Maximal Flexibility and Optimal Decoupling in Task Scheduling Problems

Master's thesis

Leon Endhoven

Maximal Flexibility and Optimal Decoupling in Task Scheduling Problems

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

APPLIED MATHEMATICS

by

Leon Endhoven
born in Poeldijk, the Netherlands

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

Maximal Flexibility and Optimal Decoupling in Task Scheduling Problems

Author: Leon Endhoven
Student id: 1275917
Email: leon.end@gmail.com

Abstract

This thesis focuses on the properties of (multi-agent) task scheduling instances represented as Simple Temporal Problems (STP). By defining a subclass $\text{STP}_{\prec}$ of STPs that contain these task scheduling instances, existing algorithms for arbitrary STPs can be improved if applied to task scheduling STPs, allowing arbitrary schedules and temporal decouplings to be created more efficiently.

With the introduction of a new flexibility metric in this thesis, a Linear Programming (LP) formulation as well as an alternative Maximum Flexibility Algorithm is given to create maximally flexible open schedules from which an optimal temporal decoupling can be derived. This thesis also contains a proof that in task scheduling instances, contrary to intuition, an optimal temporal decoupling does not reduce the flexibility of the system.

In order to ensure fair decouplings or open schedules for either the tasks or the agents, three types of egalitarian flexibility problem formulations are presented including LP formulations to solve these problems.

Thesis Committee:

Chair:	Prof. Dr. ir. K.I. Aardal, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. C. Witteveen, Faculty EEMCS, TU Delft
Committee Member:	Dr. T.B. Klos, Faculty EEMCS, TU Delft

Contents

Chapter 1

Introduction

We are confronted with it every day: tasks that have to be completed at a specific time or in a predetermined order. Some tasks we perform on the fly, while other, possibly larger, tasks require some more planning in advance. Such a task scheduling problem can be solved by creating a *schedule*, which assigns a starting time and/or a completion time to a task. Since scheduling problems occur in many domains, its applications are widely used and are therefore an important area of research. In this thesis we will focus on *task scheduling problems* which are a subclass of the class of scheduling problems and have some distinct properties including a specific ordering of the tasks. Combined with the time-constraints placed on each task, this allows us to assign starting times to each task and thereby create a schedule solving the task scheduling problem.

Our task scheduling problems will in contrast to most scheduling problems feature multiple parties that are involved in the scheduling and execution of their specific set of tasks. An important case where such a *distributed* scheduling problem is featured, is the case of NedTrain, a Dutch company.

NedTrain Case. *The Dutch company NedTrain has several locations in the Netherlands where they have stationed there maintenance centers, which are used to perform the maintenance on the rolling stock units of the Dutch railway company NS. In every maintenance center there is a fixed amount of time available for the maintenance of each rolling stock unit, since there is an agreed upon arrival and departure time of the train scheduled for maintenance. During this time, multiple tasks have to be completed, where some are in a specific order, meaning that a task t has to be completed before task t' is allowed to start. Each task has to be completed by a specialized workcrew, from which there are three present.*

Once a task is started, it takes a certain time to be completed, which is referred to as the processing time *of the task. Some tasks have additional constraints placed on them, which can occur due to some resource limitations. These constraints involve a specific time after which the task is allowed to be executed, which is referred to as a* release date*. Due to the agreed upon departure time, all tasks have a fixed* due date*, which functions as a deadline for each task. A* workplan *can be created that represents all release dates, due dates (or deadline of the total plan), processing times and relations between the tasks, as is shown in*

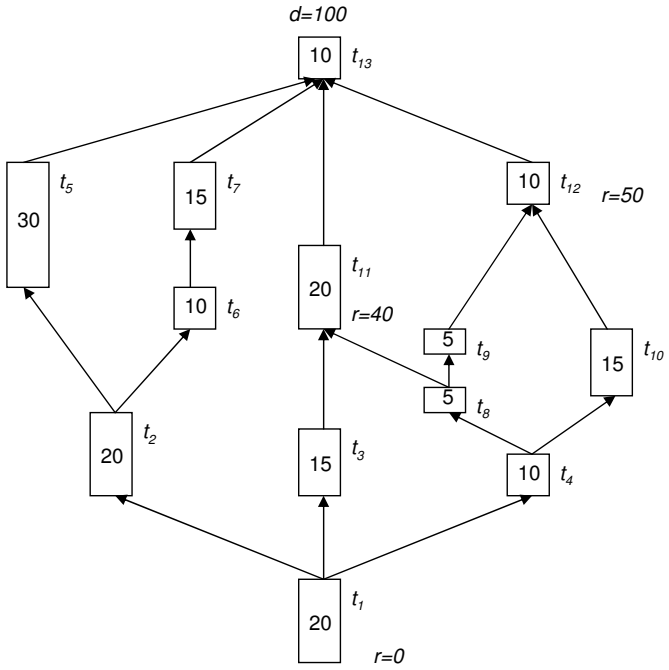

Figure 1.1: Workplan, where each node represents a task i that is labeled with t_i, each node contains its processing time, may have a release date r or due date d attached and every edge between two nodes denotes that the task at the beginning of the edge has to be completed before the task at the end is allowed to start

Figure 1.1. With this workplan alone however, the NedTrain management has no working schedule to present to its employees. Since its employees at the site where this workplan is used are divided into three different specialized workcrews, as is shown in Figure 1.2, they do not yet know when to start their tasks, as some tasks depend on preceeding tasks that have to be executed by one of the other two workcrews. Starting tasks on the fly for each workcrew is not an option, since this may lead to workcrews waiting for each other of starting a task that is not allowed to start yet. Furthermore, it gives no guarantee that all tasks are completed before the deadline of the project. NedTrain is however also interested in another approach: Apart from giving the workcrews a single schedule to work with, they are interested to see if the workcrews benefit from creating their own schedules and how this affects the entire process. They however want to maintain a fair working environment, which is why the management wants to ensure that created schedules do not favor a specific workcrew.

From this NedTrain case, we can see that several problems are encountered by the NedTrain

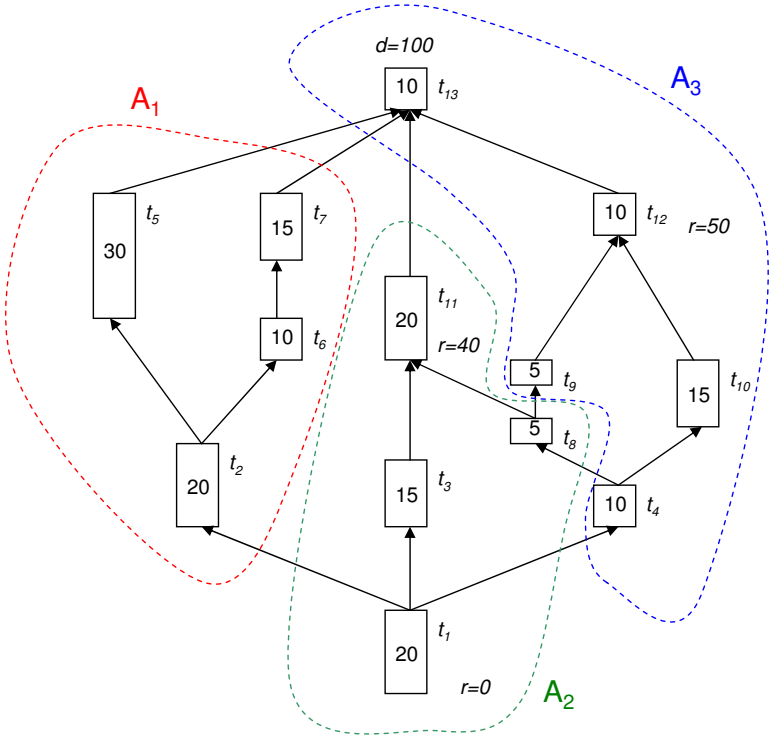

Figure 1.2: Workplan with workcrews A_i

management. The main problem is that with only the two presented workplans in Figure 1.1 and Figure 1.2, they find themselves at a loss on how to create suitable schedules for their employees. If they were able to create schedules based on these workplans, then they would like to know if they can leave the scheduling to the workcrews, in order to give them more freedom or adapt to any unforseen changes. They fear however that this freedom could cause one or more workcrews to be favored over the others, which is why they are interested in 'fair' schedules.

This leads to three main questions:

1. How can a schedule be created efficiently based on the two provided workplans?
2. Is it possible to allow the workcrews to create their own schedules and what is the effect of this on the entire process?
3. Is it possible to allow the workcrews to create their own schedules, such that their schedules are considered equally fair with respect to the freedom they have?

In order to answer the first question, we first have to know if it is possible to use the two workplans to at least create a schedule. If we know this to be true, then we can investigate on the reachable efficiency of creating such a schedule.

The second question is even more complex than it looks, since the workplans already show us that a lot of tasks are related to each other in some way. Such relations can make it hard to let each workcrew create its own independent schedule, without conflicting with the schedules created by the other two workcrews. We assume that all workcrews want to have as much freedom in their own schedules as possible, and to determine this effect on the entire process, we have to investigate how to create a schedule that contains as much freedomn as possible, i.e., a schedule that is maximally adaptable to unforseen influences. If we want to research the effect on the entire process even more, we also have to look into the relation between the total freedom in the created schedules by the workcrews based on the added restrictions, and the maximum freedom in a schedule created for all the workcrews together. Is there a difference in total freedom between these two approaches and if so, is there a bound on this difference?

The third question is based on the second question, with relation to the creation of schedules by the workcrews, but in this case we have an added constraint that these schedules have to contain an equal amount of freedom.

The result is that we can answer the three main questions by answering the more detailed questions given above. This leads to the following six research questions:

1. How can we use a workplan in order to create a schedule?
2. How can we efficiently create an arbitrary schedule?
3. Which restrictions do we have to add in order to allow each workcrew to create their own schedules without conflicting with the other schedules?
4. How can we determine a schedule that can maximally adapt to any unforseen influences?
5. Is there a trade-off between creating restrictions for independent schedules and creating maximally adaptable schedules?
6. How can we add independency restrictions to each workcrew such that their schedules are just as adaptable as the schedules of the other workcrews?

As we can see, the first main question is answered by answering research question 1 and 2. The second main question is anwered by research questions 3, 4 and 5, while the third main question is answered by research questions 3 and 4, but mainly by 6.

Let us review the six research questions one at a time and see if an answer can be found in the literature. The answer to the first research question can be found in [6] and [9]. They show that, based on the given processing times, release dates and due dates in the workplans, that it is possible to create a *Simple Temporal Problem* with a corresponding *Simple Temporal Network* to represent the scheduling problem. We will discuss this in Chapters 2 and 3. From this representation, an arbitrary schedule can be derived in $O(n^3)$ if there are n tasks present. This also somewhat answers the second research question, but we will prove in Chapter 3 that an arbitrary schedule for this type of problem can be created more efficiently, namely in $O(n^2)$ time, which we also showed in our paper [7].

The third research question is answered in [8] and [9], where it is shown that the *temporal decoupling problem* is equivalent to the problem stated in this research question. In Chapter 4 we will go into the temporal decoupling problem in more detail and show the approaches known in literature. We will also show that the currently known approaches are not as efficient as is possible in the case of this problem, since the known approached take $O(n^3)$ time, with n tasks present, but we will show a way to solve this problem in only $O(n^2)$ time, as we also did in our paper [7].

The fourth question is one that in our opinion is not adequately answered in the literature. Attempts have been made to define *metrics* that can measure the 'freedom' in a network, see [1], [2] and [8], but none of them give an exact and accurate amount of, how we define, freedom in a schedule. We will therefore introduce our own *flexibility* metric, that can measure exact and accurately the total freedom, or total *flexibility* as we will call it, in a schedule. We will show and prove in Chapter 5 that there exist multiple procedures to determine a schedule with maximal flexibility in polynomial time.

The fifth research question requires the answer to the third and fourth research questions and therefore also its related literature sources. However, we already stated that the answer to the fourth question cannot yet be found to our satisfaction in the literature, which is why we will evaluate this problem in Chapter 5. We will show there that, contrary to what intuition may say, the mentioned restrictions can be added to the constraints of the problem without limiting the total flexibility in a maximally flexible or adaptable schedule.

The last research question has to the best of our knowledge not yet been answered in the literature in relation to our current type of problem, which is why we will explain in detail three different problem formulations, all based on a different interpretation of 'fairness', that can be derived from it in Chapter 6. For all these problem formulations we will present solution methods in order to create the necessary restrictions to create fair schedules for the workcrews.

Chapter 2

Task Scheduling

While some people just use a rough outline of a schedule for the coming day, others plan their day almost to the minute. This first group probably prefers a more loose scheduling, which allows tasks to start or finish a bit later than anticipated. Both groups have something in common however, and that is that some tasks have to be executed in a specific order.

Apart from the specific ordering, tasks can also have a specific time at which they have to be completed. If we look at the NedTrain case presented in the introduction, we can see that the final task, labeled with the number 13 has to be completed within 100 time-units from the start of the first task, which is the *due date* of this task and also of the entire schedule. We can see in Figure 1.1 that this task has multiple *predecessors*, i.e., tasks that have to be completed before this task is allowed to start. Vice versa, we can denote task 13 to be the *successor* of the tasks preceeding this task, meaning that all preceeding tasks have to be completed before their successor is allowed to start. Apart from these precedence relations, we can also see that task 11 and 12 have a specified minimum starting time, which is the corresponding *release date* of these tasks.

What we see from this case is that tasks can have their own constraints on allowed starting times and completion times and can also influence each other. The NedTrain case is a specific type of a *task scheduling problem*, where each task t has to be assigned a specific starting time or time-interval.

Definition 1. *A* multi-agent task scheduling problem $\mathcal{S} = (T, \mathcal{A}, r, d, p, \prec)$ *consists of:*

- *a set of n tasks* $T = \{t_1, t_2, \ldots, t_n\}$
- *a set of k controlling* agents $\mathcal{A} = \{A_1, A_2, \ldots, A_k\}$*, where each agent* A_i *is assigned a disjoint set* $T_i \subseteq T$*, such that* $\bigcup_{i=1}^{k} T_i = T$ *and* $\forall i, j$ *with* $i \neq j$ *it holds that* $T_i \cap T_j = \emptyset$
- *a set of n* release dates r_i*, denoting the earliest allowed starting time for a task* $t_i \in T$
- *a set of n* due dates d_i*, denoting the latest allowed completion time for a task* $t_i \in T$
- *a set of n* processing times p_i*, denoting the required time between starting and completing a task* $t_i \in T$

- *a set of* precedence relations $\prec$ *between two tasks* $t_i, t_j \in T$*, indicating that if* $t_i \prec t_j$ *then* t_i *has to be completed before* t_j *is allowed to start, inducing a partial ordering on* T*.*

Remark. *Throughout this thesis we will not always specify the set of agents* $\mathcal{A}$ *or their corresponding disjoint tasksets* $\{T_i\}_{i=1}^{k}$*, but only specify them if they are of importance to the problem that is currently treated, since some task scheduling problems do not have multiple agents involved.*

In Definition 1 we formalized a number of new notions, which included the set of agents $\mathcal{A}$. An agent A_i can be a person or a group of people that is tasked with either the scheduling of the tasks T_i they control or also the execution of this set of tasks. It is important to note that a task t can only be controlled by exactly one agent, i.e., all sets T_i, T_j are disjoint, because if multiple agents would be in control of this task, it could result in a conflict of preferences, where one agent wants to schedule it early and another agent wants to schedule it at a later time.

Every task $t_i \in T$ receives a non-negative release date r_i, even if it is not specified, in which case $r_i = 0$, i.e., the start of the time-horizon. Apart from a release date, a due date d_i can be assigned to a task $t_i \in T$. In case a due date d_i is absent for a certain task $t_i \in T$, we take d_i equal to the due date of the schedule. It is important to note that a due date d_i is always considered to be strict, indicating that a task t_i has to be completed before its due date. Furthermore, a processing time p_i is specified for each task, and we allow an agent A_l to execute multiple tasks $t_i, t_j \in T_l$ in parallel.

The set of precedence relations $\prec$ contains the transitive precedence relations, indicating that if we have $t_i \prec t_j$ and also $t_j \prec t_k$, then also $t_i \prec t_k$. Note that these precedence relations are all strict, which means that if $t_i \prec t_j$, t_j is not allowed to start earlier than t_i is completed.

Example 1. *In the NedTrain case, maintenance for a rolling stock unit has to be performed by three different workcrews, where each crew has its own set of tasks to complete. Each workcrew is specialized in a certain type of maintenance, which gives them a specific set of tasks that can only be completed by them. If we revisit Figure 1.2, we see that each workcrew is represented by an agent* A_i*, resulting in* $T_1 = \{t_2, t_5, t_6, t_7\}$*,* $T_2 = \{t_1, t_3, t_8, t_{11}\}$ *and* $T_3 = \{t_4, t_9, t_{10}, t_{12}, t_{13}\}$*.*

The schedule starts with release date $r = 0$*, which basically means that the first task* t_1 *has* $r_1 = 0$*. All other tasks without a specified release date will get* $r_i = 0$*, but two other release dates are specified, namely* $r_{11} = 40$ *and* $r_{12} = 50$*. With respect to the due dates, we only have a final due date for the entire schedule, meaning that the last task* t_{13} *has a due date of* $d_{13} = 100$*. All other tasks without a specified due date will also receive* $d_i = 100$*.*

Definition 2. *We say that* t_i directly precedes t_j*, which is denoted by* $t_i \ll t_j$*, if* $t_i \prec t_j$*, but there exists no* $t_k \in T$ *such that* $t_i \prec t_k$ *and* $t_k \prec t_j$*. We denote the set of* predecessors *of* t_i *by* $pre(t_i) = \{t_j \mid t_j \prec t_i\}$*, while the set of* direct predecessors *of* t_i *is denoted by* $pre_{\ll}(t_i) = \{t_j \mid t_j \ll t_i\}$*. The set of* successors *of* $t_i \in T$ *is denoted by* $suc(t_i) = \{t_j \mid t_i \prec t_j\}$ *and the set of* direct successors *is denoted by the set* $suc_{\ll}(t_i) = \{t_j \mid t_i \ll t_j\}$*.*

As we can see, the $\ll$-relation is stronger than the $\prec$-relation, since if $t_i \ll t_j$, then certainly $t_i \prec t_j$, but if $t_i \prec t_j$, then $t_i \ll t_j$ does not have to hold. From the above definition we can see

that for all $t_i \in T$ we have $pre_{\ll}(t_i) \subseteq pre(t_i)$ and $suc_{\ll}(t_i) \subseteq suc(t_i)$, due to the $\ll$-relation being stronger than the $\prec$-relation.

Example 2. *Apart from the assignment of the tasks over the three workcrews, there is also a relation between a number of tasks. The combining of the different modules in the rolling stock unit can only be done once all the other maintenance has been completed, just as painting can only be done once certain damage has been repaired. These precedence relations are taken into account in the workplan in Figure 1.2. We can see that for example* $t_2 \prec t_5$ *and* $t_2 \prec t_6$*, i.e.,* t_2 *has to be completed before* t_5 *and* t_6 *are allowed to start. There can also be inter-agent precedence relations, we see in the same figure for example that* $t_1 \prec t_2$*, which means that agent* A_1 *has to wait with task* t_2 *for* A_2 *to finish* t_1*.*

Definition 3. *Given a multi-agent task scheduling problem* $\mathcal{S} = (T, \mathcal{A}, r, d, p, \prec)$*, we define a* solution *or* schedule σ_i *for agent* A_i *in* $\mathcal{S}$ *to be an assignment* $\sigma_i : T_i \to \mathbb{R}$ *which assigns to every* $t_j \in T_i$ *a starting time, such that all constraints are satisfied, that is* $\forall t_j \in T_i$ *we have* $\sigma_i(t_j) \geq r_j$*,* $\sigma_i(t_j) + p_j \leq d_j$ *and* $\forall t_j, t_k \in T_i$ *it holds that if* $t_j \prec t_k$ *then* $\sigma_i(t_j) + p_j \leq \sigma_i(t_k)$*.*

Furthermore, we define a joint schedule $\sigma = \bigcup_{i=1}^{k} \sigma_i$ *for* $\mathcal{S}$ *to be an assignment* $\sigma : T \to \mathbb{R}$ *if every* σ_i *is a solution for* A_i *and for all* $t_i, t_j \in T$ *with* $t_i \prec t_j$ *we have* $\sigma(t_i) + p_i \leq \sigma(t_j)$*.*

Remark. *Note that at this point we will only allow single starting times to be assigned to tasks. Later in this thesis we will also consider* open schedules*, which allow starting time-intervals to be assigned to tasks.*

If agents are of lower importance or there is only one agent present in the task scheduling problem, we will refer to a joint solution σ *as a solution without specifying if it is a combination of multiple solutions* σ_i*.*

The precedence relation constraints do not have to be stated for all $t_i \prec t_j$, because we can reduce this number of constraints by only stating them for all $t_i \ll t_j$. The transitive property of this $\ll$-relation translates it to the $\prec$-relation, which can significantly reduce the number of constraints.

Example 3. *NedTrain is interested in a schedule for which all workcrews can perform their tasks within the time-limit. Since we have seen that the precedence constraints have to be satisfied, we will state some of them, again based on Figure 1.2. We see that* $t_1 \prec t_2$*, which means that we have* $\sigma(t_1) + p_1 \leq \sigma(t_2)$*. Since we also have* $t_2 \prec t_5$*, we can also state* $\sigma(t_2) + p_2 \leq \sigma(t_5)$*. It is also possible to now state the relation* $t_1 \prec t_5$*, but due to the transitive relation, this constraint can be derived from the previous two precedence constraints and will always be satisfied if these other two constraints are satisfied.*

With respect to the release dates, this would result in $\sigma(t_i) \geq 0$ *for all* $i \neq 11$ *and* $i \neq 12$*, since we have* $\sigma(t_{11}) \geq 40$ *and* $\sigma(t_{12}) \geq 50$*. The due dates are for all tasks the same, since there is only a final due date specified for the entire schedule:* $d_i = 100$*, i.e.* $\sigma(t_i) + p_i \leq 100$ *for all* $t_i \in T$*.*

We have seen so far that from Figure 1.2 of the NedTrain case, we can extract a lot of useful information, but the graphical representation still lacks a convenient way to represent release dates, due dates and time-relations between tasks. Apart from the graphical representation, it can also be convenient if we can represent the variables and their solutions in a

more simple manner. In the next chapter we will use the definitions of our task scheduling problem to search for mechanisms that allow us to (efficiently) create solutions.

Chapter 3

Simple Temporal Problem

With the definition of a task scheduling problem given in the previous chapter, we can now expand on this and enhance our representation of the problem. As was shown in [6] and [9], we can represent our task scheduling problem as a Simple Temporal Problem (STP), which gives us a tool to find suitable schedules. For a task scheduling problem, we represented a schedule σ by assigning a starting time to every task $t \in T$, which we represented by $\sigma(t)$. It is also possible to represent each task t by a corresponding *time-point variable* which is also indicated by t. A schedule would then be an assignment of starting times to these time-point variables which satisfies the constraints that can now be represented even more simple.

Definition 4. *A Simple Temporal Problem S is a pair (T,C), where $T = \{t_0, t_1, \ldots, t_n\}$ is a set of time-point variables and C is a finite set of binary linear constraints on T, each constraint having the form $t_j - t_i \leq \delta$, for some constant δ. The time-point t_0 represents an arbitrary, fixed, reference point on the timeline. A* solution σ *to an STP S is an assignment σ of values (time-points) to the time-point variables in T, such that all constraints in C are satisfied.*

We can see that one of the differences with the task scheduling problem is that we introduce a time-point variable t_0, which can also be represented by the symbol z and is usually set equal to 0. Since each constraint is now of the form $t_j - t_i \leq \delta$, we have to adapt some of the task scheduling constraints to fit this new framework. With respect to release dates and due dates, the constraints for a task t_i transform to: $z - t_i \leq -r_i$ and $t_i - z \leq d_i - p_i$. Although this may at first sight seem more complicated, this fixed type of constraints gives us a solid structure which allows for an easier way to solve the problem. With respect to the precedence relations, we can add for every set of tasks $t_i, t_j \in T$ with $t_i \prec t_j$ a constraint $t_i - t_j \leq -p_i$, but that would give us a large number of constraints. It would be less complicated to include only the constraints where $t_i \ll t_j$. In all other situations the constraints can be considered unbounded, i.e., $t_i - t_j \leq \infty$, since there is either no relation between the two tasks or a combination of other constraints already imply a constraint between these tasks due to transitivity.

With the new symbolical representation, we can also introduce a new graphical representation, which is in case of an STP a directed labeled graph. More formally, we represent an STP $S = (T,C)$ by a directed labeled graph $G_s = (T,E,l)$, where the labels l are attached

to the edges (t_i,t_j) in such a way that $l(t_i,t_j) = [a,b]$, where $a \leq t_j - t_i \leq b$ are constraints in C, i.e., $t_j - t_i \leq b$ and $t_i - t_j \leq -a$ are both in C. This graph is called a *Simple Temporal Network* (STN), which in our case will always be a transitive graph, since we will only represent the relevant constraints by excluding directed edges (t_i,t_j) if $t_i \ll t_j$ for all $t_i,t_j \in T$ with $i \neq j$ and $t_i,t_j \neq t_0$.

Example 4. *We have seen in Figure 1.2 that there exists a workplan with corresponding crews that can represent our task scheduling problem. We will however create an STN representing the problem in our newly described representation, with the result in Figure 3.1. As we can see in Figure 3.1, we have replaced the tasks by circles with their corresponding*

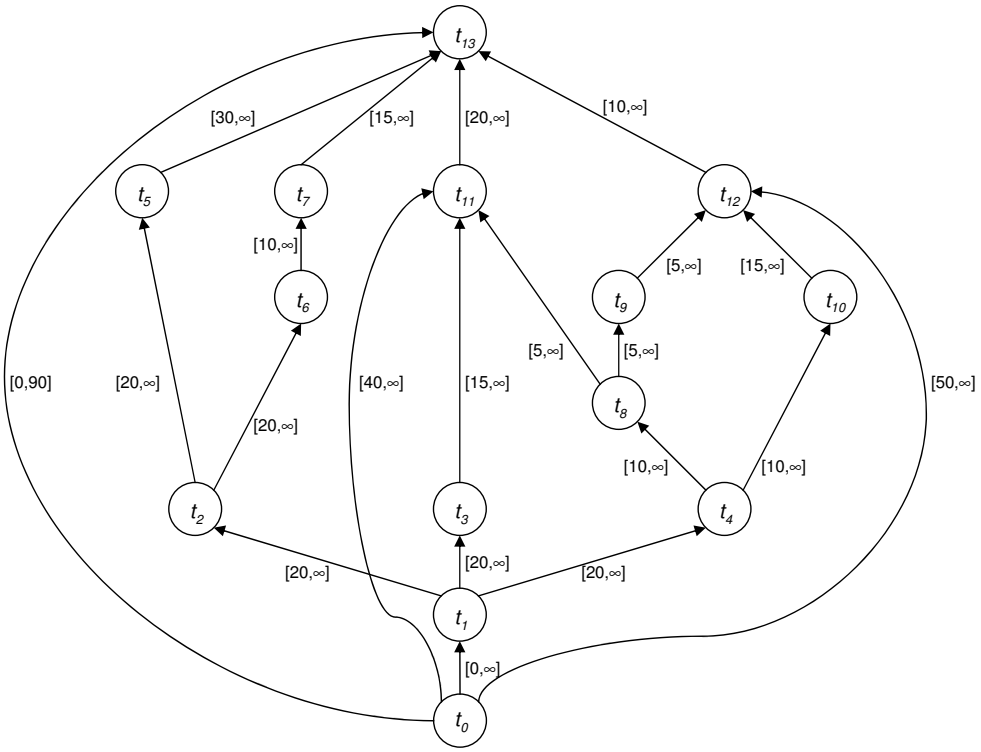

Figure 3.1: STN representing the NedTrain case

task number. The edges have been labeled with an interval $[a,b]$, which is derived from the constraints $a \leq t_j - t_i \leq b$ if $t_i \ll t_j$. We however also have to include the two specific noted release dates of tasks t_{11} and t_{12}, which is represented respectively by the constraints $t_0 - t_{11} \leq -40$ and $t_0 - t_{12} \leq -50$, where 40 and 50 form the a in both intervals. The upper bound b is ∞, since there is no due date specified. Both constraints are added as an arc (t_0,t_{11}) and (t_0,t_{12}). The due date of the complete schedule, which is equal to 100, is in fact only directly applicable to t_{13}, which results in the constraint $t_{13} - t_0 \leq d_{13} - p_{13}$, i.e.,

$t_{13} - t_0 \leq 90$. *Just like the release dates, this adds a new edge from* t_0 *to* t_{13} *representing in this case the interval* $[0, 90]$.

This way of representing a task scheduling problem as an STP or STN has also as an advantage that we know that there are methods to achieve an abritrary schedule. If we consider an arbitrary STP, i.e., an STP that is not necessarily a task scheduling instance, it is possible to find an abritrary schedule in $O(n^3)$ time , see [5] and [6], if n is the number of time-point variables present.

3.1 Complexity of calculating solutions for STPs

So far we have only shown a task scheduling instance that was converted to an STP and STN, but the class of STPs is much larger than just task scheduling instances. We will discuss a general method to acquire an arbitrary schedule in $O(n^3)$ time, as shown by [5] and [6], before we again focus on task scheduling instances due to the special properties these instances hold which allows us to compute an arbitrary schedule in only $O(m+n)$ time.

Acquiring an arbitrary schedule in $O(n^3)$ is based on the following idea: Suppose we have an arbitrary STP $S = (T, C)$ that is represented by an STN. Based on the constraint set C, we can construct a distance matrix D of size $(n+1) \times (n+1)$, in which every entry $D(i, j)$ holds the 'temporal distance' between t_i and t_j, i.e., the value in the constraint $t_j - t_i \leq D(i, j)$. Based on these constraints, we can find the (implicit) constraints in the first row and column of D, which contains all constraints involving t_0.

The first row in D, which contains the $n+1$ entries $D(t_0, t_i)$, where $i = 0, 1, \ldots, n$, represent the *earliest starting times* of the tasks t_1 up to t_n. The earliest starting time of a task t_i, denoted by $est(t_i)$, represents the first point in time at which this task can be scheduled, such that it does not violate any constraints. The entries in the first row are implied by the constraints in C and can also be found in the corresponding STN through the use of a longest path calculation from t_0 to t_i.

The first column in D, also containing $n+1$ entries $D(t_i, t_0)$, with $i = 0, 1, \ldots, n$, represents in this case the opposite of the $est()$ values, namely the *latest starting times*, which we denote by $lst()$, and represents the latest point in time at which a task is allowed to be scheduled, such that it does not violate any constraints and again allows for the creation of a feasible schedule. Like in the $est()$ case, the entries in this row are also implied by other constraints in C.

An arbitrary schedule can then be created in $O(n^3)$ time by fixing a certain $t_i \in T$ at a certain time $\sigma(t_i)$, such that $est(t_i) \leq \sigma(t_i) \leq lst(t_i)$, then recalculate D based on the chosen $\sigma(t_i)$. Then the next task $t_j \in T$ can be fixed, again with $est(t_j) \leq \sigma(t_j) \leq lst(t_j)$ based on the updated matrix D. Since an update of D requires $O(n^2)$ time and there are n iterations, an arbitrary schedule is found in $O(n^3)$.

This however gives rise to a question: Can we do better in case of a task scheduling instance?

3.2 Task scheduling instances in STP$_\prec$

Before we answer the question from the previous section, we first have to state some important properties of a task scheduling STP. These properties define the special subclass of task scheduling STPs when compared to the class of STPs.

Definition 5. *The class of task scheduling STPs is denoted by STP$_\prec$ and contains all task scheduling problems represented as an STP. A task scheduling instance $S = (T,C)$, with $S \in$ STP$_\prec$ has as properties that it contains a partial ordering of the tasks in T, rendering its corresponding STN acyclic. Furthermore it does not contain constraints of the form $t_j - t_i \leq b$ with $b \neq \infty$ if $i \neq 0$ and $t_i - t_j \leq -a$ with $a \geq p_i$ if $i \neq 0$. Therefore, STP$_\prec \subset$ STP.*

The partial ordering that is present in the taskset T of an $S \in$ STP$_\prec$ is induced by the precedence relations that are present in C. A schedule σ can only be constructed if the constraints match certain criteria, which include that the precedence constraints do not give rise to a cycle in the corresponding STN, which works both ways, resulting also in the statement that if there is a partial ordering present in T, then the corresponding STN is acyclic.

Representing a task scheduling problem $\mathcal{S}$ as a task scheduling STP $S = (T,C)$ can be done by comparing Definition 1 with Definition 5 and as the reader can also verify, we see that they both contain a set T containing n tasks, where T can be divided into k disjoint[1] sets T_i, one for each agent A_i such that $\bigcup_{i=1}^{k} T_i = T$. The remaining properties of tasks can be represented in the constraint set C, where we add $z - t_i \leq -r_i$ to satisfy the release date r_i of task t_i, and $t_i - z \leq d_i - p_i$ to satisfy the due date d_i of t_i by also using its processing time p_i. The final precedence relation property between two tasks $t_i, t_j \in T$ can be satisfied by adding the constraint $t_i - t_j \leq -p_i$ to C if $t_i \prec t_j$.

In an arbitrary STP $S = (T,C)$ however, precedence relations do not have to be present, which means that it is possible that $t_j - t_i < p_i$ for any $t_i, t_j \in T$, indicating that task t_j starts before t_i has finished.

Definition 6. *A solution σ for an STP $S = (T,C)$ is called* feasible *if it satisfies all constraints in C. An STP $S = (T,C)$ is called* consistent *if at least one feasible solution σ exists.*

Remark. *From here we will assume that all STPs we refer to are consistent and if we refer to a schedule σ we will assume that it is feasible, unless stated otherwise.*

The precedence relation property of task scheduling STPs can be exploited in multiple ways in order to reduce the complexity of calculations.

Due to the resemblance in notation and properties, we can apply and use the properties of Definition 2 also in the case of task scheduling STPs. Using this notation and the properties of task scheduling STPs, we can create a less complex way to compute the first row and column of the temporal distance matrix D, by no longer using the remaining n^2 entries in the matrix. Due to the precedence relations, a task $t_j \in T$ can only start once all of its direct predecessors $t_i \ll t_j$ have been completed.

[1] For simplicity reasons, we neglect here the presence of t_0 in the sets T_i in an STP, since t_0 does not represent an actual task.

Observation 1. *In a task scheduling STP $S = (T,C)$, if for a $t_i \in T$ a feasible solution is fixed at $\sigma(t_i) = s$, then the following inequalities hold:*

1. *for all $t_j \in suc(t_i)$, $\sigma(t_j) \geq s + p_j$,*
2. *for all $t_j \in pre(t_i)$, $\sigma(t_j) \leq s - p_j$.*

This observation follows from the fact that in a feasible solution σ, every assignment $\sigma(t_i)$ has to satisfy all related constraints, which include the precedence relations, release dates and due dates, i.e., $\sigma(t_j) \geq \sigma(t_i) + p_j$ if $t_i \ll t_j$, $\sigma(t_j) + p_j \leq \sigma(t_i)$ if $t_j \ll t_i$, $\sigma(t_i) \geq r_i$ and $\sigma(t_i) + p_i \leq d_i$. From this we can see that the earliest starting time of a task t_i, $est(t_i)$, is only depending on its release date and its direct predecessors in a task scheduling STP. This gives us the following relation:

$$est(t_i) = \max(\{est(t_j) + p_j \mid t_j \in pre_{\ll}(t_i)\} \cup r_i) \tag{3.1}$$

Since the release dates are known for all $t_i \in T$, the only requirement for computing $est(t_i)$ is knowing every $est(t_j)$ for all $t_j \in pre_{\ll}(t_i)$. Based on the partial ordering of the set T, we can compute every $est()$ value in that order, which requires us to traverse every arc and node in the corresponding STN exactly once, resulting in a $O(m+n)$ computation complexity if m represents the number of arcs and n the number of tasks/nodes.

Example 5. *Since the NedTrain case is a classical example of a task scheduling problem, which we have already represented as an STP and STN in Figure 3.1, we can use our given $est()$ calculation to determine the earliest starting time for all tasks. Since t_0 can be considered to be a dummy task without a processing time, the earliest starting time of t_1 is equal to its release date, i.e., $est(t_1) = 0$. From here, we can move on to t_2, which only depends on $est(t_1)$ (in the abscence of an explicit release date), resulting in $est(t_2) = est(t_1) + p_1 = 0 + 20 = 20$. We can continue these calculations and see that $est(t_3) = est(t_4) = 20$, $est(t_5) = est(t_6) = 40$, $est(t_7) = 50$, $est(t_8) = 30$, $est(t_9) = 35$, $est(t_{10}) = 30$, $est(t_{11}) = 40$ due to its explicit release date being greater than the effect of its direct predecessors, $est(t_{12}) = 50$ where again the release date is responsible, and $est(t_{13}) = 70$.*

From this example we can derive the answer to the first research question: *How can we use a workplan in order to create a schedule?* By using the workplan to construct an STP $S = (T,C)$ such that $S \in \text{STP}_\prec$ and matches Definitions 4 and 5 and subsequently creating the corresponding STN, we have converted a task scheduling problem to a task scheduling Simple Temporal Problem with corresponding STN. By then using the calculated $est()$ values as fixed starting times for each task $t_i \in T$, we get a schedule σ where for each task $t_i \in T$ we have $\sigma(t_i) = est(t_i)$.

We can also use the second inequality of Observation 1 to likewise create a more efficient calculation method for the latest starting time of a task $t_i \in T$. This second inequality shows us that for a latest starting time, we only have to consider the precedence constraints and a corresponding due date, if present. This results in the following relation:

$$lst(t_i) = \min(\{lst(t_j) - p_i \mid t_j \in suc_{\ll}(t_i)\} \cup d_i - p_i) \tag{3.2}$$

Again, the partial ordering can be used to compute all $lst()$ values effectively, although in this case we will apply them in a reverse order, since the latest starting times of all direct successors of a task $t_i \in T$ have to be known before we can start to determine $lst(t_i)$. In the corresponding STN this can be seen as a 'top-to-bottom' calculation of the $lst()$ values, in which we again have to traverse each arc and node exactly once. This results, like in the $est()$-case, in a $O(m+n)$ computation complexity.

Example 6. *As in example 5 we can also determine the latest starting times for all tasks $t_1, t_2, \ldots, t_{13}$. We can begin with t_{13}, since this task has no successors and is therefore only bounded by its due date: $lst(t_{13}) = 100 - 10 = 90$. We can work our way down the STN by calculating $lst(t_{12}) = 90 - 10 = 80$ and $lst(t_{11}) = 90 - 20 = 70$. The remaining results are shown in the following table, as can be checked by the reader:*

task	*est*	*lst*
1	*0*	*20*
2	*20*	*40*
3	*20*	*55*
4	*20*	*55*
5	*40*	*60*
6	*40*	*65*
7	*50*	*75*
8	*30*	*70*
9	*35*	*75*
10	*30*	*65*
11	*40*	*70*
12	*50*	*80*
13	*70*	*90*

As we can see from this table, for every task the difference between the earliest and latest starting time ranges from 20 to 40 time-units.

In Example 6 we have seen that in that case every task has a certain time-interval created by the difference between the earliest and latest starting time of that task. Since $est(t_i)$ represents the earliest time t_i is allowed to start and $lst(t_i)$ is the latest time t_i is allowed to start in order to not break any constraints, we can derive the following:

Proposition 1. *Let the STP $S = (T, C)$ be a task scheduling instance. Then the following holds:*

1. *for every feasible schedule σ of S and for every $t_i \in T$: $est(t_i) \leq \sigma(t_i) \leq lst(t_i)$,*
2. *an arbitrary schedule σ of S can be computed in $O(m+n)$ time.*

Proof. Suppose that we have a schedule σ where for some $t_i \in T$: $\sigma(t_i) < est(t_i)$. Then by Equation 3.1 either $\sigma(t_i) < r_i$, which violates its release date, or $\sigma(t_i) < est(t_j) + p_j$ for some $t_j \in pre_{\ll}(t_i)$ and thereby violating the precedence constraints. By Equation 3.2, the

same holds if for some $t_i \in T$: $\sigma(t_i) > lst(t_i)$, since then either $\sigma(t_i) > d_i - p_i$, which violates its due date, or $\sigma(t_i) > lst(t_j) - p_i$ which violates the precedence constraint. From this we conclude that the first statement must hold in every feasible schedule.

We have already found that the $est()$ and $lst()$ calculations for all tasks $t_i \in T$ can be done in $O(m+n)$ time. Based on the equations 3.1 and 3.2, we can create an arbitrary schedule by a 'bottom-to-top' procedure. The idea behind the 'bottom-to-top' procedures is to first calculate for all tasks the $lst()$ value. We then start in t_0 which is fixed at 0 and, by following the partial ordering in T, calculate for the next t_j the value $est(t_j)$ based on the previously assigned $\sigma(t_i)$ values to t_i, where $t_i \in pre_{\ll}(t_j)$. We fix t_j at some $\sigma(t_j)$ where $est(t_j) \leq \sigma(t_j) \leq lst(t_j)$ and continue to the next t_k, where $k = j+1$, in the partial ordering of T. Since for every fixed $\sigma(t_j)$ it holds that $est(t_j) \leq \sigma(t_j) \leq lst(t_j)$ and $\sigma(t_j) \geq \sigma(t_i) + p_i$ with $t_i \in pre_{\ll}(t_j)$, this resulting schedule σ is feasible. Furthermore, every arc and node is traversed exactly two times (once for the $lst()$ calculations and once for the $est()$ calculations), this leads to an arbitrary schedule in $O(m+n)$ time. □

Remark. *In a complete graph the relation between the number of arcs m and the nodes n is that $m < n^2$. Since the STN of a task scheduling STP contains at most one edge between two task t_i and t_j, the same relation $m < n^2$ will also hold in a task scheduling STN. This shows us that an arbitrary schedule can be computed in $O(m+n) = O(n^2)$ time. If we compare that to the $O(n^3)$ time that is required for an arbitrary STP, we see that by considering a special subclass STP$_\prec$ of STPs we can significantly improve the computation time of arbitrary schedules. Furthermore, it shows us that we can rebuild the distance matrix D also in $O(n^2)$ time by using our est() and lst() computations, instead of the $O(n^3)$ that is required for arbitrary STPs, which we will prove with help of the following Proposition.*

Proposition 2. *Let $S = (T,C)$ be a task scheduling instance. Then the temporal distance matrix D can be computed using a partial ordering on T in $O(n^2)$ time.*

Proof. As we already stated in the previous remark, we can rebuild the matrix D by using our previously defined $est()$ and $lst()$ calculations from Equations 3.1 and 3.2. If $t_i \ll t_j$, then the entries $D(t_i,t_j)$ and $D(t_j,t_i)$ are found directly by calculating respectively all $est()$ and $lst()$ values. However, all values $D(t_i,t_j)$ where $t_i \prec t_j$ can also be derived from the $est()$ calculations, since if $t_i \prec t_j$, but not $t_i \ll t_j$, then $D(t_i,t_j) = \max_{t_k \in pre_{\ll}(t_j)}\{D(t_i,t_k) + D(t_k,t_j)\}$. The value $D(t_j,t_i)$ can be derived once the $lst()$ calculations are in progress in a similar way as the $D(t_i,t_j)$ calculations. Since all nodes and arcs only have to be examined exactly twice, this can be done in $O(n^2)$ time. □

We have seen in this chapter the answer to the second research question: *How can we efficiently create an arbitrary schedule?* In Proposition 1 we have shown how to create a schedule for a task scheduling STP $S = (T,C)$ in $O(m+n)$ time, based on the $est()$ and $lst()$ computations.

However, so far we have not considered the workcrews that are present in our NedTrain case and apart from that we have only assigned an arbitrary single time-point to each task $t_i \in T$. The workcrews may prefer some autonomy over the creation of their schedules, but with a single person assigning time-points to tasks, they have none. In the next chapter we will show how to add restrictions in order to give the workcrews more autonomy.

Chapter 4

Temporal Decoupling

Autonomy is something that many people prefer to have in most situations, since it gives them a certain control over the current or coming situation. Problems may arise however if multiple persons are involved in a single situation and all demand a certain level of autonomy. In a task scheduling instance, we have seen that there are precedence relations between certain tasks, specifying a relation between both the tasks and the controlling agents. If we give all agents full control over their tasks in such a situation, scheduling conflicts are almost bound to happen.

Example 7. *Let us review Example 6: we see that in the table of est() and lst() values, each task has been assigned one of these values, based on the constraints. In Figure 1.2 we have seen that agent A_1 controls t_2, t_5, t_6 and t_7, A_2 controls t_1, t_3, t_8 and t_{11} and A_3 is in control of t_4, t_9, t_{10}, t_{12} and t_{13}. If A_1 chooses to schedule his task t_2 at $\sigma(t_2) = 25$ and A_2 schedules t_1 at $\sigma(t_1) = 10$, we can see that both chosen times fit the constraints such that $0 \leq \sigma(t_1) \leq 20$ and $20 \leq \sigma(t_2) \leq 40$, but there is one constraint that is not satisfied, namely the precedence constraint $t_1 + 20 \leq t_2$.*

The problem shown in Example 7 is a serious one, and shows us that we cannot give full control to the agents over their tasks without placing some limitations on their possibilities. A possibile way to overcome this situation is to somehow limit the available time-points for A_1 with respect to t_2 and also for A_2 with respect to t_1.

Example 8. *With the problem introduced in Example 7 we will add two new constraints to C, in order to prevent a violation of the precedence relation between t_1 and t_2. By adding $t_1 \leq 10$ and $t_2 \geq 30$, we can see that the precedence constraint $t_1 + 20 \leq t_2$ will always be satisfied for any t_1 or t_2 that also satisfy the two new constraints.*

We solved the problem presented in Example 7 by adding a release date for t_2 and a due date for t_1 in Example 8. By doing this, we have removed the direct relation between t_1 and t_2, and have given A_1 and A_2 two sets of tasks that are now independent of each other. We refer to this as a *temporal decoupling*, which was among others described in [8] and [11].

Definition 7. *Let $S = (T, C)$ be a task scheduling STP where T is a set $\{t_0, t_1, ..., t_n\}$ of time-point variables and C is a finite set of binary constraints on those variables. Suppose that*

$T = \{T_i\}_{i=1}^{k}$ is partitioned in k subsets T_i. Then the temporal decoupling problem *is to find k subproblems $S_i = (T_i \cup \{z\}, C_i')$, where C_i' contains constraints relevant to $T_i \cup \{z\}$, such that, whenever $\sigma_1, \ldots, \sigma_k$ are solutions of the individual STPs $S_1, \ldots, S_k$, respectively, their merge $\sigma = \bigcup_{i=1}^{k} \sigma_i$ is also a solution of the original STP S.*

What is important to note in this definition is that in each subproblem S_i controlled by agent A_i, the corresponding constraint set C_i contains only constraints that contain tasks from T_i, i.e., there will be no constraint present in C_i such that $t_j - t_k \leq \delta$ with either $t_j, t_k \in T_l$ with $i \neq l$. We have seen in Example 8 how to begin to derive two independent subproblems S_1 and S_2 from a larger problem S, but the question that rises then is how many constraints do we have to add in order to create independent subproblems?

Example 9. *It is for example possible to also add two constraints between the tasks t_2 and t_5 in the NedTrain case, but both tasks are already controlled by agent A_1 and therefore have to satisfy the precedence constraint $t_5 - t_2 \leq -p_2$. Adding two new constraints for this relation would only over-constrain the problem without creating some independency for A_1 from the other agents A_2 and A_3.*

In order to create a partitioning of the original problem S into independent subproblems $S_1, \ldots, S_k$ we only have to pay attention to the so-called *inter-agent constraints*, i.e., the constraints concerning $t_i, t_j \in T$, such that $t_i \ll t_j$ with $t_i \in T_k$ and $t_j \in T_l$, where $k \neq l$. Based on this observation, we will state a decoupling procedure:

Decoupling Procedure. *Given a task scheduling STP $S = (T, C)$, in order to create a temporal decoupling, we have to:*

1. *choose a value δ_{ij} for every pair of tasks t_i, t_j belonging to different agents, with $t_i \ll t_j$, and $lst(t_i) + p_i > est(t_j)$, such that $est(t_i) + p_i \leq \delta_{ij} \leq lst(t_i) + p_i$ and $est(t_j) \leq \delta_{ij} \leq lst(t_j)$.*
2. *add a constraint $t_i - z \leq \delta_{ij} - p_i$ to the set of constraints in $S_l = (T_l, C_l)$ of agent A_l if $t_i \in T_l$ and add the constraint $z - t_j \leq -\delta_{ij}$ to the set of constraints in $S_m = (T_m, C_m)$ of agent A_m if $t_j \in T_m$. Here, C_l and C_m denote the restriction of the total set of constraints C to the set of constraints over $T_l \cup \{z\}$ and $T_m \cup \{z\}$, respectively.*

The set $\{S_l\}_{l=1}^{k}$, with $S_l = (T_l, C_l)$ then forms a decoupling of S.

Proposition 3. *The stated Decoupling Procedure gives a feasible temporal decoupling of a task scheduling STP $S = (T, C)$.*

Proof. As Definition 7 states, the k subproblems S_l form a temporal decoupling of S if $\sigma_1, \sigma_2, \ldots, \sigma_k$ are feasible solutions of these k subproblems, respectively, and their merge $\sigma = \bigcup_{l=1}^{k} \sigma_l$ is a feasible solution of the STP S.

The merge $\sigma = \bigcup_{l=1}^{k} \sigma_l$ can only be an infeasible schedule for S if one or more constraints in C are violated. Since all constraint sets C_l contain the relevant release and due date constraints for T_l, all schedules σ_l will respect the release and due dates, i.e., a merge of these schedules σ will also respect all release and due dates.

The remaining constraints are then the precedence constraints. Since they will be satisfied for all $t_i, t_j \in T_l$ in a feasible solution σ_l for all k subproblems S_l, the only remaining precedence constraints are the inter-agent constraints. For any pair $t_i, t_j \in T$ with $t_i \ll t_j$ such that $t_i \in T_l$ and $t_j \in T_m$, with $l \neq m$ and $lst(t_i) + p_i > est(t_j)$, the Decoupling Procedure chooses a value δ_{ij} such that $est(t_i) + p_i \leq \delta_{ij} \leq lst(t_i) + p_i$ and $est(t_j) \leq \delta_{ij} \leq lst(t_j)$ and adds a constraint $t_i - z \leq \delta_{ij} - p_i$ to C_l and adds $z - t_j \leq -\delta_{ij}$ to C_m. The result is then that for any $\sigma_l(t_i)$ and $\sigma_m(t_j)$ it holds that $\sigma_l(t_i) + p_i \leq \delta_{ij} \leq \sigma_m(t_j)$, which shows that in the merge σ this precedence relation is also satisfied.

From this we can conclude that for any feasible solutions $\sigma_1, \sigma_2, \ldots, \sigma_k$ of the k subproblems S_l, their merge $\sigma = \bigcup_{l=1}^{k} \sigma_l$ will always be a feasible solution of the STP $S = (T, C)$, since all constraints in C are satisfied. □

Example 10. *In the NedTrain case we can see (for example in Figure 1.2) that the inter-agent constraints of A_1 are between the following tasks: t_1 and t_2, t_5 and t_{13}, t_7 and t_{13}. As we can see, both t_5 and t_7 are direct predecessors of t_{13}, which means we only have to add one constraint $t_{13} \geq \delta$ to C_2 of A_2 in order to decouple S_1. Adding two different constraints would only leave one constraint redundant and possibly over-constraining the problem. There are a lot of possibilities to decouple S_1, one for example is: If we add*[1] *$t_1 \leq 10$ to C_2 and $t_2 \geq 30$ to C_1, the inter-agent constraint between t_1 and t_2 can be removed. By furthermore adding $t_5 \leq 50$ and $t_7 \leq 65$ to C_1 and $t_{13} \geq 80$ to C_2 we have decoupled $S_1 = (T_1, C_1)$ from the other agents.*

Based on what we have seen before this example, we can create a temporal decoupling in an efficient way. In an arbitrary STP $S = (T, C)$ this would require the use of the complete distance matrix D, which required $O(n^3)$ computation time (see [8]). The Decoupling Procedure showed us that for a task scheduling STP this can be done more efficiently.

Proposition 4. *Let $S = (T, C)$ be a task scheduling STP where $|T| = n$ and m is the number of edges in the corresponding STN. Assume that each agent A_i has a disjoint subset of tasks T_i. Then an arbitrary decoupling of S can be achieved in $O(m+n)$ time.*

Proof. In Proposition 3 we have shown that using the Decoupling Procedure we can construct a feasible[2] decoupling for a task scheduling STP $S = (T, C)$ that satisfies Definition 7. Since this requires all the $est()$ and $lst()$ values to be known, these have to be calculated first.

To construct a decoupling, we have to follow the partial ordering in T and update for every task we encounter its $est()$ value. By every task $t_i \in T$ we encounter by following the partial ordering, we check if there exists a precedence relation $t_i \ll t_j$ that matches 1) in the Decoupling Procedure. If it exists, a value δ_{ij} is chosen that satisfies 1) and constraints are added as described in 2). After this, the next task in the partial ordering is chosen and the process repeats itself until there are no more tasks left in T.

[1] Note that for the sake of readability we have simplified the constraints here, but they can be easily converted to constraints that match the standard STP formulation according to Definition 4.

[2] This means that the resulting problem is still consistent.

Since we start with all $est()$ and $lst()$ calculations and later we update in each step the $est()$ value, this takes a total of $O(m+n)$ time. Since every edge is checked exactly once in the decoupling steps, the total complexity of the algorithm remains $O(m+n)$. □

The Decoupling Procedure, Proposition 4 and its corresponding proof give the answer to our third research question, *Which restrictions do we have to add in order to allow each workcrew to create their own schedules without conflicting with the other schedules?*, since it shows how to create a decoupling for any task scheduling STP. Every workcrew is represented by an agent and the added decoupling constraints for agent A_1 for the NedTrain case are shown in Example 10.

Proposition 4 and its related proof show us that we can establish an arbitrary decoupling of a task scheduling STP $S = (T,C)$ in only $O(m+n) = O(n^2)$ time, which is an improvement over the $O(n^3)$ time required to create an arbitrary decoupling based on the temporal distance matrix D when working with an arbitrary STP S.

However, an arbitrary decoupling may not be in the best interest of the agents involved, since the chosen values δ_{ij} are chosen arbitrarily. Agents can have many preferences, for example having a minimum makespan, meaning that the latest completion time of his tasks is as small as possible, which can be achieved by choosing every δ_{ij} as small as possible. The result however would be that every task is fixed at a single time-point, leaving no room for any on the fly changes in the schedule at all. We can ask ourselves if that would result in a workable situation, if every task has to start to the minute at a pre-determined point in time. In the next chapter we will show how to create schedules that enable the agents to have more freedom in choosing their starting times, by assigning not a single time-point to a task, but by assigning a starting time-interval to a task. We will also investigate what the result of this is with respect to the decoupling and if either the decoupling affects this freedom or vice versa.

Chapter 5

Optimal Temporal Decoupling

In the previous chapter we have seen that we can create an arbitrary decoupling of a task scheduling STP $S = (T, C)$ in $O(m+n)$ time. However, an abritrary decoupling also has its disadvantages, since it may not live up to the preferences of the corresponding agents A_i.

Example 11. *Let us revisit the NedTrain case and suppose that a decoupling has been produced, such that* $\delta_{5,13} = 90$. *The result would then be that* $est(t_{13}) = 90$ *which equals* $lst(t_{13}) = 90$. *This means that exactly 90 minutes after the maintenance has started, task* t_{13} *has to be started, and not a minute earlier or later, due to the precedence relations and due date. The question then arises if this reflects a workable schedule and if it would not be better if* $est(t_{13})$ *was a bit smaller than* 90. *That would give the workcrew and agent* A_3 *some room to correct any delay in the schedule caused by unforseen circumstances.*

In Example 11 we can see that it may be preferable to not assign a single time-point as a starting time to a certain task t_i, but instead assign a starting time-interval, to accomodate any unforseen delays or changes in the schedule. By assigning starting time-intervals to tasks, we create *flexibility* in the schedule, and we refer to such schedules as *open schedules*.

We begin this chapter by giving a formal definition of *flexibility* and *potential flexibility* and we will show to the best of our knowledge what research has been done by others on this topic. We will present the relation between flexibility and open schedules after which we will try to find open schedules that maximize flexibility. We will further investigate the relation between open schedules and temporal decouplings and the effect a temporal decoupling has on the total flexibility that can be present in the schedule or decoupling. A Linear Programming (LP) formulation will be presented that can determine an open schedule with maximal flexibility and we will also present an alternative specialized algorithm that can find an open schedule with maximal flexibility efficiently.

5.1 Flexibility

In Chapter 3 we have presented schedules $\sigma : T \to \mathbb{R}$ for an STP $S = (T, C)$, which assigned a single time-point t to a task $t_i \in T$, i.e., $\sigma(t_i) = t$. If this is done for every task, we refer to these schedules as *fixed schedules*, meaning that a controlling agent A_i of task t_i is not

allowed to choose his own starting time for this task t_i, but is bound to use the fixed time-point t. Since this removes any autonomy an agent A_i may have, a different approach may be more preferable. We can ask ourselves for example how the schedule or t_i is affected should a task $t_j \prec t_i$ be delayed by a certain amount of time. We refer to this as the *flexibility* of the STP, which allows the agents to have more freedom in the choice of their starting times.

In order to formulate a suitable metric for our problem, we first investigate existing metrics that are mentioned in the literature.

5.1.1 Existing flexibility metrics

What we are looking for in a flexibility metric for a task scheduling STP $S = (T,C)$ is to get a quantitative measure for the flexibility per task and the total flexibility in a schedule, in such a way that if we assign a starting time to a certain task $t_i \in T$, that for no other $t_j \in T$ its measure of flexibility changes. In this chapter we will discuss several existing flexibility metrics and determine if we can use them to our stated ends.

An important thesis has been written by Hunsberger [8] in which he defines many useful properties and metrics related to STPs. He defined the relative flexibility of two tasks $t_i, t_j \in T$ in an STP $S = (T,C)$ by the temporal distance based on the temporal distance matrix D, by:

$$flex_H(t_i,t_j) = D(t_i,t_j) + D(t_j,t_i) \tag{5.1}$$

Since we are dealing with task scheduling STPs, and also try to determine decouplings, we can simplify this to the relation between the time-points z and t_i, i.e.,

$$flex_H(z,t_i) = D(z,t_i) + D(t_i,z) \tag{5.2}$$

Since the z will be a fixed time-point, we can translate this to

$$flex_H(t_i) = D(z,t_i) + D(t_i,z) = lst(t_i) - est(t_i) \tag{5.3}$$

This simplification is allowed, since in task scheduling STPs, we can relate all temporal differences between t_i and t_j to the values $est(t_i)$, $lst(t_i)$, $est(t_j)$ and $lst(t_j)$ as we already stated in Section 3.2.

We are however not only interested in the flexibility of a single task, but we would like to determine the total flexibility of an STP S. Hunsberger did not explicitly mention how to calculate the total flexibility in an STP $S = (T,C)$, but he define a related formulation, namely the *rigidity* of an STP. This rigidity of Hunsberger should denote the 'inflexibility' of the STP, which is denoted by a relative value and shows by which factor an STP can adapt to changes. Before his definition of rigidity of an STP S, he also denoted *relative rigidity* of a pair of tasks, resulting in:

$$Rig(t_i,t_j) = \frac{1}{1 + D(t_i,t_j) + D(t_j,t_i)} \tag{5.4}$$

which we again can translate to tasks related to the fixed time-point z since we are dealing with a task scheduling STP:

$$Rig(z,t_i) = \frac{1}{1+D(z,t_i)+D(t_i,z)} = \frac{1}{1+lst(t_i)-lst(t_i)} \tag{5.5}$$

Hunsberger then states that the (Root Mean-Square) rigidity of an STP S can be determined by calculating:

$$Rig(S) = \sqrt{\frac{2}{N(N+1)}\sum_{i<j}|Rig(t_i,t_j)|^2} = \sqrt{\frac{2}{N(N+1)}\sum_{i<j}|\frac{1}{1+flex_H(t_i,t_j)}|^2} \tag{5.6}$$

Corresponding from Equation 5.6, we can see that the rigidity of an STP S is equal to 1 if none of the tasks contains any flexibility, i.e., $flex_H(t_i,t_j) = 0$ for all $t_i,t_j \in T$. If all tasks $t_i \in T$ are completely unbounded, resulting in $flex_H(t_i,t_j) = \infty$, the result is that the rigidity of the STP S is equal to 0.

The problem that rises with this rigidity metric is the summation over the individual flex values. We have already shown in Equations 5.1 up to 5.3 that we can represent the flexibility metric $flex_H$ of Hunsberger by only using the $est()$ and $lst()$ values, since the matrix D can be constructed by only using the $est()$ and $lst()$ values, resulting in the fact that the flex metric of Hunsberger can be constructed by only these values. In order to establish the rigidity of an STP S, Hunsberger then sums over all flex values $flex_H(t_i)$ with $t_i \in T$. We can ignore the root mean square and multiplication by a factor, since it is the summation that causes some worry. By summing over all flex values $flex_H(t_i)$ for all $t_i \in T$, we can derive from this a flexibility metric $flex_H(S)$ to calculate the total flexibility in an STP S, i.e.,

$$flex_H(S) = \sum_{i=1}^{n} flex_H(t_i) = \sum_{i=1}^{n} (lst(t_i) - est(t_i)) \tag{5.7}$$

What however is not taken into consideration here is that $flex_H$ values can be depending on each other. If we have a $t_i,t_j \in T$ with $t_i \prec t_j$, then $flex_H(t_i)$ may be related to $flex_H(t_j)$ and can affect each other.

Example 12. *Suppose we have two STPs $S_1 = (T_1,C_1)$ and $S_2 = (T_1,C_2)$, where $T_1 = \{t_1,t_2\}$ and both processing times are 1 time-unit. Suppose that in S_1 we have a deadline $d = 4$ and $t_i \ll t_2$ and in S_2 the deadline is $d = 3$, but a precedence relation is absent. See Figure 5.1 for the corresponding STNs.*

If we compute $flex_H(S_1)$, we will get: $flex_H(S_1) = (2-0)+(3-1) = 4$, while $flex_H(S_2)$ gives us also: $flex_H(S_2) = (2-0)+(2-0) = 4$. Let us see what happens if we fix one task at a single time-point. If we have $flex_H(t_1) = 2$, then we can choose to let t_1 start at $t = 2$, since we can choose any $t \in [0,2]$. But then task t_2 is forced to start at $t = 3$, without being able to choose another time-point, indicating that this task has no freedom in its choice of starting time left. In S_2 we can choose for task t_1 to start at $t = 2$, while we still have the same time-interval $[0,2]$ for t_2 we can choose from, as we also had for t_1.

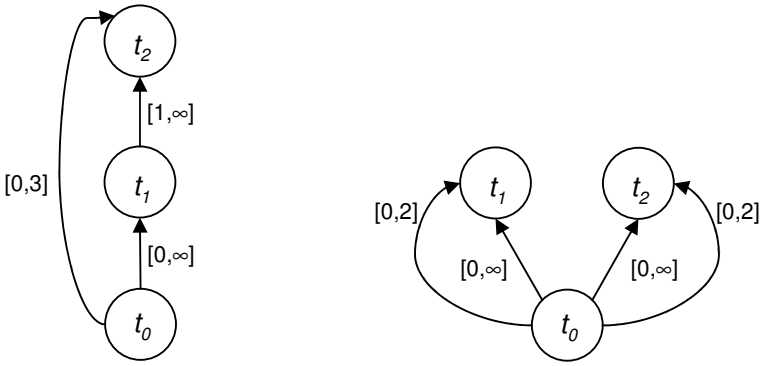

Figure 5.1: STN of S_1 and S_2 in Example 12

What we can see in Example 12 is that the freedom of choosing starting times in an STP S can depend on the presence of precedence relations. As we have seen in the example, by fixing a task at a certain starting time, other tasks may have lost their freedom to choose different starting times. As we stated at the beginning of this section, we are searching for a flexibility metric that if we fix a task at a certain starting time, that the flexibility measures of the other tasks remains unaffected. The $flex_H$ metric however calculates a different kinds of flexibility than we have in mind, which is why this metric is not suitable for our calculations.

In the literature, more flexibility metrics are defined, for example a flex metric specified by Aloulou and Portmann [1], which we will denote as $flex_A$:

$$flex_A = \frac{|\{(t_i,t_j) \mid t_i \not\prec t_j \wedge t_j \not\prec t_i\}|}{n(n-1)} \tag{5.8}$$

where $\{(t_i,t_j) \mid t_i \not\prec t_j \wedge t_j \not\prec t_i\}$ is a set that contains all pairs of tasks (t_i,t_j) which have no precedence relation $t_i \prec t_j$ or $t_j \prec t_i$, i.e., are independent of each other. This is then divided by the total number of pairs of tasks, namely $n(n-1)$ to present a relative value of how many tasks $t_i \in T$ can be modified without affecting other tasks.

Like the rigidity metric presented by Hunsberger, this $flex_A$ metric of Aloulou and Portmann also gives a relative or qualitative measure of the flexiblity in an STP S.

Example 13. *Suppose again that we have two STPs $S_1 = (T_1,C_1)$ and $S_2 = (T_1,C_2)$, with $T_1 = \{t_1,t_2\}$, where all tasks have a processing time of 1 time-unit and in both STPs we have $t_1 \ll t_2$. If S_1 has a deadline of $d = 2$ and S_2 has a deadline of $d = 4$, see Figure 5.2.*

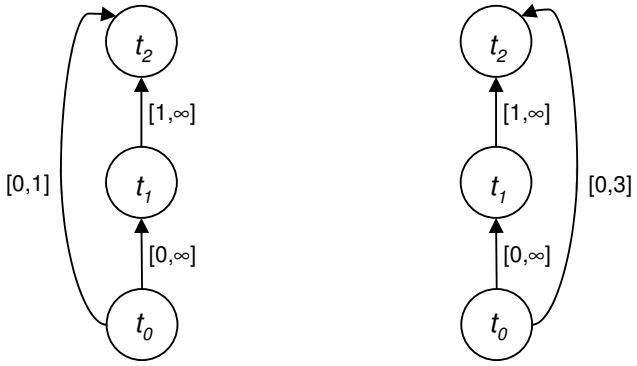

Figure 5.2: STN of S_1 and S_2 in Example 13

We can first determine the flexibility according to Hunsberger, i.e., $flex_H(S_1) = 0+0 = 0$ *and* $flex_H(S_2) = (2-0)+(3-1) = 4$. *If we fix* t_1 *in* S_2 *at* $t = 2$, *we see, like in Example 12, that task* t_2 *loses freedom due to the chosen starting time for* t_1.

If we apply $flex_A$ *to* S_1 *and* S_2, *we see the same outcome:* $flex_A(S_1) = flex_A(S_2) = 0$.

Example 13 shows us that the $flex_A$ metric of Aloulou and Portmann determines that to their notion of flexibility, there is none in both STNs. Their metric is however not a quantitative one, which we are looking for, which makes it unsuitable for our calculations.

A third flexibility metric is the *fluidity* metric of Cesta et al. [2] which is closely related to the flexibility metrics we are investigating. The fluidity metric, which we will denote by $flex_C$ is like the $flex_A$ metric a relative metric and therefore gives no information to the quantitative amount of flexibility that is present in an STP S:

$$flex_C = \sum_{i=1}^{n} \sum_{j=1 \wedge j \neq i}^{n} \frac{slack(t_i,t_j)}{H \times n \times (n-1)} \times 100 \tag{5.9}$$

where $slack(t_i,t_j)$ denotes the allowed time between the end of task t_i and the start of t_j, which in our case translates to: $slack(t_i,t_j) = lst(t_j) - est(t_i) - p_i$. The value H is the 'temporal horizon', i.e., $H = \sum_{i=1}^{n} p_i + \sum_{\forall(i,j)} r_{ij}$ and where r_{ij} is the relative release date of t_j when compared to t_i. Since we do not use relative release dates in task scheduling STPs, we can consider every r_{ij} to be equal to 0, leaving H to be the sum of all processing times.

Just like the $flex_A$ metric, here we also divide by the number of pairs of tasks $n(n-1)$ but here afterward also multiply by 100, to receive a percentage. And just like the $flex_H$ metric, we sum over all the individual flexibility values if we determine the flexibility of an STP S.

Example 14. *Suppose that we again have two STPs $S_1 = (T_1, C_1)$ and $S_2 = (T_1, C_2)$ with $T_1 = \{t_1, t_2\}$ where both tasks have a processing time of 1. In C_1 we assume that a precedence relation is defined between t_1 and t_2 such that $t_1 \ll t_2$, but in C_2 there is no such relation. Suppose S_1 has a deadline of $d = 4$ and S_2 a deadline of $d = 3$, see Figure 5.3.*

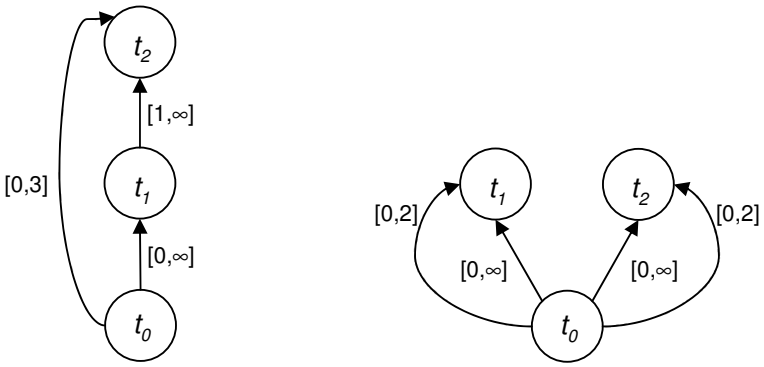

Figure 5.3: STN of S_1 and S_2 in Example 14

Then in S_1: $est(t_1) = 0$, $lst(t_1) = 2$, $est(t_2) = 1$ and $lst(t_2) = 3$. This results in: $flex_C(S_1) = \frac{(3-0-1)+(2-1-1)}{2\times2\times1} \times 100 = 50$.

In S_2 we have $est(t_1) = 0$, $lst(t_1) = 2$, $est(t_2) = 0$ and $lst(t_2) = 2$. This results in: $flex_C(S_2) = \frac{(2-0-1)+(2-0-1)}{2\times2\times1} \times 100 = 50$, which shows us that according to this flexibility metric $flex_C$ both STPs have the same flexibility. However, if we compare how both tasks are affected, we can see that if we set in S_1 $\sigma(t_1) = 2$, then t_2 will be fixed at $\sigma(t_2) = 3$, leaving only room for the 2 flexibility of t_1. In S_2 we can fix $\sigma(t_1) = 2$, but we can also choose to set $\sigma(t_2) = 2$, where for both tasks we have a flexibility of 2, giving a total of 4, indicating that S_2 fixing a starting time in S_2 does not have to affect the freedom of other tasks, contrary to S_1.

As we can see from Example 14, the fluidity metric of Cesta is not a quantitative measure of flexibility, which is why it is not suitable to our calculations. We will therefore create our own flexibility metric in the next section.

5.1.2 Open schedules and a new flexibility metric

In Section 5.1.1 we have already seen some existing flexibility metrics that are defined in the literature, but as we already stated, none of these metrics completely fits our task

scheduling STPs, since they sometimes seem to state a flexibility value that does not suite our requirements. We therefore want to create our own flexibility metric $flex$ which is based on *open schedules*, is a quantitative metric, contrary to $flex_A$ and $flex_C$, and also takes the precedence relations and their effects into account, contrary to $flex_H$. We therefore state three requirements of this flexibility metric:

1. The flexibility of a task t_i consists of being able to choose any starting time t in an interval $\sigma_o(t_i) = [m_i, n_i]$ as specified by some schedule σ_o.

2. The flexibility of a task t_i with respect to the schedule σ_o is independent of the exact time chosen for any $t_j \in T$, where $i \neq j$.

3. Any combination of starting times of any task t_i in the interval $\sigma_o(t_i)$ and any task t_j in $\sigma_o(t_j)$ gives a feasible schedule for S.

The first requirement is added to ensure that an assigned starting time-interval to a task t_i contains all feasible starting time-points of a task t_i, indicating that the size of the interval $[m_i, n_i]$ determines the flexibility of task t_i. The second requirement is based on our experiences in Sectin 5.1.1, where in other metrics precedence relations between tasks t_i and t_j would influence each others flexibility if a starting time was chosen for either t_i or t_j. The third requirement is added to ensure feasibility based on this flexibility metric.

We will give a definition of an open schedule, in order to explain the notation σ_o:

Definition 8. *Given a task scheduling STP $S = (T, C)$ an* open schedule *σ_o for S is an assignment $\sigma_o : T \to \{[m,n] : 0 \leq m \leq n < \infty\}$[1] such that for every $t_i \in T$ and every value $k_{t_i} \in \sigma_o(t_i) = [m_i, n_i]$, the assignment σ, where $\sigma(t_i) = k_{t_i}$ is a fixed schedule[2] for S.*

From this definition we can see the difference between a fixed schedule as defined in Chapter 2 and an open schedule, where in a fixed schedule every task $t_i \in T$ will get a single time-point $\sigma(t_i)$ assigned, and in an open schedule the assignment $\sigma_o(t_i)$ will be a time-interval, which does not have to contain more than one time-point, indicating that an open schedule can still be a fixed schedule if for all $t_i \in T$ we have that $m_i = n_i$.

An open schedule σ_o however allows us to determine the flexibility present in this schedule, and we will define this as follows:

Definition 9. *Given a task scheduling STP $S = (T, C)$ and a corresponding open schedule σ_o for S, the flexibility $flex_{\sigma_o}(t_i)$ of a task $t_i \in T$ w.r.t. σ_o is defined as $flex_{\sigma_o}(t_i) = n_i - m_i$ where $\sigma_o(t_i) = [m_i, n_i]$. The flexibility $flex_{\sigma_o}(S)$ of S w.r.t. σ_o is defined as the sum of the flexibilities of all the tasks occuring in S: $flex_{\sigma_o}(S) = \sum_{t_i \in T} flex_{\sigma_o}(t_i)$.*

As the reader can check, this flexibility metric fits all our requirements stated earlier in this section. This flexibility metric $flex$ allows us now to determine the quality of an open schedule σ_o, i.e., the total flexibility $flex(S)$ of the task scheduling STP S that can be contained in σ_o.

[1] This indicates that every $\sigma_o(t_i)$ is a closed interval in $\mathbb{R}^+$.

[2] Note that this definition of a fixed schedule was already presented in Chapter 2.

However, this flexibility $flex(S)$ or $flex(t_i)$ for some $t_i \in T$ can only be determined once an open schedule σ_o has been created. To get an indication of flexibility that is possible to assign to a task $t_i \in T$, we introduce a second notion of flexibility, namely the *potential flexibility* of a task.

Definition 10. *In a task scheduling STP $S = (T,C)$ we define the* potential flexibility $flex_p$ *of a task $t_i \in T$ by $flex_p(t_i) = lst(t_i) - est(t_i)$.*

We can see that this potential flexibility was in fact the flexibility metric that was introduced by Hunsberger, but in our case is used in a different manner, since we will not use $flex_p$ to determine the total flexibility of an STP S, since this can result in an overestimation as we already showed in Example 12. Since $flex_p(t_i)$ gives the maximum flexibility that can be assigned to a task $t_i \in T$, we have as a result that $flex(t_i) \leq flex_p(t_i)$.

Since $flex_p(S) = \sum_{t_i \in T} flex_p(t_i)$ will give us an upper bound on the total flexibility that can be present in an open schedule σ_o of a task scheduling STP $S = (T,C)$, we are interested if we can use our new flexibility metric $flex$ to determine open schedules that contain a maximal amount of flexibility.

5.2 Maximum flexibility

With our new flexibility metric $flex$ we can determine the flexibility that is present in an open schedule σ_o of a task scheduling STP $S = (T,C)$, but the question arises then if there is not a 'better' open schedule σ'_o for S that contains even more flexibility, i.e., $flex_{\sigma'_o}(S) > flex_{\sigma_o}(S)$. To do this, we will introduce a new definition for $flex(S)$ to denote the maximal flexibility in an STP S:

Definition 11. *Given a task scheduling STP $S = (T,C)$, the flexibility of S is defined as $flex(S) = \max\{flex_{\sigma_o}(S) \mid \sigma_o$ is an open schedule for $S\}$.*

From this definition we can see that if we know all possible open schedules σ_o of a task scheduling STP $S = (T,C)$, then we could select the one with the highest total flexibility to know the flexibility $flex(S)$ of the STP S. This can however be a very time consuming approach, since we may even have an infinite number of different open schedules σ_o, depending on the time distribution.

There is an observation we can later use to efficiently solve this dilemma:

Proposition 5. *An assignment σ_o, where for every $t_i \in T$, $\sigma_o(t_i) = [m_i, n_i]$, is an open schedule for a task scheduling STP $S = (T,C)$ iff the following conditions hold:*

1. *for every t_i, $est(t_i) \leq m_i \leq n_i \leq lst(t_i)$;*
2. *$n_i + p_i \leq m_j$ whenever $t_i \prec t_j$.*

Proof. Assume that σ_o is a flexible open schedule for $S = (T,C)$ where $\sigma_o(t_i) = [m_i, n_i]$. We show that both conditions are satisfied. Choose two fixed schedules σ_1 and σ_2 such that for all $t_i \in T$, $\sigma_1(t_i) = m_i$ and $\sigma_2(t_i) = n_i$. By Proposition 1, it follows that $est(t_i) \leq m_i \leq n_i \leq$

$lst(t_i)$ proving that Condition 1 is satisfied. To prove Condition 2, suppose, on the contrary, that there exists a $t_i \prec t_j$ such that $n_i + p_i > m_j$. Consider an assignment σ_3 such that for all $t_i \neq t_j$, $\sigma_3(t_i) = \sigma_2(t_i) = n_i$, and $\sigma_3(t_j) = m_j$. By Definition 8, σ_3 is a fixed schedule for S, in particular implying that $n_i + p_i \leq m_j$, contradicting the assumption.
Conversely, assume that we have an assignment $\rho_o(t_i) = [m_i, n_i]$ satisfying the conditions stated above. We have to show that $\rho_o(t_i)$ is an open schedule. So take an arbitrary (fixed) schedule ρ such that for every $t_i \in T$, $\rho(t_i)$ takes a value v_i in $[m_i, n_i]$. It suffices to prove that ρ satisfies all constraints in C. Since for every such value v_i we have $est(t_i) \leq v_i \leq lst(t_i)$, the release time and deadline conditions are satisfied. Let $t_i \prec t_j$ be an arbitrary precedence constraint in C. Since $n_i + p_i \leq m_j$, we have $v_i + p_i \leq n_i + p_i \leq m_j \leq v_j$, hence the precedence constraint $t_i \prec t_j$ is satisfied, too. Therefore ρ satisfies C and ρ_o is an open schedule for S. □

Proposition 5 shows us that we have to find an assignment $[m_i, n_i]$ for all tasks $t_i \in T$ that respect the constraints in order to represent a feasible open schedule for a task scheduling STP $S = (T, C)$. As it turns out, it is possible to create a Linear Programming (LP) formulation that represents the problem of creating a maximally flexible open schedule.

5.2.1 An LP formulation for determining the maximum flexibility

In Definition 4 we can see that all constraints of an STP $S = (T, C)$ are linear. Furthermore, the requirements that are stated in Proposition 5 are also linear. If we want to maximize the flexibility in S, we have to maximize the sum of flexibility values of the individual tasks, as we stated in Definition 11, which is maximizing a linear function. To summarize, we have a linear objective function and only linear constraints, from which we can conclude that we can create a Linear Program (LP) that represents the maximum flexibility problem in an STP $S = (T, C)$:

$$\begin{aligned} &\max \sum_{t_i \in T} (n_i - m_i) \\ &\text{s.t. } \forall t_i : \quad && m_i \leq n_i \\ &&& n_i \leq lst(t_i) \\ &&& m_i \geq est(t_i) \\ &\text{s.t. } \forall t_i \ll t_j, && n_i + p_i \leq m_j \end{aligned} \tag{5.10}$$

Some constraints in this LP formulation can of course be rewritten in order to maintain a $\leq$-relation, but for the sake of simplicity, we have represented the constraints in the chosen way. An advantage of this LP formulation and its corresponding flex metric is that it gives us the exact flexibility of the STP S.

Example 15. *We can also compute a maximally flexible open schedule in the NedTrain case, by applying the LP formulation to the corresponding task scheduling STP. The result of this is then an optimal solution:*

task	m_i	n_i
1	*0*	*0*
2	*20*	*20*
3	*20*	*55*
4	*20*	*20*
5	*40*	*60*
6	*40*	*65*
7	*75*	*75*
8	*30*	*30*
9	*35*	*75*
10	*30*	*65*
11	*70*	*70*
12	*80*	*80*
13	*90*	*90*

The total flexibility of this open schedule is then $flex(S) = 155$*, which is the summation over the size of the independent intervals. The total potential flexibility that was present before computing a feasible open schedule was equal to* 380*, which in turn is equal to the total Hunsberger-flexibility* $flex_H(S)$.

Another advantage of having an LP formulation of the maximum flexibility problem for a task scheduling STP is that it gives us a guarantee that an optimal solution can be found in polynomial time. This LP formulation in combination with the new flex metric also directly answers our fourth research question, as to how we can determine a schedule that can maximally adapt to any unforseen influences, i.e., determining an open schedule such that the sum of flexibility values of all the tasks is maximal.

However, with an LP formulation that can construct open schedules with maximal flexibility for task scheduling STPs, we still have not created a decoupling. The question is then in what manner decoupling affects the total flexibility that can be present in an STP and whether or not it is better to first create a decoupling and then a maximally flexible open schedule based on this decoupling, or create a decoupling based on a maximally flexible open schedule.

5.2.2 Maximally flexible decouplings

So far we have presented a manner in which to create arbitrary decouplings for task scheduling STPs in Chapter 4, which we showed can be done in $O(m+n)$ time. Arbitrary decouplings however, may affect the total flexibility that can be present in a task scheduling STP S dramatically.

Example 16. *Suppose that in the NedTrain case an arbitrary decoupling is created for which among others the constraints* $t_5 \leq 40$*,* $t_7 \leq 50$ *and* $t_{11} \leq 40$ *are added. As the reader can check, there hardly remains any room for flexibility in an open schedule and the maximum flexibility of 155 from Example 15 is no longer reachable, since these added constraints severly limit the size of the intervals of* t_5*,* t_7*,* t_{11} *and their predecessors in any open schedule.*

Example 16 raises the question if it is at all possible to construct a decoupling such that the total flexibility in a maximally flexible open schedule without a decoupling is achieved. Is there a loss in flexibility if we construct a decoupling and in which order do we have to do this: decouple first, schedule second, or schedule first and decouple second?

Before we can answer this question, we first have to define what we consider to be an *optimal temporal decoupling* of a task scheduling STP $S = (T, C)$.

Definition 12. *An* optimal temporal decoupling $\{S_i\}_{i=1}^k$ *of a task scheduling STP* $S = (T, C)$ *is a decoupling of* S *such that the sum* $\sum_{i=1}^k flex(S_i)$ *is maximal.*

From Definition 11 we can see that $flex(S)$ is defined to be the maximum over the total flexibility values of all of the feasible open schedules of S. Since a decoupling can only limit the number of feasible open schedules, it follows that $\sum_{i=1}^k flex(S_i) \leq flex(S)$. The question then is, does it hold that $\sum_{i=1}^k flex(S_i) = flex(S)$?

We will show in this section that it indeed holds that $\sum_{i=1}^k flex(S_i) = flex(S)$ for any task scheduling STP $S = (T, C)$.

Proposition 6. *Let* $\{S_i\}_{i=1}^k$ *be an optimal decoupling of* S*. Then* $\sum_{i=1}^k flex(S_i) = flex(S)$.

Proof. Consider the set of intervals $\{[m_i^*, n_i^*] : t_i \in T\}$ occurring as solutions of the LP formulation (5.10). Given the set $A = \{A_l\}_{l=1}^k$ of k agents, let $C_{inter} \subset C$ be the set of all inter-agent constraints. For every inter-agent constraint $t_i \ll t_j$, where t_i occurs in S_k and t_j occurs in S_l, add a constraint $t_i \leq n_i^*(t)$ to C_k and add $t_j \geq n_j^*$ to C_l. We show the following.

Claim 1: The resulting systems $\{S_l\}_{l=1}^k$ constitute a decoupling of S;
Claim 2: The sum of the flexibilities $flex(S_l)$ of the systems S_l in the decoupling $\{S_l\}_{l=1}^k$ equals $flex(S)$.

Ad Claim 1. Suppose on the contrary that the merge σ of some combination of individual schedules σ_k, where σ_k is a schedule for S_k, violates some constraints in C. Then σ must violate some inter-agent constraint $t_i \prec t_j$ where t_i occurs in S_k and t_j occurs in S_l. Hence, $\sigma_k(t_i) + p_i > \sigma_l(t_j)$. Since $\sigma_k(t_i) \leq n_i^*$ and $\sigma_l(t_j) \geq m_j^*$, it follows that $n_i^* + p_i > m_j^*$ violating the constraint that $n_i^* + p_i \leq m_j^*$, which is a contradiction.

Ad Claim 2. By definition, we have $\sum_{l=1}^k flex(S_l) \leq flex(S)$. Let σ_o be an open schedule for S realizing maximum flexibility where $\sigma_o(t) = [m_t^*, n_t^*]$. For every $S_i = (T_i, C_i)$ consider the assignment σ_o^i, being the restriction of σ_o to T_i. It is easy to see that each such an assignment σ_o^i is an open schedule for S_i. But then $\sum_{i=1}^k flex(\sigma_o^i) = flex(\sigma_o)$. Hence, there exists a decoupling realizing $flex(S)$. □

The result of Proposition 6 is extremely important in our search for optimal decouplings, since this shows us that by creating an optimal decoupling, we do not lose any flexibility if we compare it to creating a maximally flexible open schedule. But from this, we immediately receive a formulation to compute an optimal decoupling in polynomial time, since an optimal decoupling can be derived from a maximally flexible open schedule. If we have such a maximally flexible open schedule σ_o for a task scheduling STP $S = (T, C)$, then for any inter-agent constraint with $t_i \ll t_j$ where $t_i \in T_k$ and $t_j \in T_l$, we can add $t_i \leq n_i^*$ to C_k and $t_j \geq m_j^*$ to C_l if these constraints are more strict than the constraints currently present

in respectively C_k and C_l. Since we already have shown in Section 5.2.1 that an LP formulation can be used to solve the maximally flexible open schedule problem, we can use this formulation to determine an optimal decoupling for a task scheduling STP.

Example 17. *A maximally flexible decoupling in the NedTrain case can be created by adding the following decoupling constraints to the already existing constraints:*

$$\sigma(t_1) \leq 0 \qquad \sigma(t_8) \leq 30$$
$$\sigma(t_4) \leq 20 \qquad \sigma(t_{13)} \geq 90$$

These constraints allow for an open schedule in which a total of 155 flexibility is present, which equals the solution found by the LP program in Example 15.

Remark. *As has been shown in [11], optimal decoupling in STPs can be achieved in polynomial time if the objective is linear. The general construction in STPs however, requires more than* $|T|^3 + |T| + |C|$ *constraints and* $|T|^2$ *variables in the underlying LP. Using task scheduling instances, we are able to reduce the number of constraints in the LP to* $3|T| + |C|$ *and the number of variables to* $2|T|$.

The fifth research question stated to research if there was a trade-off between creating a decoupling and creating a maximally flexible schedule. In this section we have shown that by the use of the LP formulation 5.10 we can compute a maximally flexible open schedule for a task scheduling STP and from this we can derive an optimal decoupling by adding inter-agent constraints based on the time-intervals assigned to the corresponding tasks, since Proposition 6 shows us that the total flexibility in a maximally flexible decoupling equals that of a maximally flexible open schedule.

We now know that we can use an LP formulation to solve the optimal decoupling problem or the maximally flexible open schedule problem. Although it is proven to be solvable in polynomial time, we can ask ourselves if it is not possible to exploit more properties of a task scheduling STP. After all, among others the precedence constraints give a task scheduling STP a unique edge over arbitrary STPs and although the LP formulation uses them as constraints, it might be possible to exploit them even further. In the next section we will present a newly developed alternative polynomial time algorithm that exploits the unique properties of task scheduling STPs in order to compute a maximally flexible open schedule.

5.3 Computing optimal decouplings with a new alternative algorithm

We have seen in Section 3.2 that task scheduling STPs distinguish themselves from arbitrary STPs among others because of the existence of strict precedence relations, where a task $t_i \in T$ has to be completed before a task $t_j \in T$ is allowed to start if $t_i \ll t_j$. And it is specifically this relation that can be exploited if we want to determine a maximally flexible open schedule. The strict precedence relation induces a number of special properties on the STN that represents the task scheduling STP.

We will introduce the notion of *critically connected* STNs[3] and show that in critically connected STNs we can define a maximally flexible open schedule by determining a maximum independent set of tasks. We will then show that any task scheduling STN S can be converted to one or more critically connected STNs S_i and we will show that a *merge* of the maximally flexible open schedules σ_o of all STNs S_i is again a maximally flexible open schedule of the original STN S. Since we can determine a maximum independent set in a partially ordered transitive graph in polynomial time and the remaining steps also remain polynomial, this results in an efficient polynomial time algorithm.

5.3.1 Critically connected STNs

Before we give a definition of *critically connected* STNS, we will introduce the notion of critical paths in an STN:

Definition 13. *A path $P = \{t_i, t_j, \ldots, t_k\}$ is called an* est-critical path *if for every $t_i, t_j \in P$ with $t_i \ll t_j$ we have that $est(t_j) = est(t_i) + p_i$. If for every $t_i, t_j \in P$ with $t_i \ll t_j$ we have that $lst(t_i) = lst(t_j) - p_i$ this path is called an* lst-critical path *and traverses the directed edges in an opposite direction.*

When we calculate the $est()$ value of all tasks $t_i \in T$, we can use Equation 3.1, which shows us that if $t_j \ll t_i$, then the $est(t_i)$ value depends on one or more $t_j \in pre_{\ll}(t_i)$ or r_i. If it depends on a certain $t_j \in pre_{\ll}(t_i)$, then (t_j, t_i) is (part of) an est-critical path to t_i. Based on Equation 3.2 this holds also in the case of lst-critical paths if $t_i \ll t_j$ and $lst(t_i) = lst(t_j) - p_i$, since then (t_j, t_i) is (part of) an lst-critical path to t_j.

Definition 14. *An undirected graph $G = (V, E)$ is said to be* connected *if it contains a path from v_i to v_j for all $v_i, v_j \in V$. A directed graph $G = (V, E)$ is said to be* weakly connected *if replacing all of its directed edges by undirected edges produces a connected graph.*

Since a task scheduling STP $S = (T, C)$ is represented by an STN, we can denote an STN to be weakly connected if it matches Definition 14 of a directed graph.

Definition 15. *We define a task scheduling STN $S = (T, C)$ to be* critically connected *if it is weakly connected and every path $P = \{t_i, \ldots, t_j\}$ in the graph between any two task $t_i, t_j \in T \setminus \{z\}$ is both an est-critical and lst-critical path*[4].

Example 18. *Suppose that we have a task scheduling STN $S = (T, C)$, with $T = \{t_1, t_2, t_3, t_4\}$ as shown in Figure 5.4, where all tasks have a processing time of 1 and we have a due date $d = 5$. As the reader can verify, we have:*

task	*est*	*lst*
1	*0*	*2*
2	*1*	*3*
3	*1*	*3*
4	*2*	*4*

[3] From this point we will use the notation S to represent both the STP and its corresponding STN.

[4] Note that in this lst-critical case we traverse the path in the opposite edge directions.

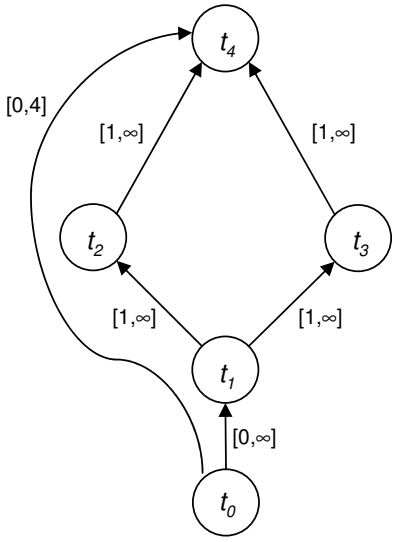

Figure 5.4: STN of example 18

As the reader also can verify, the two paths $P_1 = \{t_1, t_2, t_4\}$ and $P_2 = \{t_1, t_3, t_4\}$ are both est-critical and lst-critical, i.e., any path in S is est-critical and lst-critical, resulting in S being critically connected.

A critically connected STN $S = (T, C)$ has some special properties, since the $est()$ value of a task $t_i \in T$ depends on all $t_j \ll t_i$ with $t_j \in T$, because every path (t_j, t_i) is est-critical. The same holds for every $lst()$ value of t_i, which shows us that we have the following relation between the potential flexibility $flex_p$ for three tasks $t_i \ll t_j \ll t_k$:

Proposition 7. *In a critically connected STN $S = (T, C)$ it holds for all $t_i, t_j \in T$ that $flex_p(t_i) = flex_p(t_j)$.*

Proof. Since every path in S is both est-critical and lst-critical, it holds that for every $t_i, t_j \in T$ with $t_i \ll t_j$ that $est(t_j) = est(t_i) + p_i$ and $lst(t_i) = lst(t_j) - p_i$.

We then have:

$$\begin{aligned} flex_p(t_i) &= lst(t_i) - est(t_i) = \\ &= lst(t_j) - p_i - est(t_i) = \\ &= lst(t_j) - est(t_j) = \\ &= flex_p(t_j) \end{aligned}$$

Since this holds for any $t_i \ll t_j$, this also holds for all $t_i, t_k \in P$ for any path $P = \{t_i, \ldots t_k\}$ in S and due to S being critically connected, this holds for all $t_i, t_j \in T$. □

Example 19. *If we revisit Example 18, we can calculate that for every task $t_i \in T$ we have that $flex_p(t_i) = 2$*

One might wonder what the advantage is of a critically connected nature of a task scheduling STN compared to an arbitrary task scheduling STN. Proposition 7 shows us that in a critically connected STN $S = (T,C)$ all tasks $t_i, t_j \in T$ have the same potential flexibility $flex_p$. There is however another relation due to the critically connected relation in this STN, which we will show after the following definitions:

Definition 16. *In a task scheduling STN $S = (T,C)$ we define a* chain *of tasks $L = \{z, t_i, \ldots, t_k\}$ a partially ordered subset of T, which forms a path in S, in which each pair of tasks $t_i, t_j \in L$ has either $t_i \prec t_j$ or $t_j \prec t_i$ and t_k is a task such that $suc(t_k) = \emptyset$. An* antichain *of tasks $A = \{t_i, \ldots, t_k\}$ is a subset of T in which each pair of tasks $t_i, t_j \in A$ has no precedence relation, i.e., $t_i \not\prec t_j$ and $t_j \not\prec t_i$.*

Example 20. *If we revisit Example 18, we can find a maximum antichain by selecting both t_2 and t_3. It holds that $t_2 \not\prec t_3$ and $t_3 \not\prec t_2$, which proves that this is indeed an antichain. A larger antichain is not possible, since there are only two chains to be found in S, namely $\{t_0, t_1, t_2, t_4\}$ and $\{t_0, t_2, t_3, t_4\}$.*

A chain L of tasks in a task scheduling STN $S = (T,C)$ contains only tasks that are all related to each other due to one or more precedence constraints. If the STN S is critically connected, then for any $t_i, t_j \in L$, we have that if we increase $flex(t_i)$ we decrease the amount of flexibility that can be given to any $t_j \in L$, which we refer to as the *remaining potential flexibility*.

Definition 17. *In a critically connected task scheduling STN $S = (T,C)$ we define the* remaining potential flexibility *$flex_r$ of a task $t_i \in T$ by $flex_r(t_i) = flex_p(t_i) - \sum_{L \in L_i} \sum_{t_j \in L} flex(t_j)$ where L_i here denotes the set of all chains L that contain t_i, and where t_j is unique*[5].

The notion of remaining potential flexibility $flex_r$ is of major importance to the new iterative algorithm we will describe, since it shows us how much flexibility a task t_i can still receive if other tasks $t_j \in T$ have already received some flexibility $flex(t_j)$. From this definition, we can immediately derive the following Corollary:

Corollary 1. *Suppose we have a critically connected task scheduling STN $S = (T,C)$ with $T = \{z, t_1, t_2, \ldots, t_n\}$, then any chain $L = \{z, \ldots, t_i, \ldots, t_k\}$ contains at most $flex_p(t_i)$ flexibility, i.e., $\sum_{t_j \in L} flex(t_j) \leq flex_p(t_i)$.*

Proof. Suppose that we have an arbitrary chain L and we assign flexibility to its tasks, such that L contains the maximum flexibility it can hold, i.e., $\sum_{t_j \in L} flex(t_j)$ is maximal, then it must hold that for all $t_i \in L$ we have $flex_r(t_i) = 0$, otherwise the total flexibility would not be maximal. If $flex_r(t_i) = 0$, then according to Definition 17:

$$flex_p(t_i) - \sum_{L \in L_i} \sum_{t_j \in L} flex(t_j) = 0$$

i.e.,

$$\sum_{L \in L_i} \sum_{t_j \in L} flex(t_j) = flex_p(t_i)$$

[5] We use the notion unique here to denote that t_j is only used once in the summation, even if it occurs in multiple chains L_i that contain t_i.

Since

$$\sum_{t_j \in L} flex(t_j) \leq \sum_{L \in L_i} \sum_{t_j \in L} flex(t_j)$$

we can conclude that:

$$\sum_{t_j \in L} flex(t_j) \leq flex_p(t_i)$$

□

We can go even further than the statement in Corollary 1 by showing that the inequality sign can even be replaced by an equality sign, due to the use of antichains (see Definition 16).

Corollary 2. *Suppose we have a critically connected task scheduling STN $S = (T, C)$ with $T = \{z, t_1, t_2, \ldots, t_n\}$, then any chain $L = \{z, \ldots, t_i, \ldots, t_k\}$ contains exactly $\sum_{t_j \in L} flex(t_j) = flex_p(t_i)$ flexibility and the flexibility in L can be concentrated in one task $t_i \in L$, i.e., $flex(t_i) = flex_p(t_i)$.*

Proof. Suppose that we have the set A^* that is the largest set of antichains in S. Then any chain L must have exactly one task $t_i \in L$ that is also in A^*, otherwise A^* is not maximal. Since all $t_i, t_j \in A^*$ are independent, i.e., $t_i \not\prec t_j$ and $t_j \not\prec t_i$, we can take $flex(t_i) = flex(t_j) = flex_p(t_i)$, resulting for every L in $\sum_{t_j \in L} flex(t_j) = flex_p(t_i)$. □

Corollary 2 shows us that we can determine a maximally flexible open schedule in a critically connected task scheduling STN S by calculating a maximum antichain. Determining a maximum antichain is equivalent to determining a maximum independent set in the STN S.

Example 21. *Let us again revisit Example 18. In order to find a maximally flexible open schedule, we have to distribute as much flexibility in this network as possible. Should we for example assign flexibility to t_1, then by Definition 17 we can see that it would lower the $flex_r$ value of all other tasks. The same holds for giving flexibility to t_4. However, if we would select t_2 and give flexibility to this task, then t_3 remains unaffected and vice versa.*

By determining a maximum antichain A^*, we can find the largest set of tasks that can receive flexibility independently of each other, i.e., the $flex_r$ value of any $t_j \in A^*$ is not lowered if for some $t_i \in A^*$ the value $flex(t_i)$ is increased. In Figure 5.4 we have that $A^* = \{t_2, t_3\}$ resulting in a maximally flexible open schedule if $flex(t_2) = flex(t_3) = flex_p(t_2) = 2$.

What remains now is to show how to convert an arbitrary task scheduling STN S to one or more critically connected STNs S_i, such that the merge of the maximally flexible open schedules of the STNs S_i again gives a maximally flexible open schedule of S.

5.3.2 The Maximum Flexibility Algorithm

An arbitrary task scheduling STN $S = (T, C)$ does not have to have the property that $flex_p(t_i) = flex_p(t_j)$ for all $t_i, t_j \in T$, which is why we cannot simply define a maximum independent

set in S and create a maximally flexible open schedule by doing so. If the STN is not critically connected, we will have to find a way to convert it to one or more critically connected STNs S_i. But how do we do that? In order to answer this question we will first have to investigate the parts of an STN that prevent it from being critically connected.

Let us start with an example of a simple task scheduling STN that is not critically connected:

Example 22. *Suppose we have a task scheduling STN $S = (T,C)$ with $T = \{t_1,t_2,t_3,t_4\}$ where $p_1 = p_3 = p_4 = 1$ and $p_2 = 2$, $d = 5$ and precedence relations as can be seen in Figure 5.5. From this we can calculate the est(), lst() and corresponding $flex_p$ values.*

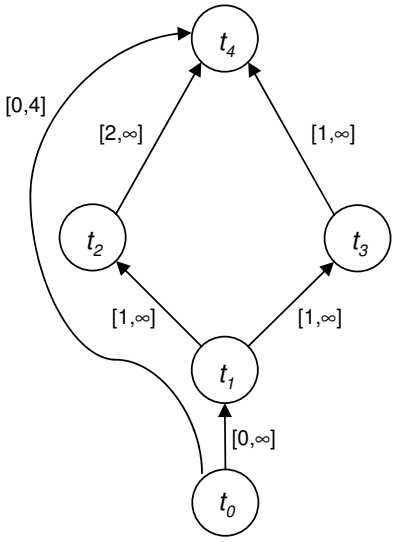

Figure 5.5: STN of example 22

task	*est*	*lst*	$flex_p$
1	*0*	*1*	*1*
2	*1*	*2*	*1*
3	*1*	*3*	*2*
4	*3*	*4*	*1*

Since not all $flex_p$ values are equal, we can conclude immediately that S is not critically connected. If we look closer at the values, we see that the path $P = \{t_1,t_2,t_4\}$ is an est-critical and lst-critical path, but that $P' = \{t_1,t_3,t_4\}$ is neither. The cause of this is that t_2 has a processing time $p_2 = 2$, which results in t_3 to have no effect on the $est(t_4)$ and $lst(t_1)$ value.

If we evaluate the differences between t_2 and t_3, we can see that t_3 is in both directions 1 short in processing time, which gives t_3 some freedom. What we can do now is create two seperate STNs. Suppose we take the original STN S and treat t_3 as if it somehow makes

up for the lower processing time such that P' as described earlier is now indeed both an est-critical and lst-critical path, where we denote this new STN by S_1. By doing this, we have 'stolen' 1 unit of potential flexibility of t_3, which should be compensated. In order to compensate this, we can create a second STN S_2 that contains only this task t_3 and is only allowed a potential flexibility $flex_p(t_3)$ of 1. We can see that S_2 is critically connected, since it contains only t_0 and t_3.

In S_1 we have now for all $t_i \in T$ that $flex_p(t_i) = 1$ and in S_2 we see also that $flex_p(t_3) = 1$. We can now determine in each STN S_1 and S_2 a maximum independent set (or maximum antichain) and give these tasks t_i their potential flexibility $flex_p(t_i)$. This results in S_1 in $flex(t_2) = 1$ and $flex(t_3) = 1$ and in S_2 in $flex(t_3) = 1$. If we combine these values again, we will get $flex(t_2) = 1$ and $flex(t_3) = 2$. The open schedule that follows from this combination is then:
$\sigma_o(t_1) \in [0,0]$, $\sigma_0(t_2) \in [1,2]$, $\sigma_o(t_3) \in [1,3]$ and $\sigma_o(t_4) \in [4,4]$, which is a maximally flexible open schedule for S.

What we demonstrated in Example 22 is key in the approach of constructing maximally flexible open schedules. During the calculations, we never had to use the $[m_i, n_i]$ values which would denote the starting time interval of t_i, but were we able to reconstruct them by simply using the assigned flexibility values $flex(t_i)$. This shows us that during the iterative solving of the separate critically connected STNs, that once we know how these STNs are created, we only have to know the potential flexibility values $flex_p$ of the tasks and determine which tasks will receive their potential flexibility as true flexibility.

We will create an algorithm that iteratively increases the flexibility given to a certain set of tasks in a task scheduling STN $S = (T, C)$. The idea of this algorithm is the following:

1. Check if S is critically connected. If true, determine a maximum independent set and give these tasks their $flex_p$ value as $flex$ value. If false, see step 2)

2. If S is not critically connected, create in each iteration i a subproblem S_i that is critically connected, such that the merge of all subproblems S_i is S again and the merge of the solutions σ_i is also a feasible solution σ for S

3. In each subproblem S_i determine a maximum independent set and raise the $flex$ value of these tasks by $flex_p$ of this current S_i. Repeat step 2) and 3) until for all tasks $flex_r = 0$, resulting in a $flex$ assignment for all tasks

Example 22 gives us an example of how to convert certain task scheduling STNs S that are not critically connected to the right form, but this does not work in all cases however. We therefore have to find a universal approach that can convert any task scheduling STN S to one or multiple critically connected task scheduling STNs. In Example 22 we have seen that due to t_3 on path P', P' was in both directions not est-critical or lst-critical. There are however cases in which a path can be est-critical, but lacks its lst-critical counterpart or vice versa. We will therefore introduce the notion of *slack* on an edge (t_i, t_j) in order to determine if an edge is est-critical and/or lst-critical.

Definition 18. *In a task scheduling STN $S = (T, C)$, if the edge between two tasks $t_i \ll t_j$ with $t_i, t_j \in T$ is not on a critical path from z to t_j, we define the* slack *from t_i to t_j, denoted*

by $s(t_i,t_j)$ on this edge (t_i,t_j), to be equal to $s(t_i,t_j) = est(t_j) - est(t_i) - p_i - flex(t_i)$. The slack from t_j to t_i is equal to $s(t_j,t_i) = lst(t_j) - lst(t_i) - p_i - flex(t_j)$.

Example 23. *If we revisit Example 22 we can determine the slack values s for all edges. Since path $P = \{t_1,t_2,t_4\}$ is both est-critical and lst-critical, there will be no slack s present on any of the traversed edges in any direction. The remaining edges (t_3,t_4) and (t_3,t_1) contain both 1 slack, due to: $s(t_3,t_4) = 3-1-1 = 1$ and $s(t_3,t_1) = 3-1-1 = 1$*

Before we continue, we will have to update the $est()$ and $lst()$ calculations given in Equations 3.1 and 3.2. If we want to create an iterative algorithm, by raising $flex$ values in each iteration, the $est()$ values of tasks can be increased by this increase of flexibility. This has to be taken into account, which can be done by the following equation:

$$est(t_i) = \max(\{est(t_j) + p_j + flex(t_j) \mid t_j \in pre_{\ll}(t_i)\} \cup r_i) \tag{5.11}$$

And also for $lst()$:

$$lst(t_i) = \min(\{lst(t_j) - p_i - flex(t_j) \mid t_j \in suc_{\ll}(t_i)\} \cup d_i - p_i) \tag{5.12}$$

Since these equatiouns lead to a recursive calculation, in Equation 5.11 all flexibility given to the predecessors of t_i is taken into account, while the same holds also for Equation 5.12, where all flexibility given to the successors of t_i is used in the calculations. This is of importance to the use of slack values during the iterations. Note that by only knowing the release dates, due dates and $flex(t_i)$ values of all t_i, we can construct an open schedule σ_o by using Equations 5.11 or 5.12

In order to deal with slack values to construct critically connected STNs S_i, we can use a similar approach as in Example 22. If we for example have $s(t_i,t_j) = c$, for some $c \in \mathbb{R}$, then this indicates that either t_i or any of its predecessors can receive c flexibility without t_j being affected. Only once this c flexibility has been given to t_i or any of its predecessors will t_j possibly be affected by a further increase of flexibility of t_i or its predecessors. However, if t_j should receive any flexibility, then t_i and its predecessors are directly affected should $s(t_j,t_i) = 0$. Therefore t_i is only independent for c flexibility of t_j, but t_j is not considered independent of t_i and its predecessors.

Example 24. *Let us revisit Example 22 and the corresponding Figure 5.5. We have shown in Example 23 that the slack values $s(t_3,t_1) = 1$ and $s(t_3,t_4) = 1$. Since the slack on (t_3,t_4) in the direction of t_4 is equal to 1, we can give 1 unit of flexibility to t_3, such that $flex(t_3) = 1$. We then see that both slack values decrease, i.e., $s(t_3,t_1) = 0$ and $s(t_3,t_4) = 0$, but for t_1, t_2 and t_4 we still have $flex_r = 1$, which remained unchanged during the increase of $flex(t_3)$. This showed us that t_3 was independent from all other tasks for a total of 1 flexibility, with respect to the $flex_r$ values.*

If we however return to our begin situation, where all $flex$ values are 0, and we increase $flex(t_4)$ by one unit, then we see that for all tasks the $flex_r$ value decreases by 1, even $flex_r(t_3)$, which shows us that t_3 is not independent from t_4 should t_4 receive an increase of flexibility, which also follows from the fact that we had $s(t_4,t_3) = 0$.

If it holds that $s(t_i,t_j) = c$, with $c > 0$ being the smallest positive slack value on any edge in S, and it also holds that $s(t_j,t_i) \geq c$, then t_i and t_j do not affect each other in both directions if the distributed total flexibility between these two tasks or there respective predecessors and successors is not greater than c. We can therefore create an STN S_l in which we allow a potential flexibility $flex_p^l(t_j)$ for each task $t_j \in T_l$ not to be greater than c and remove the edge (t_i,t_j).

Example 25. *We can apply this approach to Figure 5.5 and see that the smallest positive slack value is equal to 1, for both $s(t_3,t_1)$ and $s(t_3,t_4)$. We can therefore remove the corresponding edges (t_1,t_3) and (t_3,t_4) from the STN S. However, since S now no longer is connected, we create a seperate STN that contains only t_0 and t_3.*

By selecting the smallest positive value $c > 0$ in the original STN S, we can iteratively create seperate 'slack-free' STNs S_l by removing any edge (t_i,t_j) which has $s(t_i,t_j) \geq c$ and $s(t_j,t_i) \geq c$ from S_l. If this results in a task having no longer any predecessors and successors, the task will be removed and placed in a separate new STN S'_l. This results in the new STNs being critically connected, since each remaining edge (t_i,t_j) will now have $s(t_i,t_j) = 0$ and $s(t_j,t_i) = 0$, because if not, then either the value c was chosen too larger or an edge has not yet been removed.

This approach will form the basis of our Maximum Flexibility Algorithm. We give a more general pseudocode of the Maximum Flexibility Algorithm in Algorithm 1. We will use the notation $flex_p^*(t_i)$ to denote the remaining potential flexibility of t_i in S under the current iterative total flexibility distributed. If S is already critically connected, then $flex_p^*$ equals $flex_r$ for all $t_i \in T$.

The first For-loop executes the same steps as taken in Example 22, by assigning already some flexibility r^* if for a task $t_i \in T$ with for all $t_k,t_j \in T$ with $t_k \ll t_i \ll t_j$ it holds that $s(t_i,t_j) \geq r^*$ and $s(t_i,t_k) \geq r^*$ until one of these slack values becomes equal to 0. If this loop is completed, the algorithm starts to create seperate STNs S^* based on the $flex_p^*$ values of all task $t_i \in T$. In all these STNs S^* a maximum independent set is defined, where all tasks in this set receive the same raise in flexibility. After all $flex_p^*$ values are updated, the next iteration starts, and the algorithm stops when for every task $t_i \in T$ we have $flex_p^*(t_i) = 0$ and it returns the total assigned flexibility.

Theorem 1. *The Maximum Flexibility Algorithm correctly computes an optimal solution for the maximum flexibility problem.*

Proof. In order to prove Theorem 1, we have to show that every subproblem S_l created by the Maximum Flexibility Algorithm in iteration l is critically connected, which allows us to apply Corollary 2. Then we have to prove that the merge of all optimal solutions σ_l^* to these m subproblems S_l is again an optimal (and therefore also feasible) solution σ_S^* of S.

We will use the following five claims to prove these statements:

Claim 1: Every seperate S_l created by the Maximum Flexibility Algorithm is a critically connected STN and $flex(S_l)$ can be calculated by determining the maximum independent set in S_l.

Claim 2: If $t_i \notin T_l$ for some subproblem S_l, then $flex_p^*(t_i)$ is not affected by any solution σ_l for S_l

Algorithm 1 Maximum Flexibility Algorithm

Input: STN $S = (T, C)$
Output: $\sum_{t_i \in T} flex(t_i)$
Take $T^* := \emptyset$, $C^* := \emptyset$, $flex_p^*(t_i) := flex_p(t_i)$ and $flex(t_i) := 0 \; \forall t_i \in T$
Calculate for every edge (t_i, t_j) both $s(t_i, t_j)$ and $s(t_j, t_i)$
for all t_i **do**
  if for all t_j, t_k with $t_k \ll t_i \ll t_j$: $s(t_i, t_j) > 0$ and $s(t_i, t_k) > 0$ **then**
    $r^* = \min\{s(t_i, t_j), s(t_i, t_k)\}$
    $flex(t_i) = flex(t_i) + r^*$
    $flex_p^*(t_i) = flex_p(t_i) - r^*$
  end if
end for
while $\max_{t_i \in T}\{flex_p^*(t_i)\} > 0$ **do**
  $T^* = \{t_i \mid flex_p^*(t_i) = \max_{t_j \in T}\{flex_p^*(t_j)\}\} \cup \{z\}$
  $C^* = \{(t_i - t_j \leq \delta) \in C \mid t_i, t_j \in T^*\}$
  Create STN $S^* := (T^*, C^*)$
  $q := \infty$
  for all (t_i, t_j) with $t_i, t_j \in T^*$ **do**
    if $s(t_i, t_j) > 0$ and $s(t_j, t_i) > 0$ **then**
      $q = \min\{q, |s(t_i, t_j) - s(t_j, t_i)|\}$
      Remove the edge (t_i, t_j) from S^*
    end if
  end for
  $r = \min\{q, flex_p^*(t_i) - \max\{0, flex_p^*(t_j) \mid t_j \notin T^*\}\}$ for a $t_i \in T^*$
  Determine maximum independent set M of T^* and for all $t_i \in M$ set $flex(t_i) := flex(t_i) + r$
  Take $flex_p^*(t_i) := flex_p^*(t_i) - r \; \forall t_i \in T^*$
end while
Return: $\sum_{t_i \in T} flex(t_i)$

Claim 3: All subproblems S_k and S_l are independent for $k \neq l$, meaning that any chosen solution σ_k does not affect some other σ_l
Claim 4: The union of solutions $\bigcup_{l=1}^{m} \sigma_l$ of all the subproblems $S_1, S_2, \ldots, S_m$ is a feasible solution σ_S for S
Claim 5: The union $\sigma^* = \bigcup_{l=1}^{m} \sigma_l^*$ of optimal solutions σ_l^* for the subproblems S_l is again an optimal solution for S

The first claim will prove that all subproblems are critically connected and that the Maximum Flexibility Algorithm will find an optimal solution for these subproblems. The second claim is needed to prove the independency stated in Claim 3, which can then be used to prove the feasibility stated in Claim 4. The proof of Claim 5 will be based on the independency of the solutions for the subproblems and the feasibility of the merge of these solutions.

Ad Claim 1: We will here represent the value r used in every iteration with $flex_p^k$, which will denote the potential flexibility of each task in subproblem S_l.

In every such S_l, it holds for all t_i, t_j with $t_i \ll t_j$ that $s(t_i,t_j)$ and $s(t_j,t_i)$ are equal to 0, since if one of them is not 0, then $flex_p(t_i) \neq flex_p(t_j)$ and if both of them are unequal to 0, then the edge would have been removed. Therefore every path P in S_l is both est-critical and lst-critical, which meets the requirements of a critically connected STN. As a consequence of Corollary 2 we can determine a maximum independent set or maximum antichain to find the $flex(S_l)$ value.

Ad Claim 2: If $t_i \notin T_l$ of subproblem S_l, then there exists no est-critical and lst-critical path to any $t_j \in T_l$, because due to the absence of t_i in T_l it must hold that $flex_p^*(t_i) < flex_p^*(t_j)$ for all $t_j \in T_l$. This means that for any path $P = \{t_i, \ldots, t_x, t_y, \ldots, t_j\}$ we have an edge (t_x,t_y) such that $s(t_y,t_x) > flex_p^l$, due to the definition of r[6] in the Maximum Flexibility Algorithm, i.e., all $flex_p^l$ flexibility distributed to one or more $t_j \in T_l$ in S_l will be absorbed by all paths P and not affect $flex_p^*(t_i)$.

Ad Claim 3: For any $t_i \in T_k$ and $t_i \notin T_l$ we have due to Claim 1 that $flex_p^*(t_i)$ is not influenced by any solution σ_l of S_l. The same holds for any $t_i \notin T_k$ but with $t_i \in T_l$, where $flex_p^*(t_i)$ will not be influenced by any solution σ_k of S_k.

If $t_i \in T_k$ and $t_i \in T_l$, then we have in the union $S_k \cup S_l$ that $flex_p^{k\cup l}(t_i) = flex_p^k + flex_p^l$. Since a solution σ_k at most subtracts $flex_p^k$ from $flex_p^{k\cup l}(t_i)$, due to Corollary 1, we have that there remains at least $flex_p^l$ for t_i in S_l, indicating that σ_k does not affect σ_l by limiting its value $flex_p^l$. Therefore all subproblems S_k and S_l and their respective solutions can be considered independent.

Ad Claim 4: Due to the definition of r (or $flex_p^l$) in each iteration, it holds that $\sum_{l=1}^m flex_p^l(t_i) = flex_p(t_i)$ for all $t_i \in T$ with $S = (T,C)$. If we combine this with the result from Claim 1, then we can see that for every $t_i \in T$ we have that after each iteration it must hold that $flex_p^*(t_i) \geq 0$, since due to Claim 1 $flex_p^*(t_i)$ remains unaffected if t_i is not present in S_l, but if it is present in some S_k, then there will be at most $flex_p(t_i)$ flexibility distributed in all these subproblems together. Because $flex_p^*(t_i)$ is initially equal to $flex_p(t_i)$ at the beginning of the algorithm, it holds that at the end we have $flex_p^*(t_i) \geq 0$.

Therefore combining the solutions σ_l for all subproblems S_l, i.e., $\bigcup_{l=1}^m \sigma_l$, results for all $t_i \in T$ that $flex_p^*(t_i) \geq 0$, which proves feasibility of the solution[7].

Ad Claim 5: From Claim 4 it follows that $\bigcup_{l=1}^m \sigma_l$ is a feasible solution for S, meaning that $\sum_{l=1}^m flex(S_i) \leq flex(S)$. Since from Claim 3 it follows that all subproblems are independent and all S_l are derived from S, we can take $\bigcup_{l=1}^m S_l = S$. It then follows that, if we take $\sigma^*(S_l)$ to represent an optimal solution for S_l:

$$\sigma^*(\bigcup_{l=1}^{m} S_l) = \sigma^*(S) \tag{5.13}$$

[6] which we here represent as $flex_p^l$.

[7] Note that, as we already stated earlier, we can rebuild a solution σ_o, i.e., a time-interval assignment, of S from all independent $flex(t_i)$ values with $t_i \in T$.

Due to the independence of Claim 3 of all S_k and S_l, it then follows that:

$$\sigma^*(S_1) + \sigma^*(\bigcup_{l=2}^{m} S_l) = \sigma^*(S) \tag{5.14}$$

and

$$\sigma^*(S_1) + \sigma^*(S_2) + \sigma^*(\bigcup_{l=3}^{m} S_l) = \sigma^*(S) \tag{5.15}$$

resulting in

$$\sigma^*(S_1) + \ldots + \sigma^*(S_m) = \sigma^*(S) \tag{5.16}$$

This means that if $flex(S_l)$ represents the maximum flexibility in S_l, that:

$$flex(S_1) + \ldots + flex(S_m) = flex(S) \tag{5.17}$$

which we can rewrite to

$$\sum_{l=1}^{m} flex(S_l) = flex(S) \tag{5.18}$$

which proves our claim. □

In the Maximum Flexibility Algorithm, there is however one step in the final loop where a maximum independent set is determined. In the next section we will show an approach on how to determine maximum independent sets in polynomial time in task scheduling STNs.

5.3.3 Identifying a maximum independent set

While finding a maximum independent set of vertices is an NP-Hard problem if applied to an undirected graph, the problem is polynomial solvable if applied to a directed, acyclic (transitive) graph, as is shown by [10]. The algorithm as presented by [10] starts by creating a minimum flow (see [3] and [4]) in the graph.

Definition 19. *A task scheduling STN can represent a flow network if every edge (t_i,t_j) has a non-negative, real valued capacity $u(t_i,t_j)$ and a non-negative, real valued lower bound $l(t_i,t_j)$ and there are two nodes present that represent a source s which is connected to all t_i with $pre(t_i) = \emptyset$ by an edge (s,t_i) and a sink t that is connected to all t_j with $suc(t_j) = \emptyset$ by an edge (t_j,t). Furthermore it must hold for a flow function $f : T \times T \to \mathbb{R}$ and all nodes $t_i,t_j \in T$ with $t_i \ll t_j$ that:*

1. $l(t_i,t_j) \leq f(t_i,t_j) \leq u(t_i,t_j)$;
2. $f(t_i,t_j) = -f(t_j,t_i)$;
3. $\sum_{t_k \in pre(t_i)} f(t_k,t_i) = \sum_{t_l \in suc(t_j)} f(t_j,t_l)$

In a task scheduling STN, we can use z as a source node, since it does not represent an actual task and fits the requirements of the definition, but we will still have to add a sink node t. The goal of this specific minimum flow is to identify a minimum number of *flow paths* as specified by [10], i.e., positive flows of 1 from z to t, where each node is visited by at least one flow path. These flow paths represent a minimum number of chains to cover the STN S which we can use to identify a maximum antichain in S, because every chain contains at least one node that forced this chain to be created, otherwise if this is not the case, the chain would be redundant and we would not have a minimum number of chains to cover S.

The question is then how to ensure that every node receives at least one unit of flow, since we only allow capacities on the edges. We can do this in the following way:
For a node t_i if:

1. $|pre(t_i)| = 1$, add a lower bound of 1 to the edge pointing to t_i;
2. $|suc(t_i)| = 1$, add a lower bound of 1 to the edge pointing from t_i;
3. 1) and 2) do not hold, add a dummy node t_i'. Add an edge (t_i, t_i') with lower bound 1, and add and remove edges such that $suc(t_i') = suc(t_i)$ and $suc(t_i) = \{t_i'\}$.

We can take for each edge (t_i, t_j) the upper bound $u(t_i, t_j) = \infty$, since we are only searching for a minimum flow that satisfies the lower bounds.

We will present a minimum flow algorithm in which several notations are used that require some clarification. We define $d(t_i)$ to be the exact distance label to a task t_i, which stands for the minimum number of tasks that have to be passed starting in z to reach t_i. An edge (t_i, t_j) is called *admissable* if it holds that $d(t_j) = d(t_i) + 1$. We define $e(t_i)$ to be the flow *deficit* in a node t_i, indicating by how much the amount of outgoing flow exceeds the incoming flow, i.e., $e(t_i) = \sum_{t_k \in suc_{\ll}(t_i)} f(t_i, t_k) - \sum_{t_j \in pre_{\ll}(t_i)} f(t_j, t_i)$. A node is called *active* if $e(t_i) < 0$, i.e., more flow is entering the node t_i than leaving it. L is the set of active nodes.

The following minimum flow algorithm was presented by [4]. It was shown by [4] that this FIFO Preflow Minimum flow algorithm runs in $O(n^3)$ time and correctly computes a minimum flow. The requirement of a feasible flow can also be satisfied, since a feasible flow can be constructed within the complexity bound of this minimum flow algorithm of $O(n^3)$.

We will now show how a maximum independent set can be created by using the found minimum flow. We slightly adapt the algorithm of [10], since the networks we will be using are of a somewhat less complex structure, because we do not use upper bounds $u(t_i, t_j)$ on an edge (t_i, t_j).
To show what happens in this algorithm, we will explain the steps taken: Suppose we have a minimum flow f in our STN S, then $|f|$ is the size of the maximum independent set. We remove every edge without flow from S and we can start identifying flow paths carrying 1 flow, starting in z and ending in t. From z we choose an arbitrary edge and follow it to the task t_i at the end of this edge. We add t_i to the current path P and lower the flow on the edge(z, t_i) by 1. If the flow is then equal to 0, we can remove this edge (z, t_i) from the graph. From t_i we can again select an arbitrary edge going outward from t_i to for example a task t_j. We again add t_j to P and lower the flow on the edge (t_i, t_j) by 1 and remove it if its

Algorithm 2 FIFO Preflow Minimum flow algorithm

Input: A feasible flow f in the graph $G = (V,E)$, with $|V| = n$ and $|E| = m$
Create the residual network $G_f = (V_f, E_f)$ by removing every edge with $f(t_i,t_j) := l(t_i,t_j)$
Compute the exact distance labels d in G_f
if t has no exact distance label **then**
  f is a minimum flow
else
  $L := \emptyset$
  for every edge (t_i,t) **do**
    Set $f(t_i,t) := l(t_i,t)$
    if $e(t_i) < 0$ and $i \neq z$ **then**
      add t_i to the rear of L
    end if
  end for
  $d(t) := n$
  while $L \neq \emptyset$ **do**
    remove t_j from front of queue L
    slect the first edge (t_i,t_j) that enters t_j
    $B := 1$
    repeat
      if (t_i,t_j) is an admissable edge **then**
        pull $g = \min\{-e(t_j), r(t_i,t_j)\}$ units of flow from t_j to t_i
        if $t_i \notin L$ and $t_i \neq z$ and $t_i \neq t$ **then**
          add t_i to the rear of L
        end if
      end if
      if $e(t_j) < 0$ **then**
        if (t_i,t_j) is not the last arc that enters in t_j **then**
          select the next edge (t_i,t_j) that enters in t_j
        else
          $d(t_j) = \min\{d(t_i) \mid (t_i,t_j) \in E_f\} + 1$
          $B := 0$
        end if
      end if
    until $e(t_j) < 0$ or $B = 0$
    if $e(t_j) < 0$ or $B = 0$ **then**
      add t_j to the rear of L
    end if
  end while
end if

Algorithm 3 Maximum independent set algorithm

Input: A minimum flow f in $G = (V, E)$
for $j = 1 : |f|$ **do**
  Set $q := z$
  while $\exists f(q, t_i) > 0$ with $t_i \in V$ **do**
    $P_j = P_j \cup \{q\}$ and set $f(q, t_i) = f(q, t_i) - 1$
    Set $q := t_i$
  end while
  Set $x_j := q$
end for
Remove all t_j from all P_i if t_j occurs in multiple P_i
while $\exists x_i, x_j$ with $x_i \prec x_j$ **do**
  Set x_i equal to the latest previous node with $f(x_i) = 1$ in P_i
end while
Output: Maximum independent set $\{x_1, x_2, \ldots, x_{|f|}\}$

flow is equal to 0. This process can be repeated until we reach the sink node t. If there are outgoing edges from z left in the network, we create a new path P in the same way until no more paths can be created.

We can select the following task on the path arbitrarily, since in a feasible (minimum) flow we have the guarantee that in every node there is a flow balance, i.e., the total incoming flow equals the total outgoing flow.

Now that we have a collection of paths, we have to create a maximum independent set from these paths. From every path one independent node has to be selected that does not occur in any other path. We therefore can remove all t_j that appear in more than one of the created paths P_i. Creating the maximum independent set can be done by selecting for each path the final node not equal to t. Check if there is a currently selected node t_i that points in the original network S to any other selected node t_j in a different path. Deselect t_i and select the predecessor of t_i in that path. Repeat until there are no conflicts remaining and the result is a maximum selection of independent tasks.

5.4 Maximum Flexibility Algorithm Complexity

To determine the maximum flexibility in a directed, acyclic STN we can now apply all of the algorithms presented in the previous section. We begin by applying our Maximum Flexibility Algorithm to this STN, which takes at most n iterations in the while-loop (since there are at most n different $flex_p$ values), but applies the other algorithms in every iteration. The while-loop starts with calculating the slack for each edge and giving flexibility if possible and then continues with defining which tasks will receive flexibility, which is done by the minimum flow and maximum independent set algorithm.

With the minimum flow algorithm applied to the current network, a minimum flow is produced, which can be used to identify a maximum independent set. All tasks in this

maximum independent set will receive flexibility r, which allows us to continue to the next iteration of the Maximum Flexibility Algorithm. We first decide which edges can be removed based on the present slack values, after which we can determine a maximum independent set in the remaining network. We again add r flexibility to the tasks in this set and repeat the process until the Maximum Flexibility Algorithm terminates. The result is an iteratively calculated maximum flexibility solution.

Since the Maximum Flexibility Algorithm takes at most n iterations, the slack calculation in each iteration take $O(m)$, the minimum flow algorithm runs in $O(n^3)$ and the maximum independent set algorithm runs in $O(|f|m+n^2) = O(n^3)$. If we combine all these steps, we will get a polynomial running time of $O(n)O(n^3) = O(n^4)$ to find the maximum flexibility in a task scheduling STN, which is however not more efficient than the complexity of the LP formulation. We can see that the critical parts of this algorithm are the calculation of a minimum flow and then determining a maximum independent set, based on this flow.

However, so far we have only tried to maximize the total flexibility in a task scheduling STN, but we have not looked at the consequences of doing so. Depending on the STP, the flexibility is distributed over the tasks, but we can ask ourselves if this distribution is considered 'fair'?

Chapter 6

Egalitarian flexibility

Determining maximally flexible open schedules has now been shown to be solvable in polynomial time by either the LP formulation 5.10 or our new alternative Maximum Flexibility Algorithm 1. In Example 15 we have seen that in this maximally flexible open schedule only the tasks t_3, t_5, t_6, t_9 and t_{10} received flexibility, while the other tasks were forced to start at a single time-point. Can we consider this to be a 'fair' distribution of flexibility?

The problem that arises here is that a 'fair distribution' can be related to a number of factors. It could be fair to give each task the same flexibility as any other task, but then again, some agents control more tasks than other agents, resulting in the agents with more tasks than the others to be heavily favored in their total received flexibility.

If we want to create a 'fair' schedule, there are multiple options we have in this case, all depending on our interpretation of a 'fair' schedule. The three main types of a 'fair' schedule we use are the following:

1. All tasks receive an equal amount of flexibility, regardless of their controlling agent
2. All agents receive an equal total amount of flexibility, regardless of their number of tasks controlled
3. All agents receive an equal average amount of flexibility per task controlled

The first type ignores agents and treats all tasks in an equal way, resulting in for all i, j that $flex(t_i) = flex(t_j)$. We refer to this first option as *egalitarian task flexibility* (ETF). This egalitarian task flexibility can be applied in cases where all tasks are considered to be of equal importance and where all tasks have a certain degree of uncertainty regarding their potential starting time. Agents are of no importance in this case.

The second option now focuses on the agents instead of the tasks. All agents are considered to be equals in this case, independent of how many tasks an agent controls. The objective is to give all agents an equal portion of flexibility, which means that for all A_k, A_l it holds that $\sum_{t_i \in T_k} flex(t_i) = \sum_{t_j \in T_l} flex(t_j)$. This type of egalitarian flexibility is referred to as *egalitarian agent flexibility* (EAF). The difference between EAF and ETF is the importance of agents and tasks, where ETF only considers tasks, EAF only takes agents into account[1]. EAF is particularly usable in cases where all parties involved are considered equal

[1] Although of course precedence relations in the STN are still taken into account when creating an EAF.

and for this reason also have to receive exactly the same total flexibility.

The third option of a 'fair' schedule shows a lot of resemblance with the second option, since again agents are considered to be more important than tasks. However, in this case, we also consider tasks although, apart from the precedence relations, we only look at the size of the set T_i of agents A_i. To achieve an average amount of flexibility for every agent based on their number of tasks, we want for all agents A_k, A_l that $\sum_{t_i \in T_k} \frac{flex(t_i)}{|T_k|} = \sum_{t_j \in T_l} \frac{flex(t_j)}{|T_l|}$. We refer to this type of egalitarian flexibility as *egalitarian average agent flexibility* (EAAF), which is applicable in cases where all parties involved measure their importance by comparing the number of tasks controlled by each party.

In the coming sections we will discuss the three mentioned types of egalitarian flexibility in more detail and we will present solution methods to obtain optimal schedules in each case.

6.1 Egalitarian task flexibility

We can formalize the *egalitarian task flexibility* problem by the following definition:

Definition 20. *A solution σ_o of a task scheduling STN $S = (T, C)$ is referred to as an* egalitarian task flexibility *solution (ETF) if $\sum_{i=1}^{n} flex(t_i)$ is maximal under the condition that $flex(t_i) = flex(t_j)$ for all $t_i, t_j \in T$.*

An ETF solution of a task scheduling STN ignores the agents that are present in an STN and only ensures that all tasks receive an equal portion of flexibility. This problem is solvable in polynomial time, since we can construct an LP formulation due to the presence of a linear objective function with linear constraints.

$$\begin{aligned} &\max \sum_{t_i \in T} (n_i - m_i) \\ &\text{s.t. } \forall t_i : && m_i \leq n_i \\ & && n_i \leq lst(t_i) \\ & && m_i \geq est(t_i) \\ &\text{s.t. } \forall t_i \ll t_j, && n_i + p_i \leq m_j \\ &\text{s.t. } \forall t_i, t_j \in T && n_i - m_i = n_j - m_j \end{aligned} \tag{6.1}$$

Since we are still trying to maximize the flexibility in the task scheduling STN $S = (T, C)$, we can use the same objective function as used in LP formulation 5.10. The only difference is the addition of the last constraint, which ensures that for all $t_i, t_j \in T$ we have $flex(t_i) = flex(t_j)$.

Example 26. *If we review the NedTrain case, we can see that in a maximally flexible open schedule, the total flexibility is equal to 155, as is shown in Example 15, which is divided over 5 tasks. Since this is hardly a fair distribution if we pursue an ETF solution, we have to apply the LP formulation 6.1, which results in each task t_i receiving $flex(t_i) = 5$, since both the chains $\{t_1, t_2, t_5, t_{13}\}$ and $\{t_1, t_2, t_6, t_7, t_{13}\}$ are the limiting factors in this STN. Assigning*

any more flexibility than $flex(t_i) = 5$ to all tasks would result in either the precedence relations being violated or due dates not met.

This results in the total flexibility $flex(S) = 65$, which is considerably lower than the 155 total flexibility without the ETF constraint.

Although we have presented an LP formulation 6.1, the question arises if we can do better, i.e., compute it with lower computational complexity?

The answer to this question is at the moment unfortunately no, however, we have created an alternative $O(n^4)$ algorithm, which is based on iteratively increasing the flexibility assigned to all tasks until for one or more tasks $t_i \in T$ we reach $flex_r(t_i) = 0$. The idea behind this algorithm is to determine in each iteration the effect of a very small raise r_3 of the flexibility $flex$ of all tasks on the present slack values $s > 0$ and $flex_r$ values[2]. Once we know the effect of this raise, we determine for all slack values $s > 0$ and $flex_r$ values which one reaches 0 first if we apply r_3 a certain number of times, resulting in a raise of q. We then apply this raise, i.e., for all $t_i \in T$ we take $flex(t_i) = flex(t_i) + q$ and repeat the whole procedure. If we analyse this algorithm, we can see we start with the $est()$ and $lst()$ calculations, which can be done in $O(m+n)$ time, including the $flex_r$ determination. In the while-loop we try to remove all slack values and in each iteration at least one is set equal to 0 and will never become positive again. We therefore have to do at most m iterations in this while-loop. Calculating all slack values can be done in $O(m)$ time which has to be done at most 2 times, which includes the s^* calculations. To determine q it takes $O(m+n)$ values to take the minimum of, which gives us a total of $O(m+n)$ calculations in each iteration. The total complexity then adds up to $O(m)O(m+n) = O(m^2+mn) = O(n^4)$.

Theorem 2. *The ETF Algorithm computes a feasible maximal ETF solution.*

Proof of correctness of the ETF Algorithm. Since we start with $flex(t_i) = 0$ for all tasks $t_i \in T$ (which is a feasible schedule) and increase for all tasks in each iteration their flexibility by a value q, an egalitarian solution is ensured. The only thing remaining we have to prove now is that the final solution is still feasible (i.e., does not violate any constraints) and is maximal.

Claim 1: The solution found by the ETF algorithm with $flex(t_i) = flex(t_j)$ for all $t_i, t_j \in T$ is feasible.
Claim 2: The ETF algorithm terminates after a finite number of iterations.
Claim 3: The ETF algorithm finds a maximally flexible solution with $flex(t_i) = flex(t_j)$ for all $t_i, t_j \in T$.

Ad Claim 1: The only way to break a constraint is after we add a value q flexibility to all tasks which is too large, i.e., for some t_i we get $flex_r(t_i) < 0$.

We know that $q \leq \frac{flex_r(t_i)}{\Delta flex_p^*(t_i)}$, where $\Delta flex_r(t_i)$ denotes the number of times q will be subtracted from $flex_r(t_i)$ if the flexibility of each task is increased by q. This results in $\forall t_i \in T$ we have that if $flex(t_i) = flex(t_i) + q$, that

$$flex_r(t_i) = flex_r(t_i) - q \times \Delta flex_p^*(t_i)$$

[2]In the context of this algorithm, we define $flex_r(t_i)$ to be the flexibility that can still be assigned to t_i without violating any constraint.

Algorithm 4 Egalitarian Task Flexibility Algorithm

Input: STN $S = (T, C)$, where $|T| = n$
Output: $flex(t_i)$ for some $t_i \in T$
Calculate $est(t_i)$ and $lst(t_i)$ for all $t_i \in T$
Calculate $flex_r(t_i)$ for all $t_i \in T$
while $\min_{t_i \in T}\{flex_r(t_i)\} > 0$ **do**
  Calculate $s(t_i,t_j)$ and $s(t_j,t_i)$ for all $t_i \ll t_j$ with $t_i,t_j \in T$
  $r_1 = \min_{t_i \in T}\{flex_r(t_i)\}$ and $r_2 = \min_{s(t_i,t_j)>0}\{s(t_i,t_j)\}$ and take $r_3 = \frac{\min\{r_1,r_2\}}{n}$
  $\forall t_i \in T$ set $flex^*(t_i) = flex(t_i) + r_3$
  Calculate $est^*(t_i) := \max(\{est^*(t_j) + p_j + flex(t_j) \mid t_j \in pre_{\ll}(t_i)\} \cup r_i)\ \forall t_i \in T$
  Calculate $lst^*(t_i) := \min(\{lst^*(t_j) - p_i - flex(t_j) \mid t_j \in suc_{\ll}(t_i)\} \cup d_i - p_i)\ \forall t_i \in T$
  Calculate $flex_r^*(t_i)$ for all $t_i \in T$
  if $s(t_i,t_j) > 0$ **then**
    Calculate $s^*(t_i,t_j)$ based on $est^*(t_i)$, $lst^*(t_i)$
    $\Delta s(t_i,t_j) = \frac{s(t_i,t_j) - s^*(t_i,t_j)}{r_3}$
  end if
  $q := \infty$
  for all $t_i \in T$ **do**
    $\Delta flex_r(t_i) = \frac{flex_r(t_i) - flex_r^*(t_i)}{r_3}$
    $q = \min\{q, \frac{flex_r(t_i)}{\Delta flex_r(t_i)}\}$
  end for
  for all $\Delta s(t_i,t_j) > 0$ **do**
    $q = \min\{q, \frac{s(t_i,t_j)}{\Delta s(t_i,t_j)}\}$
  end for
  $\forall t_i \in T$: $flex(t_i) = flex(t_i) + q$.
  Calculate $flex_r(t_i)$ for all $t_i \in T$
end while
Return: $flex(t_i)$ for some $t_i \in T$

where

$$q \times \Delta flex_r(t_i) \leq \frac{flex_r(t_i)}{\Delta flex_r(t_i)} \times \Delta flex_r(t_i) = flex_r(t_i)$$

This combined gives then:

$$flex_r(t_i) = flex_r(t_i) - q \times \Delta flex_r(t_i) \geq flex_r(t_i) - flex_r(t_i) = 0$$

i.e., $flex_r(t_i) \geq 0$ if $flex(t_i) = flex(t_i) + q$, indicating that after every iteration the solution will still be feasible.

Ad Claim 2: In every iteration the value q is chosen in such a way that after updating every $flex$, either $flex_r(t_i) = 0$ for some $t_i \in T$ or some $s(t_i,t_j) > 0$ is converted to $s(t_i,t_j) = 0$. Since a slack value s will never become positive again once it has been reduced to 0, at most $2m$ iterations can be used to reduce (if necessary) all the slack values to 0, which means

that in at most $2m+1$ iterations at least one $flex_r(t_i)$ value will be 0, since $q > 0$ in each iteration.

Ad Claim 3: The maximality of the solution follows from the fact that for all $t_i, t_j \in T$ we have in each iteration that $flex(t_i) = flex(t_j)$ and the algorithm only terminates once $flex_r(t_i) = 0$ for some $t_i \in T$, indicating that $flex(t_i)$ has reached its maximum value. We can conlude from this that the ETF Algorithm finds a maximal feasible solution. □

An ETF solution can be a 'fair' solution if tasks are considered to be more important than agents controlling them. But in the case that agents are more important, we need a different approach in order to create schedules that are considered 'fair' under these circumstances, which is why we will introduce egalitarian agent flexibility.

6.2 Egalitarian agent flexibility

In the case that agents are more important than tasks, and we want to equally distribute flexibility over the tasks such that every agent gets the same total flexibility, we want to solve the *egalitarian agent flexibility* (EAF) problem. The main difference between the ETF and the EAF problem is that in the ETF problem we want to give all tasks the same flexibility, while in the EAF problem we want to give all agents the same total flexibility.

Example 27. *If we review Example 26, we see that if every task $t_i \in T$ has $flex(t_i) = 5$ we have a (maximal) ETF solution, but if we compare the total flexibility given to each agent, we see that A_1 has a total of 20, A_2 also has a total of 20 and A_3 has a total of 25 flexibility, which shows that A_3 is somewhat favored over the other two agents.*

To formalize the EAF problem, we state the following definition:

Definition 21. *A solution σ_o of a task scheduling STN $S = (T, C)$ is referred to as an* egalitarian agent flexibility *solution (EAF) if $\sum_{i=1}^{n} flex(t_i)$ is maximal under the condition that $\sum_{t_i \in T_k} flex(t_i) = \sum_{t_j \in T_l} flex(t_j)$ for every $A_k, A_l \in \mathcal{A}$.*

As we can see, we still have the same objective function as in the ETF problem, since we still want to maximize the total flexibility in σ_o, but in this case the condition under which this is done is changed, by comparing the total flexibility given to an agent A_k by summing over the flexibility values of its tasks and comparing this sum to the sum of all other agents $A_l \in \mathcal{A}$.

Since the objective function is linear, and the only constraint that is changed in comparison to the ETF formulation is also linear, we again can create an LP formulation to represent

the EAF problem:

$$\begin{aligned} &\max \sum_{t_i \in T} (n_i - m_i) \\ &\text{s.t. } \forall t_i : && m_i \leq n_i \\ & && n_i \leq lst(t_i) \\ & && m_i \geq est(t_i) \\ &\text{s.t. } \forall t_i \ll t_j, && n_i + p_i \leq m_j \\ &\text{s.t. } \forall A_k, A_l \in \mathcal{A} && \sum_{t_i \in T_k} flex(t_i) = \sum_{t_j \in T_l} flex(t_j) \end{aligned} \tag{6.2}$$

This LP formulation gives us the guarantee that the problem is solvable in polynomial time.

Example 28. *We can create an EAF solution for the NedTrain case, by applying the LP formulation 6.2 to the corresponding STN. As the reader can verify, an optimal EAF solution is:* $flex(t_3) = 35$, $flex(t_5) = 20$, $flex(t_6) = 25$, $flex(t_8) = 10$, $flex(t_9) = 10$ *and* $flex(t_{10}) = 35$*, where all the other tasks have no flexibility. Summing the values over the agents shows us that every agent receives a total of 45 flexibility, resulting in a schedule that contains a total of 135 flexibility. If we compare this to the maximally flexible schedule created in Example 15, we see that in this case the EAF solution does not lose a large amount of flexibility due to the presence of the EAF added constraint.*

If we however review the EAF solution found in Example 28, we can see that although each agent received the same total flexibility, agent A_3 has more tasks to control than the other agents, leaving less flexibility to be distributed to each of his tasks. This difference can become even larger when the differences in the number of controlled tasks become even larger. This is what we will take into account in the next section.

6.3 Egalitarian average agent flexibility

An EAF solution is a good start to ensure that all agents are considered equals. However, are two agents A_i and A_j with for example $|T_i| = 2$ and $|T_j| = 10$ also equals, and should they both receive the same total flexibility? If for example both agents receive a total of 10 flexibility, then A_i has 5 flexibility per task, while A_j has only 1 flexibility for each task. This could be considered to be 'unfair'.

A problem formulation that deals with this type of problems is the *egalitarian average agent flexibility* (EAAF) problem, and ensures that all agents receive the same average flexibility per task. More formal:

Definition 22. *A solution* σ_o *of a task scheduling STN* $S = (T, C)$ *is referred to as an* egalitarian average agent flexibility *solution (EAAF) if* $\sum_{i=1}^{n} flex(t_i)$ *is maximal under the condition that* $\sum_{t_i \in T_k} \frac{flex(t_i)}{|T_k|} = \sum_{t_j \in T_l} \frac{flex(t_j)}{|T_l|}$ *for every* $A_k, A_l \in \mathcal{A}$.

One might look at the type of solution we require the EAAF problem to produce and note that in fact the ETF problem satisfies the added EAAF constraint. The EAAF and ETF

problem are however not equivalent, since the solution space of the ETF problem can be (much) smaller than that of the EAAF problem. Demanding that all tasks have the same flexibility as in the ETF formulation automatically fulfills the EAAF requirement, but an EAAF solution does not have to fulfill the ETF requirement. We therefore have, if we denote the solution space by Sol, that: $Sol(ETF) \subseteq Sol(EAAF)$.

Like the former two egalitarian problem formulations, we can also formulate an LP problem to represent the EAAF problem:

$$\begin{aligned} &\max \sum_{t_i \in T} (n_i - m_i) \\ &\text{s.t. } \forall t_i : && m_i \leq n_i \\ & && n_i \leq lst(t_i) \\ & && m_i \geq est(t_i) \\ &\text{s.t. } \forall t_i \ll t_j, && n_i + p_i \leq m_j \\ &\text{s.t. } \forall A_k, A_l \in \mathcal{A} && \sum_{t_i \in T_k} \frac{flex(t_i)}{|T_k|} = \sum_{t_j \in T_l} \frac{flex(t_j)}{|T_l|} \end{aligned} \tag{6.3}$$

If we compare this LP formulation with the LP 6.2, we see that there is only a subtle, but important, change made to the last contraint. By dividing over the size of the set T_k and T_l of respectively the agents A_k and A_l, we divide over the number of tasks they have, resulting in equality of the average flexibility per task that is given. Note that although the average flexibility values per task per agent are equal, this is not a guarantee that giving each task this average flexibility always results in a feasible schedule. It is possible that in an EAAF solution the majority of the total flexibility of an agent A_l is concentrated in a task $t_i \in T_k$, where other tasks $t_j \in T_k$ can receive hardly any flexibility, but on an average basis, all agents are now equals.

Example 29. *If we review the EAF solution from Example 28, we see that with respect to the average flexibility per agent, A_1 has* 11.25*, A_2 has also 11.25 and A_3 has only 9. Since A_3 is at a disadvantage when compared to the other agents, we can use the LP formulation 6.3 to create an EAAF solution. The result is then: $flex(t_3) = 35$, $flex(t_5) = 20$, $flex(t_6) = 25$, $flex(t_8) = 10$, $flex(t_9) = 21.25$ and $flex(t_{10}) = 35$.*

If we compare this to Example 28, we see that only t_9 has an increase of flexibility, from 10 to 21.25. Calculating the average flexibility values results in every agent having 11.25 *average flexibility. This results in a total of* 146.25 *flexibility, which is quite near the total flexibility in Example 15.*

With our three egalitarian flexibility formulations defined, we can also give an answer to the last research question: *How can we add independency restrictions to each workcrew such that their schedules are just as adaptable as the schedules of the other workcrews?*, i.e., how to create a decoupling such that there is egalitarian flexibility.

The global answer to this question would be to apply one of our egalitarian flexibility (LP) formulations and from its solution derive a decoupling $\{S_i\}_{i=1}^{k}$. Since every egalitarian flexibility problem under its current LP formulation delivers an optimal solution, the corresponding decoupling will automatically also be optimal, i.e., maximally flexible, as was

shown in Proposition 6. However, we cannot advise the use of one egalitarian problem formulation over the other, since this entirely depends on the situation and preferences of the controlling agents. All egalitarian problem formulations however are shown to be solvable in polynomial time, either by the use of the LP formulations, or in the ETF case by the use or our own new alternative ETF Algorithm 4.

Chapter 7

Conclusion and Discussion

In this thesis we have stated the special properties of a task scheduling Simple Temporal Problem $S = (T, C)$, i.e., $S \in \text{STP}_{\prec}$, when compared to an arbitrary STP. We have shown how to represent an STP S by a Simple Temporal Network and from this derive a schedule based on the $est()$ and $lst()$ values, which answered the first research question. The special properties of task scheduling STPs allowed us to compute arbitrary schedules σ for the task scheduling STP in $O(m+n)$, by only using the first column and first row in the temporal distance matrix D, which is a significant improvement over the $O(n^3)$ time required for arbitrary STPs which require the computation of the entire matrix D. The second research question was answered by this solution, resulting in the answer to the first main question. The improvement was based on exploiting the strict precedence relations present in a task scheduling STP, which allowed us to compute the earliest starting times $est()$ and latest starting times $lst()$ more efficiently.

With the introduction of open schedules σ_o, a schedule representation has been introduced to represent starting time-intervals for all tasks in T. Based on the open schedules, temporal decouplings could be derived, which ensured that each agent would receive its own independent STP to derive its schedule from, by adding some inter-agent constraints to the constraint sets of the involved agents, which answers the third research question. However, arbitrary (open) schedules or decouplings were shown not to be always in the best interest of the agents, which is why we introduced a new flexibility metric $flex$ to give an exact representation of the total flexibility present in a created open schedule, which allowed us to create maximally flexible open schedules.

We have given an LP formulation that represents the maximum flexibility problem, which can be solved in polynomial time. Apart from this, we have also formulated a new alternative algorithm, the Maximum Flexibility Algorithm, which is able to calculate a maximally flexible open schedule in $O(n^4)$ time. The open schedules that can be derived from the solutions of both the LP formulation and the Maximum Flexibility Algorithm contain the maximum amount of flexibility that is possible according to our new flex-metric, which answers our fourth research question, since we can apply any of these two methods to acquire the required schedule.

Moreover, we have given a proof that a decoupling derived from a maximally flexible open schedule is an optimal decoupling as well, based on our $flex$ metric, i.e., the sum of

the flexibility values in the decoupled subproblems equal the sum of flexibility values in the original STP. We have therefore shown that there is no loss of flexiblity if we create a maximally flexible decoupling, when compared to a maximally flexible open schedule without the optimal decoupling constraints, which answers our fifth research question.

Maximally flexible open schedules are schedules that contain as much total flexibility as possible, however, this can be schedules that heavily favor some tasks or agents by assigning a lot of flexibility to them and leaving only little flexibility for other tasks or agents. To prevent this, we have introduced egalitarian flexibility schedules, which we divided into three categories: egalitarian task flexibility, egalitarian agent flexibility and egalitarian average agent flexibility. The first one ensures that all tasks receive the same amount of flexibility, where in the second case all agents receive the same total flexibility and the third case ensure that all agent receive an equal average amount of flexibility per task. In all cases we have presented an LP formulation to represent the problem, which shows that these problems are solvable in polynomial time, and in the ETF case we have also shown a new alternative Egalitarian Task Flexibility Algorithm, which computes an optimal solution in $O(n^4)$ time. By introducing LP formulations for all problems and also the ETF Algorithm in case of the ETF problem, we have shown how to create egalitarian flexibility schedules for any of these three problems, which answers the final research question and thereby also the third main question, since from the resulting solutions, a decoupling can be derived.

Future research can be done in the area of online schedules, since it would be interesting to see how schedules can be adapted once set in motion. If during the execution of a schedule a task $t_i \in T_k$ does not 'need' its flexibility, then it may be possible that the flexibility of t_i can be transferred to a $t_j \in T_k$ with $t_i \ll t_j$. Further research in this area of online scheduling could improve the flexibility that can be used during the execution of a schedule.

Other future work can be done by increasing the efficiency of the Maximum Flexibility Algorithm or any of its sub-algorithms to improve the current computation time of $O(n^4)$, by improving the critical parts of the algorithm, i.e., the minimum flow algorithm and the maximum independent set determination.

Since our formulations allow for the inclusion of a *relative release date* (i.e., delays) between two tasks, it would be interesting to see if this inclusion would affect the current algorithms and if it is also possible to include *relative due dates* between two tasks. Extensions can also be made by incorporating task weights related to the flexibility received, since in [10] it was shown that node weights could be included in the calculations and should not increase the complexity of the algorithm. Other research could be done in the direction of the creation of more efficient algorithms for the EAF and the EAAF case.

Bibliography

[1] M Aloulou and M.C. Portmann. An efficient proactive-reactive scheduling approach to hedge against shop floor disturbances. In *Multidiscplinary scheduling: theory and applications*, pages 223–246, 2005.

[2] A. Cestal, A. Oddi, and S.F. Smith. Profile based algorithms to solve multiple capacitated metric scheduling problems. In *In Proceedings of the 4th international conference on artificial intelligence planning systems*, pages 214–223, 1998.

[3] Laura Ciupala and Eleonor Ciurea. About preflow algorithms for the minimum flow problem. 2008.

[4] Eleonor Ciurea and Laura Ciupala. Algorithms for minimum flows. page ?, 2001.

[5] R. Dechter. *Constraint processing*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann Publishers, 2003.

[6] Rina Dechter, Itay Meiri, and Judea Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[7] Leon Endhoven, Tomas Klos, and Cees Witteveen. Maximal flexibility and optimal decoupling in task scheduling problems. 2012.

[8] Luke Hunsberger. Algorithms for a temporal decoupling problem in multi-agent planning. In *Proceedings AAAI*, 2002.

[9] Luke Hunsberger. Group decision making and temporal reasoning. 2002.

[10] Dimitrios and Kagarisi. Maximum independent sets on transitive graphs and their applications in testing and cad. page ?, 1997.

[11] Léon R. Planken, Mathijs de Weerdt, and Cees Witteveen. Optimal temporal decoupling in multiagent systems. In *Proceedings AAMAS*, pages 789–796, 2010.