TECHNISCHE UNIVERSITÄT WIEN

DIPLOMARBEIT

Methods for Intraday Scheduling in Particle Therapy

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Masterstudium Technische Mathematik

eingereicht von

Michael Höfler
Matrikelnummer 01054523

ausgeführt am Institut für Logic and Computation
der Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer/in: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Günther R. Raidl
Mitwirkung: Projektass. Dipl.-Ing. Johannes Maschler, BSc
Univ.-Ass. Dipl.-Ing. Martin Riedler, BSc

Wien, 01.10.2018

Michael Höfler	Günther R. Raidl

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Wien, am ……

……
Michael Höfler

Einverständniserklärung zur Plagiatsprüfung

Ich nehme zur Kenntnis, dass die vorgelegte Arbeit mit geeigneten und dem derzeitigen Stand der Technik entsprechenden Mitteln (Plagiat-Erkennungssoftware) elektronisch-technisch überprüft wird. Dies stellt einerseits sicher, dass bei der Erstellung der vorgelegten Arbeit die hohen Qualitätsvorgaben im Rahmen der ausgegebenen der an der TU Wien geltenden Regeln zur Sicherung guter wissenschaftlicher Praxis - „Code of Conduct“ (Mitteilungsblatt 2007, 26. Stück, Nr. 257 idgF.) an der TU Wien eingehalten wurden. Zum anderen werden durch einen Abgleich mit anderen studentischen Abschlussarbeiten Verletzungen meines persönlichen Urheberrechts vermieden.

Wien, am …….

……
Michael Höfler

Acknowledgements

First, I would like to thank my thesis advisors Günther Raidl, Johannes Maschler and Martin Riedler for their great supervision and support.

I also want to express my gratitude to EBG MedAustron GmbH[1], 2700 Wiener Neustadt, Austria, for their cooperation and for partially funding this thesis.

Last but not least, I want to thank my whole family for their loving support throughout my entire life and my partner Asmaa for always encouraging me to reach my goals.

[1] `https://www.medaustron.at`

Abstract

We consider the *Intraday Particle Therapy Patient Scheduling Problem* (*I-PTPSP*), a patient scheduling problem that arises in facilities specialized on cancer treatment with synchrotrons, a device for particle therapy treatments. This problem has a time horizon of a single day and already assigned starting times for the pending treatments from an earlier created weekly or monthly schedule. If unforeseen circumstances occur, these predefined schedules need to be adjusted to adapt to the new situation. Finding these new schedules quickly is essential to limit any negative impacts on the waiting times of the patients or the running costs of the facility. The *I-PTPSP* models this situation.

A specialty of the *I-PTPSP* is the way how the treatments are processed. The synchrotron can serve multiple treatment rooms alternatingly. The aim is to find schedules of treatments that occupy the different treatment rooms in such a way that avoids idle times of the synchrotron to increase the throughput of the facility. Beside the synchrotron and the treatment rooms, other resources are relevant and need to be considered.

In this work, we tackle the *I-PTPSP* with multiple variants of a greedy construction heuristic first. Then, we solve it with a Branch and Bound approach that incorporates some of the ideas obtained during the development of the construction heuristic. In addition, we examine and develop several traversal strategies of the Branch and Bound node tree, as well as, different techniques to obtain lower bounds for the partial solutions. Furthermore, a constraint programming (CP) model solved by a state-of-the-art CP solver is developed. Finally, we compare our solution to this reference implementation.

For that reason, we simulate different scenarios of various sizes that resemble real world use cases like a delay in a treatment room or a suddenly needed maintenance of a medical device. The so obtained problem instances are used to compare the performance of our three approaches. While all three approaches perform almost equally well on the smaller instances, the Branch and Bound approach clearly outperforms the other two on the medium to larger problem inputs.

Kurzfassung

In dieser Arbeit behandeln wir das *Intraday Particle Therapy Patient Scheduling Problem* (*I-PTPSP*), ein Planungsproblem für Behandlungstermine von Patienten, das in medizinischen Einrichtungen für Krebstherapien mit Synchrotrons, einem speziellen Typ von Teilchenbeschleunigern für die Strahlentherapie, auftritt. Das Problem beschreibt bevorstehende Behandlungen mit bereits zugewiesenen Terminen auf einen Zeithorizont von maximal einem Tag. Die Problematik liegt darin, dass durch unvorhergesehene Ereignisse diese Behandlungen nicht mehr zu den vorgesehenen Zeitpunkten durchgeführt werden können und deshalb der bestehende Zeitplan an die neue Situation angepasst werden muss. Dieser neue Zeitplan muss möglichst schnell erstellt werden, um negative Auswirkungen auf beispielsweise die Wartezeiten und laufenden Kosten der Einrichtung zu begrenzen. Das *I-PTPSP* modelliert diese Situation.

Eine Besonderheit des *I-PTPSP* liegt in der Art und Weise wie die Behandlungen durchgeführt werden. Das Synchrotron kann mehrere Behandlungsräume abwechselnd hintereinander bedienen. Das Ziel besteht nun darin, solche Belegungspläne zu finden, in denen Leerlaufzeiten des Beschleunigers möglichst vermieden werden. Neben dem Strahl und den Behandlungsräumen gibt es weitere relevante Ressourcen, die berücksichtigt werden müssen.

In dieser Arbeit betrachten wir zunächst verschiedene Varianten von Greedy Konstruktionsheuristiken, um erste Lösungen für das Problem zu erhalten. Die hierfür verwendeten Ansätze nutzen wir anschließend, um ein Branch and Bound (BAB) Verfahren zu entwickeln. Hierbei untersuchen wir sowohl verschiedene Auswahlstrategien der Knoten aus dem BAB-Baum, als auch diverse Techniken um untere Schranken für die Teillösungen zu berechnen. Weiters wird als Referenz ein Constraint Programming (CP) Modell erstellt und mithilfe eines CP Solvers gelöst. Schlussendlich vergleichen wir die Ergebnisse unserer Verfahren mit denen dieser Referenzimplementierung.

Hierfür erstellen wir verschiedene Szenarien unterschiedlicher Größe, welche reale Anwendungsfälle, wie eine Verzögerung in einem Behandlungsraum oder eine dringend benötigte Wartung eines medizinischen Gerätes, simulieren. Die so erstellten Problem-Instanzen werden anschließend verwendet, um die Verfahren miteinander zu vergleichen. Während alle drei Methoden ähnlich gute Ergebnisse bei Instanzen kleinerer Größe liefern, zeigt sich, dass der Branch and Bound Algorithmus die anderen beiden Ansätzen bereits bei Problemen ab mittlerer Größe übertrifft.

Contents

1. Introduction

Depending on the year, cancer is the first or second leading cause of death [79, 72]. There were 14.1 million new cancer cases, 8.2 million cancer deaths and 32.6 million people living with cancer within 5 years of diagnosis (excluding non-melanoma skin cancer) in 2012 worldwide [32]. The number of new cases per year is even expected to rise a further 75% to almost 25 million until 2022 [72].

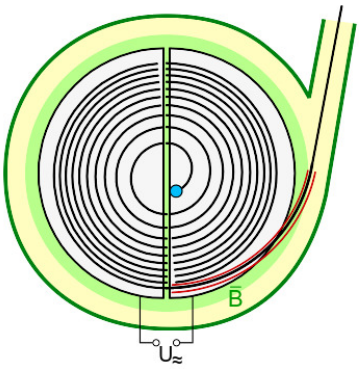
But not only is cancer an enormous burden and suffering for the people who are affected and their relatives, it has also an enormous economic impact. The economic worldwide costs of cancer were estimated to be 1.16 trillion US Dollar in 2010. This is an equivalent of 2% of the total global gross domestic product. And that is only a lower bound to the real costs as the substantial longer-term costs of families and caregivers are not included [72, 9].

As a result, a huge amount of effort has been invested to find promising treatments for cancer. One of these is the radiotherapy or radiation therapy where the cancer cells are exposed to ionizing radiation to be damaged and destroyed. As there are many different kinds of cancers in different body parts that all need to be treated differently, there are also different particle accelerators that can be used for the radiotherapy.

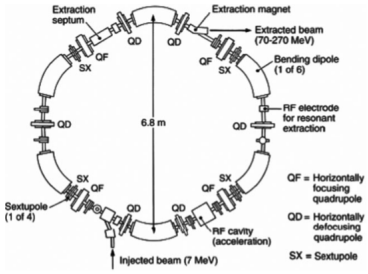
The classic devices for providing radiotherapy are linear particle accelerators, or short LINACS. As the name suggests, particles, mostly electrons, are accelerated linearly when passing through a long straight vacuum tube by electrodes. They can be used either directly or stopped abruptly to deliver supervoltage x-rays [61, 70]. In comparison to the other types of accelerators, these devices are smaller and cheaper and available in many modern hospitals that are specialized for cancer treatment. They are coupled to a treatment room, meaning that each device can be only used to treat a patient in the same room it is located.

Figure 1.1.: A LINAC for cancer treatment at the UKSH Campus Kiel [75]

Apart from linear accelerators, there are also cyclic ones. These can be divided into two types: cyclotrons and synchrotrons. They are mainly used for particle



(a) A sketch of a particle being accelerated and ejected in a cyclotron [33]



(b) Schematic diagram of a proton synchrotron [65]

Figure 1.2.: Cyclic accelerators

therapy, or more correctly hadron therapy, where mainly protons or carbon ions are used for cancer treatment. Both types of accelerators have lead to promising clinical results. The choice of usage depends on treatment method, price, available space and expertise [71, 35].

Cyclotrons are one of the most used accelerators for particle therapy with protons [71]. They overcome the length limitation of LINACS by accelerating the particle along a circular path resembling a spiral (compare to Figure 1.2a). Opposing to cyclotrons, the path of particles in synchrotrons forms a closed loop (Figure 1.2b). This allows to increase their speed with each revolution. Probably the most famous synchrotron is the *Large Hadron Collider* at *CERN* in Genf[1].

Cyclic accelerators can serve multiple treatments rooms alternatingly. Considering the multiple activities (e.g. stabilizing the patient) that need to be performed before and after the actual treatment with the beam, scheduling the beam utilization for each room in an efficient manner can increase the throughput of the facilities.

Planning, creating and managing such schedules is becoming increasingly important. Considering these huge investments that are made to obtain such medical devices, it is clear that on one hand a primary goal for medical facilities is to reduce costs and on the other hand also increase number of treated patients as well as their satisfactory levels. But constructing these schedules is complex due to the shortage of resources as well as different opposing priorities of the involved stakeholders such as facility owners, staff and the actual patients.

The aim of our work is to provide efficient algorithms that calculate such schedules. Here, we focus in particular on the planning and scheduling of the treatments for the

[1] `https://home.cern/topics/large-hadron-collider`

treatment center MedAustron in Wiener Neustadt, Austria[2] that uses a synchrotron that serves three treatment rooms. Scheduling treatments in such a facility can be divided into problems of different granularity. While other works cover the *Particle Therapy Patient Scheduling Problem* (*PTPSP*) that has a time horizon of several months [57, 53, 56, 55], we target the *Intraday Particle Therapy Patient Scheduling Problem* (*I-PTPSP*) with a time horizon of a single day in this thesis.

When scheduling treatments beforehand it is impossible to take all possible uncertain conditions into account that could somehow affect the plan. Hence, a treatment plan is produced with estimates of the actual required times. Then, during the day, when the actual times are known, the original plan often needs to be adjusted to fit the real scenario, especially if an unforeseen event occurs. Solving this problem in a short time is inevitable to not only minimizing the negative impact of such an event on the patients waiting times and the facility's running costs but also to ensure that there exists a treatment schedule that is feasible at all.

The *I-PTPSP* models this situation. We have given treatments with already assigned starting times. Due to an occurrence, like a delay in a treatment room, a no-show of a patient or a delay of a treatment device, the original plan is not feasible any more and a new plan needs to be created. Hence, we need to find a feasible schedule that reacts on the occurrence but remains as close as possible to the original plan. To this end, we aim at minimizing the total working time of the staff, the deviations to the original starting times and idle times on bottleneck resources.

We tackle this problem by introducing a greedy construction heuristic first and implementing different priority functions that will be used for guidance. Then, a Branch and Bound approach is presented. Here, we implement and test various branching tree traversal strategies. First, a depth first strategy based on the mentioned priority functions is designed. Second, a new traversal strategy based on inversion numbers as a measure for the "distance" of solutions from the original one is developed. Finally, we study a best first strategy that repeatedly "dives down" the branching tree to obtain better upper bounds. Additionally, a technique to obtain sharp lower bounds is introduced that is stepwise improved. To conclude, a constraint programming (CP) model of this problem is created.

Finally, to test our approaches, we design several scenarios that resemble real world use cases to obtain proper instances. These are then used to compare the results of the different algorithms with each other. What is more, our Branch and Bound program is tested against a state-of-the-art CP solver, namely Gecode, and it is shown that the latter is only competitive for small problem instances.

In Chapter 2, we start with a detailed problem definition and a formal mathematical description. Then, in Chapter 3, we give a brief overview of related work

[2] `https://www.medaustron.at/en/center`

and problems, especially of former work on the particle therapy patient scheduling problem. Afterwards, we present and describe our different approaches in Chapter 4. The computational study of these can be found in Chapter 5. Last but not least, the conclusion and outlook is provided in Chapter 6.

2. Problem Definition

In this chapter, we define and formalize the *Intraday Particle Therapy Patient Scheduling Problem*, or short, *I-PTPSP*. We start with a description of the assumed input data such as the resources and tasks. Then, we define the structure of a solution and which constraints need to be met to call it feasible. Afterwards, we formalize the objective function that shall be minimized and that describes the quality of such a solution. Lastly, a mathematical model that describes the problem formally, is provided.

The *I-PTPSP* aims at adapting a general plan to current situations that occur during its execution. Therefore, only the plan for the remaining day, starting from the current moment, needs to be recreated.

Our fundamental assumptions are:

- The planning period begins at a given time point and stretches to the end of the considered day.
- Solutions need to be computed within short computation times, i.e., in at most five minutes on a regular personal computer.
- Feasible solutions have to exist.

The first assumption limits the scope of the problem to the regarded time horizon. The second one comes for a simple practical reason. Unexpected occurrences happen unexpectedly but the staff needs to react as soon as possible. The earlier a new schedule is provided the better. The last assumption is needed to exclude cases that do not allow to treat all scheduled patients any more. This would lead to a new optimization problem where it needs to be decided which patients to treat and which to postpone that is not covered in this work.

2.1. Assumed Input Data

In this section we document the input data that is needed to solve the problem. While the first Section 2.1.1 is a rather technical one that depicts parameters that set up the granularity and the domain of the problem, the other two Sections define the considered resources and tasks.

2.1.1. Time

Time is defined as descrete time points in an interval, with a resolution of H^{unit} time units in an hour. The date $\tau = 0$ may be interpreted as 0:00 and $\tau = 24 \cdot H^{\text{unit}}$ as 24:00 of a specific day. Despite of the name of the problem, it is allowed to use dates that are larger than $24 \cdot H^{\text{unit}} \mathrel{\widehat{=}} 24{:}00$. This would be interpreted as the same working day that exceeds midnight. Example given, with a resolution of $H^{\text{unit}} = 60$ (i.e. minutes), the date $\tau = 360$ would be 6:00 AM in the morning, and $\tau = 1200$ would correspond to 8:00 PM in the evening.

Using this parameter allows to adapt to the scheduling needs of different facilities. Some may plan punctual to the minute, others may use only 5-minute slots. Nevetheless, throughout this work, a resolution of $H^{\text{unit}} = 60$ will be used.

Furthermore, we define the fundamental opening time window $\widetilde{W} = [\widetilde{W}^{\text{start}}, \widetilde{W}^{\text{end}})$. This is the time window in which anything must be scheduled at the considered day, including extended times outside of the regular business hours. This is a rather technical parameter that simply bounds the domain for all time variables.

2.1.2. Resources

In this Section, we define all possible properties regarding resources in our problem. Resources R, implemented by $R = \{0, \ldots, n_R - 1\}$, are some kind of infrastructure or staff that is needed to fulfill the treatments, like the particle beam, the treatment rooms, the anaesthetist, and so on. Our resources are not depletable but they can be used by only one task at the same time.

Furthermore, we introduce some time constraints on our resources to allow to model different workshifts, individual breaks or simply unavailability due to maintenance or any unforeseen occurrence. Hence, each resource $r \in R$ shall be available only within an individually predefined timeframe.

We want to differentiate between ordinary service times and extended ones. The ordinary service time windows are defined by $W_r = [W_r^{\text{start}}, W_r^{\text{end}}) \subseteq \widetilde{W}$ where $W_r^{\text{start}} \leq W_r^{\text{end}}$ are the start and end times, respectively. The extended service time windows are given by $\widehat{W}_r = [W_r^{\text{start}}, \widehat{W}_r^{\text{end}}) \subseteq \widetilde{W}$, where $\widehat{W}_r^{\text{end}}$ denotes the extended end time and $W_r^{\text{start}} \leq W_r^{\text{end}} \leq \widehat{W}_r^{\text{end}}$ holds. For some resources the extended service time window might be the same as the regular one. Therefore, we define the subset $\widehat{R} \subseteq R$ of resources with actual extended service time windows, i.e., $\widehat{R} = \{r \in R \mid W_r^{\text{end}} < \widehat{W}_r^{\text{end}}\}$. The time interval $[W_r^{\text{end}}, \widehat{W}_r^{\text{end}}]$ may be interpreted as overtime. Scheduling tasks within these time windows shall be feasible but doing so will increase the objective value of the schedule. Hence, we want to minimize the usage of extended service times to reduce costs.

In addition, each resource $r \in R$ may have unavailability time periods $\overline{W}_r = \bigcup_{w=0,\dots,\omega_r-1} \overline{W}_{r,w}$ with $\overline{W}_{r,w} = [\overline{W}^{\text{start}}_{r,w}, \overline{W}^{\text{end}}_{r,w}) \subset \widehat{W}_r$, $w = 0, \dots, \omega_r - 1$, where $\overline{W}^{\text{start}}_{r,w}$ and $\overline{W}^{\text{end}}_{r,w}$ denote the start and end times of the w-th unavailability period. All these periods are non-overlapping, and sorted according to increasing time. Unavailability periods are expected to not start directly at W^{start}_r or end at $\widehat{W}^{\text{end}}_r$ since otherwise the resources service time windows could be tightened accordingly.

Apart from minimizing extended service time, we also want to reduce idle times on specific resources to avoid an unnecessarily scattered schedule. Therefore, a set $R^{\text{scatter}} \subseteq R$ is introduced that contains all these resources $r \in R^{\text{scatter}}$ for which the idle time between the uses of resource r is to be minimized according to a priority $\varphi^{\text{scatter}}_r$.

2.1.3. Tasks

The patient treatments are the tasks we want to schedule and denoted by $T = \{0, \dots, n_T - 1\}$. Each task has a processing time $p_t \geq 0$ that describes the time needed to perform the entire task.

Beside of its overall processing time, each task requires various resources Q_t at specific sub-intervals of its processing time. For each of these required resources $r \in Q_t$, the interval $P_{t,r} = [P^{\text{start}}_{t,r}, P^{\text{end}}_{t,r}) \subseteq [0, p_t)$ denotes the time relative to the task's starting time in which resource r is needed. For instance, a task with a processing time of 30 minutes could require the room resource for this entire span but the beam resource only for 20 minutes, 5 minutes after the start of the overall treatment. These 5 minutes at the beginning and the end of the task, the particle beam is not needed because, e.g., the patient needs to be laid and fixed or otherwise prepared in the beginning or the patient needs some time for additional imaging and leave the room. We will see that it is convenient to also define the set $\overline{Q}_r \subseteq T$ or each resource $r \in R$ that contains all tasks t that require the resource r, i.e. $\overline{Q}_r \coloneqq \{t \in T \mid r \in Q_t\}$.

Furthermore, each task has an already predefined starting time $\widehat{S}_t$ that comes from the original schedule[1]. This starting time may not be preserved for some reason and has to be adapted now. The time between the newly calculated starting time and the original one represents additional waiting for the patient that we want to minimize.

In addition, we want to define hard lower and upper bounds (S^{L}_t and S^{U}_t) for the new starting time to model the patients individual constraints. A patient's treatment cannot be scheduled if he or she has not arrived yet. On the other hand, if the patient has another appointment scheduled, the treatment cannot be delayed beyond a specific time point.

[1] In our test instances, these are extracted from the solutions of the *PTPSP* [55]

2.2. Solutions, Feasibility and Objective Function

We describe a solution (or schedule) of a problem with a vector $S = (S_0, \ldots, S_{n_T-1})$ that indicates the newly calculated starting times for all tasks in T. This solution is feasible if it fulfills the following requirements:

- Each task $t \in T$ has assigned a valid starting time $S_t \in \widetilde{W}$.
- For each task t, its required resources $r \in Q_t$ are available during the specified time intervals $P_{t,r}$, i.e. $S_t + P_{t,r} \subseteq \widehat{W}_r$.
- Each resource is required by at most one task at the same time, i.e. the tasks do not overlap.

Our objective is to find a feasible solution that

- minimizes the required extended time for each resource r,
- minimizes the lateness w.r.t. the planned starting times $\widehat{S}_t$,
- minimizes the scattering of the usage of resources in R^{scatter}

The individual optimization objectives are roughly prioritized according to the order in which they are listed.

To aid modeling the objective function, we will further use the following variables:

- S_r^{last} denotes the last time resource r is needed.
- σ_t denotes the amount of time task $t \in T$ is delayed from its planned starting time $\widehat{S}_t$, i.e. $\sigma_t = \min(0, S_t - \widehat{S}_t)$

2.3. Basic *I-PTPSP* Model

Now, we can formulate a mathematical model that covers all listed aspects using the natural variables S, S_r^{last} and σ_t.

$$\min \gamma^{\text{extback}} \frac{1}{H^{\text{unit}}} \sum_{r \in \widehat{R}} \max\left(S_r^{\text{last}} - W_r^{\text{end}}, 0\right) + \tag{2.1}$$

$$\gamma^{\text{lateness}} \frac{1}{H^{\text{unit}}} \sum_{t \in T} \sigma_t + \tag{2.2}$$

$$\gamma^{\text{scatter}} \frac{1}{H^{\text{unit}}} \sum_{r \in R^{\text{scatter}}} \varphi_r^{\text{scatter}} \left(S_r^{\text{last}} - W_r^{\text{start}} - \sum_{t \in \overline{Q}_r} |P_{t,r}| \right) \tag{2.3}$$

s.t.

$$S_t + P_{t,r}^{\text{end}} \leq S_r^{\text{last}} \qquad \forall r \in \widehat{R} \cup R^{\text{scatter}}, \forall t \in \overline{Q} \tag{2.4}$$

$$W_r^{\text{start}} \leq S_r^{\text{last}} \qquad \forall r \in \widehat{R} \cup R^{\text{scatter}} \tag{2.5}$$

$$S_r^{\text{last}} \leq \widehat{W}_r^{\text{end}} \qquad \forall r \in \widehat{R} \cup R^{\text{scatter}} \tag{2.6}$$

$$S_t - \widehat{S}_t \leq \sigma_t \qquad \forall t \in T \tag{2.7}$$

$$0 \leq \sigma_t \qquad \forall t \in T \tag{2.8}$$

$$S_t^{\text{L}} \leq S_t \qquad \forall t \in T \tag{2.9}$$

$$S_t \leq S_t^{\text{U}} \qquad \forall t \in T \tag{2.10}$$

$$W_r^{\text{start}} \leq S_t + P_{t,r}^{\text{start}} \qquad \forall r \in R, \forall t \in T \tag{2.11}$$

$$S_t + P_{t,r}^{\text{end}} \leq W_r^{\text{end}} \qquad \forall r \in R \setminus \widehat{R}, \forall t \in T \tag{2.12}$$

$$S_t + P_{t,r}^{\text{end}} \leq \widehat{W}_r^{\text{end}} \qquad \forall r \in \widehat{R}, \forall t \in T \tag{2.13}$$

$$S_t + [P_{t,r}^{\text{start}}, P_{t,r}^{\text{end}}) \cap \overline{W}_r = \emptyset \qquad \forall r \in R, \forall t \in T \tag{2.14}$$

$$S_t + [P_{t,r}^{\text{start}}, P_{t,r}^{\text{end}}) \cap S_{t'} + [P_{t',r}^{\text{start}}, P_{t',r}^{\text{end}}) = \emptyset \qquad \forall r \in R, \forall t, t' \in T, t \neq t' \tag{2.15}$$

$$S_t, \sigma_t, S_r^{\text{last}} \in \mathbb{Z} \qquad \forall r \in R, \forall t \in T$$

Our model contains an objective function that we want to minimize. It consists of three parts:

1. Part 2.1 minimizes the use of the extended service time windows.
2. Part 2.2 minimizes the deviation from the planned starting time, or in other words, it minimizes the lateness of the tasks.
3. Part 2.3 helps to avoid unnecessarily scattered schedules.

These three parts are all weighted individually with the parameters γ^{extback}, γ^{lateness} and γ^{scatter} which are roughly ordered as follows:

$$\gamma^{\text{extback}} > \gamma^{\text{lateness}} \gg \gamma^{\text{scatter}}$$

Exemplary values that are used in this thesis are:

- $\gamma^{\text{extback}} = 10$
- $\gamma^{\text{lateness}} = 1$
- $\gamma^{\text{scatter}} = 0.01$

Here, we describe the constraints we used to enforce only feasible solutions as described in the previous sections:

- Inequalities 2.4, 2.5 and 2.6 define the range of S_r^{last} for all relevant resources. The first one enforces it to be not smaller than the end time point of usage for any task that requires the specified resource. The latter two enforce the variable to be within the resource's availability time windows.
- Restrictions 2.7 and 2.8 define σ, the time of delay to be the difference between the actual starting time and the planned time, but not smaller than zero.
- Inequalities 2.9 and 2.10 enforce the starting time to be within its given bounds S^{L} and S^{U}.
- Constraints 2.11, 2.12 and 2.13 demand that the task is processed only within the resource's service time. The equation 2.14 further excludes any usage of resources during their unavailability periods. Here, we used the following definition of adding an interval to a number: $A + [C, B] \coloneqq [A + C, A + B]$
- Equality 2.15 enforces that each resource is occupied by only one task at the same time.

3. Related work and problems

In this chapter we give a brief overview of related work and problems that are similar to our *I-PTPSP*.

3.1. Job shop scheduling

Job shop scheduling (JSP) is a vast field in scheduling research and is mainly known to address many problems in the manufacturing industry. The problem can be roughly (as there are lots of variations) defined as follows: There are m machines and n jobs $J_1, J_2, \ldots, J_n$ given, each consisting of n_i, $i \in \{1, \ldots, n\}$, tasks which need to be processed in a given order. These tasks have an individual processing time and require a specific machine to be processed on. Each machine can process only one task at once.

JSP is one of the well known and in practice one of the hardest scheduling problems which is NP-hard [11]. There are many different solution approaches to solve this problem. There are a vast amount of different approximation approaches, one of the best known being the shifting bottleneck procedure [1], or using well known metaheuristics, like the simulated annealing [3, 76], tabu search [64, 27], or other methods like using neural networks [78, 67], genetic algorithms [18, 59], or even hybrids of them [2]. You can find an extensive discussion on the hardness of job shop scheduling problems in [58]. And of course there is lots of research on exact approaches using dynamic programming [36], branch and bound [15, 5, 46, 52], or again, hybrids [6].

The *I-PTPSP* resembles a JSP in many aspects. The machines of the JSP represent our resources, like the beam or the treatment rooms. The jobs of the JSP are the treatments of the patients in our problem. But, in contrast to the JSP, our "jobs" often require multiple "machines" at once (among other points).

3.2. Resource constraint project scheduling

A generalization of the job shop scheduling problem is the resource constrained project scheduling (RCSP) problem. Again only a rough definition can be given, as

there are many variations: There are n projects given each consisting of n_i activities. These activities have precedence constraints, individual processing times, and require one or more resources to be processed. A large discussion on variants and extensions can be found in [38] and [14].

Similar to the JSP there is research on a large variety of approaches to solve these problems. Metaheuristics include (among others) simulated annealing [12, 13], tabu search [63, 7] and evolutionary algorithms [4, 74]. An extensive discussion on different heuristics is found in [44]. Exact approaches include Branch and Bound algorithms using different branching methods like Depth First [28, 29, 26][1], Breadth First [62], Best First [73], or A-star tree search [10].

This problem definition meets the requirement of multiple machines per job. In *I-PTPSP* the treatments of the patients are the projects of the RCSP which require different activities with different processing times with one or multiple resources. Our problem can be seen as a a highly specialized RCSP variant that introduces further features, such as unavailability periods, special treatment of specific time spans as overtime of resources, activities that require multiple resources with individual processing times and a composite objective function.

3.3. Patient scheduling problems and appointment scheduling problems

Patient scheduling problems are all kind of scheduling problems that occur in the health care sector and deal with scheduling treatment procedures in some optimal ways. These can often include probabilistic studies on different environmental parameters (like arrival time, no-shows of elective patients, uncertain service times, etc.), examination of different scheduling policies with empirical data, and of course classical solving of scheduling problems. The latter often involve dealing with the mentioned environmental parameters as well as with different arrival processes of out-patients and of course prefering individual treatments due to their urgency levels.

The health care sector is huge and so is the literature on patient scheduling as well as problem variations. Providing a detailed summary over the literature would go beyond the scope of this work and is already covered elsewhere. A great overview of the various challenges in different health care delivery systems is provided in [37, 17]. An overview of the literatur with focus on surgical scheduling can be found in [60, 16, 31]. Despite the huge amount of literature in this field, we could not find any that addresses a problem that resembles the *I-PTPSP* at least in the main points.

[1] [26] covers an extension with min-max timelags

3.4. Previous work on the particle therapy patient scheduling problem

Previous work on the particle therapy patient scheduling problem (*PTPSP*)[57] can be divided into two strands. One tackles the *PTPSP* [57], that has a time horizon of several months, with different heuristics that are repeatedly enhanced. The other one approaches the simplified intraday particle therapy patient scheduling problem (*SI-PTPSP*) [66], that has a time horizon of a single day, with matheuristics. Additionally, based on the *SI-PTPSP*, a even more abstract problem, the so called job sequencing with one common and multiple secondary resources (*JSOCMSR*) problem, was introduced and studied.

In this work, we tackle the *I-PTPSP*, which we formalized in Chapter 2. The main difference to the *SI-PTPSP* is that in the simplified version the tasks can be scheduled relatively freely and only the makespan of the entire plan is to be minimized. In the *I-PTPSP*, on the other side, the tasks have already assigned starting times. The aim is to schedule these such that (among others) their lateness is minimized.

3.4.1. Tackling the midterm planning problem with heuristics

First, in [57], the midterm planning problem, which involves a time horizon of several months, is tackled. The problem was formalized and solved via a MILP model, yet this approach did not deliver acceptable results for use in practice. Therefore, a construction heuristic was developed. This further lead to a *Greedy Randomized Adaptive Search Procedure* and an *Iterated Greedy* (IG) metaheuristic, which performed quite well on instances of practically relevant size.

Then, in [53], the *Iterated Greedy* metaheuristic, which was first presented in [57], is further enhanced by proposing a better construction operator, as well as, a superior local search heuristic that replace former rather simple components.

The *PTPSP* can be decomposed into two sub problems. A day assignment (DA) problem, in which daily treatments are assigned to their individual days and a time assignment (TA) problem, in which good starting times need to be found for the mentioned daily treatments at each day. Obviously, these two sub problems depend on each other. In previous work [57] and [53] a quite trivial lower bound for the TA problems was used that lead to avoidable overtime usages. In [56], a better time estimation for the the makespan of the TA problem is proposed. Using the Enhanced Iterated Greedy metaheuristic from [53] with these improved time estimations improved the performance significantly.

In [55] an extension of the *PTPSP*, that required that the individual treatments of each therapy are provided at roughly the same time of day, was studied. This implicates that the time assignment problems, as described in the previous paragraph, cannot be addressed independently anymore. Hence, the IG needed to be extended and refined. The resulting heuristic did not just provide promising results for the extended *PTPSP* but also improved the ones for the original *PTPSP*.

3.4.2. Approaching the simplified intraday planning problem with matheuristics

In [66], a simplified version of the intra-day problem, that minimizes the makespan, is addressed by a MIP model. The problem of such MIP approaches, when used to solve complex scheduling problems, is that the performance of *time-indexed* (TI) formulations decreases dramatically when the discrete time resolution is increased. In this paper, a *time-bucket* (TB) relaxation is considered to deal with very fine-grained TI models. These time-buckets refer to the subsets that are obtained when partitioning the set of possible starting times. Solving this relaxation is usually easier as it tends to be much smaller than the original problem. The solution of the relaxation can then be used as a lower bound for the solution of the original TI model. One the one hand, this concept is extended by iteratively subdividing some time-buckets to obtain better bounds. The work presents and compares different refinement strategies. One the other hand, the solutions of the relaxations are used as promising starting points for a heuristic that deduces feasible solutions of the original TI model. The concluded matheuristic clearly outperforms simple MILP models, no matter if defined as a discrete-event or a time indexed formulation.

3.4.3. Studying the job sequencing with one common and multiple secondary resources problem

In the work of Horn et al. [39], the job sequencing with one common and multiple secondary resources (*JSOCMSR*) problem was introduced. The *JSOCMSR* is a scheduling problem with, as the name implies, a common resource that is shared by all jobs (e.g., in *I-PTPSP* the beam resource) and secondary resources that are shared only by subsets of all jobs (e.g., in *I-PTPSP* the room resources). First, it was shown that even this simplification is NP-hard. Then, a construction heuristic and an A* algorithm in combination with a sharp lower bound calculation were presented. Afterwards, the problem was also modeled in terms of a MILP and the performance of the shown approaches were compared to the MILP formulation which was outperformed on almost all except for very small problem instances.

Then, in [54] and in [40] a prize-collecting version of the *JSOCMSR*, including a restriction that each job needs to be executed within individual time windows, was

introduced. In [54], this problem is approached with the help of multivalued decision diagrams (MDD), while in [40], the problem is addressed with an A* algorithm.

4. Approach

This chapter is a collection of different concepts and algorithms for solving the *I-PTPSP* as defined in Chapter 2. First, in Section 4.1 we describe our first greedy approach to solve the problem. Afterwards, in Section 4.2, we present our more sophisticated Branch and Bound approach with various approaches of enumerating the branching tree as well as computing the lower bounds of the nodes of the tree.

4.1. Greedy Construction Heuristic

Our *Greedy Construction Heuristic* is based on the time assignment step of the Therapy Wise Construction Heuristic introduced in *Particle Therapy Patient Scheduling: First Heuristic Approaches* [57] due to the similarity of the problem. This heuristic selects a not yet scheduled task based on some specified priorities and sets its start time as early as possible respecting all constraints.

This procedure is shown in detail in Algorithm 1. First, C_r and G are initialized. C_r represents the last time point when resource r is used by an already scheduled task. G defines the set of not yet scheduled tasks. Then, as long as G is not empty, a task t is chosen according to a priority function (this function will be discussed in Section 4.1.1). Afterwards, the starting time S_t is set to the earliest possible time point in such a way that all required resources are available when they are needed by the task t (i.e. the task t does not use any resource r before C_r) and all other constrains as defined in Chapter 2 are fulfilled. If this is not possible (e.g., because of limited resource availability), the procedure will return without a feasible solution S. At the end, all C_r values are updated and the scheduled task is removed from G.

Input: the given input instance
Output: a solution S for the *I-PTPSP*

$C_r \coloneqq W_r^{\text{start}} \qquad \forall r \in R;$
$G \coloneqq T;$
while $G \neq \emptyset$ **do**
  $t \coloneqq \arg\max_{i \in G}$ priority_func(i);
  $S_t \coloneqq$ compute_earliest_start_times(t, C);
  $C_r \coloneqq \max(C_r, S_t + P_{a,r}^{\text{end}}) \qquad \forall r \in Q_a;$
  $G \coloneqq G \setminus \{t\};$
end

Algorithm 1: Greedy Construction Heuristic

4.1.1. Priority function

The used priority function priority_func(task) in Algorithm 1 affects the performance of this heuristic critically. Hence, we tested different functions[1] for their resulting performance gains:

1. *Early planned tasks:* task t has a higher priority than task t' if $\widehat{S}_t < \widehat{S}_{t'}$.
2. *Tasks that require resources with their regular time window ending early:* task t has a higher priority than task t' if t requires a resource r with a lower W_r^{end} than any of t'.
3. *Tasks that minimize scattering of the beam resource:* task t has higher priority than task t' if the idle time that emerges on the beam resource is lower if task t is scheduled next rather than task t'.
4. *Expensive tasks regarding their relative utilization time of the beam:* task t has higher priority than task t' if the ratio of the occupied time of the beam resource and the entire processing time (i.e., $\frac{P_{t,r^{\text{B}}}^{\text{end}} - P_{t,r^{\text{B}}}^{\text{start}}}{p_t}$) of the task t is lower than the one of task t'.

Note that most of these functions relate to different parts of the objective function of the *I-PTPSP* model introduced in Section 2.3. Function 1 prioritizes early planned tasks according to their $\widehat{S}$ values and therefore naturally targets to minimize the second part of the objective function. Function 2 prioritizes tasks that require a resource that ends earlier and hence tries to minimize the first part of the objective function. The third function prioritizes tasks that lead to less scattering and therefore focuses on minimizing the third part of the objective function. Finally, the last function, prioritizes tasks with a lower ratio of beam utilization to entire processing time. These tasks are usually harder to interlock with each other. Scheduling them

[1] Some of these priority functions are based on their equivalents in [57]

earlier leaves many other tasks available that could fit and, as a result, reduce the scattering.

Ties are broken randomly. We also tried combinations of those functions in such a way that if the comparison of two tasks by a priority function results in a tie a different function is used to break it. We chose only combination with function 1 or 3 first as they performed best on our test data (refer to Section 5.3 for the results).

5. apply function 1 first, then 2
6. apply function 1 first, then 3
7. apply function 1 first, then 4
8. apply function 3 first, then 1
9. apply function 3 first, then 2
10. apply function 3 first, then 4

4.2. Branch and Bound

This algorithm is based on the branch and bound design paradigm which was first introduced in [47]. We recommend readers who are not familiar with this concept to refer to [20]. The algorithm operates on so called partial solutions S^{partial}. In our context, partial solutions describe the assignment of starting times S_t to some (not all) tasks t. These can then be gradually extended (by scheduling further tasks) until they become a classic solution S which describes the starting times S_t of all tasks.

Basically, the algorithm consists of two steps: *branching* and *bounding*. First, during the *branching* step, a given partial solution S^{partial} is extended by scheduling a new task t. This is done separately for each of these not yet scheduled tasks and results in multiple new partial solutions S_t^{partial}. Second, during the *bounding* step, lower bounds for the objective value of these partial solutions are calculated and only partial solutions with promising lower bounds are further branched. Eventually, this process will extend these partial solutions to complete ones of which the best one is returned. Each time a better (complete) solution is found, we get an upper bound for the global optimal objective value (i.e. the objective value of this solution, since the optimal value cannot be larger than the objective value of any valid solution). Partial Solutions (and their corresponding sub-trees) with lower bounds larger than the current upper bound can be pruned/bounded and dont need to be examined further.

If only partial solutions with lower bounds greater than the objective value of the current best solution are discarded during the bounding step, an exact solution of

the *I-PTPSP* can be obtained. Moreover, if one is working under time pressure, one can stop this algorithm after a short time and still obtain quite good solutions.

Algorithms 2 and 3 show this process in detail.

Algorithm 2 is quite straightforward. First, the variables *bestSolution* and *bestObj* are set to default values and a priority queue U is filled with an initial empty partial solution. Second, as long as U is not empty, the first partial solution according to a priority is removed from U and branched on if its lower bound is smaller than the current best objective value.

Input: the given input instance
Output: a solution S for the *I-PTPSP*
1 $bestSolution \coloneqq$ None;
2 $bestObj \coloneqq \infty$;
3 $U \coloneqq \{S^{\text{partial}}_{empty}\}$;
4 **while** $U \neq \emptyset$ **do**
5 | $S^{\text{partial}} \coloneqq \text{next}(U)$;
6 | **if** lowerBound(S^{partial}) $<$ *bestObj* **then**
7 | | branch(S^{partial});
8 | **end**
9 **end**

Algorithm 2: Branch and Bound algorithm

The main part happens in Algorithm 3. First, a set G consisting of all not yet scheduled tasks in a given partial solution S^{partial} is initialized. Then, step by step, we remove elements from this set as long as it is not empty. During this loop, we start by removing the task t with the lowest priority according to a priority function[2] from the set G. Next, the method $\text{schedulenext}_{\text{S}}(\text{t})$ returns a new partial solution S^{partial}_t which is created by taking the original partial solution S^{partial} and scheduling the chosen task t at the earliest feasible time. If this partial solution turns out to be complete, i.e., all tasks are scheduled, it is checked if S_t ($\coloneqq S^{\text{partial}}_t$) is a new best solution and saved in that case. Otherwise, the partial solution is inserted into the priority queue U if its lower bound is smaller than the current best objective value. As already mentioned, this procedure is repeated for every t in G.

[2]We use the same function as described in Section 4.1.1

Input: a partial solution S^{partial} that contains the starting times for already scheduled tasks

Output: a set of partial solutions $\{S_t^{\text{partial}} \mid t \text{ is not yet scheduled in } S^{\text{partial}}\}$. Each of these S_t^{partial} extends S^{partial} by the starting time of a previously not scheduled task t.

```
1  G := {t | t is not yet scheduled in S^partial};
2  while G ≠ ∅ do
3  |  t := arg min_{i∈G} priority_func(i);
4  |  G := G \ {t};
5  |  S_t^partial := schedulenext_{S^partial}(t);
6  |  if violates constraint(S_t^partial) then
7  |  |  continue;
8  |  end
9  |  if S_t^partial is complete then        // i.e.  all tasks are scheduled
   |  |  // renaming just to emphasize that this is a complete
   |  |     solution.
10 |  |  S_t := S_t^partial ;
11 |  |  if obj_val(S_t) < bestObj then
12 |  |  |  bestSolution := S_t;
13 |  |  |  bestObj := obj_val(S_t);
14 |  |  end
15 |  else if lowerBound(S_t^partial) < bestObj then
16 |  |  U := U ∪ {S_t^partial};
17 |  end
18 end
```

Algorithm 3: Branch and Bound algorithm: branching

4.2.1. Implementing the priority queue

There are many strategies on how to implement a branch and bound algorithm and often it cannot be said in advance which one will perform better. In this section, we focus on different strategies on how to decide which partial solution to choose for branching. More on branching strategies can be found in [20].

We will now introduce different variants of how to decide which node is the best to choose next.

4.2.1.1. Depth First

One of the most common strategies is the depth first search. As the name implies, this search prefers to process nodes "deeper" in the graph, or in our case, the tree.

It starts at the root and looks at all direct child nodes. Then, it selects one of these and selects a child of this one. This way, each time a node is selected, it is located at a deeper level in the tree as the previous one - at least as long the selected node has a child. When the search reaches a leave node (or in our context a complete solution S), no child can be selected. In this case, the search tracks back the search path until it finds a node with a child that has not been selected yet. If it finds one, it starts selecting child nodes again. This search is also shown in Algorithm 4. For simplicity, only an implementation for searches on trees is shown. The general case of arbitrary graphs requires additional details of how to handle circles and so on that are not necessary to get the idea of how the search works. For more information on this search strategy on general graphs refer to [21].

Input: a tree T and its root node r
Output: a list D containing all nodes $n \in T$, sorted by the order of discovery of this algorithm

```
1  c := r;
2  D := [r];
3  while true do
4  |  u := {n | n = child of c} \ asSet(D);
5  |  if u ≠ ∅ then
6  |  |  c := select out of u;
7  |  |  D . append(c);
8  |  else if c ≠ r then
9  |  |  c := parent of c;
10 |  else
11 |  |  break;
12 |  end
13 end
```

Algorithm 4: Depth First Search on a tree

The main advantage is that it constructs an upper bound (i.e., a feasible solution) for the best objective value extremely fast. Hence, it can be stopped quite soon and still provides a feasible solution. Additionally, the sooner an upper bound is found, the sooner nodes with a poor lower bound can be dropped. Also, this approach needs only limited amounts of memory as it produces only few open nodes.

We combine the Depth First Strategy with the idea of Section 4.1. We implemented this strategy by simply using the priority queue like a stack. If you refer to the branching Algorithm 3 line 3, you will realize that always the worst task according to our priority function is chosen to be scheduled next. This is done to make sure that always the most promising solution (with respect to the depth) is put on the top of the stack (Figure 4.1). This way the first complete solution that is found is identical to the one found with the *Greedy Construction Heuristic* and we start with a hopefully quite good solution right on.

This way we keep all advantages of a Depth First Strategy while introducing the feature to choose the most promising of the to be opened nodes (according to our priority function).

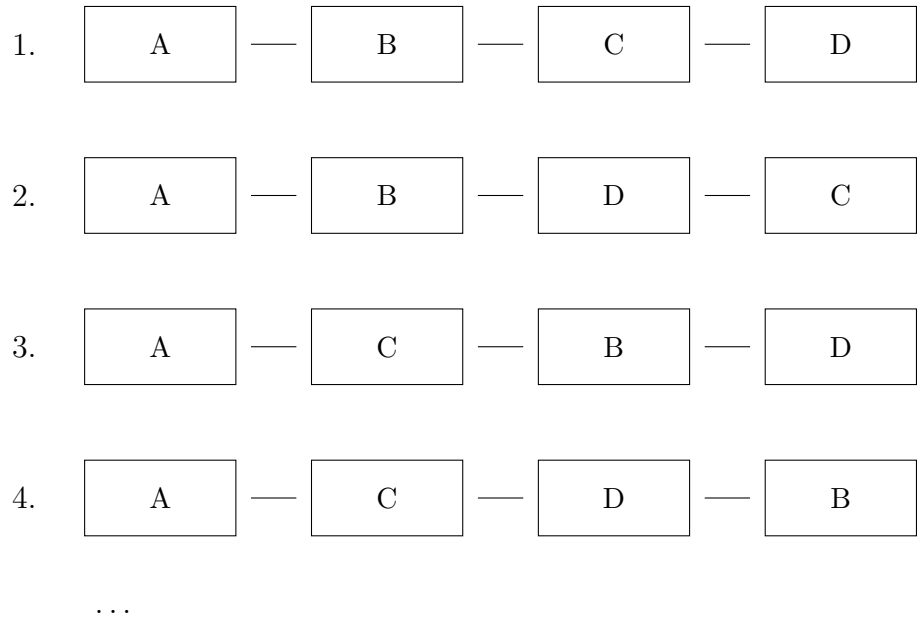


Figure 4.1.: *Depth First* enumeration strategy shown on an example with 4 tasks. We assume that A,B,C,D is the correct order according to the given priority function and that no bounding of partial solutions occurs. Then, the *Branch and Bound* with a *Depth First* strategy would examine the shown solutions in the described order.

The disadvantage of the above approach is that, regarding the order of the scheduled tasks, most of the time it permutes the tasks starting at the end of the day while keeping the order of the early tasks. Considering that, in practice, disruptions of the initial scheduled plans will occur more likely at the start of the plan, it seems more advantageous to permute rather the first tasks than the later ones.

4.2.1.2. Low Inversion Number First

Studying the *Greedy Construction Heuristic* in Section 4.1 one could assume that the solution obtained by applying said *Greedy Construction Heuristic* in combination with a well working priority function is quite "near" to the optimal solution in the solution space. Naturally, we describe a solution "near" to another if it resembles it very closely. But first, we need to define the meaning of "near" more formal. Finally, we will use this definition to find a way of how to iterate over the most similar solutions first.

Permutations and Inversions Effectively we are looking for a way to easily iterate over all possible permutations of the initial solution, starting from "near" to far away. To be able to do this, we need some terminology regarding permutations.

A common notation for a permutation a of the set $\{1, 2, \ldots, n\}$ is

$$a \coloneqq \begin{pmatrix} 1 & 2 & \cdots & n \\ a_1 & a_2 & \cdots & a_n \end{pmatrix}$$

Often, only the second row is denoted as tuple $a \coloneqq (a_1, a_2, \ldots, a_n)$, or even more abbreviated $a \coloneqq a_1 a_2 \ldots a_n$.

An important property of permutations are their *inversions*[3]. In this section we will use the definitions and properties as depicted in [43].

Definition 4.2.1. If $i < j$ and $a_i > a_j$, the pair (a_i, a_j) is called an *inversion* of the permutation. In other words, each inversion is a pair of elements that are out of sort.

Example given, the permutation 1 4 2 5 3 has three inversions: $(4, 2)$, $(4, 3)$ and $(5, 3)$.

Definition 4.2.2. The *inversion table* $b_1 b_2 \ldots b_n$ of the permutation $a_1 a_2 \ldots a_n$ is obtained by letting b_j be the number of elements to the left of j that are greater than j. In other words, b_j is the number of inversions whose second component is j.

For example, the above permutation

$$1\,4\,2\,5\,3$$

has the inversion table

$$0\,1\,2\,0\,0$$

because no number is left to 1, 4 is left to 2, 4 and 5 are left to 3, and no greater numbers are left to 4 and 5.

One important fact we will use is that the *inversion table* uniquely determines the permutation, i.e. it is another way to distinctly describe a permutation.

Example given, to reconstruct the above permutation from the inversion table 0 1 2 0 0, write down the number 5, first. Then, place 4 left to the 5, because $b_4 = 0$. Afterwards, put 3 right to both of them, since $b_3 = 2$. Similarly, 2 must be placed between 4 and 5, due to b_2 being only 1. Finally, 1 is written down left to all numbers as $b_1 = 0$.

[3] They were initially used by Cramer for his famous *Cramer's rule* which can be used to solve linear equations [22]

The advantage of using the *inversion table* instead of the above notations is that it is easier to enumerate all possible permutation since each b_i is independent of the other b_j (while a_is must be mutually distinct). They must simply satisfy

$$0 \leq b_1 \leq n-1, 0 \leq b_2, n-2, \ldots, 0 \leq b_{n-1} \leq 1, b_n = 0$$

This helps, because, often, problems in terms of permutations can be easily solved when described in terms of inversion tables. A very common problem where you may have already used inversion tables instinctively is the question for the number of possible permutations of {1,2, . . . , n}. Using the above constraints for b_i one can easily see that there are n choices for b_1, $n-1$ choices for b_2 and so on, leading to $n(n-1)\ldots 1 = n!$ choices in all.

As already mentioned, inversions describe the pairs of elements that are out of sort. And b_j is the number of inversions whose second component is j. Therefore, the sum over all b_i is the number of all inversions. It is also often referred to as *inversion number* [43].

Definition 4.2.3. The *inversion number* is defined as the sum $\sum_{i=1}^{n} b_i$

Hence, the inversion number counts the number of inversion. In fact, the inversion number is a common measure of the sortedness of a permutation [77] [49]. We will use this property of permutations to measure how "near" a solution is to the initial one.

Using the inversion number as a measure for sortedness The inversion table seems promising to solve our problem of how to permute the initial solution such that we start with the closest, i.e. as few inversions as possible, permutations and end with the fartest ones. We can simply choose the inversion tables with the lowest inversion number first and define some additional ordering among them and then proceed by increasing the inversion number. But as described above, to reconstruct the permutation from an inversion vector, we need to start with the complete inversion vector and obtain the complete permutation which describes the complete solution. This is not optimal since in a branch and bound tree we are working with only partially described solutions of which we only know the beginning. Hence, we need a similar concept to the inversion table which allows to reconstruct the first part of a solution with only knowing the first part of the table/vector.

So let us rethink the problem. We represent our nodes in the Branch and Bound tree as permutations of the first solution. This first solution is created by always scheduling the most promising task[4] next. We could obtain a very similar solution by once choosing the second best task first and all the other times the best task first. Depending on when we decide to choose the second best task to schedule

[4]according to the priority function as described in Section 4.1.1

next, we could represent these solutions by vectors of the length $n_T - 1$. Every time we choose the best task to schedule next, we insert 0. If we chose the second best task, we insert 1 and so on. This encoding can describe every possible solution unambiguously.

It seems obvious, that if we choose the second best task to schedule next only once (represented as e.g. $\langle 1, 0, \ldots, 0\rangle$), the obtained solution will resemble the initial solution to a higher degree than if we chose to schedule the second best task next more often (represented as, e.g., $\langle 1, 1, 0, \ldots, 0\rangle$), or to schedule even the third best task (represented as, e.g., $\langle 2, 0, \ldots, 0\rangle$). Similarly to the inversion table, the lower the sum of the elements of the vector, the more the initial solution is resembled by the represented one.

In fact, the described vector is the so called Lehmer code [48], or sometimes referred to as *right inversion count*, which is another possible definition of the inversion table (in our representation we always omitted the last number l_n which is always 0).

Definition 4.2.4. The *Lehmer code* $l \coloneqq l_1 l_2 \ldots l_n$ of a permutation a is defined as

$$l_i \coloneqq |\{k \mid k > j \wedge a_j < a_i\}|.$$

In other words, l_i is the number of elements in a right to a_i which are smaller than a_i

While the inversion table b_j counts the inversions (i, j) with fixed j, the Lehmer code l_i counts the inversions (i, j) with fixed i. Therefore, again, the sum of all l_i is equal to the inversion number. Similarly to the inversion table, the Lehmer code also satisfies the constraint

$$0 \leq l_1 \leq n - 1, 0 \leq l_2, n - 2, \ldots, 0 \leq l_{n-1} \leq 1, l_n = 0.$$

Functions that fulfill this property are also called *subexceedant functions*. More information about these and their connections to permutations can be found in [50].

We will show an example how to reconstruct the original permutation from a given Lehmer code. Note that you only need to know the first k numbers of the code to reconstruct the first k numbers of the represented permutation and as a result the first k tasks of the represented solution.

Let us assume we are given the following Lehmer code

$$0\,2\,0\,1\,0.$$

To reconstruct the represented permutation, we write down 1 first, as l_1 is 0, meaning that right to this element zero elements are smaller (or in our context, zero tasks are better). Next, we need to pick 4, because $l_2 = 2$ is telling us that two elements

following this one must be smaller, i.e. 2 and 3. Now, the smallest remaining number needs to be placed; i.e. 2. Afterwards, $l_4 = 1$ tells us to write down the 5. Finally, we have only the 3 left and we obtain the original permutation

$$1\,4\,2\,5\,3.$$

After finding a representation of the permutations as we need it, we can program our priority queue to prefer the nodes with a lower inversion number and among them the node that is represented by a vector with a higher number at a lower index (which permutes the earlier tasks more than the later ones).

An example is given in Figure 4.2. Note, that we do not interpret the Lehmer code as the inversion of positions of the originally ordered tasks but as inversions of choosing the best task next according to the priority function. As the priority function is calculated based on dynamic information the *best task* can be a different one depending on tasks scheduled so far and other properties of the current partial solution (refer to Section 4.1.1 for how the priority function is computed).

4.2.1.3. Most Promising First

The most promising first or best first strategy is besides the depth first one one of the most common enumeration strategies. In every iteration it selects the partial solution with the lowest lower bound of all open nodes, i.e. the most promising node. Normally this leads to a very large node tree (and, as a result, to a large memory consumption) because nodes at a lower depth in the search tree tend to have weaker lower bounds than the ones found at a greater depth and, therefore, are selected and branched first. On the other hand, compared to other strategies, much less nodes need to be examined which generally leads to finding an optimal solution faster.

To successfully use this strategy it is crucial to find a way to keep the amount of open nodes as small as possible. Otherwise, the algorithm will run out of memory very quickly. Consequently, we must bound as many nodes as possible. To accomplish this, we need a good global upper bound on the one side and sharp lower bounds on the other side.

As already mentioned, a best first approach, tends to mostly select and branch nodes at lower depths. This means, that complete solutions are found very late which also means that upper bounds are found very late. As a result, many nodes cannot be branched at an early stage of the computation due to the lack of an upper bound. To solve this issue, we implemented so called “dives”. In a predefined interval the algorithm stops branching the best node but instead completes the current partial solution similar to the *Greedy Construction Heuristic* to obtain an upper bound.

On the other hand, we need sharp lower bounds to be able to bound nodes as early as possible. This leads us to the next section.

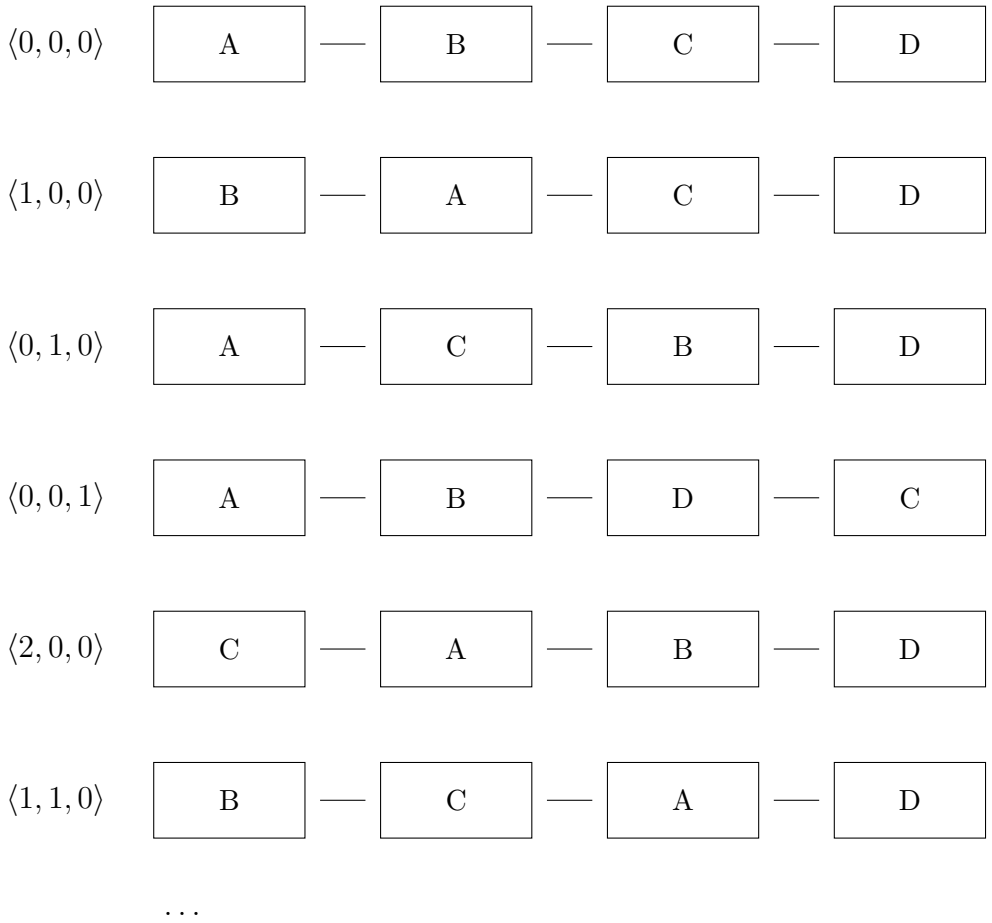


Figure 4.2.: *Low Inversion Number First* ordering of 4 tasks, assuming that A,B,C,D is the order according to the given priority function. The rows are labeled with the corresponding Lehmer code.

4.2.2. Calculating the lower bound of a partial solution

The calculation of a good lower bound is essential for the performance of a branch and bound algorithm. On the one hand, too many possibilities are evaluated if it is not sharp enough. On the other hand, calculating the lower bound should not take too much time as this will also slow down the overall performance dramatically.

In our problem, we need a lower bound for the value of the objective function as defined in Section 2.3. For easier reference it is provided here again:

$$\min \gamma^{\text{extback}} \frac{1}{H^{\text{unit}}} \sum_{r \in \widehat{R}} \max\left(S_r^{\text{last}} - W_r^{\text{end}}, 0\right) + \tag{4.1}$$

$$\gamma^{\text{lateness}} \frac{1}{H^{\text{unit}}} \sum_{t \in T} \sigma_t + \tag{4.2}$$

$$\gamma^{\text{scatter}} \frac{1}{H^{\text{unit}}} \sum_{r \in R^{\text{scatter}}} \varphi_r^{\text{scatter}} \left(S_r^{\text{last}} - S_r^{\text{first}} - \sum_{\substack{t \in T: \\ r \in Q_t}} |P_{t,r}| \right) \tag{4.3}$$

$$\text{with } \gamma^{\text{extback}} > \gamma^{\text{lateness}} \gg \gamma^{\text{scatter}} \tag{4.4}$$

We obtain the lower bound as follows. First, the objective value for a given partial solution is calculated as if it were complete. Afterwards, separate lower bounds lb_{extback} and lb_{lateness} for the terms (4.1) and (4.2) of the objective function are computed and added to the objective value. Note, that these bounds aim only for the remaining part of the final objective value that will be added when scheduling the remaining tasks. The term (4.3) is skipped (i.e., we set its associated lower bound to 0) because of its relatively small weight (as stated by line 4.4). Summing up, the lower bound we work with is $lb \coloneqq \text{obj\_val}(\text{S}^{\text{partial}}) + lb_{\text{extback}} + lb_{\text{lateness}}$, where $\text{obj\_val}(\text{S}^{\text{partial}})$ denotes the objective value of S^{partial}.

When calculating lower bounds for a problem A, it is often useful to ignore some of the constraints and try to solve the resulting "easier" problem B. Relaxing constraints enlarges the solution space $\mathcal{S}$ such that each Solution $S_A \in \mathcal{S}_A$ to the problem A is also a solution to the problem B, but a solution $S_B \in \mathcal{S}_B$ to the problem B may not be a solution to A because it could violate one of the ignored constraints. In other words: $\mathcal{S}_A \subseteq \mathcal{S}_B$. Hence, if we find an optimal solution S_B^* for the easier problem B, we also find a lower bound for the original problem A, i.e. the objective value $\text{obj\_val}(\text{S}_\text{B}^*)$:

Theorem 4.2.1. *If the solution spaces $\mathcal{S}_A$ and $\mathcal{S}_B$ for two minimization (maximization) problems A and B fulfill $\mathcal{S}_A \subseteq \mathcal{S}_B$, then the objective value an optimal solution S_B^* of B is a lower (upper) bound for the problem A.*

Moreover, a lower (upper) bound for B is also a lower (upper) bound for A.

Proof.

$$\text{obj\_val}(\text{S}_\text{B}^*) \leq \text{obj\_val}(\text{S}_\text{B}) \quad \forall S_B \in \mathcal{S}_B \Rightarrow \text{obj\_val}(\text{S}_\text{B}^*) \leq \text{obj\_val}(\text{S}_\text{A}) \quad \forall S_A \in \mathcal{S}_A$$
$$\Rightarrow \text{obj\_val}(\text{S}_\text{B}^*) \leq \text{obj\_val}(\text{S}_\text{A}^*)$$

The first inequality is simply the definition for an optimal solution. The second inequality is true because of $\mathcal{S}_A \subseteq \mathcal{S}_B$. And because of $S_A^* \in \mathcal{S}_A$, we obtain the last one. For a maximization problem, simply replace the inequality symbol $\leq$ with $\geq$.

The last statement of the theorem follows from $\text{lb}(B) \leq \text{obj\_val}(\text{S}_\text{B}^*) \leq \text{obj\_val}(\text{S}_\text{A}^*)$. □

4.2.2.1. Sharper release dates for the sub tasks

In the next sections we are going to relax our problem by splitting the original tasks t into several subtasks t_r according to the used resources such that each subtasks operates on one resource only. The release date ρ for each of this subtask t_r is calculated by $\rho_{t_r} \coloneqq \max(S_t^\text{L} + P_{t,r}^\text{start}, C_r)$.

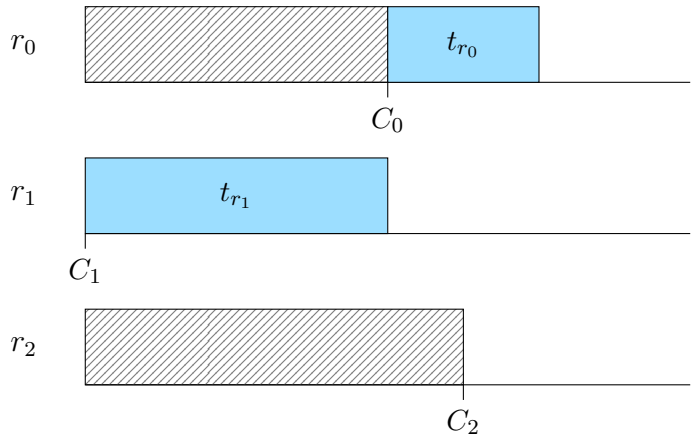
Splitting the tasks into the described subtasks without any constraints is a huge relaxation. This allows solutions with subtasks of the same original tasks being scheduled at completely unrelated parts of the day (e.g., one in the morning, one in the evening). Fortunately, we can use structural information about our tasks to sharpen the release dates of their subtasks.

For two resources, e.g., r_0 and r_1, we can calculate $\delta_t^{r_0,r_1} \coloneqq P_{t,r_1}^\text{start} - P_{t,r_0}^\text{start}$, i.e. the duration after which resource r_1 is required, calculated from the first time point at which r_0 is occupied by task t. Now, we can use this $\delta_t^{r_0,r_1}$ to define a sharper release date for the subtask t_{r_1} : $\rho_{t_{r_1}} \coloneqq \max(S_t^\text{L} + P_{t,r_1}^\text{start}, C_{r_0} + \delta_t^{r_0,r_1})$ where C_{r_0} describes the last time point at which resource r_0 stops being occupied in our partial solution S^partial. Refer to Figure 4.3 for a visual depiction of this procedure.

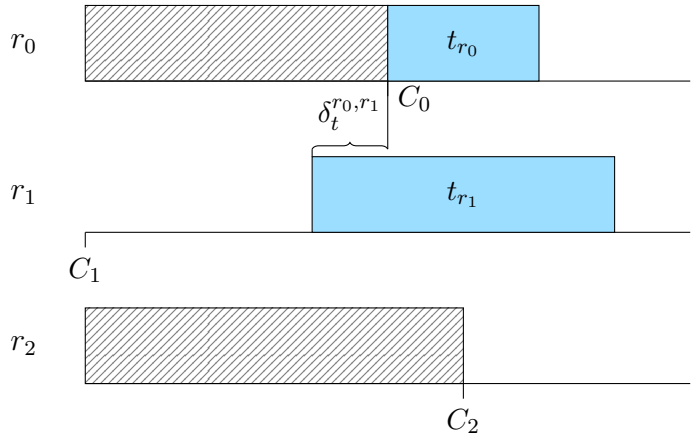
Obviously, the described approach to sharpen the release dates of subtasks is not bound to only the resources r_0 and r_1. We apply it to all resources of the selected resource set as described in the Section 4.2.2.6.

4.2.2.2. Calculating lb_extback

To calculate the lower bound lb_extback, we take the last times C_r a resource r is used by an already scheduled task t and simply add up the durations $P_{t',r}^\text{end} - P_{t',r}^\text{start}$ for all the still unscheduled tasks and the lengths of the relevant unavailability periods. Afterwards, we subtract $W_r^\text{end} - C_r$ to get the penalties for extending regular service windows. We used $W_r^\text{end} - C_r$ instead of W_r^end because we need to ignore the parts of



(a) Subtasks t_{r_0} and t_{r_1} of task t as they would be scheduled without sharpening of their release dates.



(b) Subtasks t_{r_0} and t_{r_1} of task t as they are scheduled with sharpening of their release dates. In this case the subtask t_{r_1} is scheduled later because of its modified release date $\rho_{t_{r_1}} := C_{r_0} + \delta_t^{r_0,r_1}$

Figure 4.3.: Scheduling the subtasks t_{r_0} and t_{r_1} of task t with and without sharpening of their release dates

the penalties that are already included in the objective value for the partial solution S^{partial}.

The computation is shown in more detail in Algorithm 5. First, C_r and C_r^{orig} are initialized to the last time the resource r is used. Then, W_r' is set to the maximum of W_r^{end} and C_r. We will use this variable instead of the original W^{end} to exclude any penalty that is already included in the calculated objective value for this S^{partial}. Additionally, UT is defined as the set of all unscheduled tasks.

The loop at line 4 adds the processing times of each not yet scheduled task to C_r for each resource.

Finally, in the loop at line 10 we include the times for the relevant unavailability periods and then sum up the lengths of times a resource is beeing used in its extended time window.

Input: the partial solution S^{partial} to calculate a lower bound for
Output: a lower bound lb_{extback} for the term 4.1 of the objective function
1 $C_r \coloneqq C_r^{\text{orig}} \coloneqq$ last time the resource r is used in $S^{\text{partial}} \qquad \forall r \in \widehat{R}$;
2 $W'_r \coloneqq \max(W_r^{\text{end}}, C_r) \qquad \forall r \in \widehat{R}$;
3 $UT \coloneqq$list of all unscheduled tasks;
4 **foreach** $t \in UT$ **do**
5 **foreach** $r \in Q_t$ **do**
6 $C_r \mathrel{+}= P_{t,r}^{\text{end}} - P_{t,r}^{\text{start}}$;
7 **end**
8 **end**
9 $sum \coloneqq 0$;
10 **foreach** $r \in \widehat{R}$ **do**
11 $w \coloneqq min(\{w' \mid C_r^{\text{orig}} \leq \overline{W}_{r,w'}^{\text{start}}\})$;
12 **while** $\exists \overline{W}_{r,w} \wedge \overline{W}_{r,w}^{\text{start}} \leq C_r$ **do**
13 $C_r \mathrel{+}= |\overline{W}_{r,w}|$;
14 $w \mathrel{+}= +1$;
15 **end**
16 $sum \mathrel{+}= \max(0, C_r - W'_r)$;
17 **end**
18 **return** $\gamma^{\text{extback}} \cdot \frac{1}{H^{\text{unit}}} \cdot sum$;

Algorithm 5: Calculating the lower bound lb_{extback}

4.2.2.3. Calculating lb_{lateness}

Calculating the lower bound lb_{lateness} turns out to be much harder as it seems to be on the first sight. First, let us summarize, what we are trying to do. We have tasks that require different resources which can be used by only one task at the same moment and we need to schedule these tasks in a way such that they start being processed before a specified timepoint. In our case, this timepoint is $\widehat{S}_t$. Scheduling tasks later than this specified timepoint is allowed but not wanted. Scheduling them earlier does not improve the quality of the solution. We can formalize this by $\max(s - tp, 0)$ where s denotes the tasks scheduled starting time and tp the formerly described specified timepoint and call this function the task's *tardiness*. The quality of a solution is measured by it's *total tardiness* which is the sum over the tardinesses of each task.

Moreover, we have lower bounds S^{L} for their starting times that need to be respected. These lower bounds are also called *release dates* in scheduling theory.

Furthermore, these tasks can be only processed in one block, i.e. they cannot be interrupted. In scheduling theory this sort of tasks are also called non-preemptive. Hence, it does not matter whether we demand the tasks starting at a specified timepoint a or ending at a timepoint $b := a + p$ where p denotes their processing time because the difference between a and b is constant in case of non-preemptive tasks. These timepoints b are also referred to as *due dates*.

Tasks require different resources at different timepoints with different processing times. To reduce complexity, we split the tasks into different subtasks for each resource that is required. Each of these subtasks is assigned a due date that we compute by $\widehat{S}_t + P_{t,r}^{\text{end}}$. Now, we can decouple these subtasks and try to solve this scheduling problem for each resource on its own. Since every solution to the original problem is also a solution to this one, we can apply Theorem 4.2.1 and consequently focus on solving (or at least obtaining a lower bound to) this "easier" problem.

To sum up, we are trying to schedule non-preemptive tasks which must not overlap. These tasks t have a release date r_i, a due date d_i and a processing time p_i and we want to find a solution S that minimizes the total tardiness

$$TT(S) := \sum_t \max(c_i - d_i, 0)$$

where c_i denotes the task's completion time and each task's starting time must be greater or equal to it's release date r_i.

In scheduling theory this is a well studied problem and (the simplified version without release dates) is called the *Single Machine Total Tardiness Scheduling Problem* or simply *SMTTSP*.

Although, we have already simplified the original problem quite a lot, the current problem is still NP-hard [30]. A very common practice, when trying to simplify a scheduling problem, is to relax the non-preemptiveness. This means that we allow to process the tasks partwise. This way we are able to fill any open "gaps" with pieces of still unprocessed tasks which generally leads to better solutions. But still, even this problem is NP-hard [19].

Nevertheless, as already mentioned, it is a well studied problem, so we can learn from many different approaches to solve this problem. Most of the Branch and Bound approaches used a lower bound based on Chu's lower bound [19] [45]. We even found a similar method being used to solve this problem in 1974 [8].

4.2.2.4. Chu's lower bound

Chu's lower bound [19] requires the *Shortest Remaining Processing Time (SRPT)* rule. Therefore, we will explore this one first:

Scheduling tasks by the SRPT rule is quite simple: We start at the timepoint 0 and increase the current time by one with each step. At each timepoint we schedule a part of length 1 of the task with the shortest remaining processing time of all tasks that can be currently scheduled (i.e. tasks with release dates smaller or equal to the current timepoint). Consequently, the remaining processing time of this task is reduced by one. We are done, when the remaining processing times of all tasks is 0. It is clear that as soon as the task with the shortest remaining processing time is found it surely keeps this property until it finishes (i.e. its remaining processing time becomes 0) or a new task becomes available (i.e. we reach it's release date). Therefore, it suffices to only consider times when a task ends or becomes available.

So let us consider a given non-preemptive problem Π with n tasks and it's optimal total tardiness $TT^*(\Pi)$ and construct a schedule S according to the currently explained SRPT rule. Next, we order the completion times c_i of the tasks i in S and their due dates d_i separately in non-decreasing order. We denote these new ordered series by $(c'_1, c'_2, \ldots, c'_n)$ and $(d'_1, d'_2, \ldots, d'_n)$, respectively. Chu has proven that the following relation holds [19]:

$$\sum_{i=1}^{n} \max(c'_i - d'_i, 0) \leq TT^*(\Pi) \tag{4.5}$$

In other words, Chu's lower bound compares the i-th completion time with the i-th due date and states that a such computed tardiness is surely not greater than the real one.

We have not mentioned any unavailability periods ($[\overline{W}^{\text{start}}, \overline{W}^{\text{end}}]$) yet. They are no part of the definition of the *SMTTSP* but they can be easily integrated in our problem if we just transform them into regular tasks by setting its release date to $r := \overline{W}^{\text{start}}$, due date to $d := \overline{W}^{\text{end}}$ and processing time to $p := \overline{W}^{\text{end}} - \overline{W}^{\text{start}}$. Obviously, treating unavailability periods as delayable tasks instead of rigid unavailability periods will relax our problem even more.

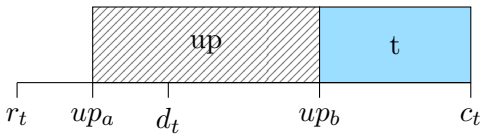
4.2.2.5. Modified Chu's lower bound

Until now, we treated unavailability periods like schedulable tasks to be able to use algorithms and results developed for problems without such unavailability periods. In this section we will show how this can affect the sharpness of the calculated lower bounds and how just little adaptions to the original algorithms allow us to treat unavailability periods exactly as such.

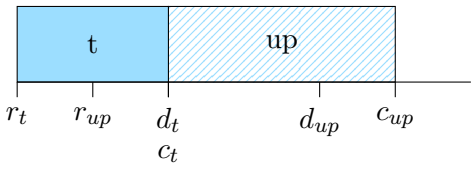
Consider the following simple problem. We need to schedule a task t with $r_t = 0$, $d_t = 2$ and $p_t = 2$ and an unavailability period *up* in the time interval $[1, 4]$. Clearly, the optimal solution is to schedule the task to start at timepoint 4 yielding a total tardiness of 4. If we apply Chu's lower bound as defined in Section 4.2.2.4, task t is

scheduled from 0 to 2 and the unavailability period *up* (treated as a task) from 2 to 5 resulting in a lower bound of only 1. Obviously we could improve the sharpness of our lower bound if we treated unavailability periods as such (see Figure 4.4).

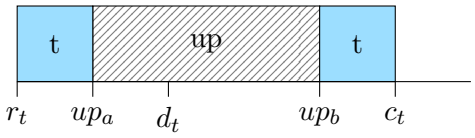
But to do this, we need to prove that the relation 4.5 still holds. We will see, that the needed proofs are almost equal to the original ones.



(a) Optimal Solution: $TT(S) = 4$



(b) Chu's lower bound: $lb(S) = 1$



(c) Adapted Chu's lower bound $lb^a(S) = 3$

Figure 4.4.: Comparing lower bounds

When proving the mentioned relation, Chu made use of the fact that the SRPT rule minimizes the current number not completed jobs at any given time point. Hence, we need to adapt the SRPT rule to respect unavailability periods and prove that the adapted version keeps this property. Here we will adapt the proof of Schrage [68] (in the original version this was proved for a continuous timeline. We are using a discrete one, so additionally to introducing unavailability periods, we will proof a discrete version).

Definition 4.2.5. The adapted SRPT rule (ASRPT) shall be defined as follows: Start at timepoint 0 and increase it by 1 with each step. If the current timepoint is included within a given unavailability period, do nothing. No task is allowed to be processed. If, on the other hand, the current timepoint is outside of an unavailability period, simply apply the normal SRPT rule as defined in Section 4.2.2.4.

Theorem 4.2.2 (Optimality of ASRPT)**.** *Consider a scheduling problem with n preemptive tasks, release dates r_i and processing times p_i on one machine that can process only one task at the same time and assume that the problem starts at time 0. Additionally, unavailability periods, i.e., time intervals $[a_q, b_q]$, are given, during*

which no task can be processed. Then the current number of not completed tasks at any timepoint is minimized by a schedule fulfilling the ASRPT *rule.*

Proof. We will show that any schedule S not following the ASRPT rule can be improved by performing said rule. Therefore, no schedule not following the ASRPT rule can be optimal.

But first, we need to introduce some additional notation: $\delta_i(t)$ determines whether a task i is being processed at timepoint t

$$\delta_i(t) \coloneqq \begin{cases} 1 & \text{if the machine is processing task } i \text{ at timepoint } t \\ 0 & \text{else} \end{cases}$$

Due to the restrictions of the problem, we know that $\forall t : (\sum_{i=1}^{n} \delta_i(t) \leq 1)$ and for any unavailability period q that $\forall t \in [a_q, b_q] : (\sum_{i=1}^{n} \delta_i(t) = 0)$

$p_i(t)$ denotes the processing time remaining for task i *before* timepoint t.

$$p_i(t) \coloneqq \begin{cases} p_i & \text{if } t = 0 \\ p_i - \sum_{k=0}^{t-1} \delta_i(k) & \end{cases}$$

furthermore $\theta(t)$ refers to the set of tasks which are available at timepoint t

$$\theta(t) \coloneqq \{i \mid r_i \leq t\}$$

and finally c_i which describes the completion time of task i, i.e. the timepoint t *after* which the task will be completed.

$$c_i \coloneqq \min\{t \mid p_i(t+1) = 0\}$$

We know for any $t' < t$ that

$$p_i(t) = p_i - \sum_{k=0}^{t-1} \delta_i(k) = p_i - \sum_{k=0}^{t'-1} \delta_i(k) - \sum_{k=t'}^{t-1} \delta_i(k) = p_i(t') - \sum_{k=t'}^{t-1} \delta_i(k)$$

This implies for any $t' \leq \min\{t \mid p_i(t+1) = 0\}$ that

$$c_i = \min\left\{t \mid p_i(t') - \sum_{k=t'}^{t} \delta_i(k) = 0\right\} = \min\left\{t \mid p_i(t') = \sum_{k=t'}^{t} \delta_i(k)\right\} \tag{4.6}$$

Furthermore, we need the fact that minimizing the current number of not completed tasks at any timepoint is equal to minimizing $\sum_{i=0}^{n} c_i$. This is simply because reducing number of not completed tasks at t is equivalent to reducing the completion time of a task i which does not end before t (i.e. $c_i >= t$) such that it ends before t (i.e. $c_i < t$).

Now, we have everything we need to proof said theorem. We will use the notation with an "o" as superscript $(p_i^o(t), c_i^o, \ldots)$ for variables of the original non-ASRPT schedule S, an "n" as superscript $(p_i^n(t), c_i^n, \ldots)$ for the new ASRPT schedule S^n we want to construct from S and the normal notation without any superscript if these variables are equal for both Schedules S and S^n.

Assuming S does not respect the ASRPT rule implies that there exists a timepoint $t^* \notin [a_q, b_q]$ for any unavailability period q such that

$$\exists i, j \in \theta^o(t^*) : (p_i^o(t^*) < p_j^o(t^*) \wedge \delta_i^o(t^*) = 0)$$

and one of the following cases:

$$\text{(a) } \forall k \in \theta^o(t^*) : (\delta_k^o(t^*) = 0)$$

or

$$\text{(b) } \exists! j \in \theta^o(t^*) : (\delta_j^o(t^*) = 1)$$

In the first case, the machine is idle, and c_i^o can be reduced by setting $\delta_i^n(t^*) = 1$ (and $\delta_i^n(c_i^o) = 0$, see also Figure 4.5a on page 51) without altering the value of any $c_j \forall j \neq i$. Hence, the objective value has improved and S was not optimal.

Suppose the second case is true. We define the set $\omega \coloneqq \{t \mid t \geq t^* \wedge \delta_i^o(t) + \delta_j^o(t) = 1\}$, i.e., the set of all timepoints t when either task i or task j are processed. We will now construct a rearrangement of $\delta_i^o(t)$ and $\delta_j^o(t)$ for $t \in \omega$ to construct a better schedule S^n.

Obviously $c_i + c_j = \min(c_i, c_j) + \max(c_i, c_j)$. Due to the equation 4.6, we can show

$$\begin{aligned}\max(c_i^o, c_j^o) &= \min\left\{t \mid p_i(t^*) + p_j(t^*) = \sum_{k=t^*+1}^{t+1} \delta_i^o(k) + \delta_j^o(k)\right\} \\ &= \min\left\{t \mid p_i(t^*) + p_j(t^*) = \sum_{k=t^*+1}^{t+1} \delta_i^n(k) + \delta_j^n(k)\right\} = \max(c_i^n, c_j^n)\end{aligned}$$

regardless of how we interchange the processing times of these two tasks during ω.

From the facts that $p_i^o(t^*) < p_j^o(t^*)$ and $\delta_i^o(t^*) = 0 \wedge \delta_j^o(t^*) = 1$, it is easy to see that if we follow the ASRPT rule and interchange the processing of i and j at t^* and c_i^o (i.e., setting $\delta_i^n(t^*) = 1 \wedge \delta_j^n(t^*) = 0$ and $\delta_i^n(c_i^o) = 0 \wedge \delta_j^n(c_i^o) = 1$, see also Figure 4.5b on page 51), the function

$$\min\left\{t \mid p_l(t^*) = \sum_{k=t^*}^{t} \delta_l(k)\right\}$$

is minimized by setting $l = i$.

This implies that

$$\min(c_i^n, c_j^n) < \min(c_i^o, c_j^o)$$

We know now, that

- $c_l^o = c_l^n$ for any $l \neq i, j$
- $\max(c_i^o, c_j^o) = \max(c_i^n, c_j^n)$
- $\min(c_i^o, c_j^o) > \min(c_i^o, c_j^o)$

Hence, $\sum_{l=0}^{n} c_i^o > \sum_{i=0}^{n} c_i^n$ which shows that S could be improved by following the ASRPT rule. Therefore the number of not completed jobs at any timepoint in an ASRPT schedule is always less than for any non-ASRPT schedule. □

From now on we also need the following notation.

Definition 4.2.6. Consider a series $(a_1, a_2, \ldots, a_n)$. Then $a_{[i]}$ denotes the i-th element in the non-decreasingly ordered series $(a_{\pi(1)}, a_{\pi(2)}, \ldots, a_{\pi(n)})$.

We also need the following Lemma. We already used a similar one at the end of the previous proof. Chu also used it (Lemma 3 in [19]). We will provide it here and adapt the notation for easier readability. Chu did not provide a proof, so we do it now.

Lemma 4.2.1 (Chu's Lemma 3)**.** *If there exist two jobs i and j for the Schedules S and S' of our problem Π such that each of the following holds*

(a) $\min(c'_i, c'_j) \leq \min(c_i, c_j)$

(b) $\max(c'_i, c'_j) \leq \max(c_i, c_j)$

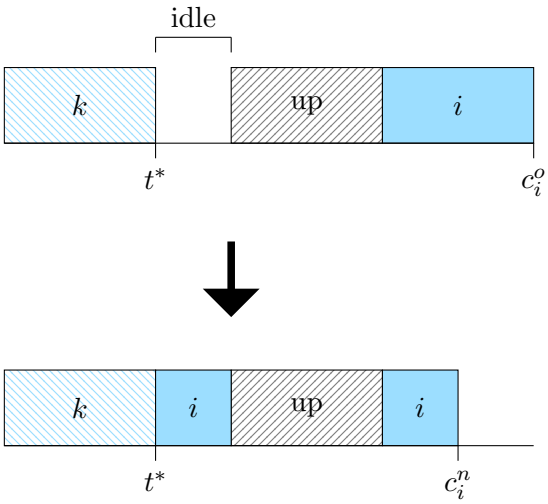
(c) $c'_k = c_k \quad \forall k \neq i \wedge k \neq j$

then

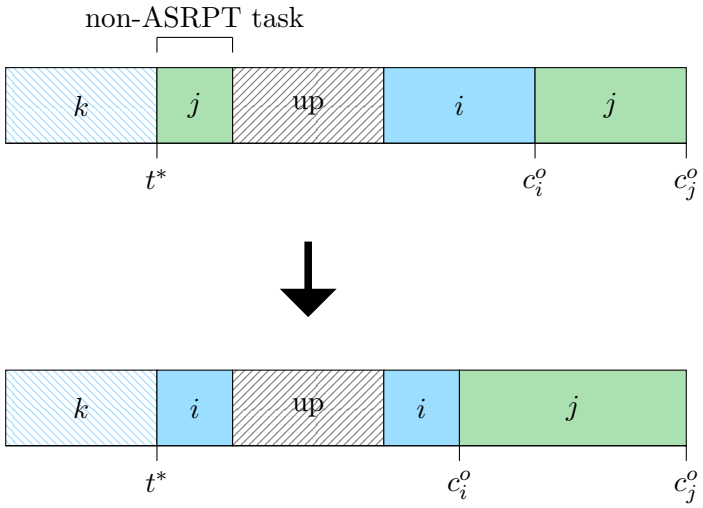
$$c'_{[l]} \leq c_{[l]} \quad \forall l$$

Proof. Let π be a permutation of $\{1, \ldots, n\}$ such that the series $(c_{\pi(1)}, c_{\pi(2)}, \ldots, c_{\pi(n)})$ is in non-decreasing order and, therefore, $c_\pi(l) = c_{[l]}$.

Because of equation (c) we know that $c'_{\pi(l)} \leq c_{\pi(l)} \quad \forall \pi(l) \neq i, j$.



(a) Case 1: idle time during which no task is processed



(b) Case 2: task j with a larger remaining processing time than task i is beeing processed at t^*.

Figure 4.5.: The two cases mentionend in the proof of Theorem 4.2.2.

W.l.o.g. we can assume that $c_i = \min(c_i, c_j)$ (otherwise interchange i and j in the following proof). Let $a \coloneqq \pi^{-1}(i)$ and $b \coloneqq \pi^{-1}(j)$. Now, we define a new permutation

π' as follows

$$\pi' \coloneqq \begin{cases} \pi'(k) = \pi(k) \quad \text{if } k \neq a, b \\ \pi'(a) = \begin{cases} \pi(a) & \text{if } c'_{\pi(a)} = \min(c'_i, c'_j) \\ \pi(b) & \text{if } c'_{\pi(b)} = \min(c'_i, c'_j) \end{cases} \\ \pi'(b) = \begin{cases} \pi(b) & \text{if } c'_{\pi(b)} = \max(c'_i, c'_j) \\ \pi(a) & \text{if } c'_{\pi(a)} = \max(c'_i, c'_j). \end{cases} \end{cases}$$

We defined π' such that $c'_{\pi'(l)} \leq c_{\pi(l)}$ holds for all $l = 1, \ldots, n$.

If the order did not change (or only $c'_{\pi'(a)}$ and $c'_{\pi'(b)}$ switched places), we have

$$c'_{[l]} = c'_{\pi'(l)} \leq c_{\pi(l)} = c_{[l]} \quad \forall l$$

and are done.

Otherwise, $c'_{\pi'(a)}$ (or $c'_{\pi'(b)}$ or both of them) need to switch places with some $c'_{\pi'(k)}$ with $k \neq a$ (or $k \neq b$, respectively) to form a non-decreasingly ordered series. Other cases are not possible, because of the definition of π and because at most two variables changed their values.

We will only show that the above equation holds for the first case. The other cases are done analogous:

So, let us assume that $c'_{\pi'(a)}$ needs to switch place with some $c'_{\pi'(k)}$ to form an ordered series. Because of

$$c'_{\pi'(a)} = \min(c'_i, c'_j) \leq \min(c_i, c_j) = c_{\pi(a)}$$

we know that $c'_{\pi'(a)}$ can become only smaller and therefore can only change with a previous entry in the series, i.e. $k < a$.

But, because of π is forming an ordered series, we can deduce an even stronger inequality as shown above:

$$c'_{\pi'(p)} \leq c_{\pi(p)} \leq c_{\pi(l)} \quad \forall l \ \forall p \leq l \tag{4.7}$$

Let σ be the ordering of (c'_i) (in non-decreasing order).

We get

$$c'_{\sigma(k)} = c'_{\pi'(a)} \leq c'_{\pi'(k)} \leq c_{\pi(k)}.$$

The first inequality is true, because if not, they would not had to change places to form a non-decreasing order.

And

$$c'_{\sigma(a)} = c'_{\pi'(k)} \leq c_{\pi(a)}.$$

The inequality is true because of $k < a$ and inequality 4.7.

As already explained above, all other c's stay at the same place and we obtain

$$c'_{[l]} = c'_{\sigma(l)} \leq c_{\pi(l)} = c_{[l]} \quad \forall l.$$

□

We also need Chu's Lemma 1 in [19] (refer to it for the proof):

Lemma 4.2.2 (Chu's Lemma 1)**.** *Let two series of numbers $(x_1, x_2, \ldots, x_n)$ and $(y_1, y_2, \ldots, y_n)$ and an ordered series $(x'_1, x'_2, \ldots, x'_n)$ be so that $x'_1 \leq x'_2 \leq \ldots \leq x'_n$ and $x'_i \leq x_i \quad \forall i$. If $(y'_1, y'_2, \ldots, y'_n)$ is the series obtained by sorting the series $(y_1, y_2, \ldots, y_n)$ in non-decreasing order, the following relation holds:*

$$\sum_{i=1}^{n} \max(x'_i - y'_i, 0) \leq \sum_{i=1}^{n} \max(x_i - y_i, 0)$$

Now, we have everything we need for

Theorem 4.2.3. *Adapted Chu's lower bound: Consider a SMTTSP problem Π with n tasks, different release dates and unavailability periods. If the problem is relaxed to allow preemptive tasks and a schedule Σ is constructed with the ASRPT rule (see Definition 4.2.5), then the following relation holds*

$$\sum_{i=1}^{n} \max(c_{[i]} - d_{[i]}, 0) \leq TT^*(\Pi)$$

where $(c_1, c_2, \ldots, c_n)$ are the completion times of the tasks in Σ, $(d_1, d_2, \ldots, d_n)$ their due dates, and $TT^*(\Pi)$ the optimal total tardiness of the problem.

Proof. This proof is very similar to the one published by Chu [19] for the problem without unavailability periods.

If we can show that for any Solution S of the relaxed problem the inequality

$$c^{\Sigma}_{[i]} \leq c^{S}_{[i]} \quad \forall i \tag{4.8}$$

holds, we can use the Lemma 4.2.2 to obtain, we can use the Lemma 4.2.2 to obtain

$$\sum_{i=0}^{n} \max(c^{\Sigma}_{[i]} - d_{[i]}, 0) \leq \sum_{i=0}^{n} \max(c^{S}_{[i]} - d_{[i]}, 0).$$

This inequality holds especially for the optimal solution S^* of the relaxed problem. And because of $TT(S^*) \leq TT^*(\Pi)$, all we need is to prove the inequality 4.8 for all S of the relaxed problem.

Let us assume such an Solution S. If it obeys the ASRPT rule, we can simply use the Theorem 4.2.2 to follow that the inequality 4.8 holds.

Therefore, let us assume that S does not obey the ASRPT rule. The idea of the following is similar to the proof of the Theorem 4.2.2. We will construct a schedule S' from S according to the ASRPT rule and show that the inequality holds for these two (Σ replaced by S'). Finally as the inequality also holds for S' we can deduce transitively that it also holds for the original S.

Let t^* be the first time that S does not obey the ASRPT rule. Again, we have two cases. Due to the similarity, we will use the same notation as in Theorem 4.2.2.

(a) The machine is idle at time $t^* \notin [a_q, b_q]$ for any unavailability period q and there exists a task i that is available at t^*: We reschedule task i to be processed at t^* such that it ends earlier (i.e. setting $\delta'_i(t^*) = 1 \wedge \delta'_i(c_i) = 0$). Therefore, there surely is a task j such that

- $\min(c'_i, c'_j) \leq \min(c_i, c_j)$
- $\max(c'_i, c'_j) \leq \max(c_i, c_j)$
- $c'_k = c_k \quad \forall k \neq i \wedge k \neq j$

Now we can apply Lemma 4.2.1 and obtain $c^{S'}_{[i]} \leq c^{S}_{[i]} \quad \forall k$.

(b) There is a task i such that its processing time is less than than that of task j at time t^*. Still, task j is processed at t^*. Again, let $\omega \coloneqq \{t \mid t \geq t^* \wedge \delta_i(t) + \delta_j(t) = 1\}$. With the same argument as in the proof of Theorem 4.2.2 we can reschedule task i and task j in ω, such that i is being processed at t^* instead of j and obtain

- $\min(c'_i, c'_j) < \min(c_i, c_j)$
- $\max(c'_i, c'_j) = \max(c_i, c_j)$
- $c'_k = c_k \quad \forall k \neq i \wedge k \neq j$

Again, with Lemma 4.2.1 we show that $c^{S'}_{[i]} \leq c^{S}_{[i]} \quad \forall k$.

Continuing this way, we can iteratively apply the ASRPT rule to construct a $S^{(n)}$ from S which at the end does obey the ASRPT. As a result, we we have shown that equality 4.8 holds for any S

$$c^{\Sigma}_{[i]} = c^{S(n)}_{[i]} \leq \ldots \leq c^{S'}_{[i]} \leq c^{S}_{[i]} \quad \forall k.$$

□

4.2.2.6. Applying the lower bound computation over different resources

The Adapted Chu's lower bound gives us a lower bound for the SMTTSP with different release dates and unavailability periods. The *Single Machine* in SMTTSP is equivalent to single resource in our problem. Of course this is already a lower bound to our original problem. To sharpen it, we could compute it for each resource separately and then take the maximum of each result.

We need to compute new lower bounds at every single expanded node. As this happens quite often a balance of time consuming computation and acquiring sharp lower bounds has to be found. Therefore, we need to decide which resources we will use for the computation. Naturally, we will only include the resources which we expect to be bottleneck resources and, as a consequence, yield larger lower bounds.

The way our instances are created, some resources can be expected to be bottleneck resources and some not. Hence, we tried the following approaches which will be evaluated in the Computational Results Section 5.5:

1. The simplest solution is to only compute the lower bound for the *beam* resource as almost all tasks require this resource for quite a long time. So this resources is naturally a bottleneck resource.

2. Another group of resources that is expected to be bottleneck resources are the *room* resources. So, to keep the computation time short we only include the *beam* and the *room* resources in our calculation.

3. We also want to investigate the effect on the lower bound quality when including all resources. But, as *patient* resources are needed by at most one task and are available for the entire day, it is very unlikely that these will ever be bottleneck resources. Therefore, we include all but *patient* resources.

4.3. CP model

As we already defined our problem mathematically in Chapter 2 it should be quite easy formulate it as a CP model that can then be solved by a CP solver (as done in Section 5.6). Here, we use the same notation and mostly the same inequalities as defined and described in Section 2.3.

$$\min \gamma^{\text{extback}} \frac{1}{H^{\text{unit}}} \sum_{r \in \widehat{R}} \max\left(S_r^{\text{last}} - W_r^{\text{end}}, 0\right) +$$
$$\gamma^{\text{lateness}} \frac{1}{H^{\text{unit}}} \sum_{t \in T} \sigma_t +$$
$$\gamma^{\text{scatter}} \frac{1}{H^{\text{unit}}} \sum_{r \in R^{\text{scatter}}} \varphi_r^{\text{scatter}} \left(S_r^{\text{last}} - W_r^{\text{start}} - \sum_{t \in \overline{Q}_r} |P_{t,r}| \right)$$

$$\text{disjunctive}\Big(\{(S_t + P_{t,r}^{\text{start}}, S_t + P_{t,r}^{\text{end}}) \mid t \in \overline{Q}_r\} \tag{4.9}$$
$$\cup \ \{(\widetilde{W}^{\text{start}}, W_r^{\text{start}})\} \cup \{(\widehat{W}_r^{\text{end}}, \widetilde{W}^{\text{end}})\} \tag{4.10}$$
$$\cup \ \left\{(\overline{W}_{r,w}^{\text{start}}, \overline{W}_{r,w}^{\text{end}}) \mid w \in \{0, \ldots, \omega_r - 1\}\right\}\Big) \qquad r \in R \tag{4.11}$$

$$S_t^{\text{L}} \leq S_t \wedge S_t \leq S_t^{\text{U}} \qquad t \in T \tag{4.12}$$

$$S_t + P_{t,r}^{\text{end}} \leq S_r^{\text{last}} \qquad r \in \widehat{R} \cup R^{\text{scatter}}, t \in \overline{Q}_t \tag{4.13}$$

$$W_r^{\text{start}} \leq S_r^{\text{last}} \wedge S_r^{\text{last}} \leq \widehat{W}_r^{\text{end}} \qquad r \in \widehat{R} \cup R^{\text{scatter}} \tag{4.14}$$

$$S_r^{\text{last}} = \widetilde{W}^{\text{start}} \qquad r \in R \setminus (\widehat{R} \cup R^{\text{scatter}}) \tag{4.15}$$

$$S_t - \widehat{S}_t \leq \sigma_t \qquad t \in T \tag{4.16}$$

$$S_t \in [\widetilde{W}^{\text{start}}, \ldots, \widetilde{W}^{\text{end}}] \qquad t \in T \tag{4.17}$$

$$S_r^{\text{last}} \in [\widetilde{W}^{\text{start}}, \ldots, \widetilde{W}^{\text{end}}] \qquad r \in R \tag{4.18}$$

$$\sigma_t \in [0, \ldots, \widetilde{W}^{\text{end}} - \widetilde{W}^{\text{start}}] \qquad t \in T \tag{4.19}$$

The objective function in our CP model looks exactly the same as in 2.3, hence we skip straight to the constraints. The first constraint spans the lines 4.9, 4.10 and 4.11. It uses the global constraint *disjunctive* that ensures that the defined intervals do not overlap. The first line 4.9 includes the processing times of all tasks that require the given resource to prevent any overlapping of tasks. The intervals in line 4.10 enforce that tasks do not use resources outside of their service times. Similarly, including the unavailability periods in line 4.11 forbids the usage of the resources during these times.

The next constraints are rather self-explanatory. The constraint in line 4.12 ensures that the tasks' starting times respect their lower and upper bounds. The next three constraints deal with the helper variables S_r^{last}. In restriction 4.13 it is defined to the last time resource r is needed. We need this information only for resources with extended service time windows ($\widehat{R}$) or resources for which scattering is penalized (R^{scatter}). Therefore, we can set it to an arbitrary value for the remaining resources to prune the solution space in line 4.15. Finally, line 4.16 defines σ_t for all tasks $t \in T$. The last three lines describe the domains for these variables.

5. Computational Study

In this chapter, we provide the results obtained by applying our different approaches to predefined test instances of the *I-PTPSP*.

First, in Section 5.1, we present how we created our test instances to test our approaches against and why we think that they may represent real world instances. In Section 5.2, we depict briefly which machines were used to obtain the data, and which technologies were used to develop our ideas. Next, in Section 5.3, we show the results for our greedy approach when used with the different priority functions. Then, in Section 5.4, the performances of the different enumeration strategies for our *Branch and Bound* algorithm, as well as the various lower bound calculation techniques, are shown. Finally, in Section 5.6, we compare the results of our approaches against a reference implementation.

5.1. Generating instance sets

To test our algorithms, instances that resemble real-world situations need to be generated. We will not create them from scratch but base them on *PTPSP* instances and their generated solutions as described in [57] and [66]. An *I-PTPSP* instance represents one day of the calculated *PTPSP* plan. In this section, we will describe how we did this.

5.1.1. Resources and tasks in the original *PTPSP* instances

The original *PTPSP* instances that were used to create the test instances for our *I-PTPSP* described working days of 24 hours with a resolution of $H^{\text{unit}} = 60$, i.e. $\widetilde{W} = [0, 1440)$. Beside of the patient resources which were assumed to be available the entire day, the instances defined a beam and three room resources, with a regular service window of 14 hours, an anesthetist that is available for the first 7 hours and 10 different assigned radio oncologists working in two overlapping shifts, each of them lasting approximately 9 hours.

Each of the tasks requires the beam resource and one of the three rooms with a total processing time of about 20 to 45 minutes. About 5% of the treatments also

require an anesthetist and some require the attendance of an oncologist. The exact details on how the tasks are created can be found in [57].

5.1.2. Simulating different scenarios

The *I-PTPSP* is designed to model the situation when due to some unexpected occurrences the original schedule needs to be adapted to the new scenario. We want to simulate different scenarios to test our algorithms against various circumstances. We will do this by extracting a created schedule for a given day from a *PTPSP* solution and then introducing some disturbances to simulate such a situation. In total we create 4 different scenarios. For each scenario we generate two variants of the instances where the considered time horizon starts after one and two hours after the start of the day. We decided not to create more versions that start even later as there would be only fewer tasks left leaving problem instances too small to challenge the algorithms.

The first two instance sets, sets `1` and `2`, simulate a general day to day scenario. Nothing spectacular happened, some tasks that have not started yet although scheduled and others that started at a different time as expected. The next two instance sets (`3` and `4`) model an unexpected delay in the preparation of the beam. All treatments requiring the beam could not be processed and won't until the beam is ready. Sets `5` and `6` depict an unexpected upcoming unavailability of the beam resource. This could be caused by an urgently needed maintenance work. Finally, sets `7` and `8`, simulate an unexpected upcoming unavailability of a room. This could have similar reasons as in the previous scenarios but affects a room instead of the beam resource.

One can assume that, in general, patients are not waiting for their therapy from the very start of the day. To resemble this fact, the value S_t^{L} of tasks t is set to $\widehat{S}_t - 30 * H^{\mathrm{unit}}$ to all described instances of set 1 to 8. Also, we set $H^{\mathrm{unit}} := 60$ to represent a time granularity of minutes.

Instance sets `1` and `2`: general scenario We assume that a solution for the *I-PTPSP* might be required at any time during the day. To simulate this, we suppose a start time t^{s} at which a solution is needed. All tasks t that are assumed to be already completed (i.e., $\widehat{S}_t + p_t < t^{\mathrm{s}}$) are removed from the instance. Next, all tasks t' that are currently running are also removed. Additionally, the starting times W_r^{start} of the resources r that were used by these tasks t' are set such that they become available after the currently running tasks were finished occupying them.

Furthermore, currently running or already completed tasks are disturbed to simulate real world scenarios better in the following way:

- Tasks t that are already completed according to the original schedule are reintroduced with a probability of

$$\alpha = lb + \frac{\widehat{S}_t - \widetilde{W}^{\text{start}}}{t^{\text{s}} - \widetilde{W}^{\text{start}}} \cdot (ub - lb)$$

  Value α lies between lb and ub and is the higher the later task t starts. We used 0.01 and 0.05 for the parameters lb and ub, respectively.

  Reintroducing tasks means that we assume that they were not processed for some unknown reason (e.g. a patient was late) and still need to be scheduled.

- The start time $\widehat{S}_t$ of currently running tasks t is disturbed by X, where X is a random sample from the triangular distribution with a lower limit of -30, an upper limit of 30 and a peak of 0.

We obtain our first instance sets `1` and `2` by applying these steps with $t^{\text{s}} = 60$ and $t^{\text{s}} = 120$, respectively.

Instance sets `3` and `4`: unavailable beam We want to simulate an unavailable beam. To do this, we perform the same steps as described above, first. Additionally, we assume that the beam resource is unavailable for another 60 minutes (i.e. until $t^{\text{s}} + 60$). This means that all tasks in the past that needed the beam resource could not be processed and are still needed to be scheduled.

The instance sets `3` and `4` are obtained by setting $t^{\text{s}} = 60$ and $t^{\text{s}} = 120$, respectively.

Instance sets `5` and `6`: planned unavailability period on the beam resource These sets simulate an unavailability period on the beam resource. This could be caused by a suddenly and urgently needed maintenance of the beam.

To obtain the instance sets `5` and `6`, we apply the same steps as for the sets `1` and `2` and insert an unavailability period of an hour on the beam 60 minutes after t^{s}, respectively.

Instance sets `7` and `8`: planned unavailability period on a room resource Similar to the previous instance sets we want to simulate a planned unavailability period. But this time it shall be applied to a room. The instance sets `7` and `8` are obtained with the same parameters as the sets `5` and `6` except that the unavailability period is applied to the second most intensively used room of the given day.

5.1.3. Final instance set used for our benchmarks

The steps applied in the previous section to each day of a *PTPSP* solution will lead to *I-PTPSP* instance sets of different size (i.e., different amount of tasks to be scheduled) due to natural fluctuations of the amount of treatments per day. To be able to compare the performance of our approaches to instances of different sizes, we need equally distributed instance sizes.

Therefore, we transformed multiple *PTPSP* instances and their corresponding solutions into *I-PTPSP* instance sets of type `1` to `8`. Afterwards, we divided them into 6 different bins according to their sizes: 1-10 tasks, 11-20 tasks, ... and 51-60 tasks. Finally, we picked 40 instances of each type out of every bin resulting in an instance set of 1920 instances.

5.2. Implementation Details

The greedy construction heuristic and the *Branch and Bound* algorithm are implemented in C++ 14, compiled with g++-6 (version 6.4.0)[1]. The CP model was implemented in MiniZinc v2.1.5 and solved with Gecode v5.0.0.

The results in this chapter was obtained by running the approaches on an Intel Xeon E5-2640 v4, 2.40GHz, 4GB RAM, single-threaded[2] with a time limit of 5 minutes per problem instance.

5.3. Greedy Construction Heuristic

We want to compare impacts on the performance of *Greedy Construction Heuristic* obtained by applying different priority functions priority_func(task) defined in Section 4.1.1. For this purpose, we applied them to each instance of the sets 1 to 8 as described in Section 5.1.3. They are listed here again for an easier reference.

1. *Early planned tasks:* task t has a higher priority than task t' if $\widehat{S}_t < \widehat{S}_{t'}$.
2. *Tasks that require resources with their regular time window ending early:* task t has a higher priority than task t' if t requires a resource r with a lower W_r^{end} than any of t'.
3. *Tasks that minimize scattering of the beam resource:* task t has higher priority than task t' if the idle time that emerges on the beam resource is lower if task t is scheduled next rather than task t'.

[1] We also developed a visualisation tool for the generated solutions as shown in the Appendix B

[2] More details on our grid engine are found at `https://www.ac.tuwien.ac.at/students/grid-engine/`

4. *Expensive tasks regarding their relative utilization time of the beam:* task t has higher priority than task t' if the ratio of the occupied time of the beam resource and the entire processing time (i.e., $\frac{P^{\text{end}}_{t,r^{\text{B}}} - P^{\text{start}}_{t,r^{\text{B}}}}{p_t}$) of the task t is lower than the one of task t'.
5. apply function 1 first, then 2
6. apply function 1 first, then 3
7. apply function 1 first, then 4
8. apply function 3 first, then 1
9. apply function 3 first, then 2
10. apply function 3 first, then 4

The results are shown in Table 5.1. The first column specifies the used priority function. The next four columns show the mean, standard deviation, minimum and maximum of the objective values of the obtained solutions. The last columns shows the relative amount of times a feasible solution was found.

We start by comparing the simple functions of type 1 to 4 first. Function 1 and 3 clearly outperform functions 2 and 4. Not only are the objective values of their solutions multiple times worse, they also fail to find a feasible solution for a third of the instances.

The main reason for the better performance of function 1 and 3 is that both functions focus on minimizing lateness. The first function does this directly. The third function prioritizes tasks that lead to the least scattered utilization of the beam resource. Due to the fact, that tasks in our instances have a lower bound on their allowed starting times set to 30 minutes before their initially planned starting time, scheduling them according to function 3 will automatically prioritize initially early scheduled tasks and therefore minimizing their lateness.

It is true that lateness is only the second highest weighted part of the priority function but disturbances of the initially planned schedule can easily lead to every single task beeing late, even in an optimal solution, easily adding up to a quite large objective value. While on the other side, if a resource's extended time window must be used, only a few tasks directly affect the objective value. What is more, focusing on minimizing lateness also leads to a solution that is similar the original schedule. If we assume that the original *PTPSP* solution is well scheduled, we can conclude that trying to resemble the original solution will lead to a good solution.

The main difference between function 1 and 3 is that the former sticks rigidly to the order of the tasks in the original schedule while the latter does this more flexible. If a task that was initially scheduled first cannot be processed due to one of its needed resources being blocked, function 3 will prioritize an early task that

does not depend on this blocked resource while function 1 will stick to the first task leaving the other resources unused.

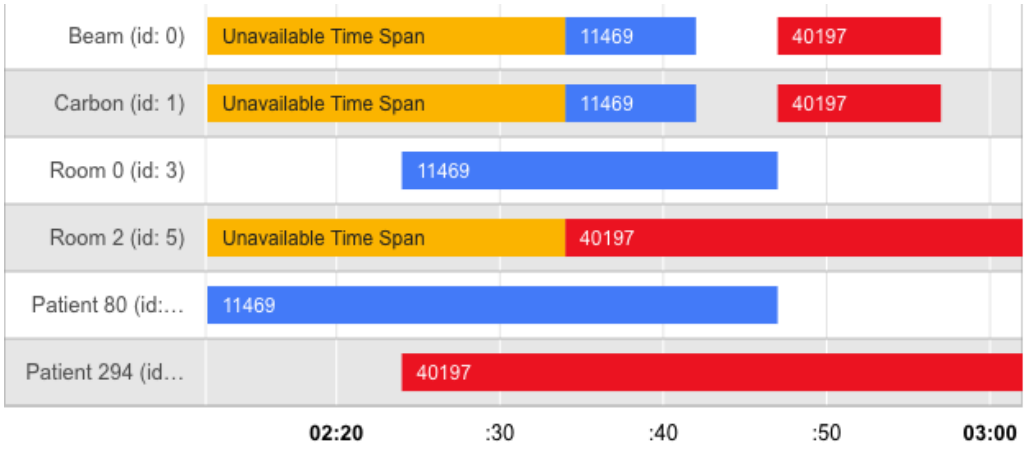
Figure 5.1 shows an example. Task `40197` has a lower $\widehat{S}$ than task `11469`. Hence, function 1 will prioritize the former task over the latter leading to the *Greedy Construction Heuristic* scheduling this one first, although, the room `2` is blocked (refer to Subfigure 5.1a). Function 3, on the other hand, will prioritize task `11469` due to it not depending on a blocked resource. In this example, it is even possible to process the entire task `11469` without delaying the task `40197` (as can bee seen in Subfigure 5.1b).

Finally, when comparing the results of applying the composed function 5-10, on can see that applying a second priority function in case of ties leads to moderate performance gains.

priority function	mean(obj)	sd(obj)	min(obj)	max(obj)	runs
1	27.197	35.502	0.004	217.337	100%
2	209.396	223.716	0.012	831.072	70.9%
3	19.806	28.338	0.004	258.059	100%
4	264.038	291.533	0.028	1619.937	67.6%
5	27.465	36.055	0.004	217.337	100%
6	26.846	34.831	0.004	217.337	100%
7	27.197	35.502	0.004	217.337	100%
8	15.146	21.191	0.004	148.671	100%
9	16.095	21.077	0.004	157.29	100%
10	19.806	28.338	0.004	258.059	100%

Table 5.1.: Comparison of the impact of the different priority functions on the performance of the Greedy Construction Heuristic. The first column specifies the used priority function. The next four columns show the mean, standard deviation, minimum and maximum of the objective values of the obtained solutions. The last columns shows the relative amount of times a feasible solution was found.

(a) Solution obtained by *Greedy Construction Heuristic* with priority function 1



(b) Solution obtained by *Greedy Construction Heuristic* with priority function 3

Figure 5.1.: Comparing the effects of priority function 1 and 3 on the results obtained by the *Greedy Construction Heuristic*. Task `40197` has a lower $\widehat{S}$ than task `11469`. Hence, function 1 will prioritize the former over the latter leading to the *Greedy Construction Heuristic* scheduling this one first although the room `2` is blocked (as can be seen in Subfigure 5.1a). Function 3, on the other hand, will prioritize task `11469` due to it not depending on a blocked resource. In this example, it is even possible to process the entire task `11469` without delaying the task `40197` (as can bee seen in Subfigure 5.1b

5.4. Branch and Bound

5.4.1. Optimality Gap

In the following sections we will often use the *optimality gap* as a measure for the performance of our algorithms. It shows the difference of the lowest lower bound of all open partial solutions in our branching tree and the objective value of the currently best known solution as relative amount of the latter. The lowest lower bound of all open partial solutions is a lower bound to the global optimum of the given problem. The currently best known objective value is an upper bound to the global optimum. Hence, the optimality gap is an upper bound to the relative difference of our current solution to the global optimum. If it is zero, that is, when the current solution is equal to the lowest lower bound of our solution tree, whe have proven, that our current solution is optimal.

Definition 5.4.1 (Optimality gap)**.** The *optimality gap* fo minimization problems is defined as

$$\text{optimality gap} = 100 \cdot \frac{ub - lb}{ub}$$

where ub denotes the upper bound (i.e. the currently best known objective value), and lb the lower bound (i.e. the currently lowest lower bound of all partial solutions).

5.4.2. Comparing the impact of different priority functions on *Branch and Bound*

In the last section, we compared the performance of the different priority functions. We saw that the functions 3, 8 and 9 provided on average the best solutions on the *Greedy Construction Heuristic*. In this section, we will have a closer look on their impact on the performance when used for the branch and bound algorithm as described in Section 4.2. Their impact on the performance on the *Branch and Bound* may slightly differ from the one on the *Greedy Construction Heuristic* as not only the values of the objective function but also the time needed to compute it influence the performance of the *Branch and Bound*.

The data in this section is obtained by applying the *Branch and Bound* with the *Most Promising First* strategy (refer to Section 4.2.1.3 for more information on this) and using the lower bound computed by the *modified Chu's lower bound* algorithm (Section 4.2.2.5) and the resource set 3 as described in Section 4.2.2.6.

Table 5.2 and Table 5.4 show the mean of the optimality gap as well as the relative number of instances for which it could be proved that the solution found was optimal (i.e., the *Branch and Bound* was not canceled prematurely due to reaching its time limit). These values are shown for each of the considered priority function. While

the first Table 5.2 summarizes this information for all instances of specific sizes (i.e., number of tasks to be scheduled), the second Table 5.4 also differentiates between the scenarios as described in Section 5.1.2. The lowest mean of the optimality gap in each row is written in bold font.

Of the three functions the function 9 tends to produce the best results. While the number solutions that could be proved to be optimal is equal for all three of them, the mean of optimality gaps reported by function 9 is almost always the lowest. This tendency is not broken, when considering the other two tables.

Table 5.3 and Table 5.5 are similarly structured. Again, the results are shown for the three considered priority functions and instance sets, either summarized only by size or size and scenario. But this time, the mean and standard deviation of the objective values as well as the mean time needed to finish (in seconds, limited by 300 seconds) are depicted. The lowest mean of the objective value in each row is written in bold font. Opposing to the previous tables, only values in the same row should be compared. It does not make sense to compare this value for different instance sets as it does not necessarily reflect the quality of the solution.

	Priority func 3		Priority func 8		Priority func 9	
# tasks	$\overline{gap}$	opt	$\overline{gap}$	opt	$\overline{gap}$	opt
1-20	0.005	99.84%	**0.002**	99.84%	0.005	99.84%
21-30	0.643	92.19%	0.932	92.19%	**0.532**	92.19%
31-40	2.476	81.88%	3.179	81.88%	**2.302**	81.88%
41-50	10.204	63.44%	10.557	63.44%	**9.69**	63.44%
51-60	29.006	33.75%	30.128	33.75%	**27.638**	33.75%

Table 5.2.: Comparing the mean of the optimality gaps and the relative number of times a solution was proven to be optimal (time limit = 300 seconds) for each of the three considered priority functions when used in the *Branch and Bound*. The size of the instances that were used is shown on the left hand side. The bold numbers mark the minimum for each row.

	Priority func 3			Priority func 8			Priority func 9		
# tasks	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$
1-20	**3.02**	3.35	1.4	**3.02**	3.35	1.5	**3.02**	3.35	1.3
21-30	6.25	5.92	29.2	6.33	6.2	30.8	**6.22**	5.82	28.2
31-40	8.56	6.89	79.9	8.79	7.51	80.2	**8.51**	6.74	77.7
41-50	12.5	12.9	135.8	12.77	13.25	135	**12.18**	12.07	135.1
51-60	27.35	26.17	219.7	28.17	26.5	217.7	**25.93**	24.25	219.7

Table 5.3.: Comparing the mean and standard deviation of the objective values as well as the mean of the time in seconds needed to finish (time limit = 300 seconds) for each of the three considered priority functions when used in the *Branch and Bound*. The size of the instances that were used is shown on the left hand size. The bold numbers mark the minimum for each row.

		Priority func 3		Priority func 8		Priority func 9	
# tasks	set	$\overline{gap}$	opt	$\overline{gap}$	opt	$\overline{gap}$	opt
1-20	1-2	**0**	100%	**0**	100%	**0**	100%
1-20	3-4	0.021	99.38%	**0.007**	99.38%	0.019	99.38%
1-20	5-6	**0**	100%	**0**	100%	**0**	100%
1-20	7-8	**0**	100%	**0**	100%	**0**	100%
21-30	1-2	**0**	100%	**0**	100%	**0**	100%
21-30	3-4	2.297	75%	3.447	75%	**1.843**	75%
21-30	5-6	**0.274**	93.75%	0.281	93.75%	0.284	93.75%
21-30	7-8	**0**	100%	**0**	100%	**0**	100%
31-40	1-2	0.114	97.5%	0.146	97.5%	**0.098**	97.5%
31-40	3-4	6.115	48.75%	8.295	48.75%	**5.656**	48.75%
31-40	5-6	3.359	82.5%	3.969	82.5%	**3.156**	82.5%
31-40	7-8	0.318	98.75%	0.306	98.75%	**0.297**	98.75%
41-50	1-2	1.965	93.75%	2.322	93.75%	**1.911**	93.75%
41-50	3-4	17.989	22.5%	20.049	22.5%	**16.93**	22.5%
41-50	5-6	16.58	51.25%	15.772	51.25%	**15.725**	51.25%
41-50	7-8	4.283	86.25%	**4.083**	86.25%	4.194	86.25%
51-60	1-2	10.925	70%	10.955	70%	**10.857**	70%
51-60	3-4	44.324	1.25%	47.259	1.25%	**41.625**	1.25%
51-60	5-6	42.216	13.75%	43.622	13.75%	**39.939**	13.75%
51-60	7-8	18.557	50%	18.677	50%	**18.132**	50%

Table 5.4.: Comparing the mean of the optimality gaps and the relative number of times a solution was proven to be optimal (time limit = 300 seconds) for each of the three considered priority functions when used in the *Branch and Bound*. The size as well as the scenarios (Section 5.1.2) that should be simulated by the used instances are depicted on the left hand size. The bold numbers mark the minimum for each row.

		Priority func 3			Priority func 8			Priority func 9		
# tasks	set	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$
1-20	1-2	**1.24**	1.26	0	**1.24**	1.26	0	**1.24**	1.26	0
1-20	3-4	**6.17**	4.05	3.4	**6.17**	4.05	3.8	**6.17**	4.05	3.2
1-20	5-6	**3.12**	2.99	0.1	**3.12**	2.99	0.1	**3.12**	2.99	0.1
1-20	7-8	**1.57**	1.7	2	**1.57**	1.7	2.3	**1.57**	1.7	2
21-30	1-2	**2.31**	2.07	0.8	**2.31**	2.07	0.8	**2.31**	2.07	0.8
21-30	3-4	12.58	6.61	88	12.91	7.28	92.5	**12.45**	6.36	86
21-30	5-6	**7**	4.67	25.6	**7**	4.67	27.4	**7**	4.68	23.5
21-30	7-8	**3.1**	2.2	2.4	**3.1**	2.2	2.5	**3.1**	2.2	2.6
31-40	1-2	**3.16**	3.15	13.4	**3.16**	3.17	16	**3.16**	3.15	13.3
31-40	3-4	16.19	4.96	206.7	16.84	6.07	207.6	**16.05**	4.62	201.1
31-40	5-6	10.65	6.21	73.7	10.93	7.13	74.9	**10.58**	6.03	70.6
31-40	7-8	**4.24**	2.57	25.8	**4.24**	2.56	22.5	**4.24**	2.56	25.7
41-50	1-2	**3.95**	5.28	35.6	4.14	6.32	33.8	**3.95**	5.35	35.6
41-50	3-4	21.16	9.32	265.9	22.09	10.4	265.6	**20.55**	8.14	263.7
41-50	5-6	18.99	16.85	175.9	18.97	16.59	172	**18.31**	15.44	175
41-50	7-8	5.9	6.02	66	5.9	5.92	68.7	**5.89**	6.07	66.1
51-60	1-2	8.54	10.09	119.4	8.64	10.42	115.7	**8.49**	10.04	118.1
51-60	3-4	44.69	22.53	299.4	46.99	23.09	300	**42**	20.87	299.2
51-60	5-6	42.5	29.48	273.7	43.4	28.49	272.8	**39.74**	26.89	273.5
51-60	7-8	13.68	14.24	186.1	13.66	13.77	182.3	**13.49**	14.02	187.9

Table 5.5.: Comparing the mean and standard deviation of the objective values as well as the mean of the time in seconds needed to finish (time limit = 300 seconds) for each of the three considered priority functions when used in the *Branch and Bound*. The size as well as the scenarios (Section 5.1.2) that should be simulated by the used instances are depicted at the left hand side. The bold numbers mark the minimum for each row.

5.4.3. Comparing the performance of the Most Promising First strategy with different "dive" frequencies

In this section we compare how different frequencies of "diving" affect the performance of the Most Promising First strategy which is explained in Section 4.2.1.3.

The data in this section is obtained by applying the *Branch and Bound* with the *Most Promising First* strategy and using the lower bound computed by the *modified Chu's lower bound* algorithm (Section 4.2.2.5), the resource set 3 as described in Section 4.2.2.6 and priority function 9 (Section 4.1.1).

Table 5.6 shows the aggregated results over all instance sets for the diving frequencies 1, 3, 4, 5, 7, 10, 25, 50, 100, 500 and 1000. The first row depicts the number of instances (of total 1920) it could be proved that the solution found was optimal. The other three rows show the means of the objective value, the optimality gap and the time needed to solve (limited with 300 seconds), respectively.

We got the best results, when we set the diving frequency to 4. Here, we obtained the best optimality gaps. While diving more often seems to find marginally better solutions (on average) but does not perform that well in proving. Diving more rarely than every 5 to 7 times on the other hand, decreases the the quality of the solutions as well as the gaps.

dive at iteration:	1	3	4	5	7	10	25	50	100	500	1000
# optimal	1491	1500	**1506**	1501	1496	1488	1499	1501	1478	1480	1460
$\overline{obj.val}$	**9.737**	9.793	9.764	9.821	9.849	9.888	9.964	10.129	10.315	10.728	11.023
$\overline{gap}$	7.081	6.867	**6.663**	6.957	7.064	7.142	7.109	7.317	7.892	8.57	9.307
$\overline{time}$	78.489	**76.82**	77.074	77.55	77.999	78.081	78.21	79.05	81.222	83.208	85.379

Table 5.6.: Comparing the performance of the Most promising first strategy with different "diving-strategies", i.e. it will dive every 1, 3, 4, 5, 7, 10, 25, 50, 100, 500, 1000 iterations. The first row depicts the number of instances (of total 1920) it could be proved that the solution found was optimal. The other three rows show the means of the objective value, the optimality gap and the time needed to solve (limited with 300 seconds), respectively.

5.4.4. Comparing the performance of the different branching strategies

In this section, we compare the performance of the different branching strategies, which are introduced in Section 4.2.1.

We obtained the data, which is provided in this section, by applying the *Branch and Bound* with the different branching strategies. We used the lower bound computed by the *modified Chu's lower bound* algorithm (Section 4.2.2.5), the resource set 3 as described in Section 4.2.2.6 and priority function 9 (Section 4.1.1). The Most Promising First strategy was applied with a diving frequency of every 4 steps. Each run is given a time limit of 300 seconds. If the algorithm did not end (and proved the optimality of the solution) until then, the process is stopped and the best solution found so far is returned.

The structure of the tables in this section is similar to the one in Section 5.4.2. The first two tables show the data aggregated over all instances of specific size, the latter two show the same data broken down to the different instance sets (which describe the different use cases, see Section 5.1.2 for more information about this). In each table, the best results of each row are written in bold font.

The Tables 5.7 and 5.9 depict the mean of the optimality gaps as well as the relative number of times the found solution was proven to be optimal. The Tables 5.8 and 5.10, on the other hand, show the mean and the standard deviation of the objective values, as well as the mean of time needed to solve the given instances.

As assumed, the sophisticated approach of the *Low Inversion First* strategy produces better results as the naive *Depth First* procedure. But still, the branching strategy *Most Promising First* outperforms both of them. Not only are its optimality gaps smaller, but the found solutions are – thanks to the "dives" – also better in almost all cases.

We also examined how strong the different branching strategies influence the time until good solutions are found. To do this, we plotted the currently best known objective values relative (in percent) to the best known value for this instance for each time point from 0 to 300 seconds.

Figure 5.2 shows the aggregated data over all instances, while Figure 5.3 breaks it down to different instance sizes. The red line shows the data for the *Branch and Bound* with the *Depth First* strategy, the green one depicts the *Low Inversion First* approach and the blue line stands for the *Most Promising First* strategy. The horizontal axis on the graph shows the time from 0 to 300 seconds, the vertical axis shows the means of the percentages of the currently best known objective values to the best known ones. Note that the y-axes are differently scaled for each Figure.

	Depth First		Low Inv. First		Most Prom. First	
# tasks	$\overline{gap}$	opt	$\overline{gap}$	opt	$\overline{gap}$	opt
1-20	**0**	99.84%	**0**	99.84%	0.004	99.84%
21-30	1.205	92.19%	1.662	92.19%	**0.525**	92.19%
31-40	7.153	82.19%	5.588	82.19%	**2.249**	82.19%
41-50	22.864	63.44%	18.233	63.44%	**9.497**	63.44%
51-60	48.463	34.38%	40.098	34.38%	**27.548**	34.38%

Table 5.7.: Comparing the mean of the optimality gaps and the relative number of times a solution was proven to be optimal (time limit = 300 seconds) for each of the labeled branching strategies. The size of the instances that were used is depicted on the left hand side. The bold numbers mark the minimum for each row.

	Depth First			Low Inv. First			Most Prom. First		
# tasks	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$
1-20	**3.02**	3.35	0.4	**3.02**	3.35	0.5	**3.02**	3.35	1.3
21-30	6.24	5.91	22.9	6.26	5.94	28.7	**6.21**	5.82	27.8
31-40	8.79	7.42	74.8	8.54	6.86	73.5	**8.49**	6.72	76.7
41-50	14.53	15.86	143.5	12.33	12.16	138.4	**12.1**	12.08	135.6
51-60	32.78	31.14	229.7	26.28	25.04	222.6	**25.69**	23.95	219.2

Table 5.8.: Comparing the mean and standard deviation of the objective values as well as the mean of the time in seconds needed to finish (time limit = 300 seconds) for each of the labeled branching strategies. The size of the instances that were used is depicted on the left hand side. The bold numbers mark the minimum for each row.

# tasks	set	Depth First		Low Inv. First		Most Prom. First	
		$\overline{gap}$	opt	$\overline{gap}$	opt	$\overline{gap}$	opt
1-20	1-2	**0**	100%	**0**	100%	**0**	100%
1-20	3-4	**0**	99.38%	**0**	99.38%	0.017	99.38%
1-20	5-6	**0**	100%	**0**	100%	**0**	100%
1-20	7-8	**0**	100%	**0**	100%	**0**	100%
21-30	1-2	**0**	100%	**0**	100%	**0**	100%
21-30	3-4	3.472	75%	5.352	75%	**1.827**	75%
21-30	5-6	1.35	93.75%	1.298	93.75%	**0.271**	93.75%
21-30	7-8	**0**	100%	**0**	100%	**0**	100%
31-40	1-2	0.564	97.5%	0.251	97.5%	**0.084**	97.5%
31-40	3-4	15.754	50%	14.832	50%	**5.433**	50%
31-40	5-6	11.457	82.5%	6.72	82.5%	**3.185**	82.5%
31-40	7-8	0.837	98.75%	0.551	98.75%	**0.296**	98.75%
41-50	1-2	5.866	93.75%	3.371	93.75%	**1.907**	93.75%
41-50	3-4	36.767	22.5%	32.598	22.5%	**16.757**	22.5%
41-50	5-6	36.543	51.25%	27.874	51.25%	**15.231**	51.25%
41-50	7-8	12.281	86.25%	9.09	86.25%	**4.091**	86.25%
51-60	1-2	24.611	71.25%	18.272	71.25%	**10.813**	71.25%
51-60	3-4	58.059	1.25%	52.037	1.25%	**41.409**	1.25%
51-60	5-6	70.814	13.75%	59.04	13.75%	**40.342**	13.75%
51-60	7-8	40.367	51.25%	31.045	51.25%	**17.628**	51.25%

Table 5.9.: Comparing the mean and standard deviation of the objective values as well as the mean of the time in seconds needed to finish (time limit = 300 seconds) for each of the labeled branching strategies. The size as well as the uses cases (Section 5.1.2) that should be simulated by the used instances are depicted on the left hand side. The bold numbers mark the minimum for each row.

		Depth First			Low Inv. First			Most Prom. First		
# tasks	set	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$	$\overline{obj}$	sd(obj)	$\overline{\text{t[s]}}$
1-20	1-2	**1.24**	1.26	0	**1.24**	1.26	0	**1.24**	1.26	0
1-20	3-4	**6.17**	4.05	1	**6.17**	4.05	1.5	**6.17**	4.05	3.3
1-20	5-6	**3.12**	2.99	0.2	**3.12**	2.99	0.1	**3.12**	2.99	0.1
1-20	7-8	**1.57**	1.7	0.3	**1.57**	1.7	0.4	**1.57**	1.7	2
21-30	1-2	**2.31**	2.07	2	**2.31**	2.07	1.4	**2.31**	2.07	0.8
21-30	3-4	**12.45**	6.41	69	12.61	6.65	83.2	**12.45**	6.37	84.6
21-30	5-6	7.13	5.02	17.4	7.01	4.7	28	**7**	4.67	23.2
21-30	7-8	**3.1**	2.2	3.3	**3.1**	2.2	2.3	**3.1**	2.2	2.4
31-40	1-2	3.17	3.23	15.7	**3.15**	3.13	15.2	**3.15**	3.14	13.1
31-40	3-4	16.35	5.5	180.3	16.08	4.76	185.1	**15.99**	4.56	199.9
31-40	5-6	11.43	7.56	83.9	10.71	6.39	74.7	**10.59**	6.06	69.8
31-40	7-8	**4.22**	2.52	19.3	**4.22**	2.52	18.9	4.24	2.56	23.9
41-50	1-2	4.41	6.85	44.7	**3.9**	5.03	34	3.94	5.25	34.8
41-50	3-4	24	13.11	256.9	20.74	7.93	262.1	**20.45**	8.19	264.2
41-50	5-6	22.83	19.66	190.9	18.81	15.61	187.5	**18.17**	15.62	177.9
41-50	7-8	6.88	9.3	81.7	5.87	6.05	70.2	**5.85**	5.96	65.4
51-60	1-2	10.27	14.6	142.5	8.58	10.43	121.2	**8.51**	10.23	116.3
51-60	3-4	48.77	26.78	300	**40.87**	19.34	300.6	41.33	20.41	299.4
51-60	5-6	54.48	33.77	284	42.71	29.47	284.8	**39.59**	26.42	275.2
51-60	7-8	17.59	19.02	192.2	**12.96**	13.96	183.7	13.34	14.13	186.1

Table 5.10.: Comparing the mean and standard deviation of the objective values as well as the mean of the time in seconds needed to finish (time limit = 300 seconds) for each of the labeled branching strategies. The size as well as the uses cases (Section 5.1.2) that should be simulated by the used instances are depicted on the left hand side. The bold numbers mark the minimum for each row.

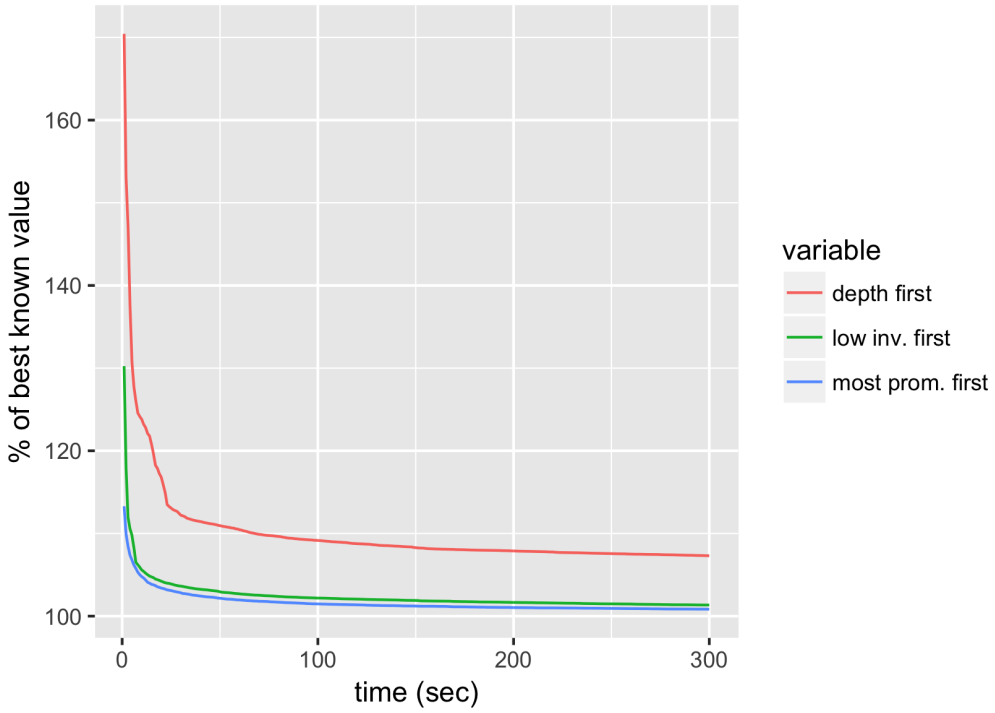
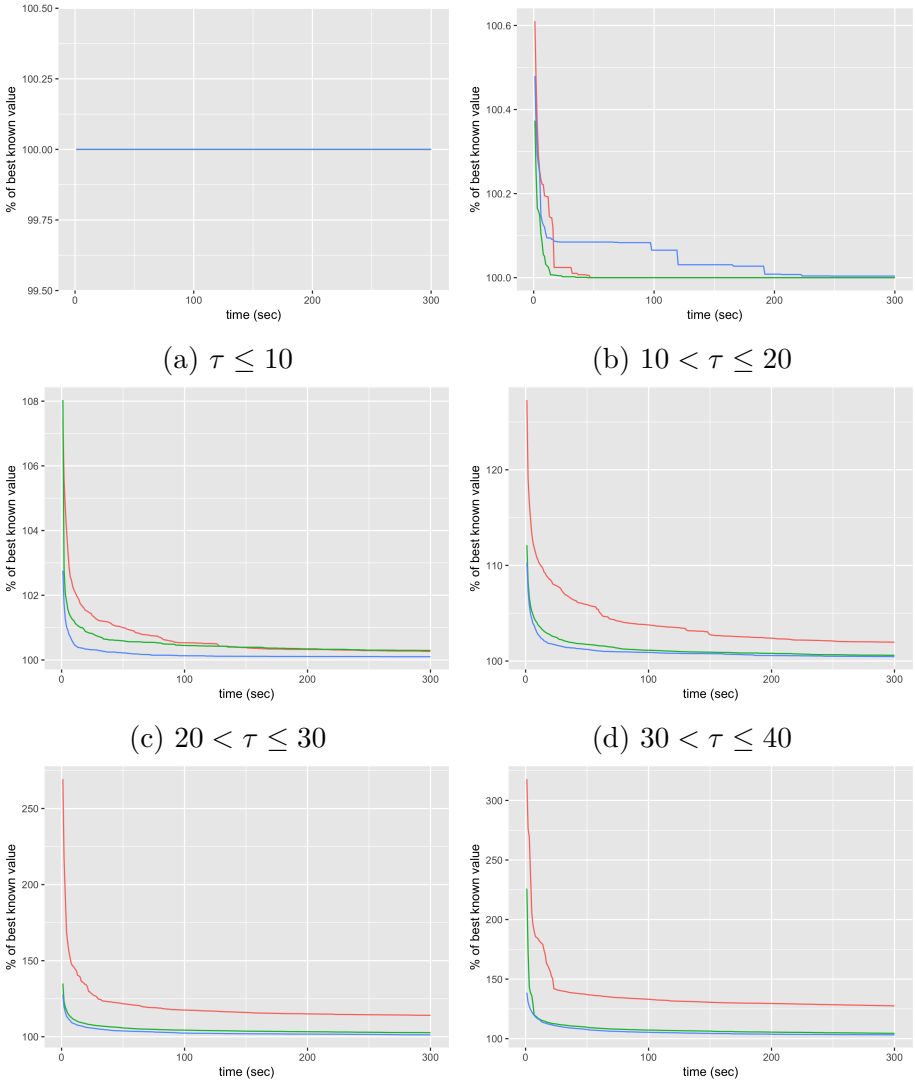


Figure 5.2.: Means of the currently best known objective values relative (in %) to the best known value for this instance, plotted by time. The red line shows the data for the *Branch and Bound* with the *Depth First* strategy, the green one depicts the *Low Inversion First* approach and the blue line stands for the *Most Promising First* strategy.



(a) $\tau \leq 10$ (b) $10 < \tau \leq 20$

(c) $20 < \tau \leq 30$ (d) $30 < \tau \leq 40$

(e) $40 < \tau \leq 50$ (f) $50 < \tau \leq 60$

Figure 5.3.: means of the currently best known objective value relative (in %) to the best known value for this instance, plotted by time, split by the size of the instances. The red line shows the data for the *Branch and Bound* with the *Depth First* strategy, the green one depicts the *Low Inversion First* approach and the blue line stands for the *Most Promising First* strategy. Note that the y-axes are differently scaled in each subfigure.

5.5. Comparing the performance of the different lower bounds when used in the *Branch and Bound*

In this section, we compare the performance of the different lower bounds, which are introduced in Section 4.2.2. The first subsection will focus on the different calculation techniques, the second will examine the differences of the various resource sets.

5.5.1. Comparing the different calculation techniques

In Section 4.2.2.3, we provided different approaches to calculate a lower bound for the lb_{lateness}-part of objective function. Here, we will examine their impacts on the performance of the *Branch and Bound* algorithm.

These are the three different approaches we tested:

no lateness The lower bound consists of the objective value of the current partial solution and the lower bound for the remaining lb_{extback} part of the objective function (as described in Section 4.2.2).

Chu This lower bound consists of the previous one plus an additional part for the remaining lb_{lateness} part of the objective function. This is calculated according to *Chu* as shown in Section 4.2.2.4.

modified Chu This lower bound is similar to the previous one, but instead of calculating the remaining lb_{lateness} part according to *Chu*, we use the theorem we developed to respect the unavailability periods in Section 4.2.2.5.

We obtained the data, which is provided in this section, by applying the *Branch and Bound* with the different lower bound calculation techniques in combination with the *Most Promising First*[3] strategy (refer to Section 4.2.1.3 for more information on this enumeration strategy), the resource set 3 as described in Section 4.2.2.6 and priority function 9 (Section 4.1.1). Each run is given a time limit of 300 seconds. If the algorithm did not end (and proved the optimality of the solution) until then, the process is stopped and the best solution found so far is returned.

Table 5.11 shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

[3]You can find the results for other combinations in the Appendix C.

When comparing the first variant with the other two, it can be clearly seen that calculating a lower bound for the lb_{lateness} part of the objective function is essential for better results.

	no lateness		Chu		modified Chu	
# tasks	$\overline{gap}$	time [s]	$\overline{gap}$	time [s]	$\overline{gap}$	time [s]
1-20	35.209	128.55	0.046	1.78	**0.004**	1.34
21-30	95.879	294.26	1.439	40.42	**0.525**	27.75
31-40	98.776	300.31	5.681	101.06	**2.249**	76.66
41-50	97.315	300.9	14.959	155.91	**9.497**	135.58
51-60	93.897	301.25	35.594	232.1	**27.548**	219.24

Table 5.11.: Comparing results for different lower bound techniques when used in combination with the *Most Promising First* strategy (refer to Section 4.2.1.3) and the resource set 3 (Section 4.2.2.6). The Table shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

Due to using lower bounds to bound unpromising nodes and as a result to minimize the number of visited nodes in a *Branch and Bound* calculation, it is also interesting to examine how well they serve this purpose. Figure 5.4 compares these indicators for problem instances with a size of 12 tasks.

Subfigure 5.4a shows a Boxplot of the total visited node for each of the three lower bound calculation variants against the depth in the node tree.

Naturally, the number of visited nodes starts growing with the depth, i.e., the number of scheduled tasks, as there are more possibilities to schedule them in different order (worst case $\frac{n!}{(n-d)!}$, where n is number of all tasks and d is the depth). But on the other hand, the more tasks are scheduled, the sharper is the lower bound and we should be able to bound more nodes which leads to less visited nodes. These effects lead to a graph that will first increase and finally decrease with a single maximum somewhere in between. Better lower bounds lead to flatter curves.

Subfigure 5.4b shows a Boxplot of the relative number of bounded nodes for each of the three lower bound calculation techniques against the depth in the node tree.

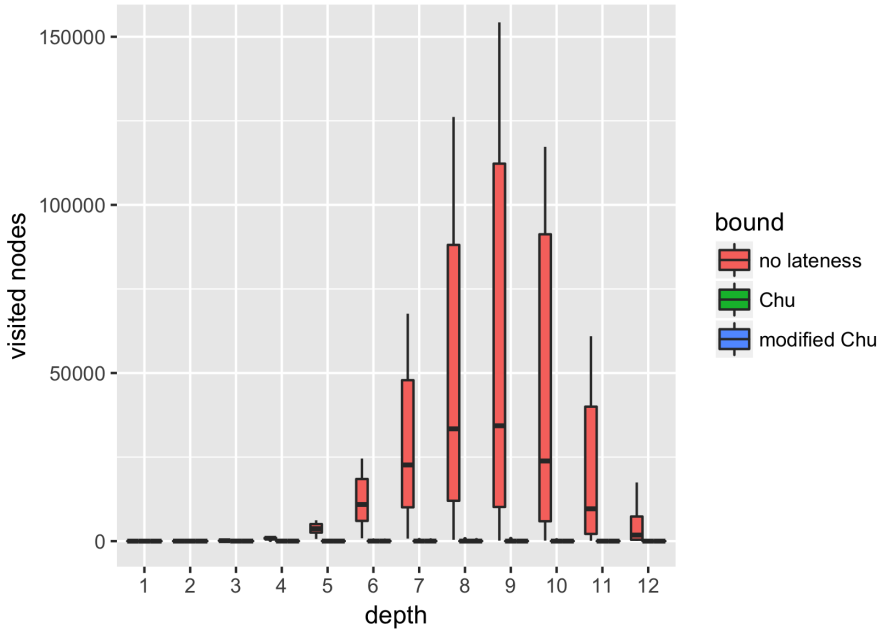
One can see that the shape of the first lower bound variant clearly differs from the one of the other two.

The *no lateness* variant produces relatively weak lower bounds which leads to only few bounded nodes in the beginning. The more tasks are scheduled, i.e. with a rising depth, the bounds become sharper and we are able to bound more and more nodes.

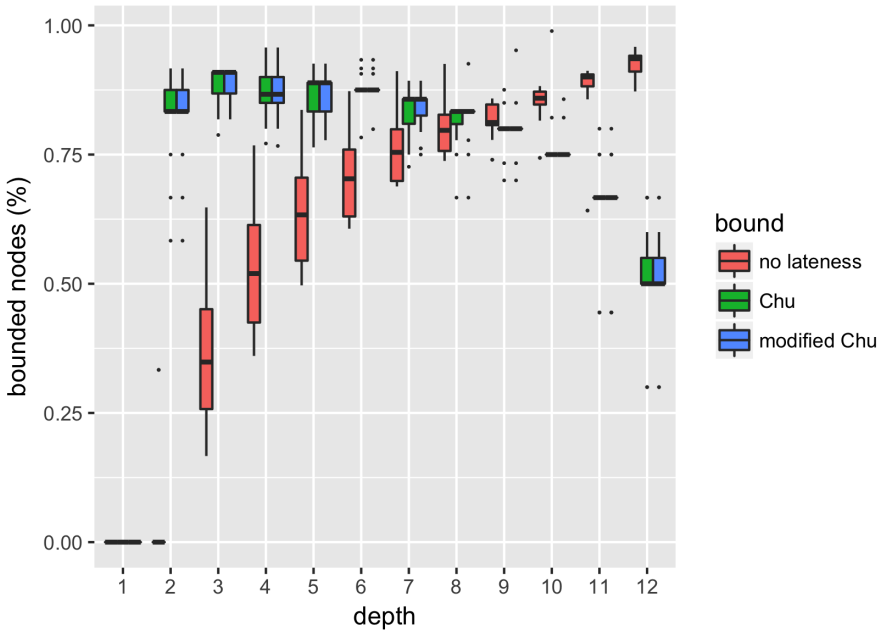
The other two techniques, on the other hand, produce quite sharp lower bounds quite early. As a result, we can bound many nodes very early. This leads to only few very strong partial solutions being available in greater depths of the node tree. This is an explanation why the first variant is able to bound more nodes at greater depths than the other two techniques. Many of the nodes, that are bounded by the *no lateness* technique at greater depths, cannot be bounded by the other two variants because their ancestors were already bounded at much lower depths in the node tree.

Finally, Figure 5.5 compares the sharpness of the three lower bounds. Here, we took the branch in the node tree that lead to the optimal solution and calculated the *sharpness* $lb_{v,d}/val*$, where $lb_{v,d}$ describes the value of the lower bound of the variant v at depth d in the selected branch and $val*$ denotes the objective value of the best solution (i.e., the last node of the selected branch). We use the lower bound to guess the value of the optimal objective value, therefore, the larger the sharpness, the better the lower bound. This *lower bound* is, precisely spoken, a lower bound of the best objective value of the sub tree in the entire node tree that has the selected node as root. Hence, we expect the sharpness of lower bounds to increase with greater depth, as the amount of possible solutions decreases or, in other words, as there is more information about how the optimal solution might look like.

This Figure shows the sharpness for the solutions of problem instances of size 30 on the y-axis against the depth on the x-axis. As previously argued, it can be clearly seen how the latter two variants of the lower bound calculation outperform the first one regarding their sharpness.



(a) Shows the total number of visitied nodes against the depth in the node tree for each of the three lower bound variants.



(b) Shows the relative number of bounded nodes against the depth in the node tree for each of the three lower bound variants

Figure 5.4.: Comparing the node trees of a *Branch and Bound* that uses the three different lower bound variants. Only problem instances with exactly 12 tasks were used for this data.

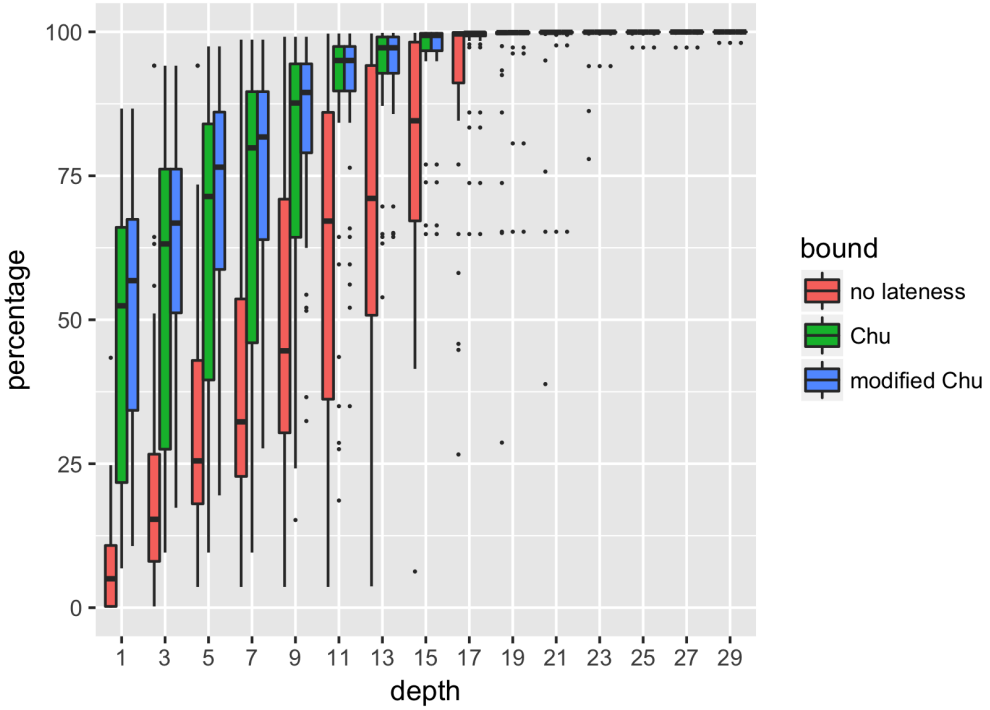


Figure 5.5.: This Figure shows the sharpness of the three lower bound calculation techniques for the solutions of problem instances of size 30.

5.5.2. Comparing the different resource sets

In Section 4.2.2.6, we described different resource sets that could be used to calculate the lower bounds. The larger the sets are, the slower is the calculation time but on the other hand the sharper is the lower bound. Now, we will examine their impacts on the performance of the *Branch and Bound*.

Here we provide a short description of these sets for a better reference. For more information on them refer to Section 4.2.2.6.

beam Computes the lower bound only for the *beam* resource.

beam & rooms Computes the lower bound for the *beam* and *room* resources.

all Computes the lower bound for all but *patient* resources.

We obtained the data, which is provided in this section, by applying the *Branch and Bound* with the different lower bound calculation techniques in combination with the *Most Promising First*[4] strategy (refer to Section 4.2.1.3 for more information on this enumeration strategy), the *modified Chu's lower bound* technique as described in Section 4.2.2.5 and priority function 9 (Section 4.1.1). Each run is given a time limit of 300 seconds. If the algorithm did not end (and proved the optimality of the solution) until then, the process is stopped and the best solution found so far is returned.

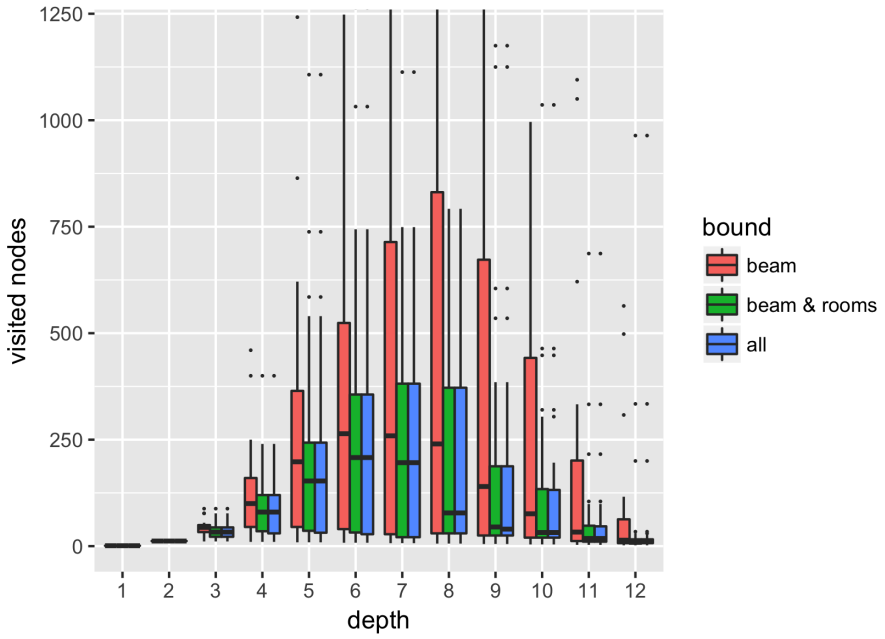
Table 5.12 shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three resource sets, aggregated by the problem size. The best gap of each row is written in bold font.

Opposing to the different calculation variants, the resource sets do not strongly differ in their impact on the performance. But still, it seems that using the largest resource set and, therefore, obtaining sharper lower bounds outweigh the additional calculation time.

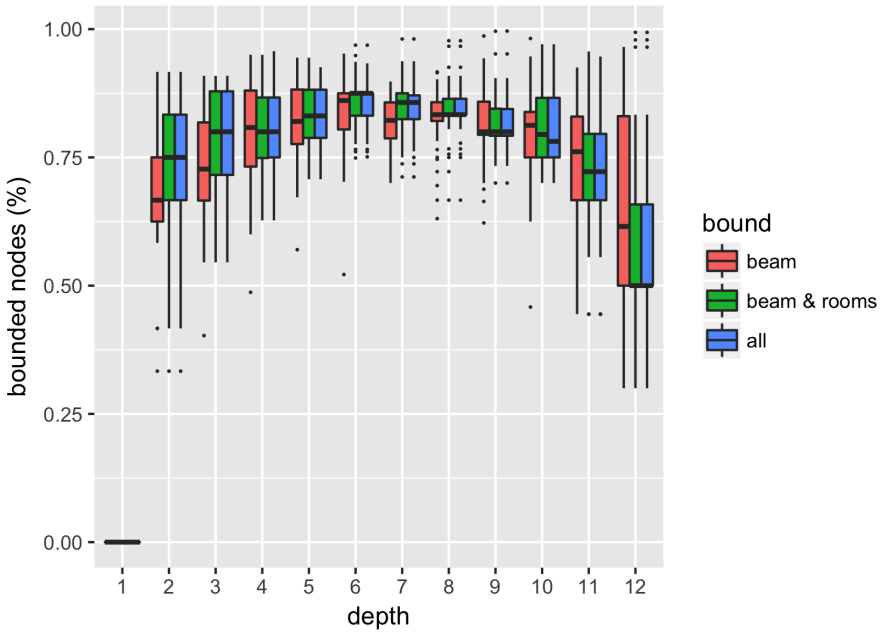
Figure 5.6 examines the node tree. Subfigure 5.6a shows the total number of visited nodes against the depth in the node tree for each of the three lower bound variants. Subfigure 5.6b depicts the relative amount of bounded nodes against the depth. Similar to the data in Table 5.12, one can see, that the different resource sets affect the node tree only slightly (at least when comparing the effects of the three different calculation variants).

Finally, Figure 5.7 shows the sharpness of the lower bounds when calculated over the three different resource sets. The sharpness of the lower bounds, when calculated only for the beam resource, is continuously smaller in the first half of the solution

[4]You can find the results for other combinations in the Appendix C



(a) Shows the total number of visitied nodes against the depth in the node tree for each of the three lower bound variants.



(b) Shows the relative number of bounded nodes against the depth in the node tree for each of the three lower bound variants

Figure 5.6.: Comparing the node trees of a *Branch and Bound* that uses different resource sets for the lower bound calculation. Only problem instances with exactly 12 tasks were used for this data.

	beam		beam & rooms		all	
# tasks	$\overline{gap}$	time [s]	$\overline{gap}$	time [s]	$\overline{gap}$	time [s]
1-20	0.09	2.61	**0.004**	1.31	**0.004**	1.34
21-30	0.653	34.32	0.559	28.25	**0.525**	27.75
31-40	3.323	95.82	2.352	77.39	**2.249**	76.66
41-50	11.204	145.63	9.606	136.47	**9.497**	135.58
51-60	29.752	226.88	27.771	219.81	**27.548**	219.24

Table 5.12.: Comparing results for different lower bound techniques when used in combination with the *Most Promising First* strategy (refer to Section 4.2.1.3) and the *modified Chu's lower bound* technique as described in Section 4.2.2.5. The Table shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

branch (refer to the previous Section for how we define sharpness). The sharpness for the latter two resource set differs only slightly.

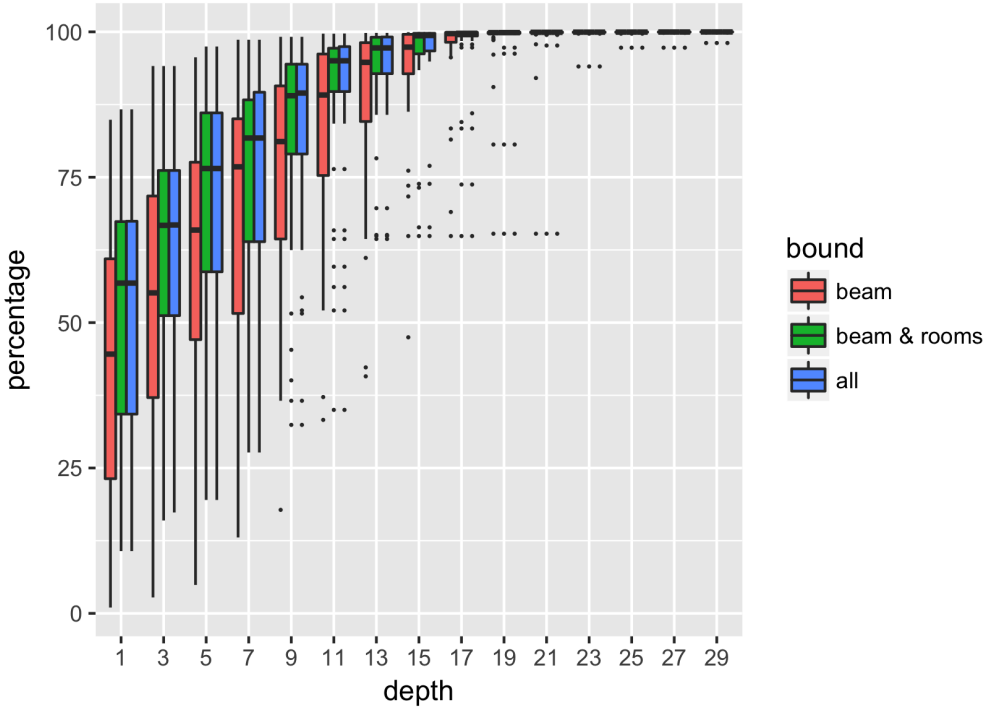


Figure 5.7.: his Figure shows the sharpness of the three lower bound resource sets for the solutions of problem instances of size 30.

5.6. Reference implementation in a CP solver

When implementing and examining new approaches to solve a specific problem it is also interesting to see how the new approach performs compared to general purpose solutions. Hence, we created a CP model for our problem in Section 4.3 that we want to solve with a CP solver.

While many solvers come with their proprietary modeling languages[5], there are also more generic modeling languages like *AMPL* [34] and MiniZinc [23] that are supported by various solvers. We decided to use MiniZinc for our implementation as it is a widely supported[6] open source modeling language actively developed at the Monash University with Data61 Decision Sciences and the University of Melbourne[7]. Furthermore, MiniZinc is a quite easily understood language[8] as it is very similar to the formal language of mathematics. Therefore, we can implement the problem with only little risk of introducing errors and use the results as an additional test for the correctness of the implementation of our *Branch and Bound* algorithm.[9]

As a solver, we decided to use *Gecode* [69], an open-source constraint solver with state-of-the-art-performance that won multiple gold medals at the MiniZinc Challenges in several years[10]. We used the MiniZinc bundle version 2.1.5 and the included *Gecode* solver version 5.0.0 to obtain the data in this section.

5.6.1. Search annotations

MiniZinc supports the concept of search annotations. These are predefined ways of telling the solver backend how to solve a given model. These annotations are not mandatory. In fact, it is not required that the solver backends implement und understand them at all [51]. A complete set of all different search annotations is found in the MiniZinc Standard Library [25]. As the number of all possible combinations of different annotations is quite limited we simply tried all of them to find the best combination (which we set to the solution variable S of the model).

There are three different types of annotations that can be set: *variable selection annotations*, *value choice annotations* and *exploration strategy annotations*. Currently the MiniZinc Standard Library defines only one exploration strategy (namely the *complete strategy*, which tells the backend to perform a complete search), we will, therefore, focus only on the former two.

[5] e.g. IBM's *CPLEX* [41] comes with its proprietary *OPL* [42]
[6] A list of all solvers can be found at[24].
[7] `http://www.minizinc.org`
[8] A good tutorial that is officially recommended can be found in [51].
[9] For details on the implementation in MiniZinc refer to the Appendix D
[10] `http://www.gecode.org/index.html`

Their impact on the *Gecode* solver was quite big as can be seen in Table 5.13. For better readability, we included only the five best and two worst combinations regarding the number of times an optimal solution was found. It is interesting that the combinations of annotations that lead to the largest number of optimal solutions can be quite bad at finding a solution at all. While the combination *occurrence-indomain_split* only found a solution to 52% of all problem instances, it was able to find an optimal solution for 41% of all instances, the highest score of all combinations of search annotations.

Annotations					
search ann	choice ann	$\overline{\text{obj}}$	$\overline{\text{time [s]}}$	sol. found	optim. found
occurrence	indomain_split	2.9927	181.4565	51.61%	40.99%
dom_w_deg	indomain_split	27.0992	185.9616	100.00%	39.64%
dom_w_deg	indomain_min	27.2321	190.1396	100.00%	38.65%
anti_first_fail	indomain_split	3.1124	191.6132	54.95%	37.66%
smallest	indomain_split	18.4388	193.1991	100.00%	37.03%
...	...	...	...	...	...
anti_first_fail	indomain_reverse_split	246.8801	265.0679	20.83%	12.24%
smallest	indomain_reverse_split	245.951	276.7355	14.58%	8.12%

Table 5.13.: Comparing the results of different search and choice annotations on *Gecode*, ordered by the relative number of times an optimum was found. The columns depict the mean of the objective values of the solutions, the mean of the time needed to find a solution (limited to 300 seconds), the relative number of times a solution was found and the relative number of times the optimum was proved. Only the best five and worst two combinations are shown.

For an easier reference, we provide the definitions (according to the *MiniZinc Standard Library* [25]) of some selected search annotations here:

Variable selection annotations

occurence choose the variable with the largest number of attached constraints

anti_first_fail choose the variable with the largest domain

smallest choose the variable with the smallest value in its domain

dom_w_deg choose the variable with largest domain, divided by the number of attached constraints weighted by how often they have caused failure

Value choice annotations

indomain assign values in ascending order

indomain_split bisect the domain, excluding the upper half first

indomain_reverse_split bisect the domain, excluding the lower half first

indomain_min assign the smallest value in the domain

indomain_max assign the largest value in the domain

Due to our problem requirements, we need to focus only on search annotations that ideally find a solution in every given instance. A comparison of the combinations that found solutions for every given problem instance can be found in Table 5.14. Again only the best five and worst two combinations are shown for a better readability.

Annotations					
search ann	choice ann	$\overline{\text{obj}}$	$\overline{\text{time [s]}}$	sol. found	optim. found
smallest	indomain_split	18.4388	193.1991	100.00%	37.03%
smallest	indomain_min	18.6163	201.0388	100.00%	34.79%
smallest	indomain	18.7217	204.9384	100.00%	33.39%
dom_w_deg	indomain_split	27.0992	185.9616	100.00%	39.64%
dom_w_deg	indomain_min	27.2321	190.1396	100.00%	38.65%
...	...	...	...	...	...
largest	indomain_max	700.2672	260.4428	100.00%	14.06%
largest	indomain_reverse_split	708.2842	257.1837	100.00%	15.05%

Table 5.14.: Comparing the results of different search and choice annotations in gecode, ordered by the mean of the objective value of the solution found. The columns depict the mean of the objective values of the solutions, the mean of the time needed to find a solution (limited to 300 seconds), the relative number of times a solution was found and the relative number of times the optimum was proved. Only combinations that found a solution in every run are considered and only the best and worst combinations are shown.

5.6.2. Comparing the performance of our *Branch and Bound* approach to the CP solver

In this section we compare the performance of our *Branch and Bound* approach versus the chosen CP solver.

On the one side, the results are obtained by applying the *Branch and Bound* with the *Most Promising First* strategy (Section 4.2.1.3), using the lower bound computed by the *modified Chu's lower bound* algorithm (Section 4.2.2.5), the resource set 3 as described in Section 4.2.2.6) and priority function 9 (Section 4.1.1). On the other side, the results for the CP approach are produced by the *Gecode* solver and the search annotations *smallest—indomain_split* as described in the previous section.

Both solvers were given exactly 5 minutes to solve each instance of our instance set as described in Section 5.1.3.

The following two tables compare the number of times the *Branch and Bound* solver performed better, equal or worse than the gecode solver. First, in Table 5.15, the total number of times an approach found a better solution than the other is shown. Second, in Table 5.16, these are split by different problem sizes and instance sets.

	BaB perf. better	equally well	*Gecode* perf. better
number of occurences	1155	763	2
rel. number of occurences	60.16%	39.74%	0.10%

Table 5.15.: Comparison of the number of times the solution of the *BaB* solver was better, equal or worse than the one of the *Gecode* solver

The next two tables are similar to the previous ones. But this time the relative quality of the solutions, i.e. $\frac{\text{objval}_{\text{BaB}}}{\text{objval}_{\text{Gecode}}}$, is compared. Again, the first Table 5.17 shows these summed up and the second Table 5.18 split by different problem sizes and instance sets.

When examining the Tables 5.16 and 5.18, it can be seen that the larger the problem instances are, the more is the *Gecode* solver outperformed by our *Branch and Bound* solver. For the largest instance sets the *Branch and Bound* solver even returned solutions that were twice as good as the solutions returned by *Gecode*.

Finally, Table 5.19 compares the mean and standard deviation of the time needed to solve the given problem instances in seconds and split by problem sizes and different instance sets. The time of execution was limited to 300 seconds.

# tasks	set	*BaB* perf. better	equally well	*Gecode* perf. better
1-20	1-2	6.88%	93.12%	0.00%
1-20	3-4	28.12%	71.88%	0.00%
1-20	5-6	28.12%	71.88%	0.00%
1-20	7-8	11.25%	88.75%	0.00%
21-30	1-2	35.00%	65.00%	0.00%
21-30	3-4	70.00%	30.00%	0.00%
21-30	5-6	80.00%	20.00%	0.00%
21-30	7-8	55.00%	45.00%	0.00%
31-40	1-2	52.50%	47.50%	0.00%
31-40	3-4	97.50%	2.50%	0.00%
31-40	5-6	92.50%	6.25%	1.25%
31-40	7-8	72.50%	27.50%	0.00%
41-50	1-2	67.50%	32.50%	0.00%
41-50	3-4	100.00%	0.00%	0.00%
41-50	5-6	100.00%	0.00%	0.00%
41-50	7-8	85.00%	15.00%	0.00%
51-60	1-2	90.00%	8.75%	1.25%
51-60	3-4	100.00%	0.00%	0.00%
51-60	5-6	100.00%	0.00%	0.00%
51-60	7-8	97.50%	2.50%	0.00%

Table 5.16.: Comparison of the number of times the solution of the *BaB* solver was better, equal or worse than the one of the *Gecode* solver — split by problem sizes and different instance sets

	$\text{objval}_{\text{BaB}}/\text{objval}_{\text{Gecode}}$
mean	0.749
median	0.843
sd	0.278

Table 5.17.: Relative comparison of the objective values, i.e. $\frac{\text{objval}_{\text{BaB}}}{\text{objval}_{\text{Gecode}}}$

Instances		$\text{objval}_{\text{BaB}}/\text{objval}_{\text{Gecode}}$		
# tasks	set	mean	median	sd
1-20	1-2	0.978	1	0.106
	3-4	0.96	1	0.076
	5-6	0.938	1	0.14
	7-8	0.959	1	0.148
21-30	1-2	0.852	1	0.266
	3-4	0.859	0.861	0.129
	5-6	0.786	0.802	0.183
	7-8	0.8	0.929	0.254
31-40	1-2	0.724	0.894	0.319
	3-4	0.762	0.754	0.125
	5-6	0.669	0.628	0.193
	7-8	0.673	0.638	0.269
41-50	1-2	0.637	0.614	0.333
	3-4	0.65	0.674	0.148
	5-6	0.491	0.469	0.195
	7-8	0.518	0.471	0.287
51-60	1-2	0.437	0.388	0.307
	3-4	0.537	0.524	0.183
	5-6	0.473	0.464	0.19
	7-8	0.423	0.361	0.229

Table 5.18.: Relative comparison of the objective values, i.e. $\frac{\text{objval}_{\text{BaB}}}{\text{objval}_{\text{Gecode}}}$ — split by problem sizes and different instance sets

Instances		*BaB*		*Gecode*	
# tasks	set	$\overline{\text{time [s]}}$	sd(time)	$\overline{\text{time [s]}}$	sd(time)
1-20	1-2	**0.02**	0.12	32.16	91.01
	3-4	**3.26**	24.74	99.48	137.9
	5-6	**0.06**	0.19	121.9	143.12
	7-8	**2.01**	23.73	51.19	111.11
21-30	1-2	**0.83**	2.88	114.67	143.14
	3-4	**84.6**	127.73	233.63	120.45
	5-6	**23.16**	73.06	266.74	85.8
	7-8	**2.42**	7.6	178.34	146.43
31-40	1-2	**13.08**	48.92	165.98	147.77
	3-4	**199.92**	124.54	296.47	31.58
	5-6	**69.79**	118.32	293.87	38.56
	7-8	**23.87**	63.27	237.31	120.31
41-50	1-2	**34.78**	85.52	214.56	131.33
	3-4	**264.22**	87.82	300	0.01
	5-6	**177.91**	136.22	300	0.01
	7-8	**65.42**	111.62	255.6	106.41
51-60	1-2	**116.25**	131.05	276.53	79.26
	3-4	**299.37**	15.93	300	0
	5-6	**275.23**	78.81	300	0.01
	7-8	**186.11**	133.71	293.62	40.73

Table 5.19.: Comparing the mean and standard deviation of the time needed to solve the given problem instances in seconds and split by problem sizes and different instance sets. The time of execution was limited to 300 seconds. The shorter time in each row is written in bold font.

5.7. Summarizing our results and comparing our three approaches

In this section, we want to summarize the obtained results and compare the performance of our three methods with their individually best configuration.

First, we use the *Greedy Construction Heuristic* in combination with the priority function 8 (as described in Section 4.1.1). Second, we apply the *Branch and Bound* approach with the *Most Promising First* strategy and a diving frequency of 4 (Section 4.2.1.3), the lower bound calculation technique *modified Chu's lower bound* (Section 4.2.2.5), the resource set 3 (Section 4.2.2.6), and the priority function 9 (Section 4.1.1). Last but not least, we take the results of the CP solver that are produced by *Gecode* using the search annotation combination *smallest* and *indomain_split* as described in Section 5.6.1.

We assumed 5 minutes of waiting time for the recreation of a treatment schedule to be an acceptable time span. Hence, every solver was given 5 minutes to solve each problem instance of our instance set that we described in Section 5.1.3. If this limit was reached, the solve process was stopped and the best solution found so far returned. Of course, the Greedy Construction Heuristic never reached this time limit as no solve process even exceeded 15 milliseconds.

The following two tables compare the mean of the objective values for each solution obtained by the three solvers. Furthermore, the standard deviation of the objective values is depicted. In addition, the results for the Branch and Bound approach also contain the mean of the optimality gap (refer to Section 5.4.1 for the definition we used). While the first Table 5.20 shows this information aggregated over problem instances of similar size, the second Table 5.21 differentiates also between different scenarios as described in Section 5.1.2.

While all three approaches deliver reasonably good results for smaller instances, the Branch and Bound solver clearly outperforms the other two on larger instances. Interestingly, when comparing the Greedy Construction Heuristic and the Gecode solver, on can see that the general purpose solution performs slightly better on small instances but is clearly outrun by our construction heuristic on the larger instances.

	Greedy CH		BaB			Gecode	
# tasks	$\overline{obj}$	sd(obj)	$\overline{obj}$	sd(obj)	$\overline{gap}$	$\overline{obj}$	sd(obj)
1-20	3.94	4.58	**3.02**	3.35	0	3.39	4.06
21-30	8.78	8.39	**6.21**	5.82	0.52	8.37	8.44
31-40	13.06	10.39	**8.49**	6.72	2.25	13.68	12.72
41-50	20.19	19.58	**12.1**	12.08	9.5	25.54	28.33
51-60	40.96	33.27	**25.69**	23.95	27.55	56.27	47.47

Table 5.20.: Comparing the the mean of the objective values obtained when solving with our three presented solvers: the *Greedy Construction Heuristic*, the *Branch and Bound* approach and the CP solver *Gecode* (each of them had a time limit of 300 seconds per instance). Additionally, the standard deviation of the objective value is provided for all three solvers and the Branch and Bound section also includes the mean of the optimality gaps. The size of the instances that were used is depicted on the left hand side. The bold numbers mark the minimum of the mean of the objective values for each row.

Instances		Greedy CH		BaB			Gecode	
# tasks	set	$\overline{obj}$	sd(obj)	$\overline{obj}$	sd(obj)	$\overline{gap}$	$\overline{obj}$	sd(obj)
1-20	1-2	1.73	1.81	**1.24**	1.26	0	1.34	1.5
1-20	3-4	7.47	5.03	**6.17**	4.05	0.02	6.65	4.75
1-20	5-6	4.26	5.01	**3.12**	2.99	0	3.68	4.24
1-20	7-8	2.31	3.23	**1.57**	1.7	0	1.87	2.45
21-30	1-2	3.44	3.09	**2.31**	2.07	0	3.17	2.96
21-30	3-4	16.28	9.09	**12.45**	6.37	1.83	15.67	9.69
21-30	5-6	10.75	8.51	**7**	4.67	0.27	9.96	8.41
21-30	7-8	4.64	3.5	**3.1**	2.2	0	4.69	4.1
31-40	1-2	4.94	4.92	**3.15**	3.14	0.08	6.79	12.77
31-40	3-4	22.47	7.46	**15.99**	4.56	5.43	22.22	10.21
31-40	5-6	17.9	10.83	**10.59**	6.06	3.18	17.91	12.98
31-40	7-8	6.93	4.55	**4.24**	2.56	0.3	7.79	6.47
41-50	1-2	7.04	11.37	**3.94**	5.25	1.91	7.94	9.27
41-50	3-4	31.9	16.31	**20.45**	8.19	16.76	34	18.39
41-50	5-6	30.65	21.76	**18.17**	15.62	15.23	42.86	37.49
41-50	7-8	11.17	13.14	**5.85**	5.96	4.09	17.36	25.55
51-60	1-2	14.68	15.57	**8.51**	10.23	10.81	21.85	20.68
51-60	3-4	60.11	25.33	**41.33**	20.41	41.41	84.37	46.7
51-60	5-6	65.7	33.74	**39.59**	26.42	40.34	86.08	49.27
51-60	7-8	23.34	21.01	**13.34**	14.13	17.63	32.77	24.59

Table 5.21.: Comparing the the mean of the objective values obtained when solving with our three presented solvers: the *Greedy Construction Heuristic*, the *Branch and Bound* approach and the CP solver *Gecode* (each of them had a time limit of 300 seconds per instance). Additionally, the standard deviation of the objective value is provided for all three solvers and the Branch and Bound section also includes the mean of the optimality gaps. The size of the instances that were used is depicted on the left hand side. The bold numbers mark the minimum of the mean of the objective values for each row.

6. Conclusion and Outlook

In this thesis we tackled the *I-PTPSP*, a particle therapy scheduling problem with a time horizon of a single day and already assigned starting times for the pending treatments. Due to the many uncertainties when creating a treatment schedule, it frequently happens that the original schedule must be dynamically altered to adapt to the current situation like a suddenly unavailable treatment room, a no-show of a patient or a delay of a treatment device. Solving the new problem within a short time frame is crucial to minimize the negative effects of such incidents on the throughput and the running costs of the facility.

First, we introduced a greedy construction heuristic and proposed several priority functions that could be used to improve the obtained solutions. Then, we presented a Branch and Bound algorithm, capable of solving the problem exactly. We provided three different implementations of the priority queue, namely a Depth First Variant, a Low Inversion Number First strategy and a Most Promising First approach. The first one combines the ideas of depth first enumeration of the node tree of the Branch and Bound algorithm with the greedy construction heuristic and its priority functions, we introduced before. The second approach was to enumerate the schedules that most resemble the original one first. The idea was that, first, a slightly changed scenario should require only slightly changed solutions and second, as one of the main objective is to minimize the lateness of the tasks, their order, according to their starting times, should not be mixed up completely. To do this, we used a smart encoding of the "distance" to the original solution that involved the use of inversion numbers. Finally, we applied the Most Promising First, or sometimes also referred to as Best First, strategy.

Furthermore, a sharp lower bound was developed in several steps. First, we used a simple approach to calculate a lower bound for the extended service time part of the objective function. Additionally, the calculated lower bounds were further refined by considering the structural characteristics of the remaining tasks. Then, we adapted Chu's proposal of a lower bound for total tardiness scheduling problems [19] to work in our case as well. Afterwards, we extended it even more to also support unavailability periods as such. Finally, we studied three different resource sets to be used for the lower bound computation.

Lastly, we compared the performance of the different proposed variants of our Branch and Bound algorithm. For this purpose, a large instance set was created that resembled real world scenarios. In the end, we compared the results of our

best Branch and Bound variant against the greedy construction heuristic and the ones obtained by a state-of-the-art CP solver. It was shown that the Branch and Bound method clearly outperformed the other two approaches. Interestingly, also the construction heuristic performed better than the CP solver on the largest problem instances.

In future work, the *I-PTPSP* should be extended to meet further possible requirements. First, beam changing times when switching beetween rooms and proton and carbon therapies may be included. Furthermore, the objective function will be extended to penalize multiple changes of the type of beam. Second, the problem formulation can be further extended by precedence graphs for different tasks. This allows to depict multiple tasks that belong to the same treatment of a single patient and that need to be completed in a specified order. This would allow to model tasks that are loosely coupled to the "main" task and possibly do not require bottleneck resources like the beam but still affect the overall schedule, e.g. an additional image processing. Furthermore, minimum and maximum time lag constraints beetween these connected treatments will be introduced. For instance, the minimum time lags could reflect the required transfer times between rooms within the facility. The maximum time lags, on the other hand, could be used to prevent schedules that require patients to wait for their succeeding appointment for an exceptionally long time. Moreover, the objective function will be extended, to consider these maximum time lag constraints that result from tasks that already completed and affect tasks that still need to be scheduled.

In addition to that, the proposed Branch and Bound algorithm needs to be extended with a type of domination strategy to diminish symmetries in nodes of the solution tree that result from introducing the mentioned precedence graphs.

Bibliography

[1] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Manage. Sci.*, 34(3):391–401, Mar. 1988.

[2] M. Al Bashir, Z. Islam, and A. K. M. Masud. Approach to job-shop scheduling problem using rule extraction neural network model. *Global Journal of Computer Science and Technology*, 11(7), 2011.

[3] A. Albrecht. Two simulated annealing-based heuristics for the job shop scheduling problem. *European Journal of Operational Research*, 118:524–548, 1999.

[4] J. Alcaraz and C. Maroto. A robust genetic algorithm for resource allocation in project scheduling. *Annals of Operations Research*, 102(1):83–109, Feb 2001.

[5] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[6] C. Artigues and D. Feillet. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals of Operations Research*, 159(1):135–159, Mar 2008.

[7] T. Baar, P. Brucker, and S. Knust. *Tabu Search Algorithms and Lower Bounds for the Resource-Constrained Project Scheduling Problem*, pages 1–18. Springer US, Boston, MA, 1999.

[8] K. R. Baker and Z.-S. Su. Sequencing with due-dates and early start times to minimize maximum tardiness. *Naval Research Logistics*, 21(1):171–176, 1974.

[9] N. Beaulieu, D. E. Bloom, L. R. Bloom, and R. M. Stein. Breakaway: The global burden of cancer – challenges and opportunities. *The Economist Group*, 2009.

[10] C. E. Bell and K. Park. Solving resource-constrained project scheduling problems by a* search. *Naval Research Logistics (NRL)*, 37(1):61–84, 1990.

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

[12] F. F. Boctor. Resource-constrained project scheduling by simulated annealing. *International Journal of Production Research*, 34(8):2335–2351, 1996.

[13] K. Bouleimen and H. Lecocq. A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. *European Journal of Operational Research*, 149(2):268 – 281, 2003.

[14] P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods, 1999.

[15] P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49:107–127, 03. 1994.

[16] B. Cardoen, E. Demeulemeester, and J. Beliën. Operating room planning and scheduling: A literature review. *European Journal of Operational Research*, 201(3):921 – 932, 2010.

[17] T. Cayirli and E. Veral. Outpatient scheduling in health care: A review of literature. *Production and Operations Management*, 12:519 – 549, 01. 2009.

[18] R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms—i. representation. *Computers And Industrial Engineering*, 30(4):983 – 997, 1996.

[19] C. Chu. A branch-and-bound algorithm to minimize total tardiness with different release dates. *Naval Research Logistics (NRL)*, 39(2):265–283, 1992.

[20] J. Clausen. Branch and bound algorithms – principles and examples, 1999.

[21] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3 edition, 2009.

[22] G. Cramer. *Introduction a l'analyse des lignes courbes algébriques*. chez les freres Cramer et Cl. Philibert, 1750.

[23] C. Data 61. Specification of minizinc. `http://www.minizinc.org/doc-lib/minizinc-spec.pdf`.

[24] C. Data 61. Flatzinc implementations. `http://www.minizinc.org/software.html#flatzinc`, 06. 2018.

[25] C. Data 61. Minizinc documentation - standard library. `http://www.minizinc.org/doc-lib/doc.html`, 06. 2018.

[26] B. de Reyck and W. Herroelen. A branch-and-bound procedure for the resource-constrained project scheduling problem with generalized precedence relations. *European Journal of Operational Research*, 111(1):152 – 174, 1998.

[27] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41(3):231–252, Sep 1993.

[28] E. Demeulemeester and W. Herroelen. A branch-and-bound procedure for the multiple resource-constrained project scheduling problem. *Management Science*, 38(12):1803–1818, 1992.

[29] E. L. Demeulemeester and W. S. Herroelen. A branch-and-bound procedure for the generalized resource-constrained project scheduling problem. *Operations Research*, 45(2):201–212, 1997.

[30] J. Du and J. Y.-T. Leung. Minimizing total tardiness on one machine is np-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.

[31] S. A. Erdogan and B. T. Denton. Surgery planning and scheduling. *Wiley Encyclopedia of Operations Research and Management Science*, 2011.

[32] J. Ferlay, I. Soerjomataram, and E. M. Cancer incidence, mortality and prevalence worldwide in 2012. `http://globocan.iarc.fr`, 06. 2018.

[33] K. Foehl. Zyklotron prinzipskizze. Wikimedia, `https://en.wikipedia.org/wiki/File:Zyklotron_Prinzipskizze02.svg`, 05. 2009.

[34] R. Fourer, D. M. Gay, and B. W. Kernigham. A modeling language for mathematical programming. *Management Science*, 36:519–554, 1990.

[35] S. Fukumoto. Cyclotron versus synchrotron for proton beam therapy. In *Proceedings, 14th International Conference on Cyclotrons and their Applications (CYCLOTRONS 95), Faure, Cape Town, South Africa, 8-13 Oct 1995*, page J13, 1995.

[36] J. Gromicho, J. Van Hoorn, F. Saldanha-da Gama, and G. Timmer. Solving the job-shop scheduling problem optimally by dynamic programming. *Applied Physics Letters*, 39, 12. 2012.

[37] D. Gupta and B. Denton. Appointment scheduling in health care: Challenges and opportunities. *IIE Transactions*, 40(9):800–819, 2008.

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

[39] M. Horn, G. R. Raidl, and C. Blum. Job sequencing with one common and multiple secondary resources: A problem motivated from particle therapy for cancer treatment. In G. Giuffrida, G. Nicosia, P. Pardalos, and R. Umeton, editors, *MOD 2017: Machine Learning, Optimization, and Big Data – Third International Conference*, volume 10710 of *LNCS*, pages 506–518. Springer, 2017.

[40] M. Horn, G. R. Raidl, and E. Rönnberg. An a* algorithm for solving a prize-collecting sequencing problem with one common and multiple secondary resources and time windows. In *Submitted to PATAT 2018*, Vienna, Austria, 2018.

[41] IBM. *IBM ILOG CPLEX Optimization Studio CP Optimizer User's Manual*. IBM Corporation, version 12, release 8 edition, 2017.

[42] IBM. *IBM ILOG CPLEX Optimization Studio OPL Language Reference Manual*. IBM Corporation, version 12, release 7 edition, 2017.

[43] D. Knuth. *The art of computer programming*, volume 3. Addison-Wesley, 1973.

[44] R. Kolisch and S. Hartmann. *Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis*, pages 147–178. Springer US, Boston, MA, 1999.

[45] C. Koulamas. The single-machine total tardiness scheduling problem: Review and extensions. *European Journal of Operational Research*, 202(1):1–7, 2010.

[46] B. J. Lageweg, J. K. Lenstra, and A. H. G. R. Kan. Job-shop scheduling by implicit enumeration. *Management Science*, 24(4):441–450, 1977.

[47] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3):497–520, 1960.

[48] D. H. Lehmer. Teaching combinatorial tricks to a computer. *Proc. Sympos. Appl. Math.*, 10:179–193, 1960.

[49] H. Mannila. Measures of presortedness and optimal sorting algorithms. *IEEE Transactions on Computers*, C-34(4):318–325, 1985.

[50] R. Mantaci and F. Rokotondrajao. A permutation representation that knows what eulerian means. *Discrete Mathematics and Theoretical COmputer Science*, 4:101–108, 2001.

[51] K. Marriott and P. J. Stuckey. A minizinc tutorial. `http://www.minizinc.org/downloads/doc-latest/minizinc-tute.pdf`.

[52] P. Martin and D. B. Shmoys. A new approach to computing optimal schedules for the job-shop scheduling problem. In *Proceedings of the 5th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 389–403, 1996.

[53] J. Maschler, T. Hackl, M. Riedler, and G. R. Raidl. An enhanced iterated greedy metaheuristic for the particle therapy patient scheduling problem. In *Proceedings of the 12th Metaheuristics International Conference*, pages 465–474, 2017.

[54] J. Maschler and G. R. Raidl. Multivalued decision diagrams for a prize-collecting sequencing problem. In *PATAT 2018: Proceedings of the 12th International Conference of the Practice and Theory of Automated Timetabling*, Vienna, Austria, 2018. to appear.

[55] J. Maschler and G. R. Raidl. Particle Therapy Patient Scheduling with Limited Starting Time Variations of Daily Treatments. *International Transactions in Operational Research*, to appear. technical report available at https://www.ac.tuwien.ac.at/files/tr/ac-tr-18-005.pdf.

[56] J. Maschler, M. Riedler, and G. R. Raidl. Particle therapy patient scheduling: Time estimation for scheduling sets of treatments. In *Computer Aided Systems Theory – EUROCAST 2017*, pages 364–372. Springer International Publishing, 2018.

[57] J. Maschler, M. Riedler, M. Stock, and G. R. Raidl. Particle therapy patient scheduling: First heuristic approaches. In *PATAT 2016: Proceedings of the 11th International Conference of the Practice and Theory of Automated Timetabling*, pages 223–244, Udine, Italy, 2016.

[58] M. Mastrolilli and O. Svensson. Hardness of approximating flow and job shop scheduling problems. *J. ACM*, 58(5):20:1–20:32, Oct. 2011.

[59] D. C. Mattfeld and C. Bierwirth. An efficient genetic algorithm for job shop scheduling with tardiness objectives. *European Journal of Operational Research*, 155(3):616–630, 2004.

[60] J. H. May, W. E. Spangler, D. P. Strum, and L. G. Vargas. The surgical scheduling problem: Current research and future opportunities. *Production and Operations Management*, 20(3):392–405, 2011.

[61] Mosby. *Mosby's Medical Dictionary*, volume 9. Elsevier, 2009.

[62] T. Nazareth, S. Verma, S. Bhattacharya, and A. Bagchi. The multiple resource constrained project scheduling problem: A breadth-first approach. *European Journal of Operational Research*, 112(2):347 – 366, 1999.

[63] K. Nonobe and T. Ibaraki. *Formulation and Tabu Search Algorithm for the Resource Constrained Project Scheduling Problem*, pages 557–588. Springer US, Boston, MA, 2002.

[64] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. *Manage. Sci.*, 42(6):797–813, June 1996.

[65] K. Peach, P. Wilson, and B. Jones. Accelerator science in medical physics. *The British Journal of Radiology*, 84:4–10, 2011.

[66] M. Riedler, T. Jatschka, J. Maschler, and G. R. Raidl. An iterative time-bucket refinement algorithm for a high-resolution resource-constrained project scheduling problem. *International Transactions in Operational Research*, 2017.

[67] A. S. Jain and S. Meeran. Job-shop scheduling using neural networks. *International Journal of Production Research*, 36, 11. 1998.

[68] L. Schrage. A proof of the optimality of the shortest remaining processing time discipline. *Operations Research*, 16(3):687–690, 1968.

[69] C. Schulte, G. Tack, and M. Z. Lagerkvist. *Modeling and Programming with Gecode*. Gecode, 2018.

[70] J. Segen. *The Dictionary of Modern Medicine*. CRC Press, 2012.

[71] J. M. Shippers. Cyclotrons for particle therapy. *CERN Yellow Reports: School Proceedings*, 2017.

[72] B. Stewart, C. P. Wild, et al. World cancer report 2014. *World*, 2015.

[73] J. P. Stinson, E. W. Davis, and B. M. Khumawala. Multiple resource–constrained scheduling using branch and bound. *A I I E Transactions*, 10(3):252–259, 1978.

[74] D.-H. Tran, M.-Y. Cheng, and M.-T. Cao. Solving resource-constrained project scheduling problems using hybrid artificial bee colony with differential evolution. *Journal of Computing in Civil Engineering*, 30(4):04015065, 2016.

[75] S. I. K. Universitätsklinikum Schleswig-Holstein. Image: Linearbeschleuniger campus kiel. Wikimedia, `https://de.wikipedia.org/wiki/Datei:Linearbeschleuniger_Kiel.jpg`, 06 2017.

[76] P. J. M. van Laarhoven, E. H. L. Aarts, and J. K. Lenstra. Job shop scheduling by simulated annealing. *Oper. Res.*, 40(1):113–125, Jan. 1992.

[77] J. Vitter and P. Flajolet. Average-case analysis of algorithms and data structures. In J. van Leeuwen, editor, *Algorithms and Complexity*, chapter 3.1, page 459. Elsevier., 1990.

[78] T. Watanabe, H. Tokumaru, and Y. Hashimoto. Job-shop scheduling using neural networks. *Control Engineering Practice*, 1(6):957 – 961, 1993.

[79] WHO. Cancer key facts. `http://www.who.int/news-room/fact-sheets/detail/cancer`, 06. 2018.

A. Symbols

$\varphi_r^{\text{scatter}}$	priority value for avoiding scattering of resource $r \in R^{\text{scatter}}$
H^{unit}	time resolution for an hour
γ^{extback}	weight for the penalization of the use of extended service time windows
γ^{lateness}	weight for the penalization of tasks that are delayed from their initial starting time
γ^{scatter}	weight for the penalization of an scattered schedule
n_R	number of resources
n_T	number of tasks
R	set of all resources
R^{scatter}	subset of resources for which scattering should be avoided
$\widehat{R}$	subset of resources having an extended service time window
r^{B}	resource representing the beam
S	represents a solution, i.e. a schedule
S_t	starting time of task t in the solution S
S_r^{first}	the first time when resource r is used
S_t^{L}	lower bound for the starting time of task t
S_r^{last}	the last time when resource r is used
S^{partial}	represents a partial solution, i.e. a schedule that does not define starting times for every task yet
S_t^{U}	upper bound for the starting time of task t
$\widehat{S}_t$	initial starting time of task t
σ_t	the amount of time task t is delayed from its initial starting time
T	set of all tasks
$P_{t,r}$	the time interval when task t requires resource r relative to the task's starting time
$P_{t,r}^{\text{end}}$	time until which task t requires resource r relative to its starting time

$P_{t,r}^{\text{start}}$	time from which on task t requires resource r relative to its starting time
p_t	entire processing time of task t
Q_t	set of resources required by task t
$\overline{Q}_r$	set of tasks that require resource r
W_r	service time window of resource r
W_r^{end}	end of the service time window of resource r
W_r^{start}	start of the service time window of resource r
$\widetilde{W}$	fundamental opening time interval
$\widetilde{W}^{\text{end}}$	end time of the fundamental opening time interval
$\widetilde{W}^{\text{start}}$	start time of the fundamental opening time interval
$\overline{W}_r$	unavailability periods of resource r
$\overline{W}_{r,w}^{\text{end}}$	end of the w-th unavailability period of resource r
$\overline{W}_{r,w}^{\text{start}}$	start of the w-th unavailability period of resource r
$\widehat{W}_r$	extended service time window of resource r
$\widehat{W}_r^{\text{end}}$	end of the extended service time window of resource r
$\widehat{W}_r^{\text{start}}$	start of the extended service time window of resource r

B. Tool for the visualisation of solutions

We created a tool that visualizes our *I-PTPSP* solutions for a more pleasant workflow.

The *Visualisation Tool* basically consists of two sections. The first section shows all the details that are preceded by number signs (#) in the JSON files. These usually contain information about which parameters were used to create the instance and solution, the objective value, the weight parameters of the objective function and so on.

The second section depicts the resource assignment of the various tasks in a timeline chart as shown in Figure B.1. Additionally, unavailability periods and extended time spans of the resources are shown and further information on the tasks can be obtained from the related tooltips. Furthermore, a range filter is located at the bottom. This allows the user to zoom into the chart and is especially useful for solutions with many tasks that are spread over the entire day.

This tool is developed using web technologies to allow for platform independent usage. Hence, it is written in *HTML*, *CSS* and *Javascript*. Especially, it uses the *Google Charts* libraries[1].

[1] `https://developers.google.com/chart/`

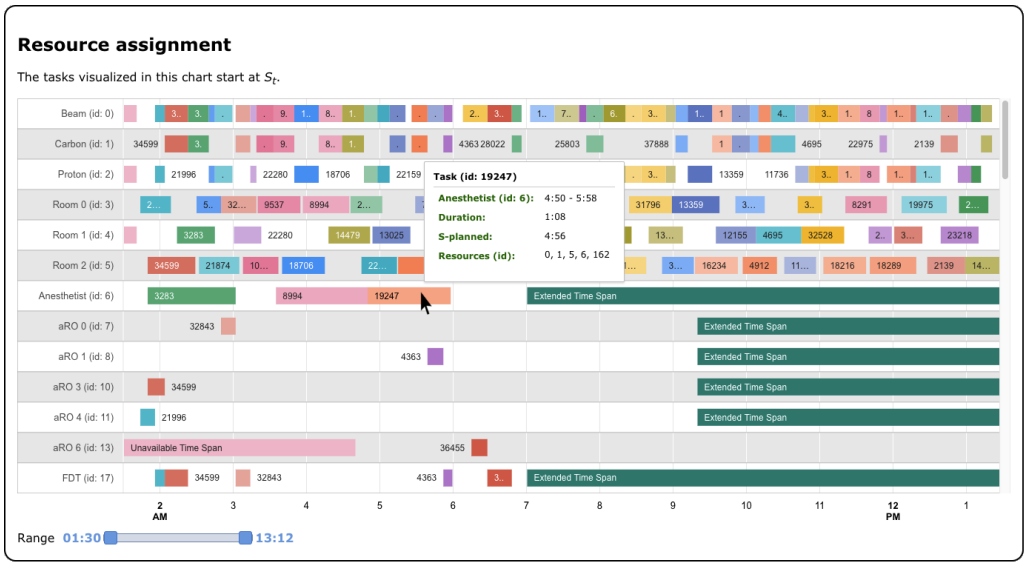


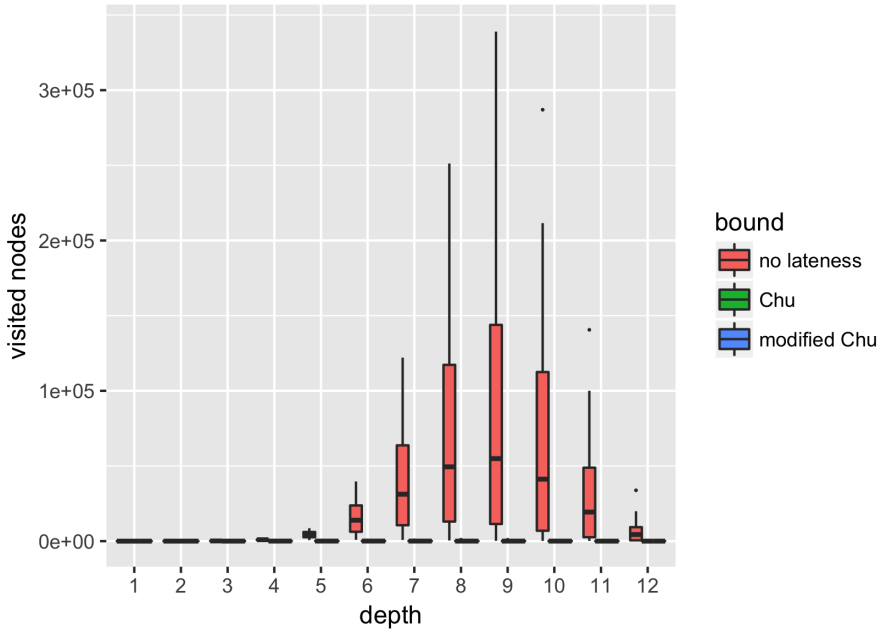
Figure B.1.: *Visualisation Tool*, showing a timeline chart to visualize a solution

C. Additional Results

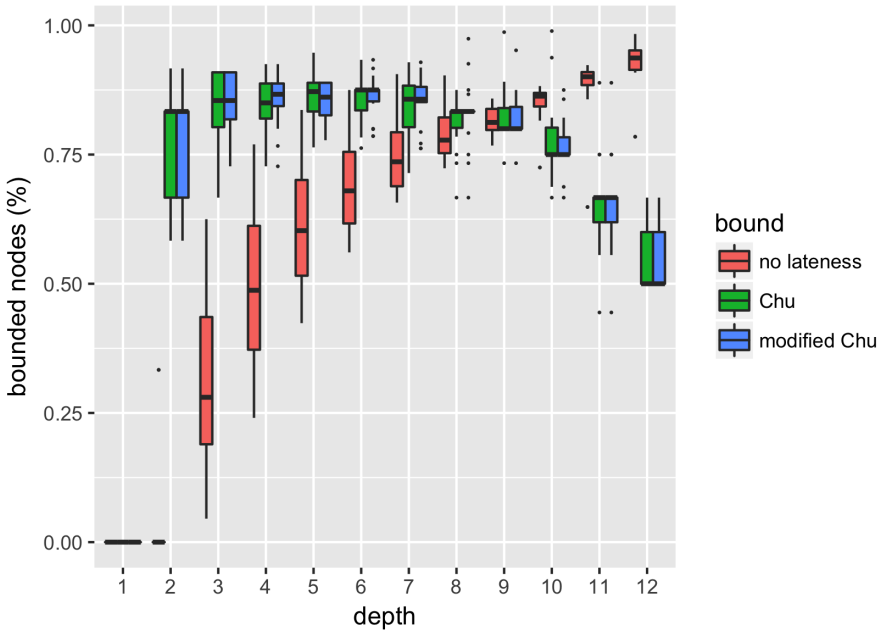
C.1. Comparing lower bounds when used in combination with the *Low Inversion Number First* strategy

In this section, we provide similar data as was introduced and explained in Section 5.5 but obtained with *Branch and Bound* in combination with the *Low Inversion Number First* strategy instead of *Most Promising First*. For further explanation and information on these tables and Figures refer to Section 5.5. For further information on the *Low Inversion Number First* strategy refer to Section 4.2.1.2.

C.1.1. Comparing the different calculation techniques



(a) Shows the total number of visitied nodes against the depth in the node tree for each of the three lower bound variants.



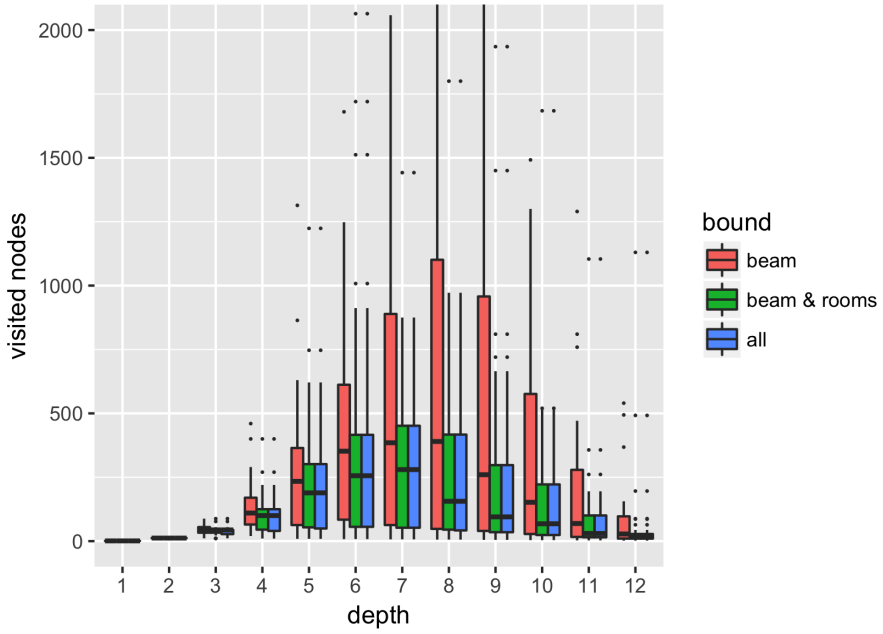
(b) Shows the relative number of bounded nodes against the depth in the node tree for each of the three lower bound variants

Figure C.1.: Comparing the node trees of a *Branch and Bound* that uses the three different lower bound variants. Only problem instances with exactly 12 tasks were used for this data.

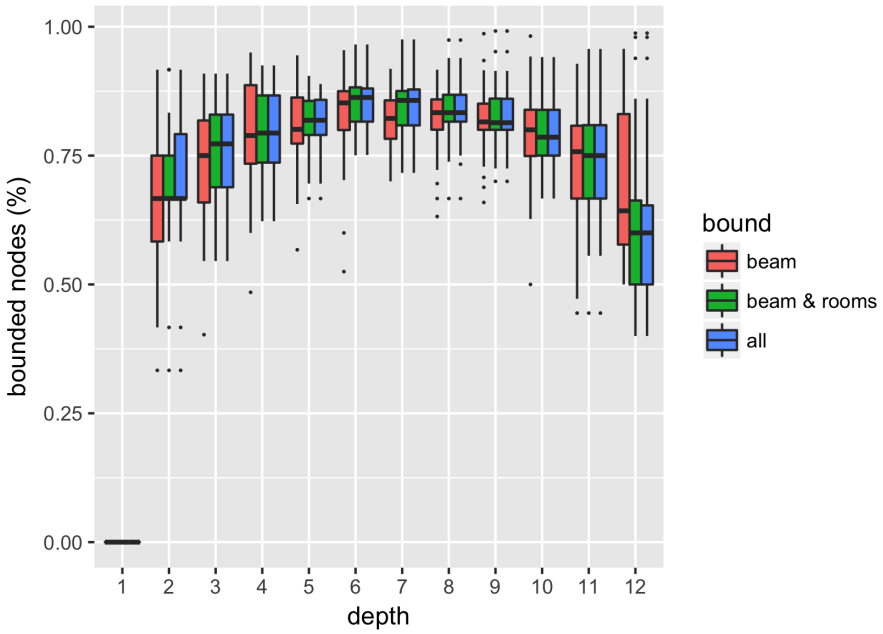
	no lateness		Chu		modified Chu	
# tasks	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$
1-20	38.782	121.61	**0**	1.06	**0**	0.52
21-30	96.754	294.9	3.613	38.11	**1.662**	28.71
31-40	98.979	300.08	11.563	97.06	**5.588**	73.49
41-50	97.978	300.47	25.986	155.79	**18.233**	138.44
51-60	95.789	300.52	48.485	230.74	**40.098**	222.59

Table C.1.: Comparing results for different lower bound techniques when used in combination with the *Low Inversion Number First* strategy (refer to Section 4.2.1.2) and the resource set 3 (Section 4.2.2.6). The Table shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

C.1.2. Comparing the different resource sets



(a) Shows the total number of visitied nodes against the depth in the node tree for each of the three lower bound variants.



(b) Shows the relative number of bounded nodes against the depth in the node tree for each of the three lower bound variants

Figure C.2.: Comparing the node trees of a *Branch and Bound* that uses different resource sets for the lower bound calculation. Only problem instances with exactly 12 tasks were used for this data.

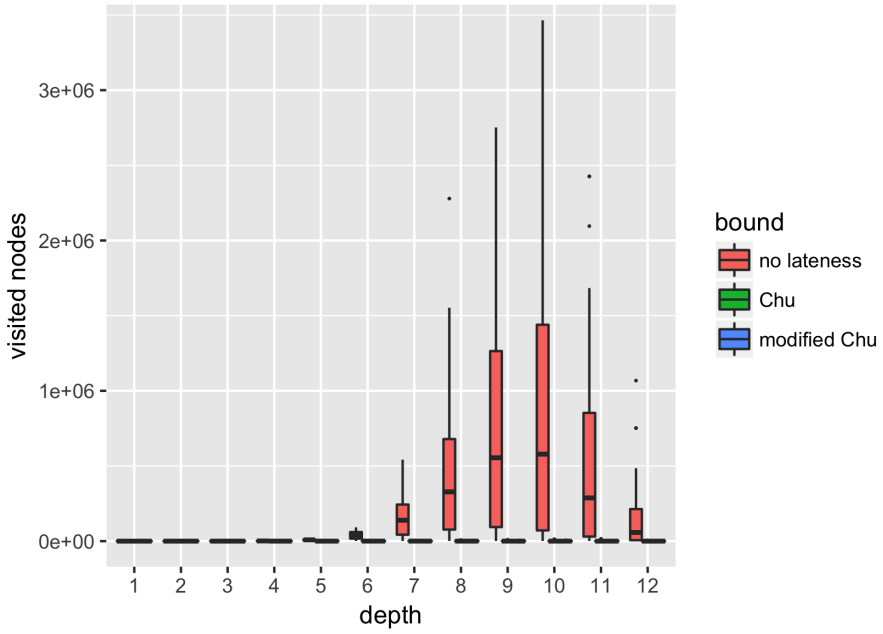
	beam		beam & rooms		all	
# tasks	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$
1-20	0.257	2.4	**0**	0.52	**0**	0.52
21-30	2.199	33.24	1.754	29.19	**1.662**	28.71
31-40	8.384	92.29	6.006	75.82	**5.588**	73.49
41-50	20.572	146.98	18.649	139.75	**18.233**	138.44
51-60	43.754	228.51	40.823	222.68	**40.098**	222.59

Table C.2.: Comparing results for different lower bound techniques when used in combination with the *Low Inversion Number First* strategy (refer to Section 4.2.1.2) and the *modified Chu's lower bound* technique as described in Section 4.2.2.5. The Table shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

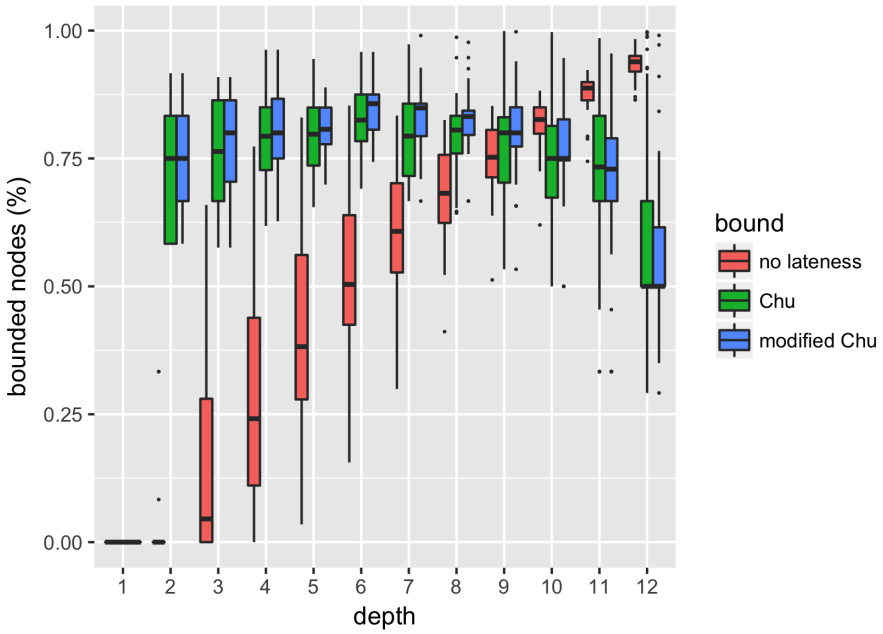
C.2. Comparing lower bounds when used in combination with the *Depth First* strategy

In this section, we provide similar data as was introduced and explained in Section 5.5 but obtained with *Branch and Bound* in combination with the *Depth First* strategy instead of *Most Promising First*. For further explanation and information on these tables and Figures refer to Section 5.5. For further information on the *Depth First* strategy refer to Section 4.2.1.1.

C.2.1. Comparing the different calculation techniques



(a) Shows the total number of visitied nodes against the depth in the node tree for each of the three lower bound variants.



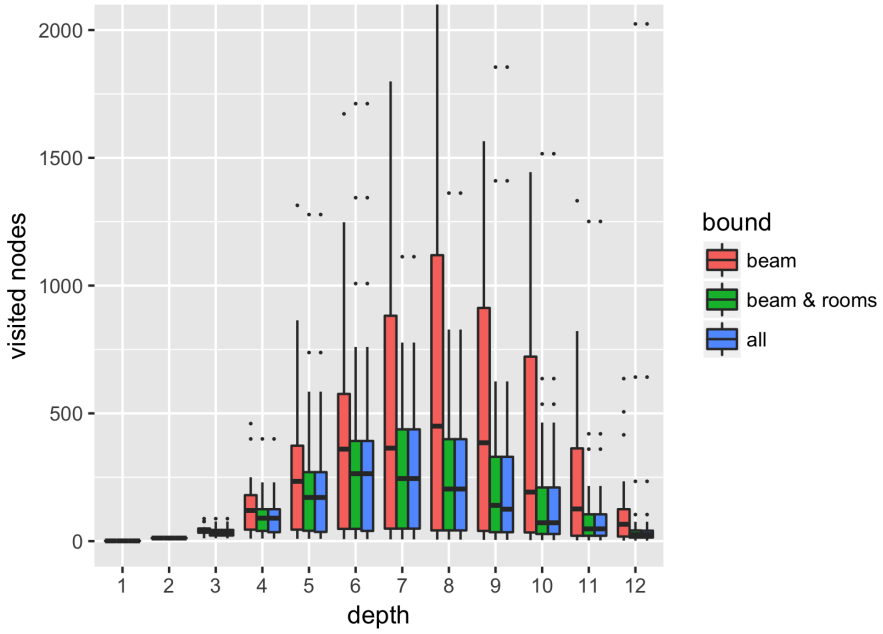
(b) Shows the relative number of bounded nodes against the depth in the node tree for each of the three lower bound variants

Figure C.3.: Comparing the node trees of a *Branch and Bound* that uses the three different lower bound variants. Only problem instances with exactly 12 tasks were used for this data.

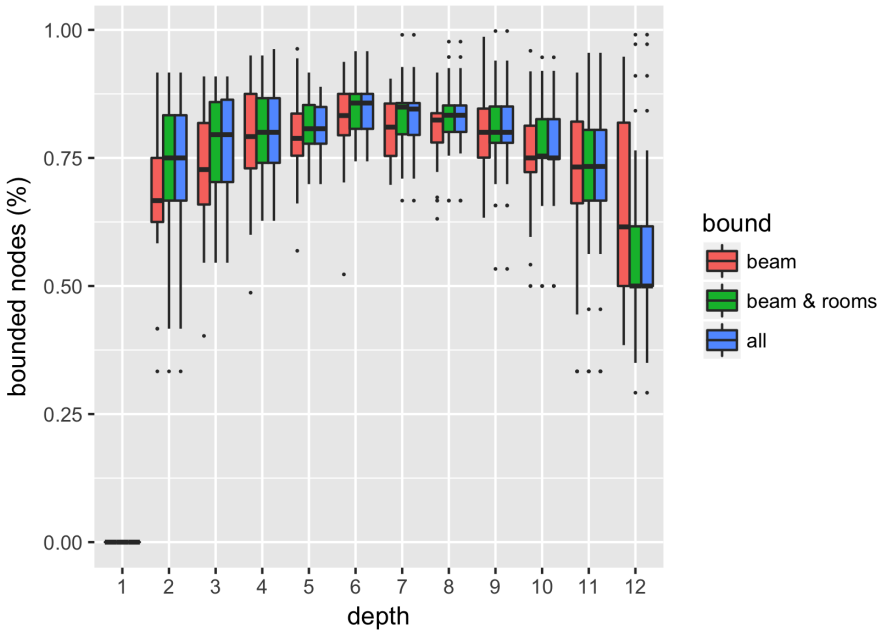
	no lateness		Chu		modified Chu	
# tasks	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$
1-20	25.885	87.17	**0**	0.56	**0**	0.36
21-30	97.913	295.51	2.46	30.43	**1.205**	22.91
31-40	99.897	300.01	11.96	95.12	**7.153**	74.81
41-50	99.225	300.01	29.198	158.58	**22.864**	143.54
51-60	97.39	300.01	54.465	236.65	**48.463**	229.69

Table C.3.: Comparing results for different lower bound techniques when used in combination with the *Depth First* strategy (refer to Section 4.2.1.1) and the resource set 3 (Section 4.2.2.6). The Table shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

C.2.2. Comparing the different resource sets



(a) Shows the total number of visitied nodes against the depth in the node tree for each of the three lower bound variants.



(b) Shows the relative number of bounded nodes against the depth in the node tree for each of the three lower bound variants

Figure C.4.: Comparing the node trees of a *Branch and Bound* that uses different resource sets for the lower bound calculation. Only problem instances with exactly 12 tasks were used for this data.

	beam		beam & rooms		all	
# tasks	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$	$\overline{gap}$	$\overline{\text{time}}$
1-20	0.054	0.86	**0**	0.31	**0**	0.36
21-30	1.31	23.83	1.232	22.02	**1.205**	22.91
31-40	8.635	82.76	**6.907**	75.49	7.153	74.81
41-50	24.252	145.38	**22.848**	141.94	22.864	143.54
51-60	50.574	231.33	**48.111**	228.45	48.463	229.69

Table C.4.: Comparing results for different lower bound techniques when used in combination with the *Depth First* strategy (refer to Section 4.2.1.1) and the *modified Chu's lower bound* technique as described in Section 4.2.2.5. The Table shows the mean of the optimality gaps, as well as the mean of the time needed to solve the problem instances (with a maximum time limit of 300 seconds) for each of the three lower bound variants, aggregated by the problem size. The best gap of each row is written in bold font.

D. Implementation in *MiniZinc*

In this section we will describe the most important and interesting parts of the implemented model in the MiniZinc modelling language. If you are already familiar with the Chapter 2 that describes the formal mathematical model of the *I-PTPSP*, you will recognize many similarities in the MiniZinc model. If not, we recommend reading it first as it will help to understand this section.

First of all, we need to define our input parameters which we will later obtain by transforming the same *JSON*-file that is needed for the *Branch and Bound* algorithm into a MiniZinc data file.

Listing D.1 describes some general parameters, like some global time input values and the gamma-values needed for the objective function. These should be quite self explanatory.

Listing D.1: MiniZinc model: general input parameters

```
% time parameters
int: Hunit;           % time units per hour
int: Wtilde_start;    % fundamental opening time
int: Wtilde_end;      % fundamental closing time
% objective function parameters
float: gamma__extback;
float: gamma__lateness;
float: gamma__scatter;
```

Next, you can find all parameters related to resources in Listing D.2. Some of these are redundant and could be computed based on other data that is provided (e.g.: `R__scatter` could be easily obtained from `phi__scatter`). But this information is already provided in the *JSON* files that serve as input for the *Branch and Bound* algorithm, so there is no sense in throwing them away and calculating them from scratch. Maybe we even gain some performance boost. An additional auxiliary variable that needed to be introduced is `max_Wbar` and `Wbars`. The minizinc modeling language does not know lists or other types of ordered datastructures with variable length [23]. Hence, these variables are needed to implement information about the unavailability periods in arrays. This comes with one limitation: if there are no unavailability periods at all, `max_Wbar` cannot be simply set to 0 without adjusting the model. So it will be set to 1 in these cases.

Listing D.2: MiniZinc model: input parameters related to resources

```
int: n_R;                    % number of Resources
set of int: R = 1..(n_R);    % set of resources
int: r__B;                   % index of the beam resource
int: r__P;                   % index of the proton
    particle r.
int: r__C;                   % index of the carbon
    particle r.
array[R] of int: R__id;        % Resources' IDs
array[R] of string: R__name; % Resources' names
% set of resources which have an extended service window
set of int: Rhat;
% set of resources which have phi_r_scatter > 0
set of int: R__scatter;
% resources' priorities to avoid  unnecessary scattering
array[R] of float: phi__scatter;
% resources' service window start times
array[R] of Wtilde_start..Wtilde_end: W__start;
% resources' service window end times
array[R] of Wtilde_start..Wtilde_end: W__end;
% resources' extended end times
array[R] of Wtilde_start..Wtilde_end: What__end;
% maximum number of poss. unavail. periods per res.
int: max_Wbar;
set of int: Wbars = 1..max_Wbar;
% Describes the starts of unavailability periods
array[R, Wbars] of Wtilde_start..Wtilde_end: Wbar__start;
% Describes the lengths of unavailability periods
array[R, Wbars] of 0..(Wtilde_end - Wtilde_start):
    Wbar__dur;
```

Finally, as the last part of the input parameters, we find those that are related to tasks in Listing D.3.

Listing D.3: MiniZinc model: input parameters related to tasks

```
int: tau;                      % number of tasks
set of int: T = 1..(tau);      % set of all tasks
array[T] of int: T__id;        % Tasks' IDs
array[T] of string: T__name; % Tasks' names
% Tasks' durations
array[T] of 0..(Wtilde_end - Wtilde_start): p;
% Tasks' initial starting times
array[T] of 0..Wtilde_end: Shat;
```

```
% Tasks' lower bound for their starting times
array[T] of 0..Wtilde_end: S__L;
% Tasks' upper bound for their starting times
array[T] of 0..Wtilde_end: S__U;
% Set of the tasks' required resources
array[T] of set of int: Q;
% Qbar[r] defines the set of all tasks that require res. r
array[R] of set of int: Qbar;
% element P__start[t][r] denotes the start time relative to the Tasks's start time at which resource r is needed
array[T, R] of 0..(Wtilde_end - Wtilde_start): P__start;
% element P__end[t][r] denotes the end time relative to the Tasks's start time at which resource r is needed
array[T, R] of 0..(Wtilde_end - Wtilde_start): P__end;
% P__dur = P__end[t][r] - P__start[t][r]
array[T, R] of 0..(Wtilde_end - Wtilde_start): P__dur;
```

The solution variables are defined in Listing D.4. `S` denotes the tasks' starting time. These are values we are looking for eventually. In addition, we defined some helper variables, that come handy for defining the objective function and improve the readability of the model. The domains of all these variables are bounded as much as possible to restrict the solution space and, as a consequence, improve the solver performance.

Listing D.4: MiniZinc model: solution variables

```
% tasks' starting times
array[T] of var Wtilde_start..Wtilde_end: S;
% helper vars
% S_r_last denotes the last time resource r is needed
array[R] of var Wtilde_start..Wtilde_end: S__last;
% sigma denotes the delay of task t with respect to its planned starting time Shat
array[T] of var 0..(Wtilde_end-Wtilde_start): sigma;
```

As already mentioned, the smaller the solution space, the better the solver performance. Hence, as `S__last` is only needed for tasks in `Rhat` and `R__scatter`, we can prune the solution space even more, by setting the other values of `S__last` to a dummy value. This is done in Listing D.5.

Listing D.5: MiniZinc model: prune the solution space. Fix unnecessary variables

```
constraint forall(r in R diff(Rhat union R__scatter))(S__last[r] = Wtilde_start);
```

For the next step, we need to define our helper variables. We can achieve this with additional constraints. Listing D.6 restricts our `S__last` and Listing D.7 specifies `sigma`. In addition, the last one also restricts our solution variable `S` to be within its lower and upper bounds.

Listing D.6: MiniZinc model: define `S__last`

```
constraint
  forall(r in (Rhat union R__scatter))(
    forall(t in Qbar[r])(
    S[t] + P__end[t,r] <= S__last[r]) /\
    W__start[r] <= S__last[r] /\ S__last[r] <=
       What__end[r]
    );
```

Listing D.7: MiniZinc model: define `sigma`

```
constraint
  forall(t in T)(
    S[t] - Shat[t] <= sigma[t] /\
    S__L[t] <= S[t] /\ S[t] <= S__U[t]
    );
```

The maybe most complex and also most interesting constraint is the next one, found in Listing D.8. We need to prevent the tasks from overlapping each other. Furthermore, we need to make sure, that also the individual starting end ending times as well as the unavailability periods of the resources are respected. Luckily, there is a so called global constraint that helps us to do this. Global constraints are generic constraints that are already defined in the MiniZincuniverse and not only make complex constraints more readable but also help to improve the solver performance as many solver-backends are able to implement these constraints very efficiently. We use the global constraint called `disjunctive(starts, durations)` which expects a list of starting points and durations that describe the given tasks that shall not overlap.[1] It also allows using durations of zero length (which will be ignored) what we will use to also respect unavailability periods (which are of length zero if not existent).

Listing D.8: MiniZinc model: prevent overlaps

```
constraint
  forall(r in R where card(Qbar[r]) > 0)(
      disjunctive(
      % startpoints
      [Wtilde_start] ++  % Resource start
      [What__end[r]] ++ % Resource end
```

[1]Refer to the MiniZincDocumentation [25] for more information on `disjunctive()` in special or global constraints in general.

```
7       [S[t] + P__start[t,r] | t in Qbar[r]] ++ % tasks
8       row(Wbar__start, r), % unavailability periods
9       % durations
10       [W__start[r]-Wtilde_start] ++ % Resource start
11       [Wtilde_end - What__end[r]] ++ % Resource end
12       [P__dur[t,r] | t in Qbar[r]] ++ % tasks
13       row(Wbar__dur,r)) % unavailability periods
14    );
```

Last but not least, we need to specify the objective function what is done in Listing D.9. The implementation of the objective function looks almost exactly the same as the definition in Section 2.3. We therefore just refer to that section for more in-depth explanation. We did not divide the objective function by *Hunit* as this is an additional calculation step and does not affect the minimization objective.

Listing D.9: MiniZinc model: objective function

```
1 solve
2   minimize (gamma__extback) * sum(r in
       Rhat)(max(S__last[r]-W__end[r],0)) +
3   (gamma__lateness) * sum(t in T)(sigma[t]) +
4   (gamma__scatter) * sum(r in R__scatter where
       card(Qbar[r]) > 0)(phi__scatter[r] *
       (S__last[r]-Wtilde_start-sum(t in
       Qbar[r])(P__dur[t,r])));
```