. By allowing maintenance teams to solve small delays themselves it removes some complexity in the scheduling process. A schedule used in a sociotechnical shop needs to find a balance between the amount of human recovery (flexibility) and constraints to ensure that deadlines are met.

A correctly constructed schedule for a sociotechnical shop must have flexibility, while also enforcing important project constraints, like deadlines and resource capacity. This allows each maintenance team a limited amount of autonomy to solve small delays with a limited need for inter-team communication. The amount of freedom in a schedule needs to be controlled so miscommunication or other social factors can not create an unsolvable situation. Creating an optimal or close to optimal sociotechnical schedule with these properties is challenging.

A sociotechnical shop will allow NedTrain to satisfy their goal of maximizing resource utilization by closely controlling team interaction while at the same time allowing each maintenance team enough autonomy to resolve small delays themselves, which helps to satisfy deadlines.

1.5 Heuristic solvers

Earlier work [15] already showed that optimal solvers are unable to create solutions in a timely manner and even if the solving time could be reduced to a couple of hours, it will generally behave like a black-box for the people using the solver. The schedule makers will have a hard time understanding why the proposed schedule is optimal. A more interactive approach will allow schedule makers to better understand the proposed schedule and will also allow them to adjust the schedule to account for odd situations unknown to the solver.

So in this work we will focus on *heuristic* solvers, which are generally able to solve problems much faster. Using rules-of-thumb or best-practices a heuristic solver attempts to find or discover a solution of satisfactory quality. Using a heuristic solver in combination with an interactive interface will allow schedule makers to better understand the scheduling problem and the solutions proposed by the solver. Heuristic solvers can also be very fast and can operate on incomplete information, which allows users to easily adjust constraints until a suitable schedule is found.

Unfortunately heuristic solvers are generally not able to find optimal solutions for harder problems and one needs to have an extensive understanding of the scheduling problem to be able to improve the quality of the created solutions. This requires a well defined problem model that closely resembles the problem in practice so the right rules-of-thumb can be applied. There are many models that describe scheduling problems. The NedTrain maintenance scheduling problem has finite resources and task ordering constraints. The *Resource Constrained Project Scheduling Problem* (RCPSP) is a well defined model that is well suited to model the NedTrain problem. The mathematical model of RCPSP was first developed by Pritsker et al. [29], and has been studied for many years [5, 16]. In Chapter 2 the RCPSP model will be investigated extensively.

Metrics play an important role in heuristic solvers. First they can help software developers to better understand the scheduling problem, which in turn allows them to create better heuristics and secondly metrics can also be used by heuristics themselves when a decision needs to be made. For example a metric that measures flexibility can be used to measure which heuristic produces a more flexible solution but it can also be used when a decision needs to be made and the possible options need to be compared with each other.

1.6 Conclusion

NedTrain is tasked with the technical maintenance of NS Reizigers rolling stock. To manage deadlines and resources from week to week, NedTrain maintains an extensive week schedule. Currently all four depots, tasked with technical maintenance, create their schedules with the aid of simple planning tools. To be able to continue to compete with the open European market, NedTrain needs to improve its internal processes, including technical maintenance and the scheduling of technical maintenance.

NedTrain wants to improve the scheduling process by using fast and interactive software that will allow schedule makers to create flexible and understandable schedules. To improve fleet availability the schedules needs to minimize the amount of slack in the schedules while still ensuring deadlines are met. They also want to create schedules that maximize resource usage while at the same time allowing maintenance teams enough freedom to resolve small delays themselves.

In this work we will focus on answering the following two questions of NedTrain:

1. How can NedTrain create flexible scheduling in a timely manner?

2. How can NedTrain maximize the freedom of maintenance teams while also maximizing resource utilization?

In Chapter 2 we will study the current and relative literature available for the project scheduling problem at NedTrain. We will conclude the chapter with a more concise formulation of the research questions that we are going to address in this thesis.

Chapter 2

Background

In this chapter we will look at the current research being done in the fields of Operational Research (OR) and Artificial Intelligence Research (AI), that is relevant to the questions at NedTrain. The field of OR is focused on operational aspects of planning and scheduling with real-world objectives, such as minimizing costs or make-span. The field of AI research is more focused on developing algorithmic techniques for scheduling and planning. In Section 2.1 we will look at modeling techniques used in OR to study project scheduling problems. To solve the RCPSP problems fast and efficiently we will introduce heuristic algorithms from the field of AI research in Section 2.4. To improve the flexibility for created schedules we also need to be able to measure flexibility so we will also investigate the current state of the art flexibility metrics in Section 2.7.

2.1 The Resource Constrained Project Scheduling Problem

The *Resource Constrained Project Scheduling Problem* (RCPSP) is a very general scheduling formalism and with a few modifications can be applied to many different real world scheduling problems. Much has been published on RCPSP and its many variations [5, 16] however the definition of RCPSP in its classical form is widely accepted as described in the following paragraph.

A project scheduling problem with resource constraints exists of a few basic elements. The first two most basic elements are tasks and resources. Each project exists of a set of tasks that need to be completed and a set of resources that may be used while a task is being executed. In the context of NedTrain a maintenance task could be replacing air conditioning filters, while a resource could be some specialized maintenance track. Each task has a processing duration while each resource has a finite amount of capacity.

Both tasks and resources are subject to practical constraints. These constraints can generally be split into two types: temporal constraints and resource constraints. Temporal constraints are constraints that restrict the start time of tasks based on time related properties, like deadlines and ordering constraints while resource constraints are associated with the resource properties of the problem instance. For example the amount of resources required by a task and the resource capacity.

An example of a practical temporal constraint at NedTrain would be to first park a train on the maintenance track before bogie (undercarriage) inspections are carried out, while an example of a resource constraint would be that only four bogies can be inspected at any one time because there are only four maintenance tracks.

The goal of the scheduling problem is to create a schedule with a short as possible running time (makespan). Creating a schedule is nothing more than defining start-times for each task so that all practical constraints are satisfied. With a correct schedule each task can be executed with the guarantee that there are enough resources available and that all required preceding tasks are completed. In the following definition we will give a concise and formal definition of RCPSP.

Definition 1 (Resource Constrained Project Scheduling Problem)**.** *An RCPSP instance is a tuple $P = \langle T, R, C \rangle$ where:*

1. *T is a set of n tasks. Each task t_i has a duration $d_i \in \mathbb{R}^{\geq 0}$ and zero or more resource requirements $req(r_k, t_i) \in \mathbb{N}^{\geq 0}$.*
2. *R is a set of m resources. Each resource r_k has a capacity $cap(r_k) \in \mathbb{N}^{\geq 0}$.*
3. *C is a set of precedence constraints between tasks $(t_i \in T \rightarrow t_j \in T)$.*

For each task and resource the following properties must always hold:

1. *For each $(t_i \rightarrow t_j) \in C$, task t_i must be completed before task t_j can start.*
2. *For each resource $r_k \in R$ and concurrent set $T_c \subset T$ the following must always hold $cap(r_k) \geq \sum_{t_i \in T_c} req(r_k, t_i)$, where T_c is a sets of tasks that can be executed concurrently.*

The solution of RCPSP is defined in Definition 2.

Definition 2 (Resource Constrained Project Scheduling Problem Solution)**.** *Given an RCPSP instance $P = \langle T, R, C \rangle$. Each task $t_i \in T$ is given a start-time variable $s_1, \ldots, s_n$, where $s_i \in \mathbb{R}^{\geq 0}$. To solve an RCPSP instance one needs to create a schedule, which is a tuple of start-time assignments $S = (s_1, \ldots, s_n)$ (fixed-time schedule), for which all constraints are satisfied and the make-span of S is minimized.*

A solving technique called *constraint satisfaction* is well suited for solving instances of RCPSP. The technique attempts to solve a problem instance by assigning values to variables in such a way that no constraints are violated. The scheduling problem RCPSP can be modeled as the more general *Constraint Satisfaction Problem* (CSP). Like scheduling there are many problems that are well suited to be modeled using CSP, for example: machine vision, belief maintenance, temporal reasoning, graph problems and floor plan design [19]. Formally, a CSP instance is defined as a tuple $\langle V, D, C \rangle$ where:

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n variables.
- $D = \{D_1, D_2, \ldots, D_n\}$ is a set of n corresponding domains for the variables in V where $v_i \in D_i$.

- $C = \{c_1, c_2, \ldots, c_m\}$ is a set of m constraints. Each constraint is a predicate (or function): $c_k : D_1 \times D_2 \times \ldots \times D_n \rightarrow \{1,0\}$. Simply put, each predicate is a function that takes a set of variable values and returns true if it satisfies the constraint and false if it does not.

A subset of CSP instances can be represented as a graph. This is when all constraints are binary, i.e., each constraint only evaluates at most two variables $c_k : D_i \times D_j \rightarrow \{1,0\}$. A CSP graph would be $G = \langle N, E \rangle$ where every node $n_i \in N$ is a variable $v_i \in V$ and an edge $(n_i, n_j) \in E$ is a constraint $c_k \in C$. Temporal constraints in RCPSP are binary and can be modeled as a graph.

The precedence constraints of RCPSP can be represented as a acyclic graph called the *project graph* [21]. A project graph is a directed graph $G = \langle N, E \rangle$ where the nodes N map to tasks T and edges $E = (t_i, t_j)$ map to the constraints $t_i \rightarrow t_j$. Figure 1.3 shows the project graph that belongs to Example 1.

As defined in Definition 1, the solution of an RCPSP instance is a schedule, which is a tuple of start times $S = (s_1, \cdots, s_n)$. If all tasks are started at their given start time and no tasks undergo a delay, then the project can be executed without any constraints being violated.

There are two states of an RCPSP problem for which no solution can exist. The first case is if there are directed cycles in the project graph. A directed cycle in the project graph means there are at least two tasks that are constrained in two different orderings at the same time. For example, Figure 2.1 shows a cycle between task t_1,t_2 and t_3. If t_2 starts before t_1 the constraint $t_1 \rightarrow t_2$ is violated and if t_1 starts before t_2 the constraints $t_2 \rightarrow t_3 \rightarrow t_1$ are violated. Checking for cycles can be done using a simple *Breadth First Search* (BFS) or *Depth First Search* (DFS).

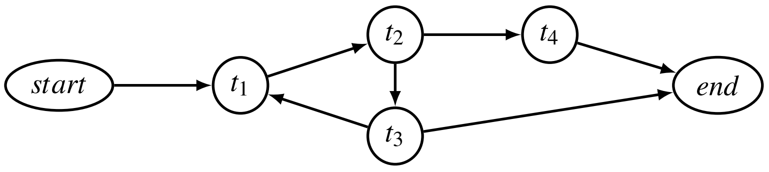


Figure 2.1: Simple project graph with precedence cycle.

The second state for which an RCPSP problem becomes unsolvable is if there is one task that, on its own, requires more resources than the total resource capacity. If this is the case, the task can never be scheduled without violating a resource constraint.

Finding an arbitrary schedule for an RCPSP problem instance becomes trivial if there are no cycles in the project graph and there is no one task that violates a resource constraint. Tasks in a project graph with no cycles can always be ordered into one sequence without violating precedence constraints. This schedule will have no parallel tasks and should then also be resource feasible. However a serial schedule is usually not considered a good solution to the RCPSP problem model because in many practical scheduling problems it is desirable to minimize the make-span. Unfortunately, minimizing the make-span schedule for an RCSPS instance is known to be *NP-hard* [2].

As stated in Definition 1, the classical RCPSP assumes that resources are renewable (the resource becomes available again after use), so it is unable to represent problems with consumable resources (resources that can only be used once). It is also unable to define deadlines or earliest start times of tasks. To solve this there are many extensions and variations of RCPSP. Two common extensions to RCPSP are:

1. the addition of *release times* and *deadlines* to tasks, where the release time is the earliest time a task may start (the time at which the task is released for execution) and the deadline (the latest time a task may finish), or

2. the addition of minimum and maximum time-lags on precedence constraints so a minimum or maximum time can be enforced between two tasks.

The latter of the two extensions is often referred to as *RCPSP/max* and is able to model a very large set of practical problems. However finding a solution is NP-complete, let alone finding an optimal one. The NP-completeness is shown in a proof by Cesta [7]. RCPSP/max is a very generalized problem model which makes it needlessly complex for our purposes.

In this work we will focus on RCPSP where tasks have a release time and a deadline. It is presented by Hartmann [16] as an extension of RCPSP with generalized time constraints. It is not explicitly named so we will refer to it as *Task RCPSP*. The most important aspects of the maintenance scheduling problem is covered by Task RCPSP. At NedTrain the release-time is the time at which a train arrives at the depot and the deadline is when the train leaves the depot. The release-time and the deadline are very strict due to the very busy national railway network and to ensure NSR is able to maintain their own public transport schedule. All the other constraints required by NedTrain are already modeled by RCPSP as defined in Definition 1. Task RCPSP is formally defined as follows:

Definition 3 (Task Resource Constrained Project Scheduling Problem)**.** *A Task RCPSP instance P_T is an extension of an RCPSP instance $P = \langle T, R, C\rangle$, where each task $t_i \in T$ has a release time $rt(t_i) \in \mathbb{N}^{\geq 0}$ and a deadline $dl(t_i) \in \mathbb{N}^{\geq 0}$. It must always hold that a task t_i starts after $rt(t_i)$ and completes before $dl(t_i)$. To solve Task RCPSP one needs to create a schedule as defined 2.*

RCPSP is generally viewed as having two different layers, each with its own set of constraints. The time layer or temporal layer contains the temporal constraint of the RCPSP problem, while the resource layer contains the resource constraints of the RCPSP problem. In the following sections we will have a closer look at both layers and their associated constraints.

2.1.1 Temporal layer

A temporal constraint is applied to the temporal variables of a problem instance, where temporal variables in a scheduling instance are the start times and durations of tasks. In other words temporal constraints restrict the possible values a temporal variable may have. In Task RCPSP there are two types of temporal constraints, deadlines and precedence constraints.

An example of a deadline constraint would be $s_i < dl(t_i)$ where s_i is the start time variable of a task and $dl(t_i)$ is the deadline of a task. This constraint states that the start time of the task must be less than the deadline of the task. An example of a precedence constraint between two tasks t_i and t_j would be $s_i < s_j$, which state that start time of task t_i has to be less than the start time of t_j, in other words task t_i has to start *before* ($\prec$) task t_j. Precedence constraints are generally expressed using this *before* relationship.

Deadline constraints and its counter part release-time constraints (a task may not start before a given release-time) are relatively simple compared to precedence constraints because they only constrain one variable. One precedence constraint by itself is simple enough to satisfy but a set of precedence constraints is able to create a complex network of interdependencies.

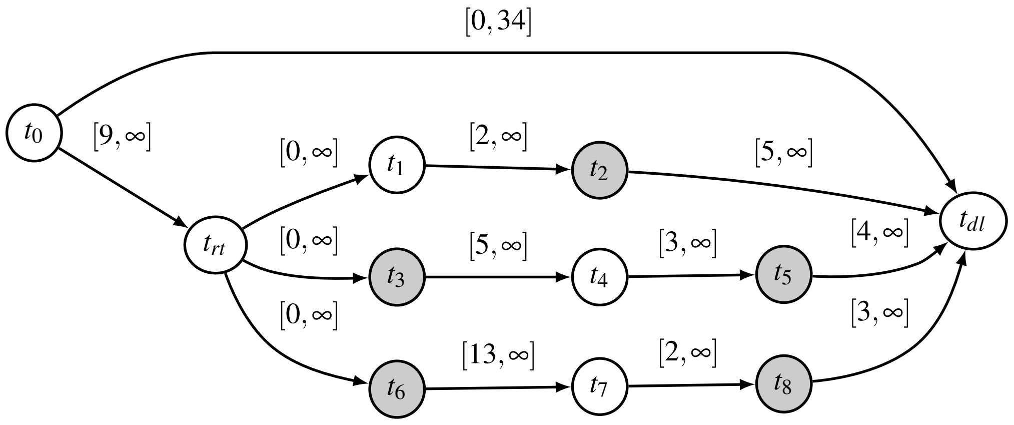


Figure 2.2: Graph representation of all the temporal constraints from Example 1 using minimum and maximum time lags. The release time is constrained by the interval between t_0 and and the dummy node t_{rt} while the deadline is enforced by the interval between t_0 and the dummy node t_{dl}. For more details see Section 2.2.

The temporal constraints in an RCPSP instance can be represented as a graph, where tasks are nodes and temporal precedence constraints are arcs between the nodes. A graph gives a natural view of the before and after relationship between tasks. Other temporal information such as durations, deadlines and release times can be noted in the nodes or encoded in the labels of arcs (see Figure 2.2) as minimum and maximum time lags. We will look more closely at this notation in Section 2.2.

There are many questions one can ask about the temporal layer of an RCPSP instance. To list a few (i) “Given a set of temporal constraints, are they consistent?, i.e, are there no contradicting constraints?”’ and (ii) “What is a possible starting time of a given task?”. A complete list of temporal questions can be found in the work by Planken [23]. When solving an RCPSP instance these kind of questions can be asked many times and therefore it is important that they are answered in a timely and efficient manner.

In the PCP algorithms which we will look at more closely in Section 2.4, it is important to do temporal consistency checks in a timely manner. A temporal consistency check consists of checking if the current temporal constraints still allow for a feasible solution. For example:

if we look at Example 1 and we add an extra precedence constraint between t_1 and t_6 ($t_1 \prec t_6$), is it still possible to create a time feasible schedule, given we if ignore resource constraints? Finding an answer to this question in a timely manner is very important for many PCP algorithms.

The temporal network of Task RCPSP and RCPSP/max will generally hold different information. For example the temporal graph of Task RCPSP is only able to define that two tasks are before or after each other and not able to define a maximum or minimum distance. In Section 2.2 we will look into the temporal constraint model called the *Simple Temporal Network*, which is generally used by RCSPS/max but can also be simplified to be used with Task RCPSP.

2.1.2 Resource layer

There are a few types of resources that can be distinguished: *consumable* or *renewable* resources and unit- or multi-capacity resources. Consumable resources are resources that can be used only once, like a volume of fuel or some construction material. On the other hand renewable resources become available again after a task is completed. Renewable resources can be split into two types: unit- and multi-capacity. A unit-capacity resource only allows one task to require the resource at any one time and a multi-capacity resource allows multiple tasks to use the resource. Unit-capacity resources are often specific machines and staff (man power), while multi-capacity resource are generally a type of machine or a set of tools.

In RCPSP and Task RCPSP, resources are renewable and are multi-capacity. The resources used for technical maintenance at NedTrain are all renewable and multi-capacity resources. In the rest of this work when we refer to resources they will have the above properties.

One of the challenges of resources in project scheduling is being able to find resource over-allocation in a timely manner. It is simple to check if a resource constraint is not satisfied if we are given a set of concurrent tasks T_c and a resource r_k, where $req(r_k,t_i)$ is the amount of resource r_k that is required by task $t_i \in T_c$ and $cap(r_k)$ is the capacity of resource r_k. If it holds that $cap(r_k) \geq \sum\limits_{t_i \in T_c} req(r_k,t_i)$, then the set A_c is resource consistent with respect to r_k. However there may be many different concurrent sets depending on the complexity of the *temporal graph* and finding and checking all possible conflict sets can become intractable with only a few tasks.

We can distinguish at least two general approaches for identifying potential resource conflicts [6]:

profile-based approaches: This approach originates from unit-capacity scheduling solutions where it is a common technique. A resource profile is built as a function of time and resource allocation. Given a maximum allocation of a resource, time points of overallocation can be identified (peaks). Each pair of tasks in a resource peak is considered to be a resource conflict. Figure 2.3 shows a profile of Example 1 where each task is started as soon as all its predecessors have completed. It shows two peaks with three tasks in each peak, which give us six conflict pairs in total. The resource profile shown below uses a single fixed-time solution.

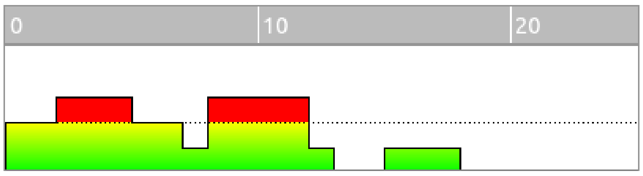


Figure 2.3: Resource profile of Example 1

clique-based approaches: The clique-based approach first constructs a conflict graph. A conflict graph is a graph where nodes are tasks and edges represent two tasks competing for the same resource. To find points of overallocation one needs to find a fully connected subgraph (clique) of tasks that require a greater amount of resource units than the capacity of the resource. Figure 2.4 shows a conflict graph of Example 1 where we can see there are four cliques of size three. These cliques are $\{t_6,t_2,t_3\},\{t_6,t_2,t_5\},\{t_5,t_2,t_8\}$ and $\{t_3,t_2,t_8\}$. With only two resource units each of these cliques is a potential resource conflict, where each edge in a clique represents a conflict pair. The clique-based approach finds two more conflict pairs than the profile approach because it finds *all* possible resource conflicts and not only the conflict using one specific schedule. Finding maximum cliques in graphs is a well-studied subject, however it is also known to be an NP-hard problem [17].

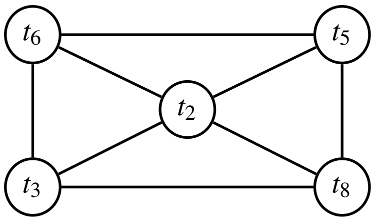


Figure 2.4: Conflict graph for the maintenance track resource of Example 1

In recent literature profile-based approaches have been the main point of focus, because the clique-based approach reduces to the well studied *maximum clique* problem [3] which is known to be a hard problem. A resource profile can be calculated in $O(n \log n)$ time, by iterating over a list of tasks sorted by earliest starting time, while maintaining an ordered set of active tasks sorted on finishing time with the current resource load. While the clique-based approach generally takes more time because it requires us to solve the decision problem of whether a weighted graph (where each weight is the resource usage of a task) contains a clique larger than a given size (resource capacity).

In this work we will focus on the profile approach because it is in the interest of NedTrain for schedules to be created in a timely manner, which is possible with the profile approach and not the clique-based approach.

Calculating an accurate resource profile can only be done on a fixed-time temporal solution. In this solution only temporal constraints are considered in the solving process. The result is a solution where all temporal constraints are satisfied but may still contain resource conflicts. With a fixed-time solution creating a resource profile can be done by finding all currently running tasks at time τ and summing their resource requirements. Given a time point τ and a resource r_k we can calculate the resource profile

$$R_p(\tau, r_k) = \sum_{\{t_i | s_i \leq \tau, s_i + d_i > \tau, t_i \in T\}} req(r_k, t_i). \tag{2.1}$$

Creating resource feasible schedules generally consists of removing all the peaks for a resource profile. Unfortunately, there may be many peaks and removing one may also remove others or even create new ones. Because of this, the choice of which peaks are removed and the order in which the peaks are resolved can impact the quality of the resulting schedule. Many different algorithms have already been created to find the best set of resource peaks, for example the backtracking algorithm by El-Koly [13] or the greedy algorithm proposed by Cesta [6]. We will have a better look at theses algorithms in Section 2.4 but first we will have a look at the temporal model STP, which is used to reason over the temporal constraints defined in RCPSP.

2.2 Simple Temporal Problem

In the previous section we looked at how Task RCPSP defines it temporal constraints but did not look at why only simple precedence constraints and deadlines are used. In this section we will have a closer look at other temporal models to better understand which abstractions are made and why. First we will have a look at the kind of time constraints that exists in real world projects and then look at how we can model them. Finally we will present the well studied temporal model called *Simple Temporal Problem* (STP).

Most temporal models have variables that represent events or the start times of tasks. To solve the problem model an assignment of start times needs to be found that satisfies all temporal constraints. If this is the case the temporal problem instance is consistent.

The best way to find the different kind of temporal constraints in a real life project is to look at an example of a real life project. Example 1 only describes constraints that can be modeled by Task RCPSP but we can take it as a starting point. Lets add some more details to the example by stating that tasks t_1, t_2 are painting tasks that, for some reason or other, requires the same brush and paint. This means that t_1 and t_2 can not be done at the same time but once one is done the other must directly follow. We can also state that t_4 can only be carried out every five hours due to some safety issues.

The above complications may seem hard to model but can actually be quite easily be modeled using inequalities. For example the paint constraint can be enforced using the following $\tau_1 + p_1 = \tau_2$ where τ_1 and τ_2 are the start events of tasks t_1, t_2 and p_1 is the duration of task t_1. The constraints of task t_4 can also be enforced using a set of inequalities that define a set of time intervals.

We can model very complex temporal problems using *Temporal Constraint Satisfaction Problem* (TCSP), described by Dechter [12]. It models each precedence constraints with a set of intervals that defines the minimum and maximum amount of allowed time between the two constrained tasks. With a dummy start task and the intervals all the above constraints can be described as inequalities. Unfortunately finding an assignment of start times for an TCSP instance is NP-Hard. So creating a schedule with only temporal constraints will already take a long time let alone that the scheduling problem at NedTrain also has resource constraints.

Fortunately Dechter also introduces the Simple Temporal Problem. This model is much simpler so it is unable to model the temporal constraint needed for t_4 but is able to model the constraints needed by t_1 and t_2. This simplification allows us to check if a problem instance is consistent in polynomial time [23].

STP is a simplification of TCSP where it only allows each precedence constraint to define one time interval between tasks. This interval describes the minimum and the maximum amount of time allowed between the two tasks. For tasks t_1 and t_2 the interval between them would be $t_1 \rightarrow t_2 = [0,0]$. It states that there must be a minimum and maximum of zero time units between t_1 and t_2, which means t_2 must directly follow t_1. A formal definition is given below.

Definition 4 (Simple Temporal Problem (STP))**.** *An STP instance is a tuple $P = \langle T,C\rangle$ where $T = \{\tau_1,\ldots\tau_n\}$ is a set of n time point variables and C is a set of binary constraints. For each constraint $c_{ij} \in C$ with $c_{ij} = (a_{ij},b_{ij})$ the following must hold: $-\tau_i + \tau_j \leq b_{ij}$ and $\tau_i - \tau_j \leq -a_{ij}$.*

A solution S for P is an assignment of values for the time point variables in T, such that all constraints hold.

The possible values for the time points $\tau_1,\ldots,\tau_n$ can be restricted using the given intervals. This allows for many temporal restrictions to be applied to the start time of a task. For example the constraint $c_{ij} = (0,0)$ would imply that time point τ_i and time point τ_j have to occur at the same time or the constraint $c_{ij} = (0,10)$ would imply that time point τ_j can occur at the same time as τ_i or at most 10 units of time later.

The STP can also be represented as a constraint graph called *Simple Temporal Network* (STN) as is shown in Figure 2.5, where time points are represented as nodes and the constraints as labeled edges.

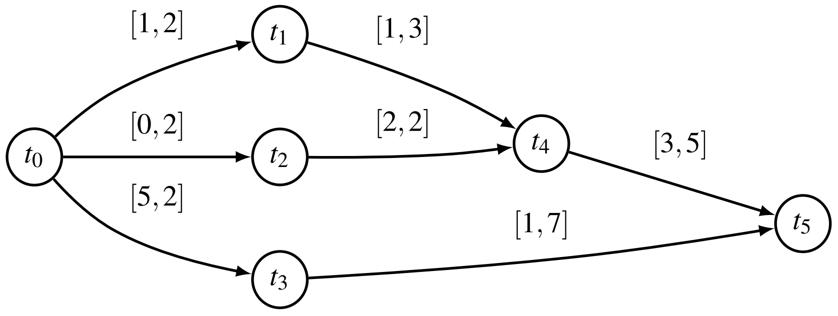


Figure 2.5: A Simple Temporal Network for Example 1

where resource constraints are not considered.

The STP problem can be solved using linear solvers because each constraint can be represented as the following two inequalities:

$$-\tau_i + \tau_j \leq b_{ij}, \tag{2.2}$$

$$\tau_i - \tau_j \leq -a_{ij}, \tag{2.3}$$

where τ are continuous variables in the linear model.

However the temporal constraints only allow a restricted set of linear inequalities to be added to the linear solver. Algorithms that take the special case of temporal constraints into account have been shown to be much faster than linear solvers [24, 10]. These special case algorithms generally work using a *distance graph* (dGraph) as defined below.

Definition 5 (Distance Graph (dGraph))**.** *Given an STP instance $P = \langle T, C\rangle$, a dGraph $G_d = \langle N, E\rangle$ is a directed graph with weighted edges, where N is a set of nodes representing the time point variables in T and E is a set of directed edges. Each edges $e_{ij} \in E$ has a weight w_{ij}. Each constraint $c_{ij} \in C$ with $c_{ij} = (a_{ij}, b_{ij})$ is mapped to two weighted edges $e_{ij}, e_{ji} \in E$ where each edge has the following weights $w_{ij} = -a_{ij}, w_{ji} = b_{ij}$. See Figure 2.6 for an example of a dGraph.*

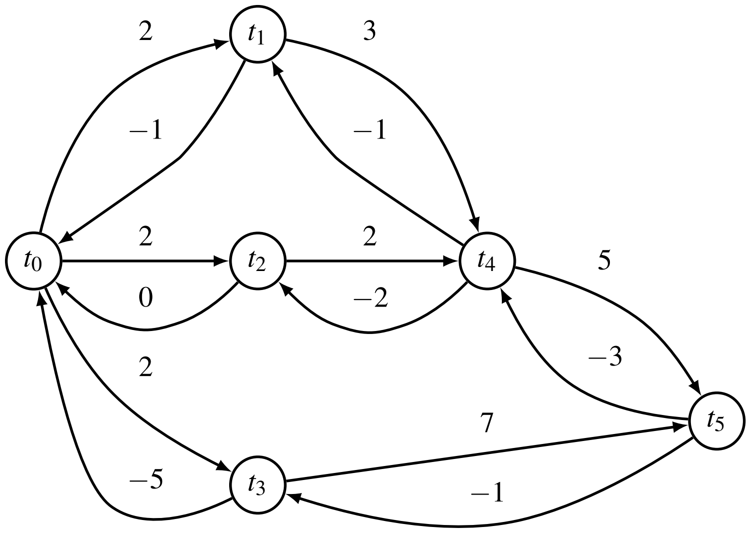


Figure 2.6: A dGraph of the STP instance shown in Figure 2.5.

If we modeled the NedTrain scheduling problem using RCPSP/max we could use an STN as the time layer. However, as stated in the previous sections Task RCPSP has enough expressive power to support the scheduling problem at NedTrain. The temporal layer of Task RCPSP is simpler than that of RCPSP/max and so we do not need to model it using STP. We still looked at the STP model because many flexibility metrics which we will look at in Section 2.7 use this model.

2.3 Scheduling solutions

Until now we considered a solution of a scheduling problem to be a schedule and the only schedule we introduced was a fixed-time schedule. A fixed time schedule does not allow for any flexibility by definition so other notions of schedules are needed to allow for flexibility. In this section we discuss two different schedule notations that allows tasks to undergo some delay without breaking the schedule. Using schedules that allow tasks some delay without breaking the schedule is important for the execution uncertainty of the maintenance projects at NedTrain. We will have a closer look at what flexibility means at NedTrain in Section 2.7, but for now we will first focus on the notation of flexible schedules.

The schedule defined in the definitions of RCPSP and STP is a mapping of tasks to start-times, $S : T \rightarrow \mathbb{R}^{\geq 0}$. This is called a *fixed-time schedule*. It assigns fixed inflexible times to tasks and any deeper information about the problem that was found during the solving process is lost. The advantage of fixed-time schedules is that all temporal or resource information can be ignored as long as the tasks are executed at the given start time and do not suffer unexpected delays. However it will be unable to continue with just the smallest delay. Even if a small delay would not break the process, the system can not continue because it is unaware of how the delay will impact the execution of other tasks.

When working with dynamic problems, fixed-time solutions fail to provide the needed flexibility. To solve this problem the scheduling process is split into two phases. In the first phase the scheduling problem is partly solved and the second phase, which is often an on-line process, converts the partial solution to a fixed-time solution. This allows the first phase to hold extra in-depth information of the problem and the second phase is capable of quickly generating new schedules without having to reanalyze the original problem constraints. The second phase need not necessarily be done by a computer, if a flexible schedule can be clearly communicated it can also be done by work crew during project proceedings.

We will focus on two partial solution representations in this work. In Section 2.3 we will look at the interval schedules which is used by Endhoven and others [14, 36] and in Section 2.3 we will look at the very well known Partial Order Schedules or POS which has been extensively studied by Policella [27, 25].

Interval schedules

Endhoven introduces an *open schedule* which, unlike a fixed-time schedule, does not assign one time point to each task. Each task is assigned an interval of time points. Each task is able to start at any time point within the interval, without having to consider the start-times of other tasks. For example if the intervals of two tasks overlap then they may start in any order without having to take resources or precedence constraints into account. The definition below shows the formal definition of an *Interval Schedule*.

Definition 6 (Interval schedule)**.** *Given an STP instance $P = \langle T, C \rangle$. An interval schedule S is a set of intervals $S = \{I_1, \ldots, I_n\}$ for each time-point variable in T, with $I_i = [a_i, b_i]$ where $a_i \leq b_i$ and for all $c_{ij} \in C$ its holds $b_i \leq a_j$.*

S is a valid interval schedule for P iff for all values between a_i, b_i is a possible assignment of the variable $\tau_i \in T$ without violating the temporal constraints in P.

Interval schedules allow for much more flexibility than fixed-time schedules, however it does not retain precedence information. So like in the case of the fixed-time schedules, if a interval deadline is violated the schedule will break but the original problem might still be feasible, unfortunately without precedence information this can not be known. In the next section we will look at *Partial Order Schedules* which does maintain structural data.

Partial Order Schedules

A fixed-time schedule is brittle when faced with unexpected and unpredictable execution changes due to each task having a fixed start- and end-time. In some practical cases when working with people the brittleness could be much less due to a person being able to oversee the implications of starting a bit earlier or finishing a bit later. However, working with more complex problems or when scheduling for machines that are unable to make small adjustments, an outdated fixed-time schedule can be a significant problem. Policella et al. [28] introduce the Partial Order Schedule (POS), which is able to introduce robustness and stability into a schedule. A POS gives only a partially ordered set of tasks without any fixed start times, using simple precedence constraints. For each task it is well defined which predecessor tasks needs to be completed first. This allows on-line algorithms to propagate delays and update outdated fixed-time schedules quickly. A POS can also be used without an on-line component by simply letting maintenance teams on the shop-floor start each task as soon as possible.

A more formal look at the POS is to see it as a graph that defines a set of feasible fixed-time schedules. During the execution of the project the set of solutions is reduced as tasks are complected and delays are propagated to future tasks. Policella formally defines a POS as follows:

Definition 7 (Partial Order Schedule (POS))**.** *A POS $G = \langle N, E\rangle$ is a directed graph, where N is a set of nodes representing temporal variables $\tau_1, \dots, \tau_n$ and E is a set of edges representing binary precedence constraints between two temporal variables $(\tau_i \to \tau_j)$. A schedule S of G is an assignment of values to N for which all precedence constraints hold. G is a valid POS of a given RCPSP instance P iff all possible schedules of G also satisfies all constraints in P.*

The robustness of a POS is gained because any external changes can be propagated in the underlaying temporal network. Any resource constraints can be ignored during propagation because the POS by definition only contains resource feasible schedules. At some point new external changes can no longer be propagated, at which point a new schedule needs to be calculated from scratch.

In the following sections we will look at how solutions are found using a technique called Precedence Constraint Posting.

2.4 The PCP paradigm

The complexity of solving RCPSP instances emerges from the set of resource constraints. Temporal constraints are much simpler to satisfy because time is assumed to be linear which allows for constraints to be propagated which reduces the search space from NP to P. Polynomial techniques for solving temporal constraints is shown in Section 2.2. Resource constraints do not have this property so search algorithms need to branch more. A branch in a search algorithm refers to a point in the search process where two or more different search directions are available and for which it is non trivial to calculate which direction will lead to a more optimal solution. Search spaces with many branches are generally much harder to navigate efficiently.

A technique called *Precedence Constraint Posting* (PCP) attempts to simplify a resource constrained problem by reducing it to only a temporal constrained problem. This technique has been well studied over the past few years [6, 31, 22]. By re-encoding the resource constraints into temporal constraints the problem becomes much easier to solve and allows even on-line solvers to generate stable and robust schedules. However the reduction step done by PCP is still a complex problem.

In classic RCPSP temporal constraints consist of precedence constraints, with no deadline. Checking the consistency of precedence constraints is a simple task. After constructing a directed graph from the precedence constraints, one only needs to note that the graph contains no cycles. If no cycles exist there exists a feasible temporal solution. Finding this solution can be done in linear time using a topological ordering algorithm.

The driving idea behind PCP is the fact that finding a temporally consistent solution is a simple task. Satisfying resource constraints has been found to be a complex problem. PCP attempts to solve this by continually adding extra precedence constraints to the time layer until all possible temporal consistent solutions are also resource consistent. For example, given two conflicting tasks trying to use the same resource unit, a precedence constraint can be posted between them so they are unable to execute at the same time, which solves the resource conflict. Simply put PCP attempts to convert a scheduling instance with resource constraints and temporal constraints into a problem with only temporal constraints.

A basic greedy PCP solver has two points at which it needs to make a search decision. The first decision is made after a set of conflicts has been found. If the set is non-empty it needs to select a conflict to solve. Once a conflict has been chosen a precedence constraint needs to be selected that will resolve the conflict. Picking a constraint is the second decision that needs to be made. In the following algorithm the basic structure of a greedy PCP solver is shown.

Algorithm 1: Greedy PCP algorithm

```
Input: A problem $P$
Output: A schedule $S$
$S \leftarrow P$
while true do
    $conflictSet \leftarrow$ FIND-RESOURCE-CONFLICTS($S$)
    if $conflictSet = \emptyset$ then return $S$
    if UNSOLVABLE($conflict - set$) then return failure
    $conflict \leftarrow$ SELECT-CONFLICT($conflictSet$)
    $constraint \leftarrow$ SELECT-PRECEDENCE-CONSTRAINT($conflict$)
    $S \leftarrow$ POST-PRECEDENCE-CONSTRAINT($S, constraint$)
```

The PCP solver continues posting extra precedence constraints until all resource conflicts are gone. It is possible for an unsolvable conflict to arise, if this is the case the problem has become unsolvable and the algorithm fails. An example of an unsolvable conflict would be if two tasks t_1 and t_2 are both competing for the same resource and can only be resolved by a posting a precedence constraint but existing constraint do not allow it. So the conflict pair t_1,t_2 are unsolvable and so the problem in its whole become unsolvable.

Finding the right precedence constraint to post has opened the way for many new heuristics. A well known heuristic is to select a conflict pair and constraint that has the least amount of negative impact on the temporal flexibly of the problem [30, 9]. This is often referred to as the *least commitment principle*. In the following subsections we will look at some variations of this heuristic proposed in literature.

The first step in the PCP approach is to find resource conflicts, where a resource conflict is generally a set (*conflict set*) of tasks competing for the same scarce resource. As mentioned in Section 2.1.2, there are two popular approaches to finding conflict sets, (i) using a resource profile, and (ii) using cliques of a sufficient size in a conflict graph. We will focus on the resource profile approach because the CFSA and ESTA algorithms, which we will discusses in Sections 4 and 2 respecify, use this approach to find resource conflicts and calculating a resource profile is generally faster than finding cliques in a conflict graph.

Resolving a conflict set (which also removes a resource peak) can be done by simply selecting two tasks at random in the conflict set and then posting a constraint between them and then continuing to post constraints until the conflict set is resolved. However this gives bad results because the posted constraint might have very little impact in solving the set. This is shown in the example below.

Example 2 (Using a random conflict selection method.)**.** *Take tasks t_2,t_5 and t_8 from Example 1 as a conflict set, where each task requires the following amount of maintenance tracks:* $1,2,1$*, respectively. The depot has* 2 *maintenance tracks available. If the pair t_2 and t_8 is selected and a constraint is added between them, the conflict will not be solved, because the combined capacity of the tasks does not exceed the resource capacity. It is task t_5 in combination with t_2 or t_8 that is responsible for the resource violation. So the pair t_2 and t_5 would be a better conflict pair to select. Selecting* wrong *constraints will cause the PCP algorithm to add extra possibly unneeded constraints.*

The above example shows there is a clear good and bad pair of tasks to select. However in bigger problem instances this is generally not as clear. So when selecting a conflict or resolving a conflict the *least-commitment strategy* is generally applied. Within the domain of constraint satisfaction problems the least-commitment strategy is the strategy of making choices that minimize the amount of restrictions added to a set of possible solutions. In other words solvers try not to commit too much to one possible set of solutions. This allows the solver to avoid dead-ends and increases its chances of finding a solution.

While following the least-commitment strategy increases a solvers chances of finding a solution it does not directly help to find solutions with a minimum make-span or increased amount of flexibility. There are two popular approaches when selecting a conflict pair. One method uses *Minimal Critical Sets* (MCS) to filter the number of conflicting pairs and the other simply attempts to pick the right conflict pairs using heuristics (*pair-wise conflict resolution*). The pair-wise approach is much faster than using MCS but has the risk of posting more constraints than needed, as is the case in Example 2 where it first resolves the conflict t_1,t_2 which reduces the size of the resource peak but does not remove the conflict set. The use of MCS was proposed by Laborie and Policella [20, 8], while direct pair-wise conflict resolution was first proposed by Smith [30]. In the following two sections we will have a closer look at the two conflict resolution techniques.

2.4.1 Minimal Critical Set

When solving a scheduling instance many resource conflict sets can be found and only some of them need to be resolved to create a usable schedule. A PCP solver can use *Minimal Critical Sets* (MCS's) to filter the number of conflict sets found in a resource peak, which will help it to solve the problem instance more effectively.

The concept of a MCS is simple. An MCS is a set of tasks for which the following two properties hold: (i) a set of tasks that can be executed at the same time and together require more of a resource than the resources capacity, and (ii) any strict subset of the tasks does not constitute a resource conflict i. e. if only one of the tasks is executed before or after one other task, there will no longer be a resource conflict. The formal definition of a MCS is given below.

Definition 8 (Minimal Critical Set). *Given an RCPSP instances $P = \langle T, R, C \rangle$ and a set $T_c \subset T$ of concurrent tasks. T_c is an MCS iff:*

1. *T_c is a resource conflict: $cap(r_k) < \sum_{t_i \in T_c} req(r_k, t_i)$, and*
2. *$\forall T'_c \subset T_c$, T'_c is not a resource conflict.*

It is proposed by Laborie and Cesta [20, 8] to only resolve conflicts that are also an MCS. This allows for less constraints to be posted because each posted constraint is guaranteed to resolve the conflict set. However if a resource peak is very large the peak may contain multiple MCS and not all of them need to be resolved because there may be a subset of tasks that can be found in more than one MCS. Due to this overlap resolving one MCS may also resolve another. Picking the right MCS to resolve first becomes an issue.

When picking a MCS the solver will generally want to resolve the most critical MCS first because once only one MCS becomes unresolvable the problem instance as a whole will also become unresolvable. A more critical MCS is considered a higher quality MCS.

The quality of an MCS is measured by the number of valid precedence constraints that are allowed to resolve the MCS. A high quality (critical) MCS allows very little constraints to resolve the MCS. In other words each MCS has a set of precedence constraints and the MCS can be resolved if only one is posted. However the size of this set varies, where a low number of constraints constitutes a high quality MCS. A high quality MCS has a higher chance of becoming unsolvable than a less critical one. By solving the most critical first the chance of unsolvable conflicts to arise later on in the PCP process is reduced.

Once the most critical MCS has been found a precedence constraint needs to be selected to resolve the MCS. A critical MCS will generally have only a few valid precedence constraints that can be posted. When following the least-commitment strategy the solver will generally select the constraint that removes the least amount of slack from the underlying temporal graph.

Unfortunately, finding the most critical MCS is an intractable problem for larger scheduling instances, which is the case at NedTrain, because there are an exponential number of

MCS's to the number of tasks in a resource peak: $\sqrt{n} \cdot 2^n$. While it is not strictly necessary to enumerate all MCS's to find an MCS that is of sufficient quality, as is shown in [22]. But this is only an approximation and is less effective than compared to using the most critical MCS.

2.4.2 Pair-wise conflict resolution

Pair-wise conflict resolution is a simpler approach to resolve resource conflicts above. It creates a set of conflict pairs from a given conflict set. Each conflict pair competes for the same scarce resource and only some of them need to be resolved to remove a resource peak. A conflict pair (t_1, t_2) is resolved by posting a constraint $t_1 \rightarrow t_2$ or $t_2 \rightarrow t_1$.

To prune the search space early and to make better search choices each conflict pair can be classified into four different cases. Using a temporal network like an STN the relationship between the two tasks in a conflict pair is defined. The classifications are mutually exclusive and every conflict pair will always satisfy one of the cases. The four cases are defined below given two tasks t_1 and t_2 in a conflict pair:

Case 1. Task t_1 cannot be before t_2 and t_2 can not be before t_1

Case 2. Task t_2 cannot be before t_1

Case 3. Task t_1 cannot be before t_2

Case 4. Both orderings are allowed.

By classifying all the conflicts early pruning can be done by removing all conflicts of Case 1 because any extra posted constraint will create an inconsistent temporal network. Cases 2 and 3 are trivial to resolve because only one ordering is possible. Case 4 conflicts are open to any ordering and often requires more calculations to resolve.

Unlike MCS finding all pair-wise conflicts is tractable due to the number of pairs being bounded by $O(n^2)$. Given a conflict set of n tasks the maximum number of conflicts would be $n(n-1)/2$. This makes pair-wise conflict resolution much faster but does so at the expense of occasionally posting too many constraints. The effectiveness of pair-wise conflict resolution is dependent on the heuristics used when selecting the next conflict pair to resolve. In the following section we will look at some methods to rank conflict pairs.

2.4.3 Heuristics for selecting pair-wise resource conflicts

Many different heuristics have be proposed for selecting resource conflict pairs. In this section we will look at two different heuristics used for selecting conflicts. One was proposed by Smith [30], which is referred to as *Min-Slack* and the other, proposed by Cesta and Smith, as ω_{res}.

Min-Slack was introduced first and it uses the slack between two tasks in a conflict pair to rank them.

Slack is the amount of temporal space between two tasks, see Figure 2.7. The slack between tasks is found by maintaining lower- and upper-bound start times of each task. The *earliest start time* (est) is the earliest possible starting time a task may have without violating

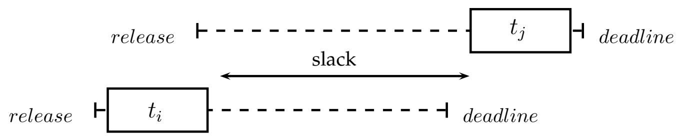


Figure 2.7: Slack between tasks $t_i \prec t_j$.

temporal constraints, while the *latest finishing time* (lft) is the latest possible finishing time a task may have without violating temporal constraints. Using est and lst the slack is calculated as follows:

$$slack(t_i \prec t_j) = lft(t_j) - est(t_i) - (d_i + d_j). \tag{2.4}$$

The heuristic ranks all conflict pairs (t_i, t_j) based on its slack, starting with the least amount of slack, as follows:

$$\min_{(t_i,t_j)} \{\min(slack(t_i \prec t_j), slack(t_j \prec t_i))\}. \tag{2.5}$$

The conflict with the least amount of slack is considered most critical because it already has very little freedom and stands a greater chance of contributing to an unsolvable resource peak. The strategy is to resolve the most critical conflicts first, such that they will not cause problems in future iterations of the PCP process.

In later work by Cesta and Smith [6] another heuristic is proposed that selects conflicts based on a flexibility metric which they refer to as ω_{res}. It measures the temporal distance between two tasks in a conflict and like *Min-Slack* selects the conflict with the least amount of flexibility. The ω_{res} is calculated using a distance graph (see Definition 5). The distance graph only describes time points and not tasks with a duration, so each task is split into a start-time and end-time where the start-time is $\tau_i = s_i$ and the end-time is $\varepsilon_i = s_i + d_i$. Formally ω_{res} the calculated as follows:

$$\omega_{res}(t_i, t_j) = \begin{cases} \frac{\min\{d(\varepsilon_i,\tau_j), d(\varepsilon_j,\tau_i)\}}{\sqrt{S}} & \text{if all conflicts are case 4} \\ \min\{d(\varepsilon_i,\tau_j), d(\varepsilon_j,\tau_i)\} & \text{else} \end{cases} \tag{2.6}$$

where $S = \min\{d(\varepsilon_i,\tau_j), d(\varepsilon_j,\tau_i)\}/\max\{d(\varepsilon_i,\tau_j), d(\varepsilon_j,\tau_i)\}$.

If all conflict sets are not of case 4 the metric behaves the same as the slack metric. However if all conflict sets are of case 4 it divides the score by S to take into account both orderings of the tasks ($t_i \rightarrow t_j$, $t_j \rightarrow t_i$) and not only the ordering with the minimum slack. In other words, for conflict pairs of case 2 and 3 there is only one valid ordering for tasks and we can simply score the one ordering, however when all conflict pairs are of case 4 there are two possible orderings of which one might be critical but the other not. To take this imbalance into account the metric adjusts the score using S. Below is an example of a situation that shows the need for S.

Example 3 (Situation where S adjusts ω_{res})**.** *Suppose two pair-wise conflicts are found:* (t_i, t_j) *and* (t_k, t_l)*. The* ω_{res} *without the* S *of both conflicts would be* $\min\{d(\varepsilon_i,\tau_j), d(\varepsilon_j,\tau_i)\} = \min\{d(\varepsilon_k,\tau_l), d(\varepsilon_l,\tau_k)\} = 2$. *Both conflicts are the same, however if we look at the two* *conflicts shown in Figure 2.8, we can intuitively see that conflict (t_k, t_l) is much more critical than conflict (t_i, t_j). For this reason the S divider was added. By dividing by S the score is scaled by the amount of slack of the maximum ordering. Now $\omega_{res}(t_k, t_l)$ scores $2/\sqrt{4}$ which is greater than the score of $\omega_{res}(t_i, t_j)$, which is $2/\sqrt{20}$.*

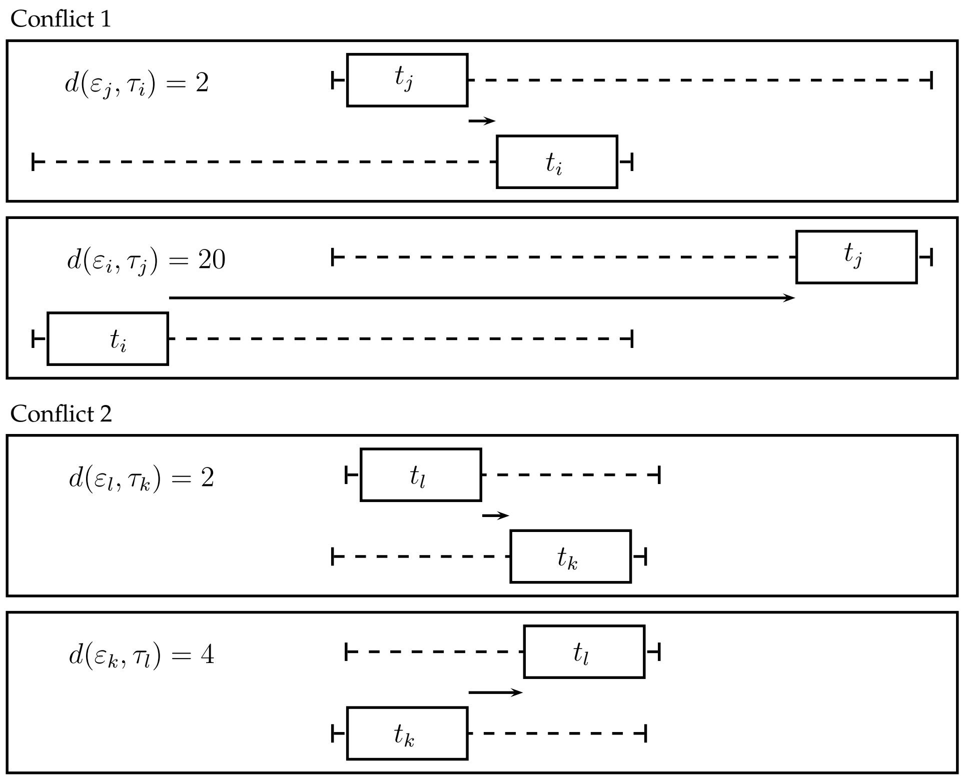

Conflict 1
$d(\varepsilon_j, \tau_i) = 2$
t_j
t_i
$d(\varepsilon_i, \tau_j) = 20$
t_j
t_i
Conflict 2
$d(\varepsilon_l, \tau_k) = 2$
t_l
t_k
$d(\varepsilon_k, \tau_l) = 4$
t_l
t_k

Figure 2.8: Two pair-wise conflicts that would tie using ω_{res} without the *S* divider.

Once a conflict pair has been selected a precedence constraint (resolver) has to be chosen to resolve the conflict. In the next section we will look at some heuristics used to solve conflicts.

2.4.4 Heuristics for conflict resolution

In the previous sections we looked at the heuristics used by a PCP solver to find conflict sets and to select conflict pairs. In this section we will look at heuristics used to resolve the conflict pairs.

Once a conflict pair has been selected a resolver needs to be selected. A *resolver* is simply a constraint $t_1 \prec t_2$ or $t_2 \prec t_3$ which needs to be added (posted) to the temporal graph to resolve the conflict, where the pair (t_1, t_2) is selected using one of the heuristics described in the previous section. The *constraint selection function* only needs to make a choice between the ordering of the two tasks.

When selecting a resolver, one generally picks the precedence constraint that is least restricting. This is done by looking at which of the two options remove fewer solutions from the solution space. However the solution space is too complex to measure so the impact of a newly added precedence constraint is hard to understand or measured. Heuristics are generally used to estimate the impact of each precedence constraint and a decision is made based on this estimation.

By posting the right constraint the amount of commitment to one set of possible solutions is reduced. Laborie [20] defines commitment as

$$commit(P,c) = 1 - \frac{card(INST(P \cup c))}{card(INST(P))}, \tag{2.7}$$

where P is a temporal problem, c is a constraint and $INST(P)$ is the set of possible solutions that satisfy the constraints in P. The $commit(P,c)$ metric is 0 iff the precedence constraint c is redundant and 1 iff the addition of c creates an inconsistent problem P.

However finding $INST(P)$ or even just finding its size is an intractable problem and so the commitment function is often estimated based on the amount of slack each task has in the temporal problem.

In the work by Cesta [6] the amount of slack that is lost by posting a precedence constraint is minimized. This is based on the assumption that a partial solution with a high level of slack represents a larger amount of different solutions. The constraint selection function selects the constraint c_i with the maximum amount of slack, as follows:

$$c_i = \begin{cases} a_i \prec b_i & \text{if } slack(a_i,b_i) > slack(b_i,a_i) \\ b_i \prec a_i & \text{else} \end{cases}. \tag{2.8}$$

This method of constraint selection is used by Smith, Cesta and Policella [30, 6, 25].

2.4.5 Conclusion

In this section we introduced the PCP paradigm. It exploits the separation of temporal constraints (precedence and deadlines) and resource constraints in the RCPSP model. It first identifies resource violations and then attempts to reduce the violation by adding extra temporal constraints in the form of precedence constraints. The temporal network will become a POS once all resource violations have been removed and the temporal network itself is still consistent.

We showed that the PCP process is an iterative process where in each loop the resource constraints are analyzed and precedence constraints are posted to the temporal network. There are two main branch points in the search space of the solver, at the first branch point the solver needs to select two tasks competing for the same resource and at the second branch point it needs to select a precedence constraint that will resolve the conflict.

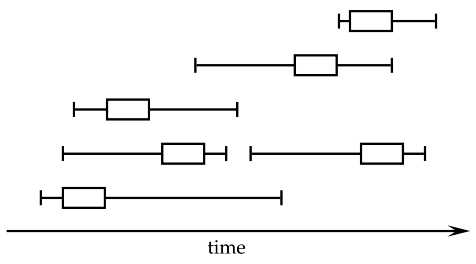
In the work by Evers [15] it was shown that the PCP paradigm is well suited for the detailed schedules of technical maintenance at NedTrain. The created POS is able to handle small changes without the need of rescheduling. At some point a change might occur that the POS can not handle and a rescheduling step will be needed. However a complete rescheduling may not be necessary. The unresolvable POS can have some of its precedence constraints removed given back to the PCP solver. If the right constraints are removed the PCP solver may be able to create a new POS very similar to the original.

In the following two sections we will look at two different PCP algorithms. First we will look at a very straightforward implementation called *Conflict Free Solution Algorithm* (CFSA) then we will look at *Solve and Robustify*.

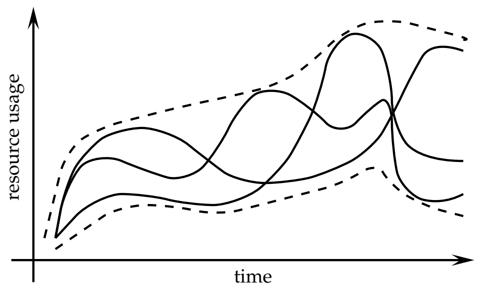
2.5 Conflict Free Solution Algorithm

The *Conflict Free Solution Algorithm*(CFSA) takes a bounding approach to solving RCPSP problems. It starts with the original problem constraints for which it is hard to find a solution and continues to add new constraints until finding a solution becomes easier. To be more precise, we keep adding precedence constraints until all resource constraints become redundant. To be able to make the resource constraints redundant, all possible solutions allowed by the time layer must satisfy the resource constraints.

CFSA uses a resource profile to find pairs of tasks that require the same resource at the same time. However, to be able to create a resource profile that can find the exact number of conflict pairs, it needs to know the exact starting time of tasks, but CFSA does not maintain the exact starting time of tasks, it only maintains temporal bounds in the form of an STN. With a temporal network it is intractable to construct an exact resource profile because of possible interactions between tasks. Instead CFSA maintains two profiles: the worst case profile (upper-bound) and a best case profile (lower-bound), which is shown by the dashed lines in Figure 2.9b.



(a) Bounded time solution.



(b) Bounded resource profile.

Figure 2.9: An example of temporal bounds (a) and resource profile (b) used by CFSA.

Using this bounded profile three types of resource peaks are distinguished:

lower-bound-peak: the lower-bound profile equals the resource capacity and the upper-bound profile is greater than the resource capacity.

unresolvable-peak: the lower-bound profile is greater than the resource capacity.

upper-bound-peak: the lower-bound profile is less than the resource capacity and the upper-bound profile is greater than the resource capacity.

If an *unresolvable-peak* is found the algorithm stops and if any unresolvable-peaks appear while it is removing other peaks it will also fail. To reduce the chances of unresolvable-peaks from appearing it will always try to remove *lower-bound-peaks* first. It does so by posting a precedence constraint between conflicting pairs of tasks. Only when there are no lower-bound-peaks will it continue to remove *upper-bound-peaks* in the same manner. Once all peaks are removed the temporal network will be a valid POS.

Unfortunately, CFSA has the tendency to post too many precedence constraints by using a bounding approach based on an estimated upper- and lower-bound of the resource profile. The resulting POS is over constrained and represents only a small set of possible schedules. In the next section we will look at another approach for which it is known to produce a less constrained solution.

2.6 Solve and Robustify

The *Solve and Robustify* approach proposed by Policella et al. [26] overcomes the uncertainty that CFSA struggles with, by splitting the solving process into two phases. Instead of using a bounding approach Solve and Robustify first attempts to find a single fixed-time schedule and then expands it to a POS. In other words the first phase solves the problem and the second phase adds flexibility or robustness, as is shown in Figure 2.10. Both phases use a PCP approach but handle resource constraints in a different manner. Policella shows that by splitting up the task of finding a solution and creating a robust solution, it allows the algorithm to focus more on one task during each phase, allowing it to select constraints in a more efficient manner. The first phase uses the *Earliest Start Time Algorithm* and then uses the *Chaining Algorithm*, which we will look at in more detail in the following sections.

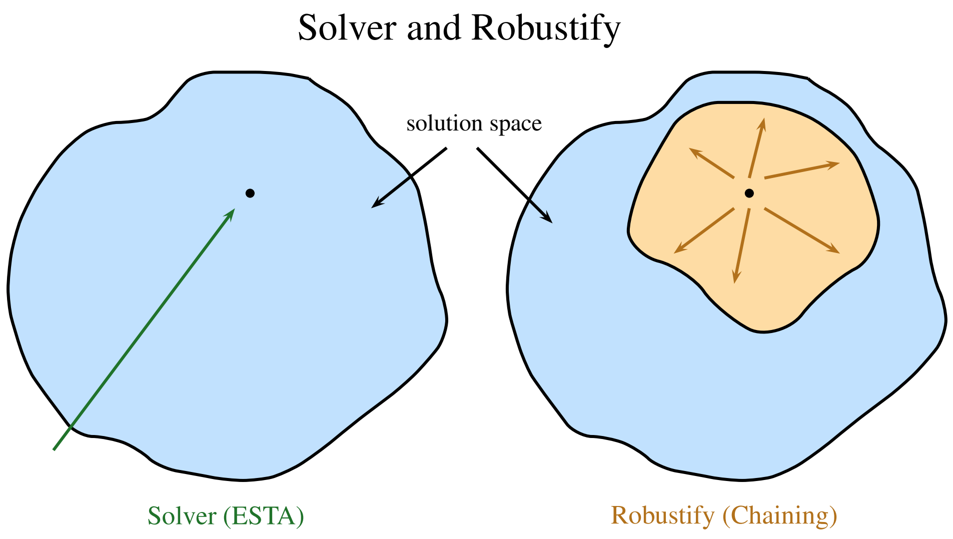


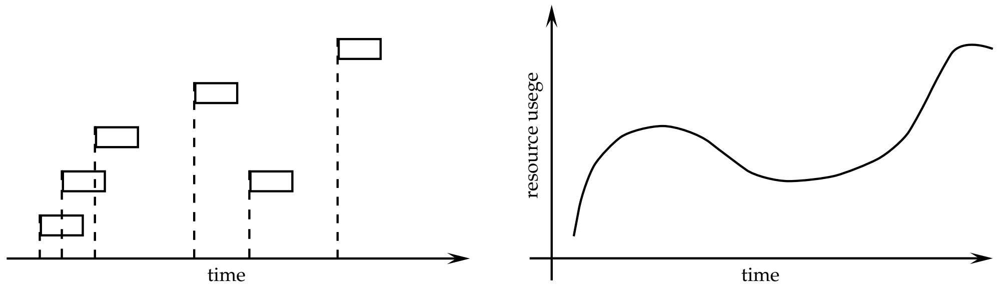
Figure 2.10: Shows the two phases of Solver and Robustify.

2.6.1 Earliest Start Time Algorithm

The *Earliest Start Time Algorithm* (ESTA) [6] is a PCP based algorithm which was first introduced to solve the *Multiple Capacitated Metric Scheduling Problem* (MCM-SP). The MCM-SP problem model is, with exception to a few minor differences, the same as RCP-SP/max. In later work by Policella [25] the ESTA algorithm is used to solve RCPSP/max. RCPSP/max is a generalization of Task RCPSP, so ESTA can be used to solve this problem model as well.

ESTA finds solutions in very much the same way as CFSA but only allows tasks to start at the earliest time allowed by the temporal network (see Figure 2.11a), it therefore only finds one fixed-time solution. With a fixed-time schedule it will be able to calculate an exact resource profile and it will also be able to select much more relevant precedence constraints.

In a temporally consistent STN each task has an interval of possible start time values $[est(t_i), lst(t_i)]$. It is known that a consistent temporal solution can be retrieved if all start time points assume the value $est(t_i)$ (lower-bound). This temporal solution is referred to as the *Earliest Start Time Solution (ESTS)*. This is also true for the upper bound $lst(t_i)$, *Latest Start Time Solution (LSTS)*, however minimizing the make-span is the desirable objective, so the lower bound is more interesting. While the ESTS and LSTS may be temporally consistent they need not be resource consistent, however it is a good starting point for a resource peak leveling algorithm using precedence constraints (PCP).



(a) Fixed-time solution. (b) Exact resource profile.

Figure 2.11: An example of an ESTS (a) and a resource profile (b) used by ESTA.

Unlike with CFSA the time points are single values and no longer based on approximate bounds, allowing an exact resource profile to be calculated. With an exact resource profile ESTA can make better deductions when picking which resource peak is to be flattened and which constraint is best suited to do so.

Algorithm 2 shows the general process of ESTA. It first calculates an ESTS and then using a resource profile finds pair-wise conflicts. If no resource conflicts are found the ESTS is also a resource consistent solution. If a conflict is found that can not be put into sequence due to temporal restrictions, then it is considered unsolvable and the algorithm fails. If this is not the case a conflict pair is selected and a constraint is posted to resolve it, at which point it will start over until a consistent ESTS is found.

Algorithm 2: Earliest Start Time Algorithm

```
Input: A problem $P$
Output: A schedule $S$
while true do
    $S \leftarrow$ CALCULATE-ESTS($P$)
    $conflictSet \leftarrow$ FIND-RESOURCE-CONFLICTS($S$)
    if $conflictSet = \emptyset$ then return $S$
    if UNSOLVABLE($conflictSet$) then return failure
    $conflict \leftarrow$ SELECT-CONFLICT($conflictSet$)
    $constraint \leftarrow$ SELECT-PRECEDENCE-CONSTRAINT($conflict$)
    POST-PRECEDENCE-CONSTRAINT($P, constraint$)
```

How well ESTA performs depends on which conflicts are solved first and which constraints are posted. Cesta [6] proposes two heuristics for selecting conflicts and constraints. Following the *least commitment principle* the most critical conflict is selected first and then the constraint that removes the least amount of slack from the temporal network is posted.

By only attempting to remove resource conflicts from the ESTS the ESTA algorithm is able to find a solution in a fast and efficient manner. Unfortunately ESTA creates fixed-time schedules. In this form the schedule has no flexibility because it only defines start times for tasks and if only one task was to undergo a delay all guarantees of resource feasibility will be gone. In the next section we will introduce the *Chaining algorithm* that converts a fixed-time schedule into a fully resource feasible POS.

2.6.2 Chaining

The Chaining algorithm is a PCP algorithm that handles resource constraints differently. It sees each resource unit as a distinct unit that flows through a sequence of tasks. This sequence of tasks is called a chain. The Chaining algorithm constructs the chains by posting precedence constraints. Any two tasks competing for the same resource *unit* will be added to the same chain and will no longer be able to conflict with each other. By starting the same number of chains as there are resources we are also guaranteed that resources are never over-utilized.

Before the Chaining algorithm can add tasks to chains it needs to sort them by the starting time defined in the given fixed-time schedule. By adding the tasks in this order we are guaranteed that each task being added will not violate a temporal constraint. Adding tasks in the order provided by the fixed-time schedule also ensures that there are enough free chains at the given start time of the task, so the make-span does not increase during the chaining process.

The POS created using the Chaining algorithm is also referred to as a POS in *chain-form* or POS_{ch}. The use of the Chaining algorithm was first proposed by Cesta [6] but it was only mentioned shortly. The chaining algorithm and $POS_c h$ is further explored by Policella [27, 26] where he shows a POS can always be converted to a POS_{ch}.

Algorithm 3: Basic Chaining Algorithm [25].

```
Input: a problem $P$ and one of its fixed-time schedules $S = s_1, \dots, s_n$
Output: A POS in chain-form
$POS \leftarrow P$
Sort task in $S$ by start times $s_i$
Initialize all chains with the start task $t_0$
forall resource $r_k$ do
    forall task $t_i$ do
        for 1 to $req(r_k, t_i)$ do
            $chain \leftarrow$ SELECTCHAIN$(t_i, r_k)$
            $t_l \leftarrow last(chain)$
            $POS \leftarrow POS \cup \{t_l \prec t_i\}$
            $last(chain) \leftarrow t_i$
return $POS$
```

Algorithm 3 shows a formal description of the Chaining algorithm. Given a fixed-time schedule $S = s_1, \dots, s_n$ the original problem P (any constraints added by ESTA are removed), tasks are sorted by their start-time s_i and for each resource unit r_{kj} (where k is the resource and j a unit of resource r_k) a chain (or queue) is created containing the first task t_0. Once a chain has been selected a constraint is posted between task t_i and the last task in the chain, defined as $last(chain)$. Once all the tasks have been assigned to chains and all the needed precedence constraints have been posted the temporal network will be a valid POS.

2.6.3 Conclusion

We first discussed CFSA, which is a very straightforward implementation of PCP and because of this it runs into problems. It attempts to reduce a RCPSP problem directly to a POS but the temporal network has too many interactions to allow it to construct an accurate resource profile. ESTA avoids this problem by only solving the problem for one fixed scheduling of the tasks. Unfortunately it is only able to create a fixed-time schedule that has no flexibility. The Solve and Robustify approach solves the shortcomings of ESTA by applying the Chaining algorithm as a post-processing step, which expands the fixed-time schedule to a POS. Existing literature [26, 6] strongly suggests that the two phase approach works much better than a direct approach.

However both approaches have merit for NedTrain. Solve and Robustify is well suited because it is already focused on creating flexible schedules. CFSA is an older approach but can potentially be improved significantly using recent flexibility techniques like Interval Schedules. Both techniques can create a POS, which is well suited for NedTrain because it can absorb much of the uncertainty in the maintenance projects. Also work done by Evers [15] showed that ESTA was able to create a week schedule for technical maintenance in a timely manner.

Evers only used ESTA to minimize make-span and did not attempt to create flexible schedules. To improve on the work by Evers we will investigate different flexibility metrics in the next section so we can improve the above two algorithms to create more flexible solutions.

2.7 Flexibility

Before getting into more detail of what flexibility is with respect to schedules in general we will have a look at what definition of flexibility is used by NedTrain. At NedTrain a schedule is considered good and flexible if three properties are met. The first property is a clear and highly reliable make-span or final deadline. This allows NedTrain to give clients (NSR) guaranties with respect to fleet availability. The second property is freedom for the maintenance teams on the shop floor. Freedom on the shop floor helps NedTrain solve two problems. First it allows small delays to be solved by the maintenance teams without the need for rescheduling and second it gives the maintenance teams some autonomy, which helps to enforce involvement. Finally the extra freedom and reliable make-span should not be at the expanse of too much slack. In short, a flexible schedule should give management reliable information and give maintenance teams autonomy while minimizing the duration of the maintenance project.

The definition above is very intuitive and hard to work with in the context of computer algorithms. We will have a closer look at the three properties and describe them in a more concise manner. The first property is reliable deadlines, which is relativity simple, one can just define a deadline and if the need to reschedule is low it will also be reliable. The second property, freedom, can be defined as the number of different executions the schedule of the project allows. With many different executions allowed by a schedule, maintenance team are free to pick which execution is best. The final property was to minimize the project duration which can be expressed as minimizing unneeded slack. In short a good flexible schedule presents a maximized set of fixed-time schedules with a clear and tight deadline.

There is one detail in our definition of flexibility that we still need to address. A schedule can be made flexible with respect to different aspects of the scheduling problem. For NedTrain we have temporal and resource constraints. With temporal flexibility the maintenance teams have more freedom to handle unexpected delays while with resource flexibility the teams will have more freedom to handle unexpected changes in resource availability. Maintenance tasks are done on trains that are not always available to NedTrain and so the condition of the trains have a high level of uncertainty, while many of the resources at a NedTrain depot are directly under the control of NedTrain and there is much less uncertainty.

In this work we will focus on temporal changes i.e. changes in task duration and start times by allowing task to start at different moments depending on previous delays and if there are any unforeseen changes in resource availability, the schedule can be patched by adding extra temporal constraints.

We have defined flexibility as the number of possible concrete schedules represented in a flexible schedule, however it is not generally possible to count the exact number of possible schedules so most flexibility metrics are estimations of flexibility. In the following sections we will look at the strong and weak points of different flexibility metrics used to measure the flexibility of Partial Order Schedules (POS).

2.7.1 Flexibility metrics

There are many methods of measuring the flexibility of a POS. Some methods give better results than others, while others are much simpler to calculate. Due to the intuitive nature of flexibility, the context in which the metric is used can have great impact on its accuracy. We will be considering metrics in the context to the technical maintenance scheduling problem at NedTrain.

The driving idea behind most of the flexibility metrics is the number of different solutions a schedule allows. Counting the number of solutions quickly becomes infeasible with bigger schedules. Because of this, many usable flexibility metrics approximate the amount of schedules represented within a solution. In the following paragraphs we will discuss a few different metrics.

A schedule with very little precedence constraints relative to the number of tasks will be able to represent more possible sequences of these tasks and thus more possible solutions. This is the main idea behind the $flex_{seq}$ metric proposed by Aloulou and Portmann [1]. Counting the number of possible sequences (or linear extensions) is a counting problem that falls within the #*P*-complete counting class [4]. In other words it takes polynomial time to find a sequence but it takes exponential time to count how many sequences there are. Therefore $flex_{seq}$ approximates the number of possible sequences by counting the number of time point pairs for which there is no precedence constraint using

$$flex_{seq}(T) = |\{t_i, t_j \in T : i < j \wedge t_i \not\prec t_j \wedge t_j \not\prec t_i\}|, \tag{2.9}$$

where T is the set of tasks in a given scheduling problem. This gives an good indication of the number of possible sequences, however it does not take the *scheduling horizon* into account, i.e., the total amount of time available to the schedule. For example, given a schedule with a total ordering, the $flex_{seq}$ metric would give the schedule a flexibility of 0, even if the deadline of all tasks still allow for a lot of delay (slack).

To overcome this oversight Aloulou and Portmann introduce another metric $flex_{time}$. It takes the ratio between the total time in which tasks may be executed and the total processing time of all the tasks. In the case of Task RCPSP the $flex_{time}$ metric would be

$$flex_{time}(T) = \frac{\max_{t_i \in T}(dl(t_i)) - \sum_{t_i \in T} d_i}{\sum_{t_i \in T} d_i}, \tag{2.10}$$

where the worst case make-span is considered to be the latest deadline.

However this metric fails to consider the precedence constraints between tasks and only looks at release and deadlines. However it was the intention of Aloulou and Portmann to combine the use of $flex_{seq}$ and $flex_{time}$. The combinations of the metrics is still very crude and does not take the structure of the scheduling into account.

In work by Cesta [6], a robustness metric, *RB*, is introduced. The *RB* metric attempts to take the precedence constraint network into account as well the scheduling horizon. *RB* is the ratio between the average distances in time between tasks and the scheduling horizon. Given that H is the scheduling horizon and $d(t_i, t_j)$ is the temporal distance between two tasks as defined by the distance graph (see Definition 5). Formally *RB* is defined as follows:

$$RB(T) = 100 \sum_{t_i,t_j \in T, i \neq j} \frac{|d(t_i,t_j) + d(t_j,t_i)|}{H(n(n-1))}. \tag{2.11}$$

While Cesta refers to it as a robustness metric, it is equivalent to our definition of a flexibility metric because it estimates the total number of different starting times of each task while taking precedence constraints and deadlines into account. This metric does a much better task of taking different aspects of a flexible schedule into account but it does so in a very crude way because it only looks at the interaction between task *pairs*.

A slack based metric $RM1$ was proposed by Chtourou [11]. It simply sums the slack of tasks, i.e., the difference in time between the earliest start time and the latest start time of each task. More precisely:

$$RM1(T) = \sum_{t_i \in T} \text{lst}(t_i) - \text{est}(t_i), \tag{2.12}$$

where $est(t_i)$ is the earliest start time of task t_i and $lst(t_i)$ is the latest start time of task t_i.

This naturally takes the scheduling horizon into account and also to some extent precedence constraints. Unfortunately, it does not take the interactions between tasks which is created by precedence constraints into account, i.e., it can not tell when two tasks can be executed in parallel or in sequence. If two tasks need to be executed in sequence, due to a precedence constraint, it will overestimate the amount of slack because while the interval $\text{lst}(t_i) - \text{est}(t_i)$ itself takes precedence constraints into account, each task is still evaluated individually and shared slack is counted multiple times.

In the next section we will look at the $flex_I$ metric which is able to take task interaction into account. We will show that this metric is more closely aligned with the intuitive definition of flexibility at NedTrain.

2.7.2 $flex_I$ metric

Before introducing $flex_I$ in more detail we will first look at the problem Endhoven [14] was working with when he proposed the $flex_I$ metric. In maintenance orientated projects there is a lot of uncertainty and it is preferable to give maintenance engineers (agents) enough freedom in their schedules to handle delays. This leads to two challenges when creating a schedule for a project:

1. How can we create a overall schedule that maximizes the amount of freedom agents have while ensuring that all project constraints are enforced.

2. How can we decouple tasks (remove interactions between tasks) in the schedule so that each agent is free to handle delays independent of the other agents working on other tasks within the project.

The scheduling problem is modeled as an STP, see Section 2.2. In addition to the constraints provided by STP each task is also assigned to an agent.

To solve the above problems an Interval Schedule (see Definition 6) is created. If the time intervals are defined properly it allows an agent all the freedom to plan within these time intervals independent of other agents while not violating overall project constrains. To be able to maximize the amount of freedom the agents have we need to be able to measure the amount of freedom, which we will come back to shortly. The amount of freedom an agent has can be interpreted as the amount of task-independent flexibility of an STN as proposed by Wilson [35].

Before we have a closer look at how we can measure the freedom of an agent or the independent slack of a task we will first look at the structure for an interval schedule. Each task $t_i \in T$, where T is a set of tasks in a scheduling problem instance, is assigned an interval $[a_i, b_i]$. Each task is free to start within this interval without taking the behavior of other tasks into account. We can define that amount of independent slack or flexibility as follows:

$$flex_I = \sum_{t_i \in T} (b_i - a_i). \tag{2.13}$$

This gives us an impression of the amount of freedom agents have in a given interval schedule. The metric $flex_I$ defined above has a lot of resemblance to the $RM1$ metric, however it uses the interval schedule and not the lower-bound (est) and upper-bound (lst) to calculate slack.

Endhoven also describes a method that creates an interval schedule with a maximum amount of freedom over all agents. The intervals schedule is created by first finding the *est* and *lst* of each task in a given STN instance. To calculate a_i and b_i the following linear model is solved:

$$\begin{aligned} &\text{maximize} \sum_{t_i \in T} (b_i - a_i) \\ &\text{subject to} \\ &\qquad \forall i : a_i \leq b_i \\ &\qquad\qquad b_i \leq lst(t_i) \\ &\qquad\qquad a_i \geq est(t_i) \\ &\forall t_i \ll t_j : b_i + d_i \leq a_j \end{aligned} \tag{2.14}$$

The linear model finds values for a_i and b_i for which $flex_I$ is maximized. The first 3 constraints ensure that the values a_i and b_i stay within the interval $[est(t_i), lst(t_i)]$, to ensure that no temporal constraints defined in the STN instance are violated. The last constraint forces the intervals of interacting tasks to not overlap, which decouples the tasks allowing each task to be scheduled within its interval without it effecting other tasks. An interval schedule can be constructed using the variables a_i and b_i.

In the work by Endhoven this interval schedule was enough to satisfy his problem conditions described above. The created interval schedule maximizes the overall freedom of the agents and the tasks start time intervals are free of interaction with each other so each agent is free to act within its own interval.

In a later publication by Wilson [35] it is pointed out that $flex_I$ is a good metric of flexibility in itself. When using $flex_I$ with Interval Schedules, it becomes clear that $flex_I$ matches with the definition of flexibility used at NedTrain. An interval schedule with a high value of $flex_I$ is able to provide maintenances teams with much freedom while still closely coordinating resource usage. It also does not overestimate the amount of slack in a schedule, which is the case of the *RM*1 metric. *RM*1 does not take serial and parallel properties into account, which causes it to sometimes count some units of slack multiple times as is shown in the following example.

Example 4 (Example where $flex_I$ does not over estimate the slack of a serial schedule.)**.** *Given five tasks of unit duration. If the tasks have a total ordering (see Figure 2.12) and a deadline of six then RM would give it a flexibility value of five, while $flex_I$ would give only one. The value of one is intuitively more accurate because while all the tasks are able to delay one unit, only* one *may do so. Once one task is delayed all the flexibility within the schedule will be gone.*

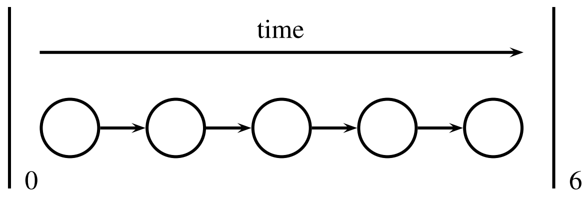


Figure 2.12: Example of 5 unit durations tasks in total ordering with a deadline of 6.

The $flex_I$ metric only sums up the slack within the independent intervals, so each unit of slack is counted at most once, which means it also describes the lower-bound of flexibility in a schedule, while *RM*1 for example can be considered an upper-bound because it counts all the units of slack a task has, even if it has to share it with other tasks.

2.7.3 Using different Objective functions

Equation 2.14 uses $\sum_{t_i \in T}(b_i - a_i)$ as objective function which maximizes the amount of slack. To be able to maximize the slack the linear solver does not distribute the slack equally between all tasks, so the Interval Schedule that is created will generally not be a very fair schedule. Only a small set of fully concurrent tasks will be given all the slack in the schedule. So while the default $flex_I$ objective finds the maximum amount of independent slack in the schedule it is not the best objective function when we also want to use the underlaying interval schedule. In this section we will look at two other objective functions that produce different distributions of slack.

By distributing the slack more equally over all tasks the total independent slack measured by $flex_I$ will be reduced. However Wilson [35] shows that sacrificing flexibility for a more equal distribution of the flexibility over tasks, produces a schedule that is much more robust.

The objective function $flex_{I_{equal}}$ is a first attempt to create an objective function that distributes the slack more equally at the expense of total flexibility. It uses the following objective function:

$$\sum_{t_i \in T} ((a_i - est(t_i)) + (lst(t_i) - b_i)) * ((a_i - est(t_i)) + (lst(t_i) - b_i)). \quad (2.15)$$

While it does distribute flexibility over more tasks it still functions poorly by producing schedules where many tasks still have zero independent slack. So, to force the solver to give each task a minimum amount of slack, a two part objective function was proposed by Wilmer et al. [33], which we call $flex_{I_{fair}}$. To ensure each task gets a fair amount of flexibility while at the same time maximizing the total flexibility the linear solver is run twice. First the solver maximizes the minimum amount of slack each task can have without creating an unsolvable instance. Using this lower-bound as an extra constraint in the original model the total slack is maximized. In other words we first define an amount of slack *min* that each task will minimally always have. This value *min* is maximized and then added as an extra constraint to the model. We then maximize the total slack like Equation 2.14 but now each task will be constrained to have at least *min* amount of slack.

The model to calculate the lower-bound is as follows:

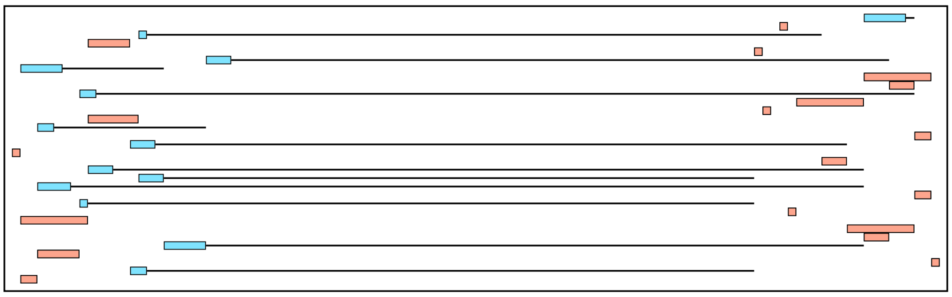
$$\begin{aligned}
&\text{maximize}\, min \\
&\text{subject to} \\
&\qquad \forall i : a_i \le b_i \\
&\qquad\quad b_i \le lst(t_i) \\
&\qquad\quad a_i \ge est(t_i) \\
&\qquad\quad b_i - a_i \ge min \\
&\forall t_i \ll t_j : b_i + d_i \le a_j
\end{aligned} \tag{2.16}$$

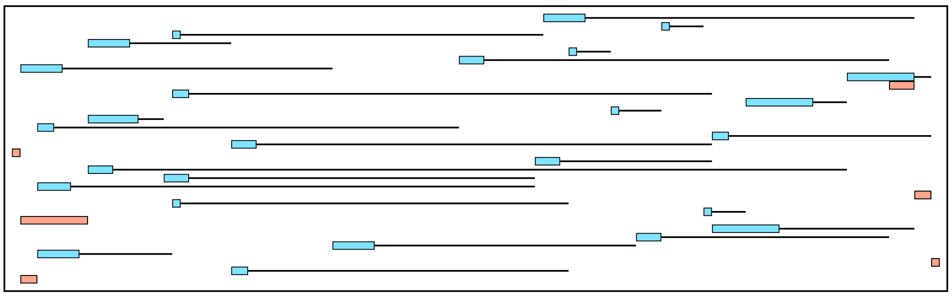
This model allows us to find the maximum *min* value at which the problem instance should still be solvable when maximizing the total slack. To ensure that each task has a minimum amount of slack an extra constraint, $b_i - a_i \ge min$ is added to the original model (see Equation 2.14), which give us the following model:

$$\begin{aligned}
&\text{maximize} \sum_{t_i \in T} (b_i - a_i) \\
&\text{subject to} \\
&\qquad \forall i : a_i \le b_i \\
&\qquad\quad b_i \le lst(t_i) \\
&\qquad\quad a_i \ge est(t_i) \\
&\qquad\quad b_i - a_i \ge min \\
&\forall t_i \ll t_j : b_i + d_i \le a_j .
\end{aligned} \tag{2.17}$$

The $flex_{I_{fair}}$ objective function requires us to solve the linear model twice but does a much better job of distributing the flexibility between tasks.

We can illustrate the differences between the objective functions by solving the same scheduling problem instance using the different objective functions. The resulting Interval Schedules can be seen in Figure 2.13. Figure 2.13a shows 19 tasks with no slack, Figure 2.13b shows six tasks with no slack and finally Figure 2.13c show that all tasks have some slack.


(a) Interval Schedule created using $flex_I$


(b) Interval Schedule created using $flex_{I_{equal}}$

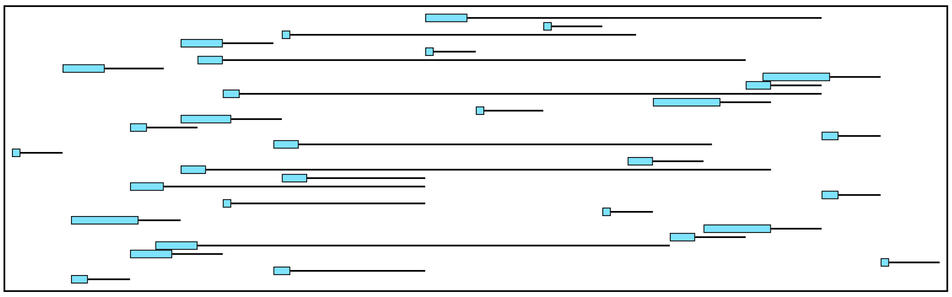

(c) Interval Schedule created using $flex_{I_{fair}}$

Figure 2.13: Three Interval Schedules created using different objective functions, where the lines after some tasks represents slack.

2.7.4 Conclusions

At the beginning of this section we discussed the intuitive aspects of flexibility and then highlighted some of the existing flexibility metrics. We took a closer look at $flex_I$ because it is not only a very intuitive way of quantifying flexibility but also creates an interval schedule in the process, which is a schedule that is good for a sociotechnical shop, which in turn is good for NedTrain because they are interested in creating a sociotechnical shop at their maintenances depots.

2.8 Conclusions and research questions

With the questions from NedTrain in mind, which we defined in Chapter 1, we looked at the current state of the art for modeling scheduling problems, heuristic solvers and flexibility metrics.

We first looked at a formal scheduling model RCPSP, which is ideal for the scheduling challenges at NedTrain. RCPSP is able to model the maintenance tasks, with respect to precedences and resource constraints. With the addition of task release times and deadlines (Task RCPSP) we are able to model the strict deadlines required by NedTrain.

We then looked at two fast heuristic PCP solvers. Speed is important because NedTrain is interested in creating software to assist schedule makers. Assistant tools need to be fast and interactive so end-users can understand the scheduling problem and easily make changes if needed.

Finally we looked at flexibility because for NedTrain flexibility in schedules is important due to maintenance projects being very uncertain in their execution. To be able to increase the flexibility in schedules we needed a way of measuring it. The $flex_I$ metric is well suited for NedTrain because it fits well with their existing definition of flexibility or shop floor freedom. Schedules with a high value of $flex_I$ give maintenance teams more freedom to solve small delays independently while maintaining deadlines. Clear deadlines are important for NedTrain from a management point of view.

This concludes the topics we looked at during this chapter. For the rest of this section we will look at how we can improve on the above.

The current state of the PCP solvers can be improved to better suit the needs of NedTrain. For example the two mentioned PCP solvers Solve and Robustify and CFSA are both focused on solving RCPSP/max instances, and not Task RCPSP. Task RCPSP is a simpler problem, so we may be able to improve the flexibility of the found solutions or improve the running time of the solvers. We also looked at the very recent concept of Interval Schedules. This new method of defining a flexible schedule may provide us with some new insights that can be used to improve the heuristics used by the solvers.

With the introduction of $flex_I$ and Interval Schedules we may be able to improve the two existing PCP algorithms. This leads to the following questions, which we will answer in the following chapters.

1. Can the $flex_I$ metric or its underlying Interval Schedule help the Solve and Robustify algorithm find a partial order schedule (or interval schedule) that has a greater value of $flex_I$?
   a) Which part of the Solve and Robustify algorithm can be changed, so the algorithm is able to use the extra insight provided by $flex_I$?
   b) Does the use of $flex_I$ help the solver create a schedule with a higher value of $flex_I$?
2. Can the $flex_I$ metric or its underlying interval schedule help the CFSA algorithm find a partial order schedule (or interval schedule) that has a greater value of $flex_I$?

a) How can $flex_I$ or an Interval Schedule help CFSA avoid creating over constrained schedules?

b) Does the use of an Interval Schedule as temporal layer help the solver create a schedule with higher value of $flex_I$?

c) Using $flex_I$ as leading measure of flexibility, which of the two above approaches creates more flexible schedules?

Answering these questions will allow NedTrain to get more insight into the benefits of using $flex_I$. With this insight NedTrain will be one step closer to producing the highly flexible schedules needed for its maintenance processes.

Chapter 3

Improving Solve and Robustify

The *Solve and Robustify* algorithm designed by Policella et al. [26] was created to solve RCPSP/max. In this chapter we will propose a change in the implementation of the Solve and Robustify to make use of the simpler Task RCPSP model. We will also make use of $flex_I$, which was recently introduced by Wilson [36]. With these improvements we hope to improve the flexibility of the created schedules for the scheduling problem at NedTrain without significantly increasing the run-time of the algorithm.

The modification of the Solve and Robustify solver (Policella-Solver) as described in Section 2.4 allows the solver to actively focus on producing solutions with a higher value of $flex_I$. First we will show an example to motivate the proposed changes, then we will show how we implemented a modified Policella-Solver to see if the modification has any positive effect. To test the solvers we will be using a modified version of the well known benchmarks designed by Kolisch et al. [18]. Finally we will look at the results and to get a better understanding of what is happening, we will do some follow up tests.

3.1 Motivation and Example

The Solve and Robustify approach is a PCP technique that first solves an RCPSP instance and then expands it to make the solution more robust or flexible. Originally a bounding technique was used where an RCPSP instance was directly reduced to a POS. However Policella shows that expanding one valid fixed-time schedule into a POS, results in a more relaxed and less constraint solution.

Solve and Robustify is based on the assumption that the performance of the two phases are independent of each other, in other words the quality of the fixed-time schedule created by the *solve phase* has no impact on the qualtiy of the POS found by the *robustify phase*. The decoupling of the two phases allows the *solve phase* to focus on minimizing the make-span and the *robustify phase* to focus on creating a flexible and open POS. While this is an interesting heuristic strategy [26], it is not an accurate assumption in general. The two phases are linked together because not all fixed-time schedules created by the first phase are equally susceptible to expansion in the second phase.

The above assumption might not be acceptable if the flexibility is set as primary objective. The first phase influences the second phase so it would benefit the resulting POS if the first phase would also take flexibility into account. Let us assume that focusing on flexibility in the first phase will help the second phase produce a more flexible solution. The main goal of the first phase must still be finding a valid fixed-time solution, otherwise there would be nothing to expand, however the solve phase can produce many different solutions, where some will result in a better POS after the second phase.

The solve phase of Solve and Robustify uses an ESTA solver to solve an RCPSP/max instance. The heuristics in ESTA are focused on finding a solution, without any particular properties because finding a solution for an RCPSP/max instance is NP-hard by itself. In this work we use Task RCPSP, which is a less complex problem. This allows the heuristics to be modified to allow the ESTA solver to focus on producing solutions that are more inclined to be expanded by the chaining algorithm. The chaining algorithm uses a fixed-time schedule as a starting point and attempts to relax the solution so that it becomes more flexible, however due to project constraints not all fixed-time schedules can be relaxed by the same amount. With this in mind we could modify the heuristics in ESTA to not only find an arbitrary fixed-time schedule but also to find one that will produce a more flexible POS after chaining.

The fixed-time schedule created by ESTA does not contain any flexibility by definition. However ESTA produces a fixed-time schedule by maintaining a temporal network. The temporal network will have a fair amount of flexibility but can also have a number of resource constraint violations. The heuristics of ESTA can be modified to not only find a ESTS (earliest start time schedule) but also to improve the flexibility of this temporal network. It is then assumed that an ESTS created from a more flexible temporal network will also produce a more flexible POS in the chaining phase.

The ESTA solver uses greedy heuristics at two distinguishable branch points. Both branch points have impact on the resulting ESTS and can be modified to help create an ESTS that is more inclined to produce a flexible POS. The first point is the selection of a resource conflict and the second is the selection of a constraint that resolves the conflict. The general idea of the first heuristic is to select the most constrained conflict because adding an extra temporal constraint will not have a great impact on the temporal network. The second heuristic selects a constraint that constrains the temporal network as little as possible. This can also be interpreted as a maximization of slack or flexibility, which would be a good point to optimize.

Which constraint is selected at the second branch point is determined by the *constraint selection function*. It takes a conflict pair (t_1, t_2) as input, where t_1 and t_2 are tasks and returns a precedence constraint: $t_1 \prec t_2$ or $t_2 \prec t_1$.

To increase the chances of finding a solution Cesta [6] points out that the constraint selection function should follow the least commitment strategy. Cesta uses a local slack metric called Min-Slack that follows this strategy. The constraint selection function measures the slack between the two conflicting tasks and selects the constraint that has the most slack. In most cases, this reduces the impact of the constraint on the whole temporal network in most cases and maximizes overall slack. This selection function was adopted by Policella when Solve and Robustify was introduced [25, 26].

We propose to improve the constraint selection function by using a global flexibility metric like $flex_I$, $RM1$ or RB. Using a global flexibility metric to select constraints does not deviate from the least commitment strategy so it should still be a good heuristic for finding solutions, however it will also be able to take the slack of tasks other than the conflicting pair of tasks into account. When resolving a conflict between two tasks that are not particularly interesting to the structure of the project graph (precedence constraints) the use of a global or local slack metric is less important, however when there is a resource conflict between two key tasks the impact of a new constraint can have an effect on many other tasks.

Example 5 (Local slack fails to maximize overall slack)**.** *With this example we show how the local slack metric fails to measure the impact of a constraint on the whole scheduling problem. Let a scheduling problem instance P have tasks* $\{t_1,t_2,t_3,t_4,t_5,t_6,t_7,t_8\}$ *with durations* $\{1,2,2,1,1,1,1,1\}$*, respectively. Each task has a deadline of* 6 *with the following constraints:*

$$\{(t_1,t_3),(t_2,t_4),(t_2,t_5),(t_2,t_6),(t_2,t_7),(t_2,t_8)\}.$$

The problem P has only one resource r with one unit. Tasks t_1 *and* t_2 *both require 1 unit of r. See Figure 3.1 for an image of the project graph.*

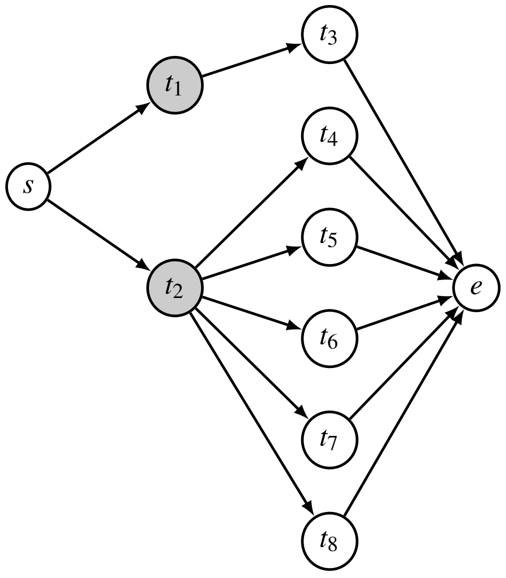


Figure 3.1: An example of a project graph with conflicting tasks t_1 and t_2 for which a constraint must be selected.

The tasks t_1 *and* t_2 *are selected as conflicting. The constraints* $t_1 \rightarrow t_2$ *and* $t_2 \rightarrow t_1$ *will resolve the conflict. Using a local slack metric the constraint* $t_1 \rightarrow t_2$ *would be selected because* $slack(t_1 \rightarrow t_2) = 2$ *and* $slack(t_2 \rightarrow t_1) = 1$*. However the total slack within the temporal network is great if* $t_2 \rightarrow t_1$ *is posted because task* t_2 *has more successors than task* t_1*. By posting* $t_1 \rightarrow t_2$*, tasks* t_4,t_5,t_6,t_7 *and* t_8 *lose 1 slack each while posting* $t_2 \rightarrow t_1$ *only* t_3 *loses 2 slack. This can be seen in Figure 3.2.*

This example shows how an added constraint impacts the whole temporal network and not just the two tasks that need to be constrained due to a resource conflict. Figure 3.2

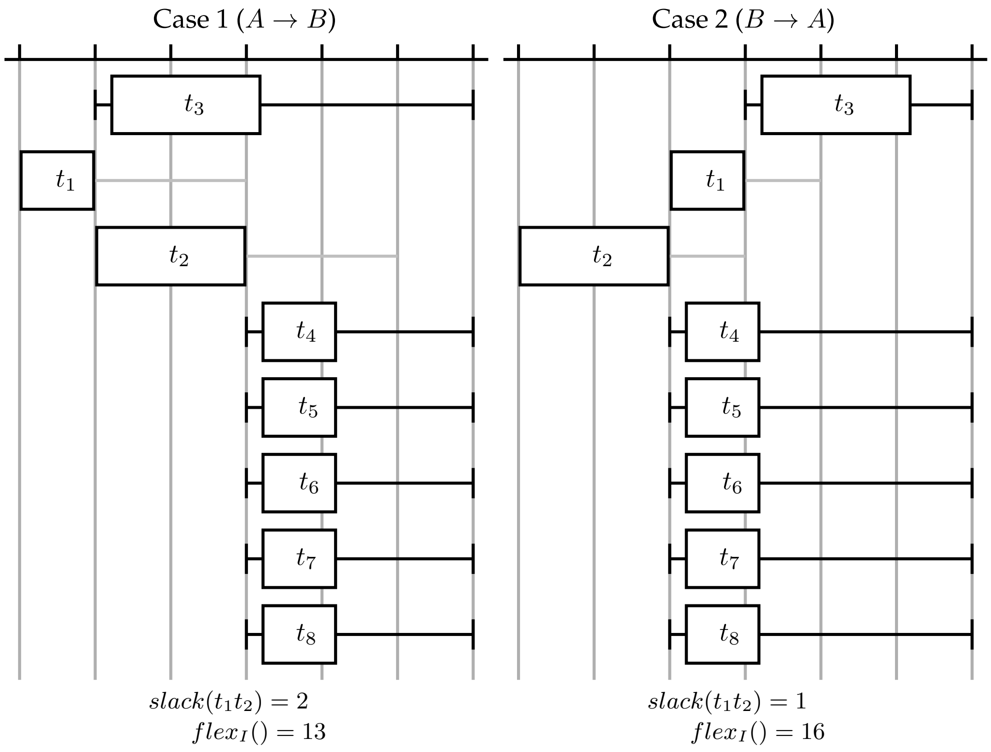


Figure 3.2: Example of local slack and $flex_I$ disagreeing on which constraint costs the least amount of slack.

*uses $flex_I$ as global metric. It is clear in the figure that Case 2 has more independent slack. Other global metrics like RM*1 *and RB (see Section 2.7) also shows that Case 2 produces a more flexible solution. What each flexibility metric would score in the two cases is shown in Table 3.1.*

Table 3.1: How different metrics score the two cases shown in Example 5.

Metric	Case 1	Case 2
*RM*1	17	18
RB	29	34
$flex_I$	13	16

A global flexibility metric like *RM*1, *RB* or $flex_I$ is able to take the complete structure of the temporal constraints into account, as is shown in Example 5 where all three metrics give Case 2 a higher value of flexibility than Case 1.

The above example is a strong example that shows that the global metric $flex_I$ will help the Solve and Robustify algorithm to produce interval schedules with more independent slack, however it is a toy example to illustrate a point. In the following sections we will present an modified ESTA solver using the $flex_I$ metric. Using a set of benchmarks we will see if the positive results shown in Example 5 are also true in general.

3.2 Implementation of Flexi-Solver

The above section shows an example of how global metrics can improve flexibility of a solution. It is challenging to show this is true in general using only examples, so to validate our hypothesis we will implement a modified version of the Policella-Solver, where the Policella-Solver is an implementation of the Solve and Robustify algorithm as described in [26]. The Policella-Solver uses the ESTA algorithm for the solve phase and the chaining algorithm for the robustify phase. We will be modifying the ESTA algorithm to use $flex_I$ as a constraint selection heuristic.

Before we can create an ESTA solver that uses $flex_I$ we need to modify the *constraint selection function* used by ESTA so it can use a global metric instead of the default local slack metric. Only the constraint function is changed, the rest of the solver is the same as described in Section 2.6.1. The original constraint selection function is given a conflicting pair of tasks t_1 and t_2 and selects a constraint ($t_1 \prec t_2$ or $t_2 \prec t_1$) which provides the most slack between the tasks. A global metric like $flex_I$ requires the whole temporal network in order to calculate a value. To find the best constraint to select, the constraint selection function needs to temporarily post both constraints in turn and measure the flexibility. It then returns the constraint that produced the highest flexibility value. A pseudocode example of the function is shown in Algorithm 4, where *metric* can be any metric that requires a temporal network as input and gives a quantitative value as output.

To create an ESTA solver that uses $flex_I$ the *metric* function in Algorithm 4 should be replaced by $flex_I$. The modified ESTA solver creates an ESTS. This ESTS is passed to the chaining algorithm. We will refer to the whole process, as the *Flexi-Solver*.

Algorithm 4: Constraint selection based on a defined metric.

```
Input: A project graph $G(N,E)$ and a conflict pair $(t_i \in N, t_j \in N)$
Output: A constraint $c$
$G1(T, E \cup \{(t_i, t_j)\})$
$G2(T, E \cup \{(t_j, t_i)\})$
if $metric(G1) > metric(G2)$ then
    return $(t_i \rightarrow t_j)$
else
    return $(t_j \rightarrow t_i)$
```

The modified constraint selection function allows us to change the selection metrics with ease and it is very simple to create Solve and Robustify solvers using other metrics like *RB* and *RM*1. We will be making use of this later. The selection function does not take ties into account because we assume that if the two options produce equally flexible temporal networks the decision is trivial. This is not an accurate assumption but adding another metric for tie breaking is outside the focus of our experiments.

To see how the Flexi-Solver will behave differently from the original implementation we will need to validate it against a set of benchmarks. In the following section we will show which benchmarks are used to validate the Flexi-Solver.

3.3 PSPLib benchmarks

To be able to test and compare the Task RCPSP solvers proposed in this work we need a dataset, which can function as test set or benchmark. Unfortunately there is no readily available data to create task RCPSP instances of the scheduling challenges at NedTrain. This is due to the scheduling currently being done by hand at each of the maintenances depots at NedTrain.

Fortunately there is a well known set of benchmarks found in literature. This set of benchmarks called *PSPLib* was designed by Kolisch [18] and provides the scientific community a way to compare RCPSP solvers with each other, which can also be used to compare the Flexi-Solver against the original implementation by Cesta and Policella.

Before we have a closer look at the actual PSPLib datasets we will first compare the benchmarks with the type of scheduling problems found at NedTrain. Kolisch introduces two different datasets: *Multi Mode* and *Single Mode*, where Multi Mode are scheduling instances where tasks may pick different resources, i.e., use a machine in different modes, while in Single Mode each task has no choice and must acquire a specific resource type. The main elements of the scheduling problem at NedTrain: tasks, precedence constraints and a finite resource capacity, are all present in the Single Mode data set, making it a good dataset to benchmark the solvers proposed in this work.

There are however, differences between the instances in PSPLib and the ones NedTrain needs to deal with. First and most important is the lack of deadlines in the PSPLib datasets. These deadlines need to be added before the benchmarks can be used to give any indication of how well our solvers work in the context of the scheduling challenges at NedTrain. Before we can add a suitable deadline we must understand what kind of impact it can have on a problem instance. RCPSP instances are always solvable given that the following two properties are true. First, there is no one task that requires more resources by itself than the number of available resources. Second there is no precedence cycle between tasks. Both these properties are true for all the PSPLib benchmarks, so all instances are solvable, however this may no longer be true once deadlines are added. It is our goal to create flexible schedules and for an unsolvable instance flexibility is irrelevant. Also NedTrain is required to always meet its maintenance deadlines so the deadline of each instance needs to be large enough to ensure it is solvable. The deadline has a great impact on the flexibility of a schedule but not on the relative performance of different solvers, so to be able to compare the flexibility of solutions created by the different solvers it is desirable to have a constant deadline.

The second difference between PSPLib instances and NedTrain scheduling instances is in the structure of the project graph (precedence constraints), the PSPLib instances were designed to be hard to solve, which means the project graph contains many precedence constraints and there are very little loosely constraint tasks. This generally is not the case with the maintenance projects at NedTrain where there are much less precedence constrains and where there are small independent groups of parallel or sequential tasks. The PSPLib datasets are considered harder to solve so it is reasonable to assume any improvements found using this dataset will also produce better results with technical maintenance scheduling because the scheduling problem at NedTrain has much less precedence constraints defined by the problem instance itself, so adding an extra constraint will effect a larger group of tasks.

The dataset of RCPSP instances from the PSPLib are split up into four benchmark sets of different sizes. Each benchmark set is named: j30, j60, j90 and j120, where the numbers refer to the number of tasks in each RCPSP instance. Each benchmark set has constant properties and variable properties. The constant properties are the number of tasks and resource types. For example the j60 data set has problem instances with 60 tasks and 4 resource types. Within each set there are 3 properties that are variable. These properties are:

NC Temporal network complexity with values $\{1.5, 1.8, 2.1\}$, which is the average number of precedence constraints per task,

RS Resource strength with values $\{0.2, 0.5, 0.7, 1.0\}$, which defines how scarce (not abundant) the resources are,

RF Resource factor with values $\{0.25, 0.50, 0.75, 1.0\}$, which defines how many different resources one task requires.

There are 48 different combinations of these three properties in a benchmark set. Each combination has 10 problem instances, which gives each benchmark set 480 instances. However 25% of the instances have trivial resource constraints, in other words the resource capacity is so high that there are no resource violations. So while there are 480 instances, only 360 instances are of interest.

The experiments conducted in the following sections are all done using the j60 problem instances, where each task has a release time of 0 and a deadline of 250. We choose to use 250 as deadline for all tasks because earlier experiments showed that all j60 instances are still solvable with 0 as release time and 250 as deadline. The experiments we did existed of solving the j60 instances with different deadlines using the Flexi-Solver and Policella-Solver. We found that all instances were solvable using a deadline of 250. We choose to use j60 instances over j30 instances or j120 instances because they are hard to solve optimally and they are small enough to run experiments in a reasonable amount of time.

3.4 Benchmark Experiments

In the following section and subsections we will present some questions that we will attempt to answer by performing experiments. By answering these questions we will be able to validate or invalidate our hypothesis, that $flex_I$ can be used to improve Solve and Robustify. We will focus on running different solvers against the same set of benchmarks, starting with the Flexi- and Policella-Solvers and then move on from there.

3.4.1 Flexi-Solver and Policella-Solver

In the above section we presented the Flexi-Solver that uses the $flex_I$ metric during constraint selection. The first experiment we will do is to answer the following question: Does the Flexi-Solver produce schedules with a higher value of $flex_I$ than the original Policella-Solver?

To answer the above question we will solve the modified j60 benchmarks described in the previous section. The j60 benchmarks are too big to solve optimally so we will only be able to compare the results between the two solvers. Based on Example 5, where we show that Min-Slack used by the Policella-Solver is unable to fully take the impact of a posted constraint into account, we can assume that the Flexi-Solver will perform better.

When the Flexi-Solver and the Policella-Solver both solve the modified j60 benchmarks, it becomes clear that no one solver performs any better or worse than the other. The bar chart in Figure 3.3 shows the average flexibility is almost the same. The variance in results are also very small as can be seen in the box plot.

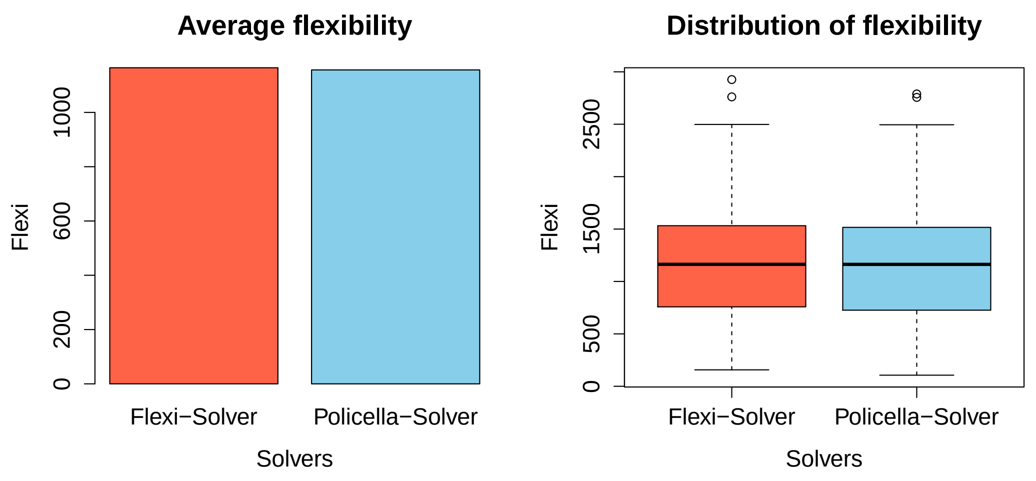


Figure 3.3: Flexibility measured using $flex_I$ of two solutions sets created by Flexi-Solver and the original Policella-Solver.

Before we conclude that using the $flex_I$ metric over the Min-Slack metric does not produce an overall improvement we need to answer another question: Are the schedules found by both solvers similar with respect to other properties, like make-span and number of precedence constraints? If we see the same similarity we can reasonably conclude that using the $flex_I$ metric as constraint selection heuristic does not help the Flexi-Solver to produce more flexible solutions.

We calculate the make-span and the number of non-transitive precedence constraints for each schedule found by the two solvers. A non-transitive precedence constraint is any constraint $t_i \prec t_j$ for which there is no other directed path in the project graph from t_i to t_j. In other words each non-transitive precedence constraint has an impact on the partial ordering of the tasks.

In Figure 3.4 we see that the average make-span and number of precedence constraints are the same. This shows that the Flexi-Solver and the Policella-Solver create schedules with very similar properties and there is no reason to assume one is better than the other.

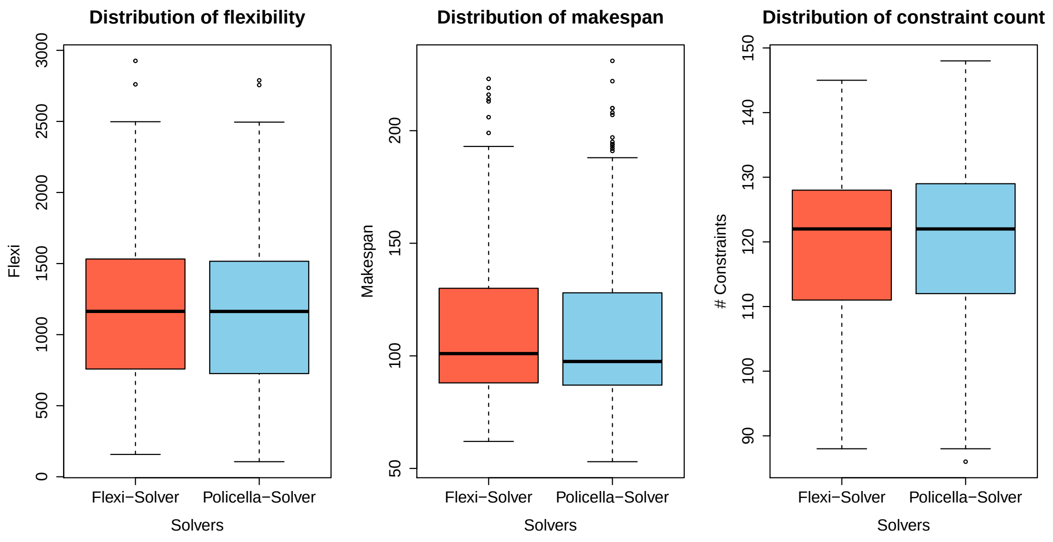


Figure 3.4: The $flex_I$, make-span and number of precedence constraints of two solutions sets created by the Flexi-Solver and the Policella-Solver.

3.4.2 RM1-Solver and RB-Solver

The results we have seen so far would let us believe that there is very little difference between using a global metric or local metric for constraint selection, but if this would be true we would need to revise our earlier assumptions and motivating example. Before we do this we will just assume that $flex_I$ is not a suitable global metric and maybe using $RM1$ or RB will produce better results. Which leads us to the next question: Is the metric $flex_I$ a less suitable metric compared to $RM1$ and RB, when used as an heuristic for resolving conflict pairs?

Answering the above question can be done by first constructing two new solvers: RM1-Solver and RB-Solver. These solvers can be created by modifying Policella-Solver in the same way as Flexi-Solver, where the $metric$ function in Algorithm 4 is replaced by $RM1$ and RB. We can then measure the flexibility of the schedules created by the two solvers and if there is again no improvement we will need to reassess some of our assumptions.

The experiments using the RM1-Solver and RB-Solver show very much the same results as the Flexi-Solver. The results of all 4 solvers Flexi-Solver, RM1-Solver, RB-Solver and Policella-Solver are shown in Figure 3.5.

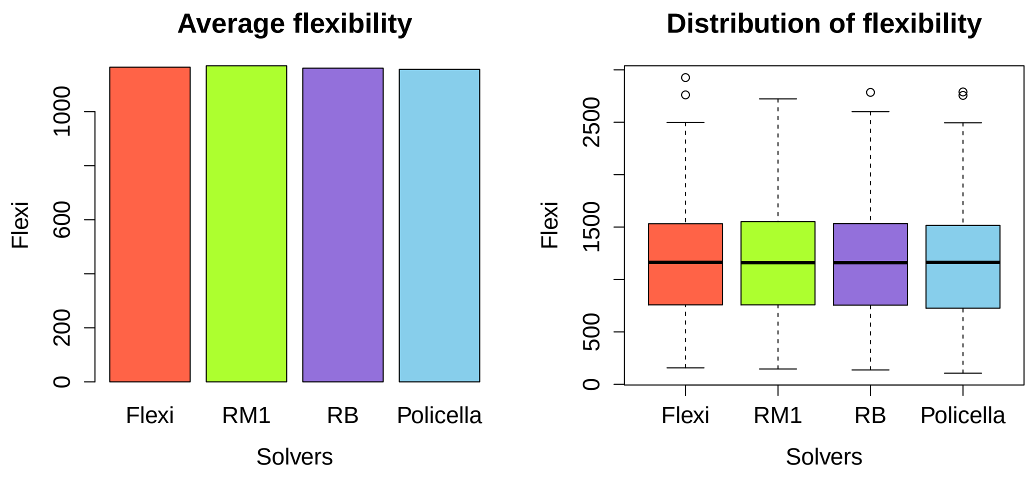


Figure 3.5: Flexibility measured using $flex_I$ of four solutions sets created by Flexi-Solver, RM1-Solver, RB-Solver and the Policella-Solver.

3.4.3 Conclusions

The above results show that the Flexi-Solver does not produce solutions with a higher value of $flex_I$. We also found that not only is there no change with respect to $flex_I$, there is also very little impact on other properties of the produced schedules, like make-span and the total number of precedence constraints. This leads to the conclusion that using $flex_I$ as heuristic has no major impact on the results.

To see if $flex_I$ was just an unsuitable metric compared to the RB and $RM1$ metrics, we also created the solvers RM1-Solver and RB-Solver. Unfortunately the Flexi-Solver, the RM1-Solver and RB-Solver all produced schedules with very similar amount of flexibility.

Based on all the above results it is clear that the proposed changes to the Solve and Robustify does not create a solver that can produce more flexible schedules. This is surprising because using a global measure of flexibility increases the solvers awareness of slack in the temporal network as shown in Example 5.

Because we are not yet fully satisfied with the above results we revise some of the assumptions we made at the start of this chapter in the following sections. In the next section we will investigate how much influence the constraint selection function has on the created schedules, which yields yet again a surprising result.

3.5 Using a random constraint selection function

In the above section we concluded that the proposed greedy solver, Flexi-Solver does not produce solutions with more flexibility than the original Solve and Robustify algorithm. This was not expected so in this section we will have a closer look at one of the assumptions we made at the start of this chapter and investigate the behavior of the Flexi-Solver with some follow up experiments.

Flexi-Solver, RM1-Solver and RB-Solver all produced very similar solutions. This similarity can be attributed to the constraint selection function not being the only component in an ESTA solver that has influence on the created schedules. ESTA based solvers have many other components like conflict selection, peak selection and chaining. When we considered to improve the constraint selection function we assumed that it would have a notable impact on the solutions.

In this section we will propose another solver that will allow us to explore the influence of the constraint selection function. This will help to answer the following question: How much, if any, influence does the constraint selection function have in the Solve and Robustify algorithm?

3.5.1 Random-Solver and Experiments

To get more insight into the impact of the constraint selection function we propose creating another solver. The four solvers we already created used some kind of metric to guide its decisions, however they all produced very similar results, which gives us very little insight into what is really happening. To get a better view of how the constraint selection in ESTA affects the solutions space we will create a solver called Random-Solver.

The Random-Solver is the same as the Flexi-Solver, RB-Solver, RM1-Solver and Policella-Solver, only the constraint selection function simply picks a constraint $t_1 \prec t_2$ or $t_2 \prec t_1$ at random. The Random-Solver will be free to blindly explore the solution space but will also perform badly on average because it does not try to understand its decision. However if we solve the same data set many times using the Random-Solver it will give an impression of the solutions that are reachable when using the right heuristics.

For our experiments we used the benchmarks described in Section 3.3. Each of the 360 instances were solved 2000 times each. For each of the 720,000 solutions we calculated the value of $flex_I$. By looking at the variance of the resulting $flex_I$ values we were able to find approximate upper and lower bounds of the solution space. These bounds allows us to see what the impact of the constraint selection function is.

To get an approximate idea of the solution space we can calculate three values, the lower-bound, the upper-bound and the average value of $flex_I$. We calculate these bounds as follows; first we solve each problem instance 2000 times, which gives us 2000 different solutions for each problem instance when we measure the flexibility of each solution using $flex_I$. The lower-bound is then the lowest $flex_I$ value of the 2000 solutions, the upper-bound is the highest $flex_I$ value of the 2000 solutions and the average is the average of all 2000 solutions. So given a problem instance P we generate a set $S_p = \{S_1, \dots, S_{2000}\}$ of 2000 solutions using the Random-Solver. Formally the upper-bound is

$$flex_{I_{UB}}(S_p) = \max\{\forall S_i \in S_p | flex_I(S_i)\}$$

and the lower-bound is

$$flex_{I_{LB}}(S_p) = \min\{\forall S_i \in S_p | flex_I(S_i)\}.$$

Figure 3.6 shows the average maximum and minimum amount of $flex_I$ found over all j60 problem instances. It shows that the average maximum $flex_I$ of solutions created by Random-Solver (see Random-Max in Figure 3.6) is 1373 while the average $flex_I$ of solutions created by the Flexi-Solver is only 1143. There is room for improvement of about 200 units of slack, which is a substantial amount for a problem with 60 tasks.

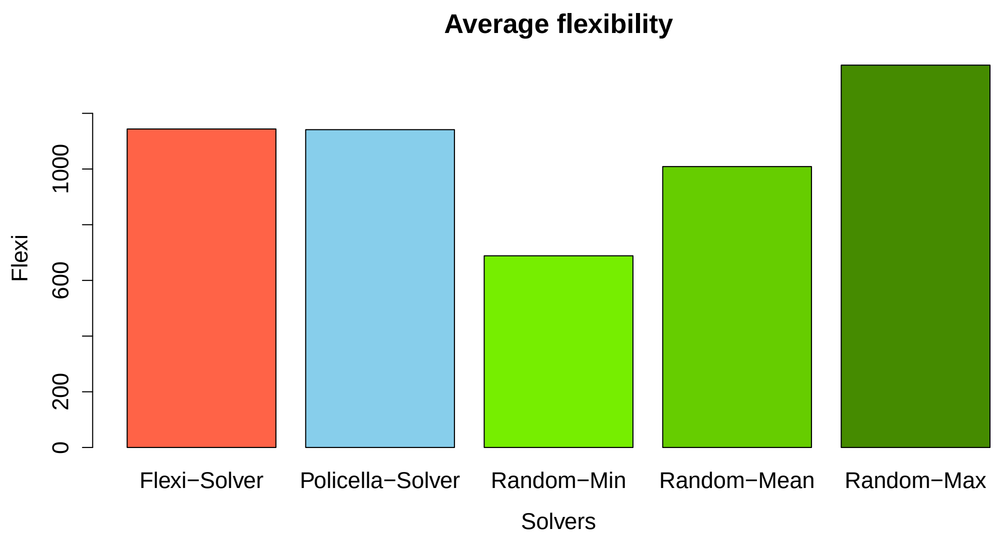


Figure 3.6: The performance of the Random-Solver compared to the Flexi-Solver and Policella-Solver.

3.5.2 Conclusions

At the start of this section we were dissatisfied by the negative results of the Flexi-Solver so we took a closer look at the solution space to get a better understanding of why the global metrics are unable to create better solutions. Specifically, we wanted to know how much the selection function influences the resulting schedules. This allowed us to better strengthen our assumption that the selection function has a notable impact on the resulting schedules.

We found that the constraint selection function does have considerable influence on the amount of flexibility in the resulting POS. This lets us conclude that it should still be possible to improve the Solver and Robustify algorithm by using better constraint selection heuristics. This is somewhat surprising because in Section 3.4 we saw that using different constraint selection heuristics did not have much impact on the results.

The Random-Solver showed that the constraint selection can be changed to allow a solver to create more flexible schedules, but it does not give us any information on how we can improve the constraint selection function, in other words we still do not know which constraint to select, we only know that it can have a significant impact on the results.

In literature the focus of constraint selection was only to reduce the creation of unresolvable conflicts, to ensure a greater chance of finding a solution. The impact of a selected constraint on the flexibility of the solutions was not considered. At the start of this chapter we assumed that the constraint selection function could be modified to not only avoid unresolvable conflicts, but to also improve the flexibility of the resulting schedules. We assumed that any extra slack or flexibility gained during the constraint selection, would be visible in the resulting schedules.

So we assumed that maximizing flexibility during constraint selection would provide schedules with a higher amount of flexibility, but this does not seem to be the case. In the next section we will investigate this more closely.

3.6 Using a lookahead approach

Based on the results from our experiments in Section 3.4, we concluded that maybe the constraint selection function is not well suited to be optimized, however in the previous section we concluded the opposite. In this section we will have a closer look at these seemingly contradicting results.

For the first experiments in Section 3.4 we created four different solvers: Flexi-Solver, Policella-Solver, RM1-Solver and RB-Solver. All four solvers produced similar results. This allows us to conclude that the constraint selection function may not have enough influence to change results. The follow-up experiments in Section 3.5 showed us that this was not the case, so we need to re-evaluate our hypothesis.

The main reason for the contradicting results is due to the Random-Solver not making greedy selection choices based on flexibility. The Random-Solver acts as a black-box, where it only looks at the final result and is only able to produce better schedules because it is executed many times so we do not know which constraints the Random-Solver selects. The Flexi-Solver did not show the same improvement as the Random-Solver, which would suggest that it does not consistently pick the right precedence constraints.

The results above could allow one to conclude that optimizing on a global flexibility metric is not beneficial, however before we can assume that this is true we will have a closer look at how the Flexi-Solver works. The Flexi-Solver selects in a greedy way the constraint that maximizes the global flexibility metric. Until now we assumed that the selection *metric* was the failing part of the constraint selection function, however all three solvers Flexi-Solver, RM1-Solver and RB-Solver selected constraints in a greedy fashion. The reason for the poor results in these three solvers may very well be the greedy approach of constraint selection.

So we have to ask the following question: Are the proposed solvers performing poorly because the greedy selection process is not good enough or is optimizing on flexibility so early in the solving process pointless?

To answer the question above we propose to create a lookahead constraint selection function. The greedy approach checks the impact of a selected constraint only at the moment it is added to the temporal network. A lookahead constraint selection function will look ahead to see what the impact of the selected constraint is in later iterations of the solving process. As we can recall the PCP paradigm used by Solve and Robustify will continue to add precedence constraints to a temporal network until all resource constraints are resolved.

The lookahead constraint selection function will calculate a number of theses PCP iterations. This will have a negative impact on the runtime performance of the Solve and Robustify algorithm but it gives us some interesting information, namely if it still produces poor results we can conclude that optimizing the constraint selection function on flexibility will never produce more flexible schedules.

In the following section we will give a short motivation why we believe a lookahead approach should provide us with a positive result. We will then describe the implementation of the lookahead constraint selection function.

3.6.1 Motivation of using lookahead

We believe a lookahead in the constraint selection function will help the function pick better constraints because it will be less shortsighted. In this section we will have a look at some results that will enforce this hypothesis.

The results from the first few experiments show that the different solvers produce on average the same solutions, but if we look at solutions of individual problem instances we will see that there is more variation. This variation can be attributed to the fact that the simple greedy constraint selection function is unable to fully take the impact of an added constraint into account. This results in a solver performing very well on one problem instance, while performing badly on the next.

The look ahead will allow the constraint selection function to gain insight at the expense of running time. It is our expectation that using a small but constant lookahead will allow the ESTA algorithm to avoid selecting constraints that only provide a short sighted improvement in flexibility, which will result in more flexible solutions.

3.6.2 Implementation

The implementation for the *Lookahead-Solver* is the same as the Flexi-Solver with only a modified constraint selection function. The constraint selection function used by the Flexi-Solver was described earlier in Algorithm 4.

The constraint selection function used by the Lookahead-Solver uses the same input as the one used by the Flexi-Solver and has the same output. However the lookahead function will execute a number of iterations of the PCP algorithm before selecting a constraint. A formal definition of this can be seen in Algorithm 5.

Algorithm 5: Constraint selection function used by Lookahead-Solver.

```
Input: A project graph $G(N,E)$ and a conflict pair $(t_i \in N, t_j \in N)$
Output: A constraint $c$
$score1 \leftarrow lookahead(G(N, E \cup \{(t_1,t_2)\}, depth, metric)$
$score2 \leftarrow lookahead(G(N, E \cup \{(t_2,t_1)\}, depth, metric)$
if $score1 > score2$ then
    return $(t_1 \rightarrow t_2)$
else
    return $(t_2 \rightarrow t_1)$
```

The *lookahead* function is a recursive method shown below. It takes a constraint and a metric as input and returns a score based on the metric. The *depth* parameter sets the number of lookahead steps.

```
Function Lookahead(G, d, metric)
    Input: A project graph $G(N,E)$ and depth count $d$ and a flexibility metric $metric$
    Output: score
    if $unsolveable(G)$ then
        return 0
    if $d = 0 \vee hasNoPeaks(G)$ then
        return $metric(G)$
    $(t_1,t_2) \leftarrow selectConflict(G)$
    $score1 \leftarrow lookahead(G(N,E \cup \{(t_1,t_2)\}, d-1, metric)$
    $score2 \leftarrow lookahead(G(N,E \cup \{(t_2,t_1)\}, d-1, metric)$
    return $\max(score1, score2)$
```

Using the above methods we can create a few different lookahead solvers using different flexibility metrics.

3.6.3 Experiments and conclusions

Using the Lookahead-Solver described above we can determine if the lack of more flexible schedules is due to the greedy approach of constraint selection or due to the flexibility metric selecting wrong constraints. With every extra lookahead step the solver will make less shortsighted decisions, however using lookahead will have great impact on the running-time of the algorithm which is impractical for NedTrain. Fortunately we only need to run the solver once to see why the modified constraint selection function is not helping the solver to create more flexible schedules.

For the following experiments we created the Lookahead-10-Solver using $flex_I$ as selection heuristic and with a lookahead of 10. A lookahead of 10 steps requires a processing time of 15 minutes per instance on a 2.33Ghz Intel Xeon core. More lookahead steps cause the solver to become even slower without any real benefit.

Despite the much longer processing time the Lookahead-10-Solver only shows a very small improvement. We can see this in Figure 3.7.

While the Lookahead-10-Solver is too slow for NedTain it did allow us to come to an interesting conclusion. The solver did a lookahead of 10 iterations, this is quite extensive and it still only performed marginally better than the Flexi-Solver, which is an interesting result. With these results and the results from the random solver we can conclude that maximizing flexibility using the constraint selection function is not going to produce more flexible schedules.

However there is still something odd about the results of the lookahead solver in combination with the results of the random solver. The lookahead solver spends much time carefully selecting constraints that lead to a more flexible temporal network, but this flexibility is almost not visible in the final schedules. In the following section we will have a look at how

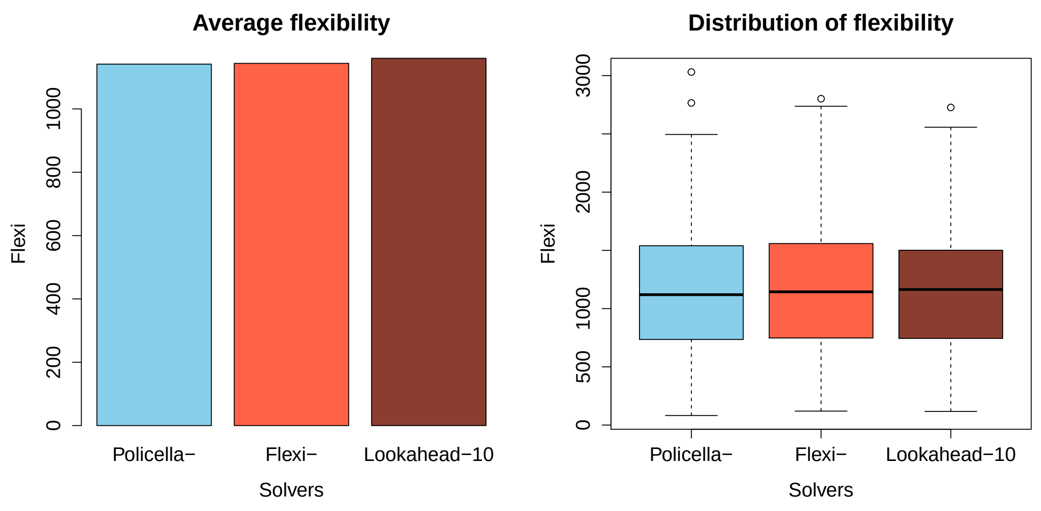


Figure 3.7: Performance of Lookahead-10 compared to Solvers from Section 3.4.

the Robustify phase (Chaining) does not put any extra flexibility hidden in the fixed-time scheduling to good use.

3.7 Solve and Robustify

To understand why the Random-Solver was able to produce more flexible schedules than the Policella-Solver and the Lookahead-10-Solver we need to go back to one of the first assumptions we made. The driving assumption used by Policella was that the Solve phase and Robustify phase are independent of each other, which allowed the solver to first focus on finding a fixed-time solution and then expanding the fixed-time solution into a POS with a certain amount of flexibility. The results from the Random-Solver showed that the first phase does have a significant influence on the flexibility of the resulting POS, however the Lookahead-10-Solver does not seem to be able to make use of this influence.

In the first phase of Solve and Robustify the ESTA solver uses a temporal network to hold deadline and precedence constraint information. The constraint selection function maximizes the flexibility of this temporal network and not the final schedule (POS). A plausible explanation for the results is that the Lookahead-10-Solver did produce a temporal network in the solve phase with a very high value of flexibility but the robustify phase did not use this extra slack. This would explain all above results. The Flexi-Solver, RM1-Solver, RB-Solver and the Lookahead-10-Solver all produced similar results because the Chaining algorithm is simply unaware of the slack in the fixed-time schedule and is unable to use it. The Random-Solver did get better results because it did not necessarily optimize the flexibility of the underlying temporal network. Unfortunately we do not know what the Random-Solver did to find the better solutions because of its black-box nature.

To better understand what is happening let us forget about the Robustify phase for a bit and measure the flexibility of the temporal network at the end of the Solve phase. This will give use a better idea of what information is passed to the Robustify phase. The three solvers: Policella-Solver, Flexi-Solver and Lookahead-Solver are run again, only now we take the flexibility measured by $flex_I$ before chaining is applied.

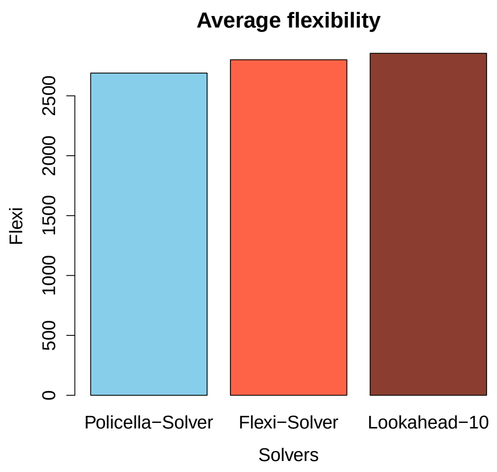


Figure 3.8: Flexibility measured using $flex_I$ before chaining is applied.

The average flexibility measured is shown in Figure 3.8. The results of the first three bars Policella-Solver, Flexi-Solver and Lookahead-Solver have the proportions we expected at the start of this chapter. When we introduced the idea of using a global metric we used Example 5 to support our hypothesis. The increase in flexibility before chaining is what we expected to see. If we compare these results with the results of the flexibility measured after chaining (see Figure 3.7) we can see that the Chaining algorithm is indeed unable to utilize the extra flexibility hidden in the temporal layer of the fixed time schedules.

For completeness we can also measure the flexibility before chaining of solutions created by the Random-Solver. Let us once again run the Random-Solver 2000 times and save the best schedules (the schedules with the highest value of $flex_I$ after chaining). We can then measure the average flexibility before chaining in the same way as we did with the three solvers above. If we compare the results with the other three solvers (see Figure 3.9) we get an interesting result. Namely the flexibility of the temporal network created by the Random-Solver before chaining is much lower than the other solvers, while in Figure 3.6 we saw that the flexibility of the best schedule found by the Random-Solver after chaining is much higher then the other solvers.

This would suggest that our original proposition of improving flexibility in the first phase of Solve and Robustify is not only unhelpful but also counterproductive. We can now conclude that improving flexibility early in the Solve and Robustify algorithm does not improve the flexibility of the created schedules and we also know that this is due to the chaining algorithm not using the extra flexibility.

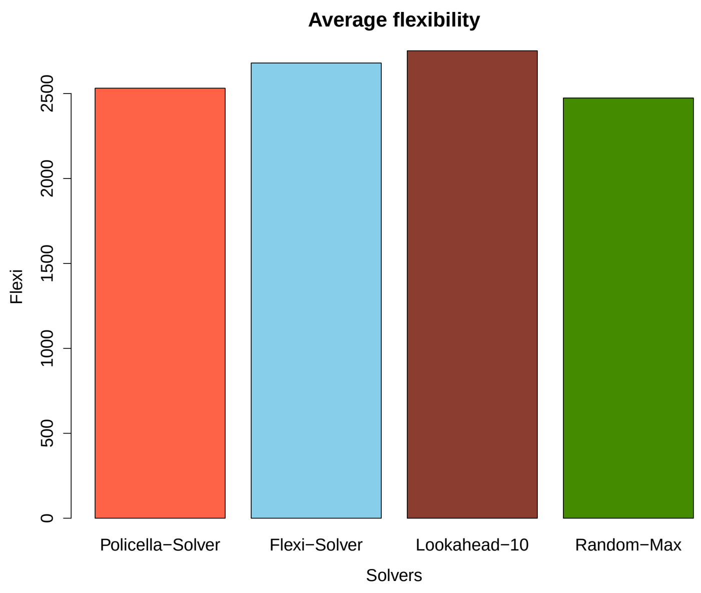


Figure 3.9: Flexibility measured using $flex_I$ before chaining is applied. With the best schedules found using the Random-Solver

3.8 Conclusions

In this chapter we attempted to improve the Solve and Robustify algorithm by changing the ESTA solver by giving the constraint selection function a more insightful metric to measure flexibility. The driving idea behind this approach was that the original Solve and Robustify is focused on the harder RCPSP/max model than the Task RCPSP model. With the extra leeway we wanted to modify the ESTA solver in such a way that it would produce an ESTS with a more flexible underlying temporal network. To test this we created a solver called Flexi-Solver.

Unfortunately using $flex_I$ or any other flexibility metric to guide the search did not improve the flexibility of the final POS. We were not satisfied with these results because not only did we see little to no improvement, there was also little to no change. So we created a Random-Solver to re-evaluate our assumption that the constraint selection function has influence on the created schedules.

The experiments showed that the constraint selection function has enough influence to produce more flexible schedules, but that Flexi-Solver does not utilize this influence in the right way. To find out if the Flexi-Solver was failing due to the use of the greedy constraint selection method or if optimizing flexibility was just ineffective, we created the Lookahead-10-Solver. Using this solver we showed that optimizing flexibility using the constraint selection function does not improve the flexibility of the resulting schedules.

To better understand why the optimization of the constraint selection function does not help to improve the solver as a whole we looked at the flexibility of the temporal network between the solve and robustify phases. This showed that the constraint selection function does improve the flexibility of the temporal network before the chaining algorithm is applied but that the chaining algorithm does not use this extra flexibility.

Finally we concluded that our intuition to focus on improving the flexibility at the constraint selection branch point of ESTA, is only correct if chaining is not applied, unfortunately chaining is a key part of the Solve and Robustify process and we are unable to create flexible schedules without it. We did however find that the constraint selection function has a significant impact on the flexibility of schedules created by Solve and Robustify and that the original Min-Slack does not exploit it. Unfortunately we do not know *how* we can modify the constraint selection function to improve it.

Chapter 4

Interval Schedules and CFSA

At the end of Chapter 2 we defined the research question: *"Can the $flex_I$ metric or its underlying Interval Schedule help the CFSA algorithm find a partial order schedule (or Interval Schedule) that has a greater value of $flex_I$?"*. So in this chapter we will propose a change to the *Conflict Free Solutions Algorithm* (CFSA) that we introduced in Section 2.5. Using Interval Schedules (see Section 2.3) we will change the way CFSA creates its resource profile, which will reduce the number of peaks and allow it to solve problems by posting less constraints than the original. We will first look at a motivation of the proposed changes and then see how CFSA can be implemented using the independent time intervals described in an Interval Schedule.

The change we will propose will cause CFSA to first create a valid Interval Schedule which can then be converted to a POS. It is our intention to improve the flexibility of both the Interval Schedule and the POS. It is not necessarily needed to convert the Interval Schedule to a POS because an Interval Schedule is already well suited for NedTrain. Interval Schedules can already provide the needed freedom for maintenance teams and it can also safeguard project constraints.

Due to time restrictions we will only run preliminary experiments to show that using *Interval Schedules* has a positive impact on the flexibility of the created schedules. We will however look at possible improvements for future work.

4.1 Inaccurate bounds used by CFSA

The Conflict Free Solutions Algorithm uses an inaccurate lower-bound (LB_D) and upper-bound (UB_D) resource (demand) profile. The inaccuracy of the bounds prevents the solver from finding efficient constraints to post, leading to over constrained solutions or no solutions at all. This was the main argument used in the past to discard CFSA when the ESTA+Chaining algorithm was introduced by Cesta and Smith.

Example 6 (CFSA creates a needlessly over constrained solution)**.** *Given is a simple problem P with 3 independent tasks t_1, t_2, t_3, each with a duration of 1. The problem has 1 resource r with a capacity of 2, and each task t_1, t_2, t_3 requires 1 unit of r.*

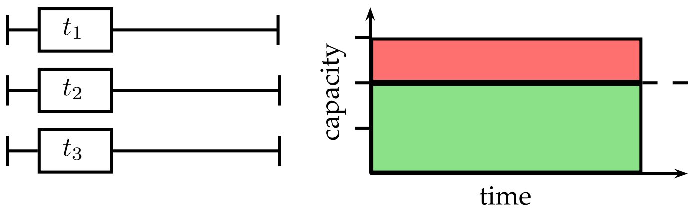


Figure 4.1: Illustration of the temporal network and resource profile of Example 6.

With 3 independent tasks and a resource limit of 2 we can see that only one constraint is needed to create a valid POS. For example posting a constraint: $t_1 \rightarrow t_2$, will ensure that never more than 2 resource units are used at any one time.

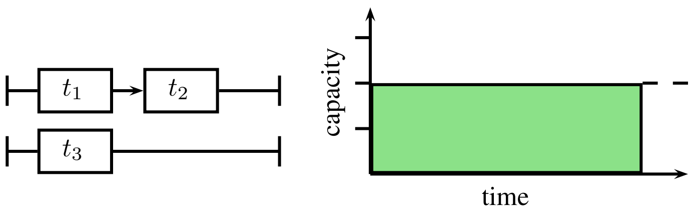


Figure 4.2: An optimal solution for Example 6.

However the CFSA algorithm calculates the resource profile using a lower- and a upper-bound. Once a constraint $t_1 \rightarrow t_2$ is posted the CFSA upper-bound will still find a peak of 3 units because it only looks at pairs of tasks and assumes the worst case. For example the peak $UB_D(s_{t_3}) = 3$, i.e. task t_3 requires 1 resource for itself and it can overlap with t_1 and t_2 which adds another 2 resource units to the resource profile. The UB_D function used by CFSA does not see that t_1 and t_2 are constrained, because it only looks at individual task pairs. To remove the upper-bound peak the CFSA algorithm posts for example another constraint $t_2 \rightarrow t_3$. The resulting POS is now completely ordered and the maximum number of concurrently used resource units is 1 (see Figure 4.3).

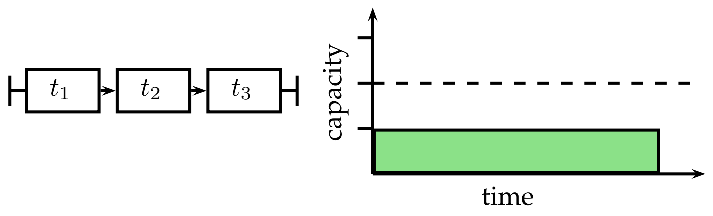


Figure 4.3: CFSA solution for Example 6.

Example 6 shows how a small schedule would be created using the original CFSA algorithm used by Cesta [6]. It shows that CFSA posts more constraints than necessary because the upper-bound is not accurate enough. This is caused by the pairwise approach of calculating the demand profile. As shown in Section 2.5, CFSA uses UB_D to calculate the upper bound of the profile. UB_D only looks at task pairs. For example if the demand profile is calculated for t_1, it will look at the resource demand of the pair (t_1,t_2) and the pair (t_1,t_3). The constraints between the tasks within the pair are considered, however any constraints between t_2 and t_3 are ignored. This causes the upper-bound demand profile to overestimate the demand, which causes extra precedence constraints to be posted.

Improving the accuracy of the upper-bound is no simple task. To find a worst case resource demand require finding the maximum concurrent clique between all the tasks within a given start time interval. In other words, if we want to find the accurate maximum demand of a given resource r during the start time window of a given task t_1, we will need to find all tasks that are concurrent with t_1 and also concurrent with respect to each other. Current implementations of CFSA use only a set $T_c(t_1)$ of tasks that are concurrent with t_1 but not necessarily concurrent with respect to each other. UB_D returns the total resource requirement of this set. To improve the accuracy we need to reduce the set $T_c(t_1)$ to a mutually concurrent set that maximizes the resource requirement. This is equivalent to the maximum clique problem which is known to be *NP-hard* [17].

4.2 Using $flex_I$ to improve the bounds

The previous section shows that the resource demand profile is inaccurate because of interactions between tasks. We propose to use $flex_I$ intervals to overcome this problem. CFSA uses start intervals and precedence constraints to test whether two tasks are concurrent, however it is unable to take all precedence constraints into account. If we use $flex_I$ intervals instead we remove the need to take any precedence constraints into account, which allows for the creation of an accurate resource profile.

Using $flex_I$ intervals allows us to create a demand profile in very much the same way as is done in ESTA. ESTA calculates the demand profile from a fixed-time schedule, namely the Earliest Start Time Schedule. Using $flex_I$ intervals we are able to create an upper-bound fixed-time schedule for which we can then create an exact resource profile. This is possible because precedence constraints are no longer relevant in an Interval Schedule. All tasks are free to start at any time within the given intervals. In the next section we will look in more detail how this upper-bound fixed-time schedule can be created.

By using an upper-bound fixed-time schedule the problems presented in Example 6 are no longer relevant. In Example 6 the resource profile has one peak demanding three resource units. The peak contains three unconstrained tasks. After the constraint $t_1 \rightarrow t_2$ is posted the intervals are adjusted using $flex_I$. With the new interval there are two peaks with a resource demand of two units each. Problem P has a resource capacity of two and so with the adjusted intervals the problem is solved. Figure 4.4 shows a visualization of the two states before and after the constraint $t_1 \rightarrow t_2$ is posted. This example was originally used by Cesta [6] to disqualify CFSA. However by using $flex_I$ intervals we are able to discredit the example.

Using an Interval Schedule to restrict the start-time windows of tasks allows us to create an accurate upper-bound profile because each task has it own independent time interval. How the slack is distributed between the independent time intervals does not have any impact on

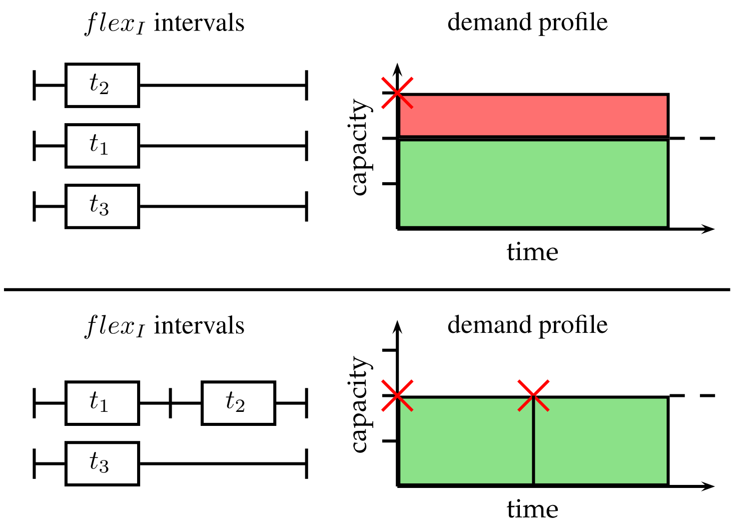


Figure 4.4: Two iterations of CFSA using $flex_I$ intervals, before and after posting $t_1 \rightarrow t_2$. On the left two Interval Schedules and on the right two resource profiles.

the accuracy of the upper-bound resource profile. However the distribution of slack does have an effect on the peaks in the resource profile. The number of overlapping intervals may change when using a different $flex_I$ objective function, which will change which tasks can be run concurrently. How a different distribution of slack can create different peaks is shown in the following example.

Example 7 (How $flex_I$ slack distribution creates different peaks)**.** *Given an RCPSP instance $P = \langle T,R,C\rangle$, where T has 8 tasks $\{t_1,t_2,\dots,t_8\}$ and where C has 9 precedence constraints*

$$\{(t_1,t_3),(t_1,t_5),(t_2,t_5),(t_3,t_4),(t_5,t_6),(t_5,t_7),(t_4,t_8),(t_6,t_8),(t_7,t_8)\}.$$

Each task has a duration of 2 and a deadline of 10. Figure 4.5 shows the problem as a project graph.

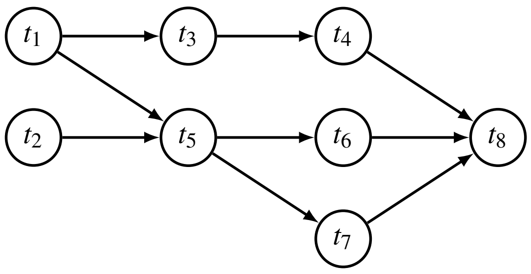


Figure 4.5: Project graph representation of problem P from Example 7.

The project graph shown in Figure 4.5 has many Interval Schedules associated with it. Let A,B,C be three different schedules shown in Figure 4.6. A resource peak can only start *when a interval starts so we only need to look at the starting points of each interval and see which other intervals are active. If we look at schedule A the first peak is $\{t_1,t_2\}$ followed by $\{t_3,t_5\},\{t_4,t_6,t_7\},\{t_8\}$. However if we look at the other two Interval Schedules B and C we can see that the peaks are different. For B the peaks are*

$$\{t_1,t_2\},\{t_3,t_5\},\{t_3,t_6,t_7\},\{t_4,t_6,t_7\},\{t_8\}$$

while for schedule C the peaks are

$$\{t_1,t_2\},\{t_3,t_5\},\{t_4,t_5\},\{t_4,t_6,t_7\},\{t_8\}.$$

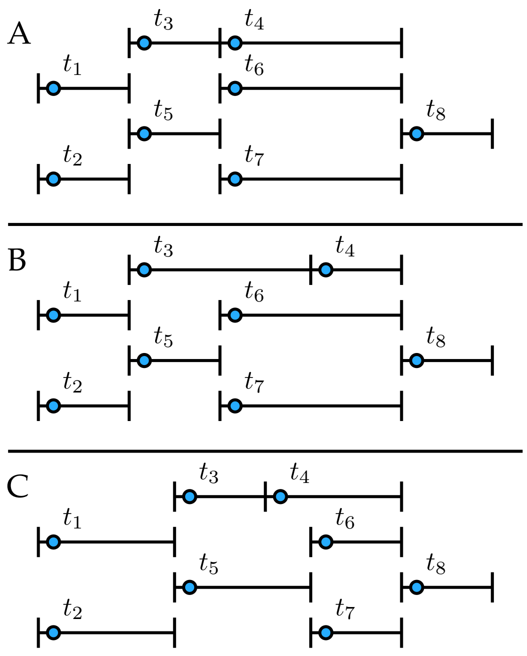


Figure 4.6: Three possible Interval Schedules for problem *P*.

It is up to the selection heuristics to select a peak that violates a resource constraint and solve it. As Example 7 shows, different peaks are found if different $flex_I$ distributions are used. This leaves us with two questions:

1. Does using different $flex_I$ distribution objectives results in different solutions?

2. Which of the different existing distribution objectives provide better results?

In the following section we will formulate some hypotheses but first we will look into the implementation details of CFSA with $flex_I$ intervals.

4.2.1 Implementation of CFSA with $flex_i$ intervals

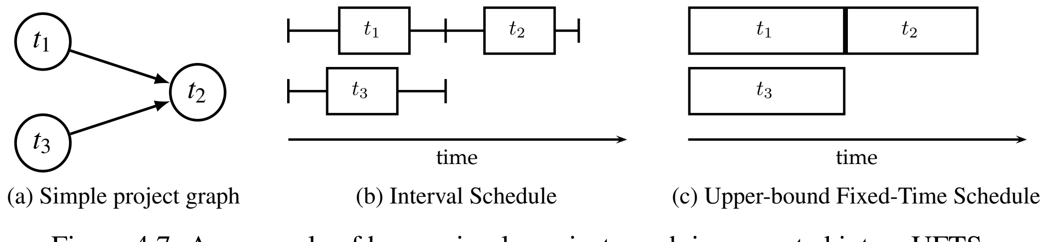
In the previous sections we proposed to use $flex_I$ intervals to calculate a tight upper-bound of the resource profile. In this section we will describe the implementation of a modified CFSA algorithm. The original CFSA algorithm is described in Chapter 2.

Before we explain how we modified CFSA we will introduce the *Upper-bound Fixed-Time Schedule* (UFTS) and how it can be created. A UFTS schedule is very much like the Earliest Start Time Schedule (ESTS) used by ESTA. An ESTS is a list of start-times for a set of tasks with a fixed duration. The UFTS is the same with the exception that the duration of the tasks has been modified.

An UFTS is created by first creating an Interval Schedule using $flex_I$. A new set of task durations is created for each task i.e. given a set of T tasks, a new set of durations is created as follows: $\{\forall t \in T | d_t = b_t - a_t\}$ where d is duration and a, b are $flex_I$ interval bounds. Algorithm 7 shows this in a more formal way. Figure 4.7 shows an example of a simple project graph being converted to a UFTS in three basic steps.

Algorithm 7: Creating an upper-bound fixed-time schedule using $flex_I$

```
**Input**: A problem $P$
**Output**: A fixed schedule $S$
$P_2 \leftarrow P$
$S_I \leftarrow createIntervalSchedule(P)$
**forall** *task* $t_i \in P_2$ **do**
    $d_{t_i} \leftarrow b_i - a_i$
$S \leftarrow getEarliestStartTimeSchedule(P_2)$
**return** $S$
```



(a) Simple project graph (b) Interval Schedule (c) Upper-bound Fixed-Time Schedule

Figure 4.7: An example of how a simple project graph is converted into a UFTS.

The ESTA algorithm is able to create a fixed-time schedule by creating an ESTS. We can also create a fixed-time schedule with CFSA by creating a UFTS. Once a UFTS is created that has no resource violations the start and end time can be extracted to create a resource feasible Interval Schedule.

The original CFSA produced a POS and not an Interval Schedule. Fortunately converting an Interval Schedule to a POS can be done in a simple and very naive way without loss of flexibility (if measured by $flex_I$), by iterating over all the tasks t_i in the Interval Schedule and posting a precedence constraint to any task t_j for which the earliest start time of t_j is greater than the latest finishing time of t_i. This process of creating a POS is very naive and adds more constraints than needed but it will not reduce the amount of independent slack ($flex_I$) in the schedule because the size of each task interval is maintained.

So the modified CFSA solver first creates an UFTS and removes a conflict pair using the same heuristics as ESTA. Then a new UFTS is created using the updated problem instance. Once a resource feasible UFTS is created a POS can be created using the naive method described above. Which maximization objective is used by $flex_I$ to create the Interval Schedules is undefined and open to modification.

In Chapter 2 we introduced different objective functions that can be used when calculating $flex_I$ intervals. The $flex_{I_{max}}$ objective function finds the maximum amount of slack in the whole schedule while the other objective functions, $flex_{I_{equal}}$ and $flex_{I_{fair}}$ sacrifice total slack for a more even distribution of slack over the tasks. All three objective functions will cause the above modified CFSA solver to behave differently. So we created three solvers, *CFSA-Flexi-Max*, *CFSA-Flexi-Equal*, *CFSA-Flexi-Fair* using $flex_{I_{max}}$, $flex_{I_{equal}}$ and $flex_{I_{fair}}$, respectively. In the following section we will look at how these solvers perform on the $j60$ benchmarks.

4.3 Experiments and results

Our first set of benchmarks was done using the modified $j60$ problem instances (see Section 3.3) and CFSA-Flexi-Max solver. It is expected that the CFSA-Flexi-Max will not perform very well because the $flex_{I_{max}}$ objective function generally clusters tasks together which will create large resource peaks and the CFSA algorithm will have to post more constraints to resolve them. When compared to the solvers created in Chapter 3, the Policella-Solver and the Flexi-Solver, we can see that they perform equally well, as shown in Figure 4.8.

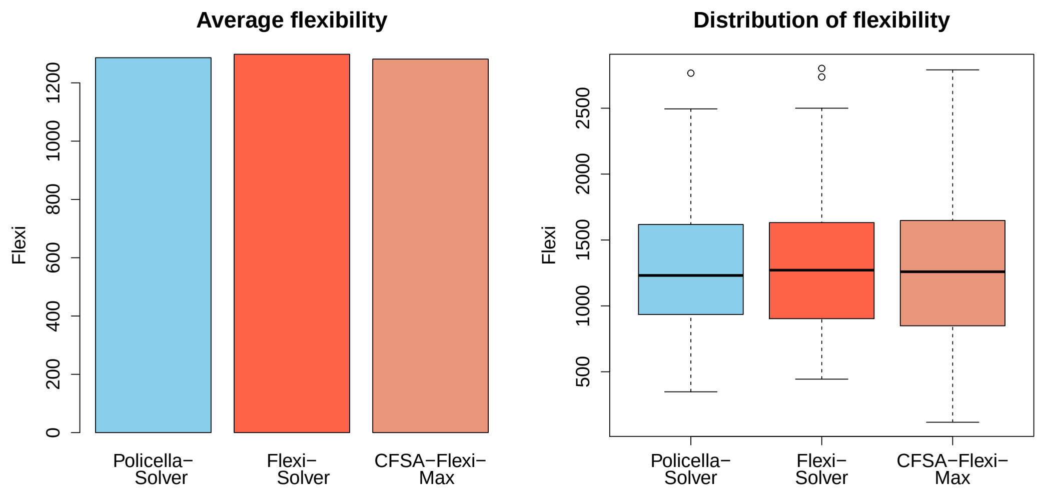


Figure 4.8: Performance of CFSA-Flexi-Max compared to Solvers from Chapter 3.

While the CFSA solver performs slightly less than the Solve and robustify solvers it is still a positive result because Cesta showed [6] that the original CFSA algorithm performed considerably less compared to ESTA. However by using an Interval Schedule we are able to create solutions with almost equal $flex_I$ values.

Next we tested the other two CFSA solvers, CFSA-Flexi-Equal, CFSA-Flexi-Fair. These solvers will generally distribute tasks more evenly over the total project time-line. So we will expect them to perform better then CFSA-Flexi-Max because it will create less resource peaks. However the results were somewhat surprising. As shown in Figure 4.9 the CFSA-Flexi-Max-Fair performed much better than CFSA-Flexi-Max but the performance of CFSA-Flexi-Equal is significantly less. It is clear that CFSA-Flexi-Fair produces much better results than the other solvers.

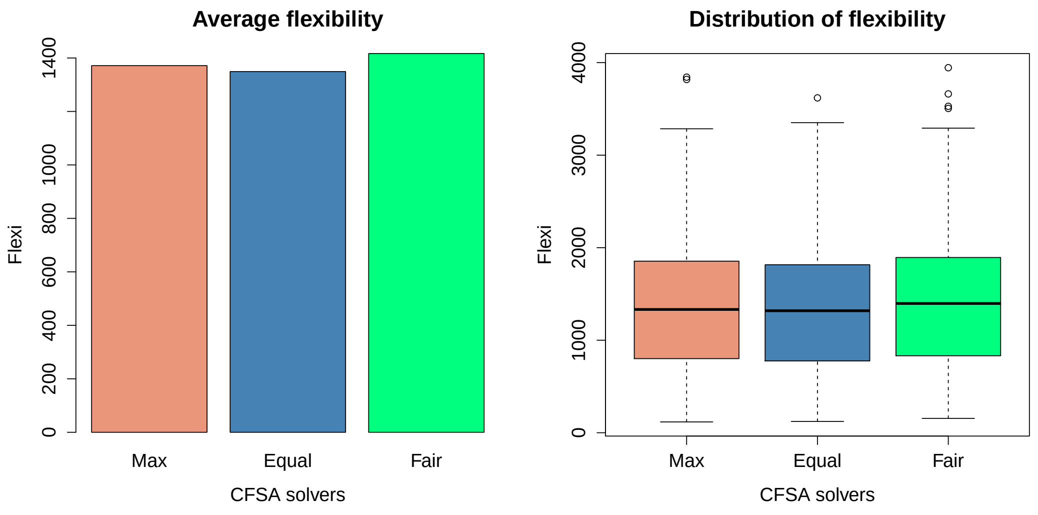


Figure 4.9: Performance of CFSA solvers compared to each other.

Initial results show that using Interval Schedules in combination with CSFA allows CFSA to produce schedules with a higher value of $flex_I$ compared to the Solve and robustify approach. There is still room for improvements which we will discuss in the next section.

4.4 Proposed improvements

Due to time restrictions we only touched upon the possibilities of using Interval Schedules within the CSFA algorithm. Preliminary results show there is merit to continue to investigate this subject matter more. There are three open questions that are open to more research:

1. Results shown in Figure 4.9, indicate that different objective functions for $flex_I$ can greatly impact the resulting schedules, however we only tested 3 different variations.

2. Due to the use of Interval Schedules the interaction between tasks is only relevant to the solver creating the Interval Schedules and so heuristics used to select conflicts and precedence constraints may no longer be well suited and should be reviewed.

3. Finally the modified CFSA initially creates an Interval Schedule and not a POS. The solvers created above currently have a naive method of converting an Interval Schedule to a POS. There is some room for improvement by posting less constraints.

The most interesting result seen in Figure 4.9 is that CFSA-Flexi-Equal creates a schedule with a $flex_I$ value even lower than the schedule created by CFSA-Flexi-Max. This was surprising because $flex_{I_{equal}}$ spreads tasks more evenly over the project lifespan than $flex_{I_{max}}$, but less than $flex_{I_{fair}}$. We are able to conclude that distributing tasks evenly between the projects release-time and deadline is not the only factor that allows CFSA-Flexi-Fair to post less constraints. This suggests that there is more going on with the CFSA-Flexi-Fair solver.

The second interesting point mentioned above is to review the heuristics used. The conflict and constraint selection heuristics used by the three new solvers still assume that tasks can interact with each other. However this is no longer the case because the heuristics are working with independent intervals, where tasks no longer interact. In particular the conflict selection function is currently very much focused on finding conflicts that are the most critical. This is measured by the amount of slack between two tasks but this can be misleading because the intervals have been reduced in size and many conflict pairs may seem much more critical than they really are.

Finally the function used to convert the resulting Interval schedule to a POS. As stated in Section 4.2, the naive method does not reduce the amount of $flex_I$ so at first sight there would be no need to improve it. However it still posts many more constraints than necessary which can cause problems when the deadlines of the Interval Schedule is relaxed. As mentioned in Chapter 2 the $flex_I$ metric gives the lower-bound flexibility of a POS. So while the flexibility, in terms of $flex_I$, of the generated POS will not be improved if less constraints are posted the flexibility in terms of more optimistic metric like $RM1$ will be improved. To clarify this let us assume we have an Interval Schedule S. At some point during the execution of the project a deadline in S is not met. At this point the Interval Schedule is no longer valid but the underlaying POS may still be valid because a POS is more relaxed. However if we use the naive method the underlying POS have too many constraints and will also be invalid.

The open questions above are many and it is clear that there is much room for more research into Intervals Schedules and the CFSA algorithm.

Chapter 5

Conclusions and future work

Planning and scheduling for real world projects with execution uncertainty is very challenging, especially were there are also hard deadlines that need to be met. Having a state of the art solver that is able to create optimal but rigid schedules will generally not work. With high execution uncertainty and human labor the need to create flexible and understandable schedules quickly and interactively is of much more value than an optimal but rigid schedule created by black box like solvers. NedTrain is tasked with the maintenance of the rolling stock of NS Reizigers, and struggles with this problem.

In this work we first described the scheduling challenges faced by NedTrain. We then studied current Operational Research (OR) and Artificial Intelligence (AI) literature with the question of NedTrain in mind. At the end of the literature study we formulated more detailed research questions and for the remainder of this work we focused on answering these questions. In the next section we will look at our results from an academic point of view and after that we will see how the results relate to the questions at NedTrain.

5.1 Research

In this work we set out to improve on two existing PCP algorithms using new insights provided by Wilson et al. [36]. Wilson proposed a new flexibility metric which we called $flex_I$ and a new method of defining flexible schedules called *Interval Schedules*. Using the flexibility metric and its underlying Interval Schedule, we proposed to improve the *Solve and Robustify* algorithm and the *Conflict Free Solution Algorithm*.

5.1.1 Research Questions

After the literature research we proposed some research questions. In Chapter 4 we answered the following questions:

- Can the $flex_I$ metric or its underlying Interval Schedule help the Solve and Robustify algorithm find a partial order schedule (or interval schedule) that has a greater value of $flex_I$?

1. Which part of the Solve and Robustify algorithm can be changed, so the algorithm is able to use the extra insight provided by $flex_I$?
2. Does the use of $flex_I$ help the solver create a schedule with a higher value of $flex_I$?

The proposed change to the Solve and Robustify algorithm was to change the *solve phase*, which uses *Earliest Start Time Algorithm* to create a single fixed-time schedule. The ESTA has only one choice point (branch) where it requires a notion of flexibility, which is a local slack metric. It uses this notion of flexibility to determine how much impact a newly added precedence constraint would have on the intermediate *Simple Temporal Network* (STN). It was our initial intention to use the $flex_I$ metric to measure this impact instead of the simple slack based metric proposed by the original paper. Unfortunately the *robustify phase* performed by the *Chaining algorithm* was unable to put the extra flexibility left in the STN to good use. This led to the conclusion that the changes to the ESTA algorithm did not persist after chaining and so improving the ESTA algorithm, by introducing a more sophisticated notion of flexibility, is not beneficial.

Next we focused on the CFSA algorithm. The CFSA algorithm was shown to create over constrained solutions with the introduction of Solve and Robustify, which was able to create much less constrained solutions. With the introduction of Interval Schedules we were able to ask the following questions:

- Can the $flex_I$ metric or its underlying interval schedule help the CFSA algorithm find a partial order schedule (or interval schedule) that has a greater value of $flex_I$?

  1. How can $flex_I$ or an Interval Schedule help CFSA avoid creating over constrained schedules?
  2. Does the use of an Interval Schedule as temporal layer help the solver create a schedule with higher value of $flex_I$?

The proposed changes to the CFSA algorithm proved more beneficial than the changes to Solve and Robustify algorithm. The modification made to the CFSA algorithm made use of the *Interval Schedule* and not $flex_I$ itself. The CFSA algorithm does not use two phases but simply takes the scheduling problem instance and continues to add precedence constraints until the temporal network becomes a valid POS. The problem with CFSA in the past was that the resulting POS was over constrained, due to the solver posting many ineffective and unneeded precedence constraints. It was unable to take existing interactions between tasks into account, which resulted in poor precedence constraints being added to the temporal network. To overcome this we proposed to use an interval schedule as the time layer of the problem instead of the typical precedence graph (project graph). This removed the interactions between tasks and allowed the CFSA algorithm to post fewer but more effective precedence constraints. In short the Interval Schedule allowed us to:

1. Remove interactions between tasks, allowing us to create an exact resource profile and not be dependent on a lower- and upper-bound profile.

2. It allowed us to reduce the size of the initial resource peaks by spreading the tasks out over the whole lifetime of the project.

We validated the above by implementing a modified CFSA. This CFSA algorithm used Interval Schedules created by $flex_I$ to create a resource profile. To distribute the tasks equally over the project lifetime we used the $flex_{I_{fair}}$ objective function. With this simple modification the CFSA algorithm already performed better than Solve and Robustify with respect to $flex_I$.

5.1.2 Future work

We were able to find satisfactory answers to all our research questions. In this subsection we will highlight some possibilities for future work.

We first looked into using $flex_I$ to improve Solve and Robustify. Due to continued negative results we concluded that having the constraint selection function optimize on flexibility is counterproductive because the chaining algorithm that is applied during the Robustify phase does not put any of the found flexibility to good use. However we also saw that the Random-Solver was able to find solutions that were better than the original Solve and Robustify solver. Unfortunately the execution time of the Random-Solver takes too long because finding a good schedule requires many executions. It does provide us with an interesting new question for future work: Which constraints does the Random-Solver select? Also can this be measured before the constraint is posted? If this is the case we could create a constraint selection heuristic that does provide the Robustifty phase with a better starting point.

During the second part of this work we focused on using Interval schedules in combination with CFSA. The preliminary experiments showed positive results, particularly when using the right interval distribution function. Unfortunately due to time considerations we did not have time to work out all the details. There are still many questions open to more research:

1. We only tested three different slack distribution functions when creating an interval schedule. Initial results showed that using the right distribution has much impact on the flexibility of the found schedules.

2. The interval schedule removes all interaction between tasks, however the constraint and conflict selection heuristics, used by CFSA, assume there is interaction and takes this into account. The heuristics can be improved to focus more on selecting constraint based on how it will affect the interval schedule in the next iteration of the PCP process.

3. We also assumed that each task should have its own independent start interval but grouping tasks together under independent agents may allow us to solve much bigger problem instances.

Using Interval Schedules as the temporal layer in the CSFA algorithm is an interesting concept with many aspects that merits more research.

5.2 NedTrain

The section above was mainly focused on a more academic view on the research done in this thesis. In this section we will focus on how the results relate to NedTrain. Generally, NedTrain is interested in increasing the availability of trains and increasing resource utilization. For Technical Maintenance it means that unnecessary slack needs to be removed and the use of resources between maintenance teams needs to be coordinated. Unfortunately in practice the execution of maintenance is very uncertain with many absolute constraints. Flexible schedules that provide management with the needed control and provide the maintenance teams enough autonomy are needed to increase the efficiency of maintenance execution. The questions at NedTrain can be divided into two main parts:

1. How can we create schedules that maximize flexibility?
2. How can we create these schedules in a fast and interactive manner using software assistant tools?

The first question has already been answered for the most part by earlier work. For example, Wilson [34] shows that the Simple Temporal Problem is well suited for modeling the temporal aspects of maintenance projects and that much uncertainty can be accounted for by using Interval Schedules [36]. As mentioned in Section 2.7, NedTrain considers a good and flexible schedule to have the following three properties:

1. reliable deadlines,
2. temporal freedom (flexibility) for maintenance teams to solve small unexpected delays,
3. finally minimizing slack that increase the project deadline.

The Interval Schedule enforces these properties by defining clear task deadlines in such away that any slack in the problem instance is distributed over the tasks in such a way that the slack is most efficiently utilized.

An Interval Schedule is flexible enough to allow maintenance teams to execute tasks with some autonomy. It allows a team to resolve many unexpected delays themselves. With clear start-times and deadlines NedTrain is able to give time guarantees to external parties. Constructing flexible interval schedules also increases resource utilization and at the same time increases maintenance team autonomy. The flexibility (the amount of slack) of a schedule can be measured using $flex_I$.

Ideally, we want to create high quality (flexible) and optimal interval schedules using some kind of software tool. For NedTrain it is important to create an interactive scheduling program that clearly communicates many high quality schedules to the end user. This software has to be fast allowing the planner to update constraints and tweak the schedule.

For the kind of software we described above we need fast solvers, so that they can provide the user with feedback in a timely manner. Work done by Evers [15] demonstrated that PCP algorithms are able to produce resource feasible solutions in a timely manner. In this work we chose to improve two of these PCP algorithms: Solve and Robustify and CFSA.

Solve and Robustify is already focused on creating robust schedules but it creates a POS and not an interval schedule. Fortunately transforming a POS to an Interval Schedule can be done optimally using a linear solver. For this reason we focused our experiments on measuring and improving the flexibility of the created POS. Unfortunately we did not manage to improve the Solve and Robustify algorithm.

The original CFSA performed worse than Solve and Robustify, however after we replaced the temporal layer with an interval schedule the solver started to produce better solutions. The advantage of this modified CFSA algorithm is that it creates a resource valid interval schedule and not a POS. There is still much unknown about CFSA in combination with interval schedules and it is open to more research.

5.3 Future work

In this section we will have a look at some open topics related to maintenance scheduling at NedTrain.

To be able to compare the solvers we used a modified version of the j60 benchmarks created by Kolisch [18]. The j60 benchmarks have many more connected project graphs than we generally see in NedTrain projects. To better compare solvers with each other in the context of the maintenance projects at NedTrain a new set of benchmarks could be created that better matches with NedTrain. To better simulate individual trains, we only want to have precedence constraints between tasks performed on the same train and no constraints between tasks performed on different trains. The final schedule will then only have precedence constraints between trains to coordinate resources used by both trains.

Another topic was already mentioned in the first half of this chapter. Using CFSA to directly create flexible interval schedules is very interesting for NedTrain. Any improvement in solver speed or schedule quality will increase the value of any planning or scheduling tool using the modified CFSA solver.

Bibliography

[1] M. A. Aloulou and M. Portmann. An efficient proactive-reactive scheduling approach to hedge against shop floor disturbances. In G. Kendall, E. K. Burke, S. Petrovic, and M. Gendreau, editors, *Multidisciplinary Scheduling: Theory and Applications*, pages 223–246. Springer, 2003.

[2] J. Blazewicz, J.K. Lenstra, and A.H.G. Kan. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, 1983.

[3] I.M. Bomze, M. Budinich, P. M. Pardalos, and M. Pelillo. The maximum clique problem. In D. Du and P.M. Pardalos, editors, *Handbook of Combinatorial Optimization (supp. Vol. A)*, pages 1–74. Kluwer Academic Publishers, 1999.

[4] G. Brightwell and P. Winkler. Counting linear extensions. *Order*, 8(3):225–242, 1991.

[5] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, January 1999.

[6] A. Cesta, A. Oddi, and S. F. Smith. Profile Based Algorithms to Solve Multiple Scheduling Problems Capacitated Metric. In R. Simmons, M. Veloso, and S. Smith, editors, *Proceedings of the 4th International Conference on Artificial Intelligence Planning Systems*, pages 214–223. AAAI Press, 1998.

[7] A. Cesta and C. Stella. A Time and Resource Problem for Planning Architectures. *Recent Advances in AI Planning*, 1348(4):117–129, 1997.

[8] Amedeo Cesta, Angelo Oddi, and Stephen F. Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, 2002.

[9] C. Cheng and S. F. Smith. Generating Feasible Schedules under Complex Metric Constraints. In B. Hayes-Roth and R. E. Korf, editors, *Proceedings of the 12th National Conference on Artificial Intelligence*, pages 1086–1091, 1994.

[10] B. Y. Choueiry and L. Xu. An efficient consistency algorithm for the temporal constraint satisfaction problem. *AI Communications*, 17(4):213–221, May 2004.

[11] H. Chtourou and M. Haouari. A two-stage-priority-rule-based algorithm for robust resource-constrained project scheduling. *Computers & Industrial Engineering*, 55(1):183–194, August 2008.

[12] R Dechter. Temporal constraint networks. *Artificial Intelligence*, 49(1-3):61–95, May 1991.

[13] A. El-Kholy and B. Richards. Temporal and Resource Reasoning in Planning: the parcPLAN approach. In W. Wahlster, editor, *Proceedings of the 12th European Conference on Artificial Intelligence*, pages 1–5. John Wiley and Sons, Chichester, 1996.

[14] L. Endhoven, T. Klos, and C. Witteveen. Maximum Flexibility and Optimal Decoupling in Task Scheduling Problems. In *Proceedings Intelligent Agent Technology*, pages 33 – 37, 2012.

[15] R. P. Evers. *Algorithms for Scheduling of Train Maintenance*. Master thesis, Delft University of Technology, 2010.

[16] S. Hartmann and D. Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, November 2010.

[17] R. M. Karp. Reducibility among combinatorial problems. In M. Jünger, T. M. Liebling, D. Naddef, G. L. Nemhauser, W. R. Pulleyblank, G. Reinelt, G. Rinaldi, and L. A. Wolsey, editors, *50 Years of Integer Programming 1958-2008*, pages 219–241. Springer Berlin Heidelberg, 2010.

[18] R. Kolisch and A. Sprecher. PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, 96(1):205–216, 1997.

[19] V. Kumar. Algorithms for constraint-satisfaction problems: A survey. *AI magazine*, 13(1):32–44, 1992.

[20] P. Laborie and M. Ghailab. Planning with sharable resource constraints. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 1643–1649. Morgan Kaufmann, 1995.

[21] M. Lombardi and M. Milano. A Precedence Constraint Posting Approach for the RCPSP with Time Lags and Variable Durations. *Principles and Practice of Constraint Programming*, 5732(1):569–583, 2009.

[22] M. Lombardi, M. Milano, and L. Benini. Robust Scheduling of Task Graphs under Execution Time Uncertainty. *IEEE Transactions on Computers*, 62(1):98–111, January 2013.

[23] L. R. Planken. *Algorithms for Simple Temporal Reasoning*. PhD thesis, Delft University of Technology, 2013.

[24] L. R. Planken, M. M. De Weerdt, and R. P. J. Van Der Krogt. P3C: A new algorithm for the simple temporal problem. In J. Rintanen, B. Nebel, C. J. Beck, and E. A. Hansen, editors, *Proceedings of the 18th International Conference on Automated Planning and Scheduling*, pages 256–263. AAAI Press, 2008.

[25] N. Policella, A. Cesta, A. Oddi, and S. F. Smith. From precedence constraint posting to partial order schedules A CSP approach to Robust Scheduling. *AI Communications*, 20(3):163–180, 2007.

[26] N. Policella, A. Cesta, A. Oddi, and S. F. Smith. Solve-and-robustify. *Journal of Scheduling*, 12(3):299–314, October 2008.

[27] N. Policella, A. Oddi, S. F. Smith, and A. Cesta. Generating robust partial order schedules. In M. Wallace, editor, *Principles and Practice of Constraint Programming*, pages 496–511. Springer, 2004.

[28] N. Policella, S. F. Smith, A. Cesta, and A. Oddi. Generating robust schedules through temporal flexibility. In S. Zilberstein, J. Koehler, and S. Koenig, editors, *Proceedings of the 14th International Conference on Automated Planning and Scheduling*, pages 209–218, 2004.

[29] A. A. B. Pritsker, L. J. Watters, and P. M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, 1969.

[30] Stephen F. Smith and Cheng-Chung Cheng. Slack-Based Heuristics For Constraint Satisfaction Scheduling. In R. Fikes and W. G. Lehnert, editors, *Proceedings of the 11th National Conference on Artificial Intelligence*, pages 139–144. AAAI Press, 1993.

[31] T. B. Smith and J. M. Pyle. An Effective Algorithm For Project Scheduling With Arbitrary Temporal Constraints. In D. L. McGuinness and G. Ferguson, editors, *Proceedings of the 19th National Conference on Artifical Intelligence*, pages 544–549. AAAI Press, 2004.

[32] V. C. S. Wiers. The relationship between shop floor autonomy and APS implementation success: evidence from two cases. *Production Planning & Control*, 20(7):576–585, October 2009.

[33] D. Wilmer, T. Klos, and M. Wilson. Distributing Flexibility to Enhance Robustness in Task Scheduling Problems. In K. Hindriks, M. de Weerdt, B. van Riemsdijk, and M. Warnier, editors, *Proceedings of the 25th Benelux Conference on Artificial Intelligence*, pages 255–262, 2013.

[34] M. Wilson, N. Roos, B. Huisman, and C. Witteveen. Efficient Workplan Management in Maintenance Tasks. In P. De Causmaecker, J. Maervoet, T. Messelis, K. Verbeeck, and T. Vermeulen, editors, *Proceedings of the 23rd Benelux Conference on Artificial Intelligence*, pages 344–351. CODeS/KaHo Sint-Lieven, 2011.

[35] M. Wilson, C. Witteveen, T. Klos, and B. Huisman. Enhancing flexibility and robustness in multi-agent task scheduling. In *OPTMAS workshop*, page 18, 2013.

[36] Michel Wilson, Tomas Klos, Cees Witteveen, and Bob Huisman. Flexibility and Decoupling in the Simple Temporal Problem. In F. Rossi, editor, *Proceedings International Joint Conference on Artificial Intelligence*, pages 2422–2428. AAAI Press, 2013.

Appendix A

TMS input format

In this thesis we ran many simulations and experimental tests. For this we used the train maintenance problem description format used by Evers[15]. In this format resources, tasks and other types of information are defined as a set of commands. For example to add a resource to the problem a new line is added to the input that starts with the letter R and any information needed to define the resource is added as arguments after this letter. A full list of commands is shown in Table A.1 and the arguments they expect are shown in Table A.2.

Command	Key	Alises
Resource	R	
Train/Project	T	J,D
Activity/Task	A	
Activity resource requirement	Q	
Precedence constraint	P	p,S

Table A.1: List of all commands.

Command	Arguments			
R	resourceId:int	capacity:int	name:string	
T	trainId:int	start:int	end:int	name:string
A	trainId:int	activityId:int	duration:int	name:string
Q	trainId:int	activityId:int	resourceId:int	amount:int
P	trainId1:int	activityId1:int	trainId2:int	activityId2:int

Table A.2: List of the arguments required by each command.

Below is the TMS input file for our running example of a NedTrain scheduling problem (see Example 1).

```
T 0 0 28 "Train 5100"
R 0 2 "Putspoor"
```

```
A 0 1 1 "t_rt"
A 0 2 2 "t_1"
A 0 3 5 "t_2"
A 0 4 5 "t_3"
A 0 5 3 "t_4"
A 0 6 4 "t_5"
A 0 7 13 "t_6"
A 0 8 2 "t_7"
A 0 9 3 "t_8"
A 0 10 1 "t_dl"

P 0 0 0 2
P 0 0 0 4
P 0 0 0 7

P 0 2 0 3

P 0 4 0 5
P 0 5 0 6

P 0 7 0 8
P 0 8 0 9

P 0 3 0 10
P 0 6 0 10
P 0 9 0 10

Q 0 3 0 1
Q 0 4 0 1
Q 0 6 0 2
Q 0 7 0 1
Q 0 9 0 1
```

Appendix B

Solver implementations

In this work we investigated two different PCP algorithms, *Conflict Free Solution Algorithm* (CFSA) and *Solve and Robustify*. To test, validate and investigate the two algorithms we implemented the two algorithms with varying options. In this chapter we will give a short introduction of the main components of the solvers, how to use them and how to reproduce the results presented in this thesis.

We created one Task RCPSP solver that takes a variety of option allowing it to use different solving techniques. Using different option we are able to create many different solvers. In this work we used the following solvers for experiments:

Policella-Solver Unmodified implementation of Solve and Robustify as described by Policella [26].

Flexi-Solver Solve and Robustify implementation where the embedded ESTA solver uses $flex_I$ for constraint selection.

RM1-Solver Solve and Robustify implementation where the embedded ESTA solver uses $RM1$ for constraint selection.

RB-Solver Solve and Robustify implementation where the embedded ESTA solver uses RB for constraint selection.

Random-Solver Solve and Robustify implementation where the embedded ESTA solver selects a random constraint to resolve a conflict.

Lookahead-10-Solver Solve and Robustify implementation where the embedded ESTA solver uses $flex_I$ and a 10 step lookahead for constraint selection.

CFSA-flexi-max A CFSA implementation where an Interval Schedule, created using $flex_{I_{max}}$, is used to find conflict pairs.

CFSA-flexi-equal A CFSA implementation where an Interval Schedule, created using $flex_{I_{equal}}$, is used to find conflict pairs.

FSA-flexi-fair A CFSA implementation where an Interval Schedule, created using $flex_{I_{fair}}$, is used to find conflict pairs.

The Task RCPSP solver implementation uses Python 2.7. We chose to use Python because it is a scripting language that allows for fast development and for a scripting language it is known to be relativity fast in its execution. Finally, Python is widely supported by many systems.

To calculate the $flex_I$ metric and to create Interval Schedules we used the Gurobi Optimizer. We chose to use Gurobi because it has a simple and easy to use Python interface, which allows use to define and solve linear models within our Python code.

B.1 Solver usage

The main solver *SolverTMS.py* can be called from command-line with the following command-line

```
$ python SolverTMS.py [TMS-problem-instance]
```

The solver has many options which can be viewed using the help flag -h. The solver can also be used with the TMS scheduling GUI using the -g flag. The TMS schedule GUI is an application that visualizes a TMS problem instance and shows how a solution is created by a PCP solver.

B.2 Benchmarks

The SolverTMS.py was mainly used in combination with the TMS scheduling GUI to get a better understanding for the behavior of the different solvers. When solving the PSPLib benchmarks, by Kolisch [18], we used the following three benchmark scripts:

```
$ python run_benchmarks_esta_cfsa.py
$ python run_benchmarks_lookahead.py
$ python run_benchmarks_random.py
```

All three scripts create CSV files in the `gen_results` folder. The CSV file can be analyzed using *R*, which is a free software environment for statistical computing and graphics.