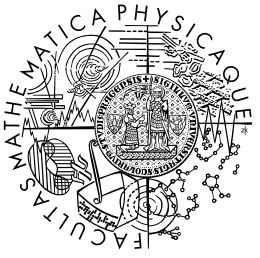


FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Lukáš Nedbálek

Bottleneck identification for constraint relaxation in resource-constrained project scheduling

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Jiří Švancara, Ph.D.

Study programme: Computer Science (B0613A140006)

Prague 2024

I declare that I carried out this bachelor thesis on my own, and only with the cited sources, literature and other professional sources. I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to express my gratitude to my supervisor, RNDr. Jiří Švancara, Ph.D., for his kind guidance, and to Ing. Antonín Novák, Ph.D., for the numerous invaluable consultations on the subject.

I would also like to thank my family for their continuous support throughout my studies.

Finally, I wish to acknowledge my debugging ducks for their supportive presence during the writing process.

Title: Bottleneck identification for constraint relaxation in resource-constrained project scheduling

Author: Lukáš Nedbálek

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Jiří Švancara, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: In modern manufacturing systems, production planners create schedules by iteratively obtaining proposed schedules and adjusting input parameters to satisfy multiple, often competing, optimization goals. The goal of this thesis is to address the problem of reducing the tardiness of a particular manufacturing order in an obtained schedule, which is a practical problem commonly arising in production scheduling. We do this by identifying bottlenecks in the schedule and proposing relaxations to constraints related to the identified bottlenecks. We develop two methods for this purpose, both utilizing constraint programming. The first baseline method adapts existing approaches from the literature and proposes general relaxations. The second method identifies potential improvements in relaxed versions of the problem and proposes relaxations targeting the specific manufacturing order. Numerical experiments show that the baseline method achieves great improvements for small costs. while the second method is more reliable in achieving improvements across various problem instances.

Keywords: scheduling, RCPSP, bottlenecks, constraint relaxation

Název práce: Identifikace úzkých hrdel pro relaxaci podmínek v rozvrhování projektů

Autor: Lukáš Nedbálek

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Jiří Švancara, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Plánovači výroby často sestavují rozvrh výroby tak, že iterovavaně získávájí návrhy na rozvrh a upravují vstupní parametry za účelem vyhovět mnohým, často protichůdným, optimalizačním cílům. Cílem této práce je zaměřit se na problém snižování zpoždění, tzv. tardiness, vybrané zakázky v obdrženém rozvrhu, jakožto běžně řešený problém při plánování výroby. Zaměříme se na identifikaci tzv. úzkých hrdel daných rozvrhů za účelem relaxace omezujících podmínek souvisejících s těmito úzkými hrdly. Pro tento účel představíme dvě metody. První adaptuje existující přístupy z literatury v kombinaci s návrhy obecných relaxací podmínek. Druhá identifikuje potenciální zlepšení v relaxovaných verzích problému a navrhuje relaxace zaměřující se na konkrétní zpožděnou zakázku. Numerické experimenty ukazují, že zatímco první metoda nachází dobrá zlepšující řešení za nízké ceny, druhá metoda je v nacházení zlepšujících řešení více konzistentní.

Klíčová slova: plánování výroby, RCPSP, úzká hrdla, relaxování omezujících podmínek

Contents

Introduction

Motivation

In modern manufacturing systems, Advanced Planning and Scheduling (APS) software is used to schedule production, manage resources, and analyze complex production data. This software helps manufacturing planners to schedule production plans and make important decisions regarding the manufacturing systems. Production needs to meet demand while performing optimally with respect to multiple, often competing, optimization goals. For humans — even knowledgeable professionals — the set of constraints and conflicting priorities becomes too complex to gain full insight and consequently decisions can be made without full understanding of their impact. Conversely, the APS software is capable of processing extensive data, however, not all information can be specified in advance, nor can all information about the real-world be exactly modeled for the software to consider. Moreover, the software lacks the intuition of a human production planner, which undoubtedly plays an important role in making decisions. This leads to users of the APS software to plan production by iteratively scheduling and fine-tuning settings to obtain an acceptable schedule.

In this thesis, we address a problem which arises frequently in manufacturing — the problem of reducing the tardiness of a selected manufacturing order in an existing production schedule. Upon obtaining a schedule constructed to meet all the extensive production demands, we observe that a particular manufacturing order is overly tardy with respect to its deadline. We would like to obtain a modified schedule which reduces the tardiness of that manufacturing order while not differing from the original schedule much. This corresponds to a scenario where, for example, after a discussion with the intended customer, we need to prioritize the production of the manufacturing order considered. However, production has already been scheduled and re-planning it entirely is not acceptable. Our goal is to find modifications to the schedule, potentially locally adjusting the settings of the system, to decrease the tardiness of the target manufacturing order with minimal impact on the remaining production.

We achieve this by considering manufacturing bottlenecks in the schedule, related to specific manufacturing constraints. Our aim is to identify these limiting constraints and relax them to improve the tardiness of a selected manufacturing order. Following the relaxation, we obtain a modified schedule and compare the improvements. An example of such a modification is illustrated in Figure 1.

To model the problem at hand, we use an extension of the Resource-Constrained Project Scheduling Problem (RCPSP) that introduces time-variant resource capacities. We develop two different methods to address the problem. The first method adapts existing approaches from the literature, which, however, address the problem in simplified scheduling models. We first need to adapt the approaches to our problem. The seconds method is developed specifically to address the problem. This method utilizes relaxations which focus specifically on the target manufacturing order. Lastly, we present a set of problem instances designed to model the addressed scheduling problem and we evaluate the two methods using the presented problem instance set.

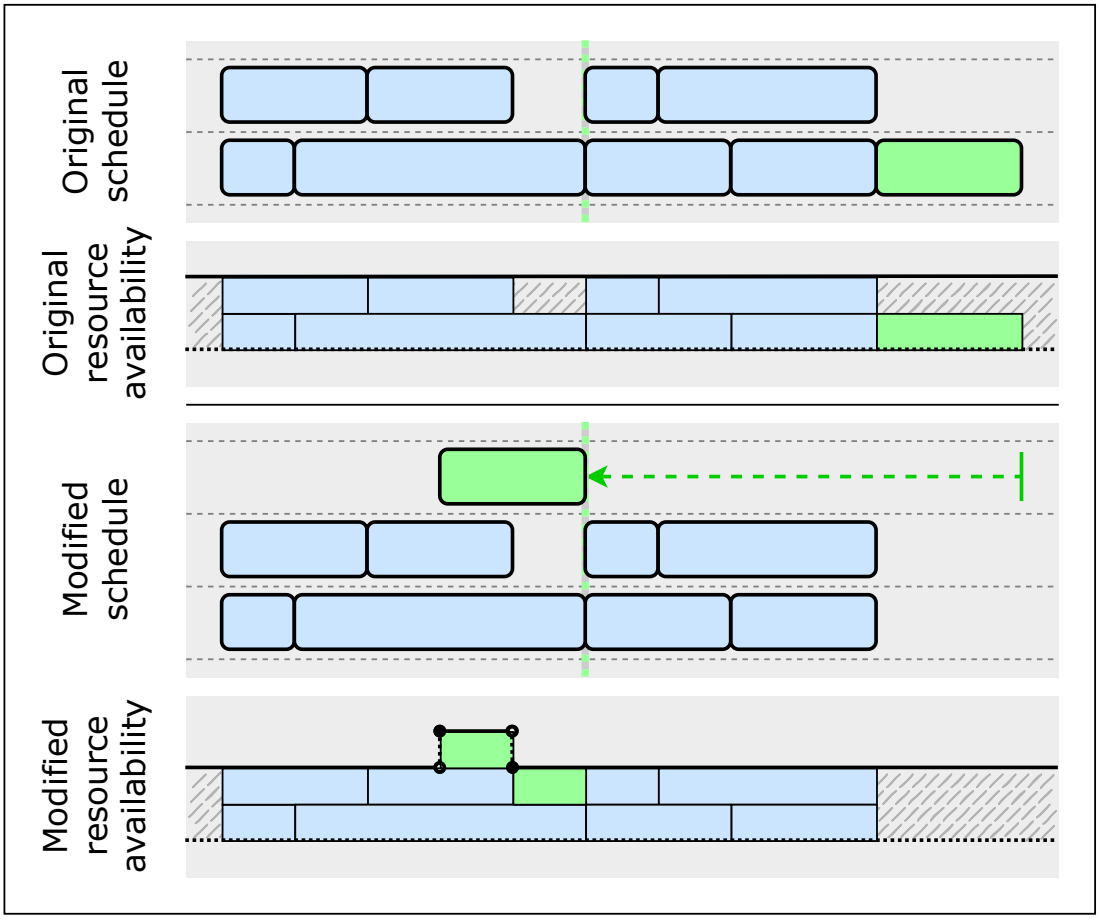


Figure 1 Schedule modification example. The green highlighted job is overly tardy with respect to its deadline (indicated by the vertical dotted line). It could not have been scheduled earlier due to the lack of remaining capacity on the resource. The resource capacity is increased within the minimal required time interval for the tardy job to be scheduled earlier. By temporarily increasing the resource capacity, we achieved improvement in the tardiness of the highlighted job, as it was possible to schedule it earlier.

Thesis outline

In Chapter 1, we introduce the problem addressed in this thesis, followed by a survey of the scheduling literature in Chapter 2 aimed to explore various existing approaches to the problem. In Chapter 3, we present our solution approach, where we design two methods for identifying bottlenecks and relaxing corresponding constraints. In Chapter 4, we conduct numerical experiments to evaluate the performance of our proposed methods and discuss achieved results. In the last chapter, we conclude the thesis by summarizing achieved results and proposing directions for future work.

1 Problem statement

In this chapter, we first state an extension of the RCPSP that we will use to model our studied problem. Following this, we describe a constraint programming method used to find solutions to the problem. Lastly, we present a formal framework for identifying bottlenecks, relaxing related constraints, and obtaining improved solutions.

1.1 Scheduling

We use the $PSm \mid intree \mid \sum_j w_j T_j$ variant[1] of the RCPSP to model the targeted real-world scheduling problem. We extend the problem by introducing several new definitions to better model the addressed problem. Furthermore, we introduce the notion of problem instances to help us distinguish between the original problem and its modifications, as described in Section 1.4. All following definitions and values are assumed to be in the integer domain.

Definition 1 (Problem instance)**.** *A* problem instance $\mathcal{I}$ *is defined as a 4-tuple* $(\mathcal{J}, \mathcal{P}, \mathcal{R}, \mathcal{T})$*, where*

- $\mathcal{J} = \{1, \ldots, n\}$ *is the set of* jobs*,*
- $\mathcal{P}$ *is the set of all* precedences *constraints,*
- $\mathcal{R} = \{1, \ldots, m\}$ *is the set of* resources*,*
- $\mathcal{T}$ *is the time horizon of the problem instance.*

Each job $j \in \mathcal{J}$ has a *duration* p_j, describing the amount of time needed to process the job j. *Preemption* of jobs is not allowed in any form — the execution of a job cannot be interrupted after its start, not even at the ends of working shifts (see variable resource capacities below). Each job j also has a *due date* d_j, stating a time horizon in which the job should be completed; otherwise, it is considered *tardy*. Subsequently, each job j also has a *tardiness weight* w_j which defines a penalty accumulated for each unit of time the job is tardy.

Execution order of jobs is constrained with *precedence constraints* $\mathcal{P}$. A precedence constraint between jobs i and j, denoted as $i \rightarrow j$ or $(i, j) \in \mathcal{P}$, states that the processing of job j can start only after the processing of job i was completed.

Interpreting jobs $\mathcal{J}$ as vertices and precedences $\mathcal{P}$ as edges between the jobs, we define the *precedence graph* $G = (\mathcal{J}, \mathcal{P})$. We assume the precedence graph to be an *inforest*, in other words, the precedences constrain the execution order of jobs in such a way that the resulting precedence graph is always an inforest. An inforest is a directed acyclic graph where each vertex has at most one successor. A connected subgraph of an inforest is an *intree*. See Figure 1.1 for a precedence graph example.

Jobs are executed on *resources* $\mathcal{R}$ — machines with time-variant renewable *capacities*. The capacity of a resource k during a time period $t \in \{1, \ldots, \mathcal{T}\}$ is

[1] $\alpha|\beta|\gamma$ three-field notation defined by Brucker et al. (1999).

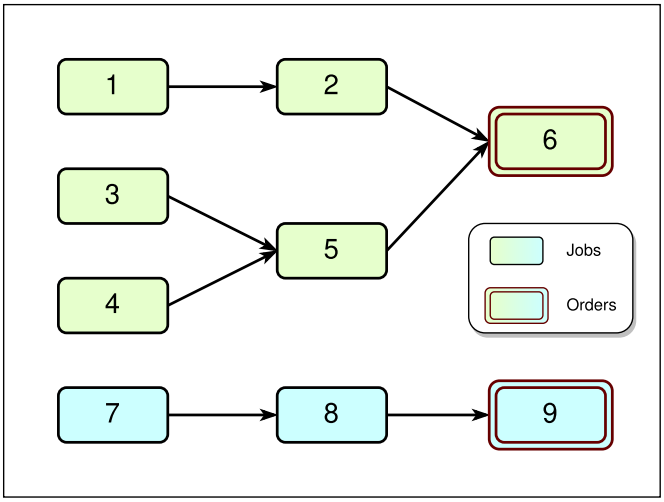


Figure 1.1 Example of a precedence graph. The inforest structure of the graph is demonstrated. Additionally, orders as the roots of the intrees are highlighted.

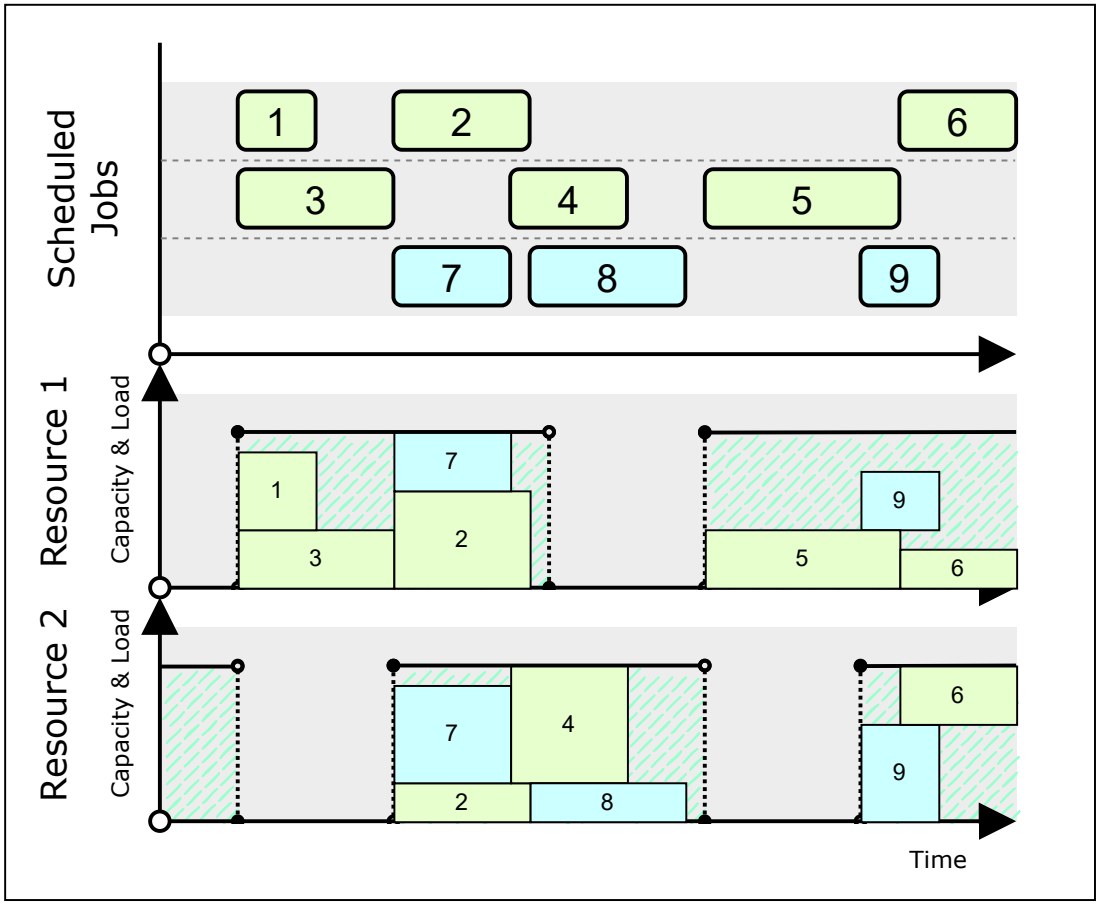


Figure 1.2 Example of a scheduled RCPSP with time-variable resource capacities. The jobs with corresponding precedences from Figure 1.1 are scheduled on two machines with partially overlapping working shifts. Job durations and capacity consumptions were chosen to illustrate the variability in resource utilization.

denoted as $R_k^{(t)}$. During their execution, jobs consume the capacities of resources. For a job j, the per-period consumption of a resource k is denoted as r_{jk}. This describes how much of the resource's capacity is consumed each period during the job's execution. Capacities are renewable, meaning that each time period the specified amount of capacity is available, regardless of prior capacity consumptions. The resource capacity functions $R_k^{(t)}$ are assumed periodic with the same period of 24. Moreover, the resource capacity function of the resource k only takes the values 0 and $R_k^{(-)}>0$. This assumption, however, is only for the initial problem instance — resource capacity functions of modified instances, as described in Section 1.4, can be non-periodical and take arbitrary non-negative integer values.

The set of *orders* $\mathcal{O} = \{j \in \mathcal{J} \mid \nexists i : j \to i\}$ is the set of roots of the precedence intrees, i.e. the set of all jobs for which no precedence successor exists. A job $j \in \mathcal{O}$ is called an *order*.

We make a simple observation; due to the intree-structure, each (weakly) connected component in the precedence graph contains exactly one order. From this, we can say that each job is either an order or is associated with a unique order within the same intree component. This is demonstrated in Figure 1.1.

Having introduced orders, we can now specify the ranges of job deadlines and tardiness weights based on whether a job belongs to the orders set $\mathcal{O}$:

$$d_j = \begin{cases} a \in \mathbb{N}_0 & \dots \text{ if } j \in \mathcal{O} \\ +\infty & \dots \text{ otherwise} \end{cases} \qquad w_j = \begin{cases} a \geq 0 & \dots \text{ if } j \in \mathcal{O} \\ 0 & \dots \text{ otherwise} \end{cases}$$

1.2 Constraint programming model

We formulate the above problem using a constraint programming model. This provides us with a well-functioning framework and allows us to use existing solvers for finding optimal solutions. The model can be stated as:

$$\begin{aligned}
\text{given} \quad & \mathcal{J} = (1, \dots, n),\ \mathcal{R} = (1, \dots, m),\ \mathcal{P}, \\
& p_1, \dots, p_n \in \mathbb{N}_0,\ d_1, \dots, d_n \in \{1, \dots, \mathcal{T}\},\ w_1, \dots, w_n \in \mathbb{N}_0, \\
& r_{11}, \dots, r_{nm} \in \mathbb{N}_0,\ R_1, \dots, R_m : \{1, \dots, \mathcal{T}\} \to \mathbb{N}_0 \\
\text{find} \quad & S = (S_1, \dots, S_n) \in \mathbb{N}^n \\
\text{minimizing} \quad & \sum_{j \in \mathcal{J}} w_j T_j && (1.1) \\
\text{subject to} \quad & C_i \leq S_j \qquad \forall i \to j \in \mathcal{P} && (1.2) \\
& \sum_{j \in \mathcal{J}} \mathrm{c}_{jk}^{(t)} \leq R_k^{(t)} \qquad \forall t \in \{1, \dots, \mathcal{T}\}\ \forall k \in \mathcal{R} && (1.3) \\
\text{where} \quad & C = (S_1 + p_1, \dots, S_n + p_n),\ T_j = \max(0, C_j - d_j), \\
& \mathrm{c}_{jk}^{(t)} \stackrel{\text{def}}{=} \begin{cases} r_{jk} & \text{if } S_j \leq t < C_j \\ 0 & \text{otherwise} \end{cases}
\end{aligned}$$

Inequalities (1.2) formulate the precedence constraints — start and finish times of jobs are according to all the precedences. Inequalities (1.3) formulate the resource capacity constraints — in every time period the combined consumption

of jobs scheduled during the period cannot exceed any of the resource's capacities. Expression (1.1) is the optimization minimization objective — the weighted tardiness of jobs.

We assume we have a solver capable of solving (1.1)–(1.3) in reasonable time through the use of constraint programming.

In the following sections, we will consider the stated constraints as potential bottlenecks and how to relax specific constraints identified as bottlenecks.

1.3 Bottlenecks

1.3.1 Definition

Our goal will be to find bottlenecks in the problem, specifically in the obtained schedule — the solution to the constraint programming model defined in Section 1.2. First, we state a general definition of an execution-level bottleneck:

Definition 2 (Execution Bottleneck Machine (EBM) (Wang et al., 2016))**.** EBM *is a machine that dominates the scheduling performance in the strongest manner at the execution level of production systems.*

The execution level of a production system stated here refers to finding bottlenecks specific to the given problem instance and the obtained solution to that instance. This is consistent with our goal of finding bottlenecks in the particular problem instances and their solutions, and improving the system performance only with respect to the currently presented problem. In contrast, we will not be interested in bottlenecks of the whole system nor in generally improving the performance of the system regardless of the problem instance. Having made the distinction, we will now discuss our interpretation of the definition suited to our problem.

Instead of identifying just a single machine as a bottleneck or listing all the machines in the order of their scheduling impact, we will focus on identifying specific time periods on specific machines as schedule bottlenecks. This allows us to relax only the specific constraints related to the identified time periods, resulting in localized modifications with minimal costs.

1.3.2 Constraints relaxation

When deciding which constraints to relax, we have two constraints to consider: precedence constraints (1.2) and resource capacity constraints (1.3). We will discuss individual segments that form the constraints, considering their possible relaxation.

(i) *Job precedences* Job precedence constraints are inherent to the problem — — jobs cannot start until all their predecessors have finished executing. We cannot remove individual precedences as a relaxation, as precedences model the technological requirements of the production system. We could imagine that scheduling jobs might require preparations, which do not necessarily require the presence of the product about to be processed. Those preparations could therefore be allowed to start even before the predecessor jobs

finish executing, which would introduce slack in the constraints[2]. However, we can assume that those preparations are completed before the start, or we can assume no preparation time at all.

(ii) *Cumulative consumption* The cumulative consumption is dependent on the constructed schedule and the given problem instance. Each job contributes to the cumulative consumption during the time periods it is scheduled. As part of the problem definition, we cannot omit this contribution, nor can it be shortened, as job durations p_j are fixed. Equally, job resource consumptions r_{jk} influence the cumulative value, but again, due to the nature of our problem, we cannot modify resource consumptions as those are inherent to the problem and the corresponding real-life execution and operation requirements.

(iii) *Resource capacities* Available capacities of resources can be modified with reasonable correspondence to modifications of the real-life problem. More specifically, the capacity of a resource during a time period can be increased or decreased. Increasing the capacity of a resource during specific time periods could correspond to increasing the number of workers operating the resource machine, assuming that increasing the number of operating workers increases the total processing capacity. While sole reduction of capacities would not relax the constraints, decreasing the capacity of one resource by a specific amount while increasing the capacity of another resource by the same amount could tighten the constraints on the former resource but relax the constraints on the latter. This decreasing of the capacity of one resource while increasing the capacity of another by the same amount could correspond to migrating workers between the resource machines, assuming that migrations of operating workers are possible.

We can conclude that the precedence constraints (1.2) cannot be relaxed. On the other hand, the resource capacity constraints (1.3) can be relaxed by modifying the capacities of resources. As discussed, these modifications can be achieved through capacity additions or migrations, both of which have corresponding realizations in real-world production systems.

1.3.3 Resource capacity modifications

We will consider *capacity additions* and *capacity migrations* as the possible relaxations of scheduling constraints, namely of the constraints (1.3). Additions are executed by increasing the capacity of a selected resource by a specified amount over a specified time interval. Migrations are executed by decreasing the capacity of a selected source resource by a specified amount over a specified time interval and increasing the capacity of a selected target resource by the specified amount over the same time interval. An illustrative example of a capacity addition and a capacity migration is shown in Figure 1.3.

[2]Scheduling with so-called *setup times* is a broadly studied subject in the literature. See the survey of Hartmann and Briskorn (2010) where the extension of setup times is described and various examples of approaches to the problem are given.

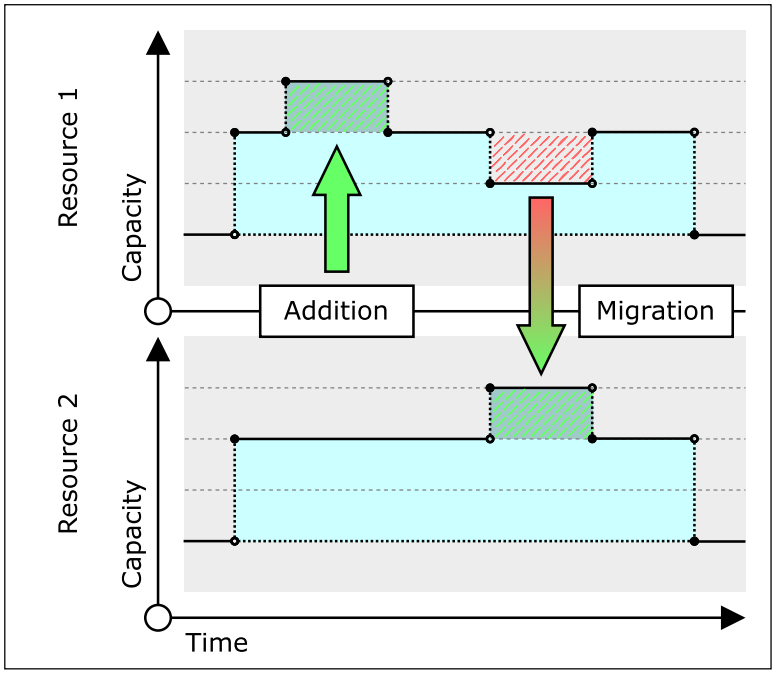


Figure 1.3 Diagram of possible capacity changes

We consider migrations only between resources, limited to the same time interval. We do not allow migrations in time — reducing the capacity of the source resource during a time interval and then increasing the capacity of a target resource during a different interval. Such migrations could correspond to migrating resource materials between resources. In such cases, the change in time would correspond to the migrated material being processed at a different time on a different machine. However, we consider production systems with workers and machine operators as main resource capacities. Considering this, migrating capacities in time would not have practical realizations, as the workers migrated from one resource during one time period would likely still be assigned to that resource during a different time period.

We associate a capacity addition with the 4-tuple (k, s, e, c), where k is the resource whose capacity is increased, s and e form the time interval $\{s, \ldots, e-1\}$ over which the addition is executed, and c is the added capacity. Analogously, we associate a capacity migration with the 5-tuple $(k_{\text{from}}, k_{\text{to}}, s, e, c)$, where k_{from} is the source resource whose capacity is lowered, k_{to} is the target resource whose capacity is increased, s and e form the time interval $\{s, \ldots, e-1\}$ over which the migration is executed, and c is the migrated capacity. For a modified instance $\mathcal{I}^*$, the sets of all migrations and additions are denoted as $\mathcal{M}^{\mathcal{I}^*}$ and $\mathcal{A}^{\mathcal{I}^*}$, respectively.

In a real-world production system, migrating capacities is generally more cost-effective than adding new capacities. For example, reassigning workers from an underutilized machine to a bottleneck machine is typically less expensive than extending workers' shifts into overtime or planning an entirely new and irregular shift. Therefore, capacity migrations are usually preferred. However, if the required capacity changes cannot be achieved through capacity migrations, capacity additions can be utilized.

1.4 Relaxed schedule

In the rest of this chapter, we define a general procedure for solving the presented problem, identifying bottlenecks and relaxing corresponding constraints by modifying the initial problem, solving the modified problem, and evaluating whether the modified solution reached a desired improvement. More precisely:

1. Suppose we obtained an optimal solution S to the problem instance $\mathcal{I}$.
2. We select the target order $o \in \mathcal{O}$ for which we want to improve the tardiness. We consider improvement to be any non-zero decrease in the objective function with respect to the selected order o and its tardiness T_o.
3. We identify bottlenecks in the solution S of the instance $\mathcal{I}$.
4. Based on the identified bottlenecks, corresponding constraints are relaxed via capacity additions and capacity migrations. Those capacity changes are captured by modified resource capacity functions $R_1^*, \ldots, R_m^*$ corresponding to a modified problem instance $\mathcal{I}^*$.
5. We obtain a solution S^* to the modified problem instance $\mathcal{I}^*$.
6. Finally, any desired evaluations can be made on the modified solution S^*, alongside comparisons to the original solution S.

2 Related works

2.1 RCPSP scheduling

To model our studied problem we chose the $PSm \mid intree \mid \sum_j w_j T_j$ version of the RCPSP. The standard RCPSP is proven to be NP-hard (Blazewicz et al., 1983). A comprehensive overview of complexity results was done by Ganian et al. (2021).

2.1.1 Time-variable resource capacity constraints

Under the general PSm (Project Scheduling) characteristic we allow arbitrary resource capacity functions. However, as stated in Sections 1.1, 1.3 and 1.4, the resource functions are mostly periodical with only a few local modifications.

Hartmann and Briskorn (2010) and Hartmann and Briskorn (2022) surveyed the variants of the RCPSP studied in the literature and alongside other variants also discussed time-variable resource capacity constraints. As in our modeled real-world problem, the surveyed literature used the time-varying capacities to model worker shifts, resource machine maintenance, or other manufacturing processes resulting in variable availabilities.

Time-variable resource availabilities are often studied with allowed preemption. Recall from Section 1.1 that when job preemption is allowed, execution of a job can be interrupted and resumed at a later period. Usually, only a specific form of preemption is allowed — interrupting the execution of a job at the end of a working shift and resuming it at the start of the following working shift. Preemption in this context can be a reasonable assumption as it can, in some manufacturing systems, correspond to a simple interruption in executed work. On the other hand, job preemption, even in the specific form mentioned, might not be available due to technical reasons.

To state a few examples of scheduling under time-variable resource capacity constraints, see the work of Klein (2000) or Nonobe and Ibaraki (2002). For an example of calendar-based availability scheduling with job preemption, see the work of Franck et al. (2001).

2.1.2 Solution approaches

Solving the RCPSP is done using exact methods, heuristics, or metaheuristics. Exact solution methods aim to solve scheduling problems systematically. They guarantee to find the optimal solution by exhaustively exploring all possible solutions within a problem's feasible solution space. However, this exhaustive search can become computationally expensive and impractical for large-scale or complex problem instances. The most prominent exact approach is the branch-and-bound method, used in many variations and with problem-specific heuristics. First branch-and-bound solution method was proposed by Demeulemeester and Herroelen (1992), another example is the work of Vanhoucke et al. (2001). An approach using a modified constraint programming and SAT solver with generalized precedence relations can be found in the work of Schnell and Hartl (2015).

In contrast, heuristics aim to quickly find good solutions but do not guarantee optimality. Heuristic approaches are simple, easy to implement, and computationally inexpensive compared to exact methods. Heuristics use priority scheduling rules, schedule-generating schemes, and approximations, but also relaxations of exact methods, such as truncated branch-and-bound or relaxed integer programming. Simple heuristics were popular as alternatives to exact methods. See the survey of Kolisch and Hartmann (1999) for an extensive survey of heuristic approaches used at the time.

Metaheuristics are higher-level problem-solving frameworks that search for good solutions in the solution space using a combination of exploration and exploitation strategies. Like simple heuristics, metaheuristics do not guarantee finding the optimal solution but offer a trade-off of computational performance. They are often more complex than simple heuristics and, as a result computationally more expensive, but they can handle larger and more complex problem instances. Metaheuristics include genetic algorithms, simulated annealing, tabu search, and particle swarm optimization. A recent survey on metaheuristics by Pellerin et al. (2020) compares a wide range of metaheuristics on PSPLIB instances.

We chose an exact solution approach by utilizing a constraint programming solver. Our problem contains many constraints, most of which model time-variable capacities of resources. We frequently modify those constraints and, therefore, need to make adjustments to the model. Constraint programming models and solvers are ideal for such use cases. The constraints can be expressed in a declarative way without having to handle specifics. This makes the process of adjusting the declared constraints and obtaining modified models straightforward. In comparison, modeling these problems for heuristics or metaheuristics requires significantly more effort, and adjusting the modeled constraints might prove even more difficult.

2.2 Bottlenecks in scheduling

Bottlenecks are broadly studied in the scheduling literature. The notion of a bottleneck is recognized to have an important role in scheduling when system performance is considered.

2.2.1 Various definitions

In Definition 2 we defined the Execution Bottleneck Machine (EBM). We chose this definition as it best corresponds to our studied problem. Namely, it emphasizes the execution phase of production within which the bottleneck is considered and doesn't specify any particular metric or measure of the bottleneck's impact on the system.

Betterton and Silver (2012) surveyed the literature on scheduling bottlenecks and found at least 11 different definitions of the term *bottleneck resource*. The first attempts at defining the term were very specific in the way the bottleneck resource is identified. Usually, the definition corresponded closely with the identification method proposed in the same work. See for example the work of Lawrence and Buss (1994), Kuo et al. (1996), or Roser et al. (2001). Throughout the years, however, the aim shifted towards defining a bottleneck as generally as possible to

cover as many specific interpretations of the term but to also best capture the important nature of a bottleneck resource in production systems. For attempts at this approach see the work of Chiang et al. (2001) or Biller et al. (2010).

Based on the definitions presented up to that point, Betterton and Silver (2012) stated the following definition.

Definition 3 (Bottleneck resource (Betterton; Silver, 2012))**.** *The* bottleneck *is the resource that affects the performance of a system in the strongest manner, that is, the resource that, for a given differential increment of change, has the largest influence on system performance.*

This definition provides a neat generalization of the term *bottleneck resource*. It also provides insight that focusing on such resources and making small adjustments can significantly influence the system's performance. Note, that the definition does not specify the influence of mentioned adjustments has to be beneficial towards the performance. Correctly identifying the bottleneck resource, but incorrectly adjusting the related settings can impact the performance negatively.

2.2.2 Bottleneck classification

Bottlenecks can be identified at different stages of production. We use the bottleneck classification proposed by Wang et al. (2016). This classification distinguishes between structural, planning, and execution bottlenecks based on the time frame at which the bottlenecks are identified.

Structural bottleneck machines frequently affect the performance of the production system, regardless of the specific problem setting. Such machines are usually identifiable by human operators, as their high impact on performance can be observed across different problem settings and schedules. In contrast to their identifiability by humans, structural bottlenecks might not be easily solvable or relaxed. It may be caused by a hard limitation of the system, for example, an inefficient production line layout or machines running at full capacity without the possibility of increasing the capacity.

Structural bottlenecks can be identified with the use of historical production data. Modern production systems collect large volumes of data concerning the production, performance of individual components, utilization of resources, etc. This data is then processed by techniques from data science or machine learning to capture various patterns in production, anomalies, or correlations among the data indicating a potential bottleneck. Such techniques can better react and adapt to dynamic changes in the production system, given that enough data regarding similar dynamic changes has been collected. A recent survey by Mukund Subramaniyan et al. (2021) provides an overview of data-driven bottleneck identification methods. To state a few examples, see the work of M. Subramaniyan et al. (2016), Roh et al. (2018) or Li (2009).

When historical data is not available, a simulation of the system might provide a sufficient approximation of the real production. Such an approach was studied by Roser et al. (2001) where the study focused on a serial production line and the bottleneck machine was identified through the average active duration on machines. Zhang and Wu (2008b) and Zhang and Wu (2012) first model an alternative relaxed optimization model for the system, identify the bottleneck

machines during the scheduling of this model, and then schedule the original problem with a designed genetic algorithm, which utilizes the obtained bottleneck information by allocating more resources to the identified machines.

Planning bottleneck machines are considered during the construction of a schedule. Identifying such machines during the scheduling process can help guide the scheduling procedure — whether exact, heuristic, or metaheuristic. The scheduling procedure can choose to allocate more resources to the identified bottleneck machine, spend more time and computational resources on scheduling jobs on such machines, etc. A notable example of such an approach is the shifting bottleneck procedure proposed by Adams et al. (1988). They construct the schedule by sequentially scheduling on singular machines, each time choosing an unscheduled machine identified as the current bottleneck, then locally reoptimizing the already scheduled machines. In their study, Mönch and Zimmermann (2010) argue that the shifting bottleneck procedure in its various versions remains a superior method for scheduling semiconductor wafer production systems. They argue that this is due to several difficulties present in scheduling such systems, challenges that the shifting bottleneck procedure can easily address. In a Job-Shop scheduling problem, Zhang and Wu (2008a) identify bottleneck machines by their sensitivity to changes to their scheduling policies. They incorporate this identification in a proposed genetic algorithm to guide its search for improved solutions.

Execution bottleneck machines are the focus of study in this thesis. These bottlenecks are considered in the constructed schedule for a given problem instance and restrain us from achieving a better schedule in terms of the given optimization goal, such as average throughput, schedule makespan, weighted tardiness, etc. While identifying structural bottlenecks primarily aims to enhance the overall system performance, identifying execution bottlenecks aims to improve performance for specific problem instances. It's worth noting that for a specific problem instance and its constructed schedule, we may identify one machine as its execution bottleneck, but for a different problem instance with different settings, the execution bottleneck may vary. Consequently, relaxing constraints related to an identified execution bottleneck aims to enhance performance for the specific problem instance; however, applying such relaxation to a different problem instance might not provide any improvement.

Limited studies have focused on this type of bottleneck. Wang et al. (2016) proposed a multi-indicator approach for identifying execution bottlenecks. Their study investigated how identified execution bottlenecks differ from identified planning bottlenecks. Computational results demonstrate that, for many problems, the identified planning bottlenecks differ from the execution bottlenecks identified in specific schedules for those problems. This highlights the importance of distinguishing between planning bottlenecks and execution bottlenecks.

2.2.3 Identification indicators

Various techniques are used for the identification of bottleneck machines. Such techniques typically involve the usage of identification indicators, either individually or in combination. An identification indicator refers to a value computed for each machine, allowing us to rank the machines and then select the

bottleneck machine based on this rank. We will state a few examples proposed in the literature:

- Machine Utilization Rate (MUR) (Lawrence; Buss, 1994):

$$\mathrm{MUR}_k = \frac{\sum_{j\in\mathcal{J}_k} p_j}{\max_{j\in\mathcal{J}_k} C_j - \min_{j\in\mathcal{J}_k} S_j}.$$

- Queue Length (QL) (Lawrence; Buss, 1994). In manufacturing systems consisting of machines with workload queues, this indicator considers the number of items in the machine's queue as the bottleneck measure.

- Average Uninterrupted Active Duration (AUAD) (Roser et al., 2001):

$$\mathrm{AUAD}_k = \frac{\sum_{i=1}^{A_k} a_{ki}}{A_k},$$

  where $a_{k1}, \ldots, a_{kA_k}$ are the lengths of uninterrupted active durations of resource k, A_k is the number of those individual durations.

- Resource Strength (RS) (Cooper, 1976) and Resource Constrainedness (RC) (Patterson, 1976):

$$\mathrm{RS}_k = \frac{R_k}{\mathrm{avg}_{j\in\mathcal{J}}\, r_{jk}} = \frac{R_k \cdot n}{\sum_{j\in\mathcal{J}} r_{jk}},$$

$$\mathrm{RC}_k = \frac{\mathrm{avg}_{j\in\mathcal{J}_k}\, r_{jk}}{R_k} = \frac{\sum_{j\in\mathcal{J}_k} r_{jk}}{R_k \cdot |\mathcal{J}_k|},$$

  where R_k is the per-period capacity of a resource k — these indicators do not account for variable resource capacities. Those indicators are usually considered when describing problem instances, specifically, the properties of resources. However, Luo et al. (2023) chose them as bottleneck identifiers and compared their effectiveness when used to guide a genetic programming algorithm.

In the formulae for MUR_k and RC_k, $\mathcal{J}_k \overset{\text{def}}{=} \{j \in \mathcal{J} : r_{jk} > 0\}$ is the set of jobs with nonzero consumption of the resource k, i.e. the jobs which are executed on the resource k.

Identification indicators have the advantage of being simple, making their implementation straightforward and their computation efficient. More complex identification methods may provide better insight into the bottleneck identification process, however, such methods are usually tailored specifically to a specific version of the problem or their implementation can be too complicated for practical use.

2.2.4 Bottlenecks in the RCPSP

To the best of our knowledge, only little research focuses on identifying bottlenecks in the RCPSP. The closest research is on bottlenecks in the Job-Shop problem, i.e. scheduling on unit-capacity resources.

Luo et al. (2023) studied how identifying bottleneck machines can guide the scheduling process of a genetic algorithm. Arkhipov et al. (2017) conducted a case study on a large-scale resource-constrained scheduling problem with over 3 thousand operations and over 50 machines. They proposed a heuristic approach for estimating project makespan and resource load profiles. Those estimations are in turn used to identify bottleneck resources for the problem. However, the identified bottleneck resources were not addressed further.

Concerning bottleneck identification indicators discussed above, we were unable to find any for the RCPSP with time-variable resource capacities. Moreover, we were unable to find any identification identifiers for the standard RCPSP which would account for time-variable consumption profiles in a schedule. Although the indicators mentioned in Section 2.2.3 can in theory be used in the RCPSP, they were originally designed for the Job-Shop problem. Identification indicators in the RCPSP could incorporate information about variable resource loads and even variable capacity functions in the time-variant capacities extension. However, the indicators designed for the Job-Shop problem do not consider this additional dimension of information. We address this further in Section 3.1.

2.2.5 Relaxing the identified bottlenecks

In their study, Zhang and Wu (2012) addressed the Job-Shop problem by relaxing its capacity constraints and then solving the modified relaxed problem to identify bottlenecks based on the solution. The obtained information was used to guide a proposed simulated annealing algorithm to find a solution to the original problem. Thus, the relaxation served only as an intermediate step toward obtaining a solution, rather than being the desired result.

Lawrence and Buss (1994) studied how identified bottlenecks shift between machines in response to introducing relaxations to the original problem. They employed a proposed "bottleneck chasing" policy for relaxing short-run bottlenecks, wherein the capacity of the identified bottleneck resource is increased, e.g., by extending its working shifts, or assigning additional employees. Then, the bottleneck identification process is run again to examine whether the bottlenecks shift to a different resource. The authors observed that while the chasing policy is effective at relaxing local bottlenecks, it also increases the "bottleneck shiftiness," resulting in a more change-sensitive system. They also studied the shiftiness of long-run bottleneck resources having the highest utilization over time. Results show that increasing the capacity of long-run bottleneck resources also increases the bottleneck shiftiness, but this shiftiness can be reduced by simultaneously increasing the capacity of non-bottleneck resources.

2.3 Contribution

As discussed in Sections 2.2.3 and 2.2.4, the scheduling literature primarily focuses on bottlenecks in the Job-Shop problem. We aim to extend the standard Job-Shop approaches to the RCPSP. Our second goal is to design an approach for identifying bottlenecks in the RCPSP extended with time-variant resource capacities with the focus on relaxing the identified bottlenecks to improve a proposed schedule.

3 Solution approach

In this chapter, we present two algorithms designed for identifying and relaxing bottlenecks in the RCPSP. Both algorithms aim to improve the tardiness of a selected order by introducing relaxations to the capacity constraints in the problem instance and finding a solution to the modified problem instance.

First, we propose an algorithm called Identification Indicator-based Relaxing Algorithm (IIRA). The IIRA combines an adaptation of existing bottleneck identification approaches from the literature with a new method for relaxing the capacity constraints. The algorithm utilizes bottleneck identification indicators to find bottleneck resources, selects periods with high improvement potential, and increases the capacities of the bottleneck resources during the selected periods.

The second algorithm we propose is the Schedule Suffix Interval Relaxing Algorithm (SSIRA). The SSIRA employs a novel approach to relaxing capacity constraints based on finding improvement intervals in partially relaxed versions of the problem. SSIRA iteratively relaxes the capacity constraints in suffixes of an obtained schedule and selects jobs that could start earlier following the relaxation. The resource capacity constraints are then relaxed with respect to a small subset of the selected jobs with improvement potential.

To help explain how the algorithms work, we illustrate[1] key moments on the schedule given in Figure 3.1. In the illustrated schedule, job 9 is scheduled past its due date and is considered tardy. This job is the target order for improvements in the examples provided for the algorithms.

3.1 Baseline solution

In this section, we propose adaptations of existing bottleneck identification indicators from the literature. Utilizing the adapted indicators, we propose the IIRA for relaxing capacity constraints of a given problem instance based on its solution.

3.1.1 Adapted identification indicators

We adapt existing identification indicators to detect bottlenecks, specifically the Machine Utilization Rate (MUR) and Average Uninterrupted Active Duration (AUAD) indicators. For precise definitions, see Section 2.2.3.

The MUR, first utilized as a bottleneck identification indicator by Lawrence and Buss (1994), considers the ratio of executed work on a resource to the total time the resource was used. In their study, Lawrence and Buss (1994) demonstrate, that despite its simplicity, the MUR indicator is effective at identifying long-run bottlenecks[2].

The AUAD, initially proposed by Roser et al. (2001), is more complex but remains a comprehensive indicator for identifying bottleneck resources. For the

[1]The figures used to illustrate the specifics of the algorithms were created in Python using the plotting library Matplotlib. See online at `https://matplotlib.org/`.

[2]Long-run bottlenecks could be viewed as structural bottlenecks — see Section 2.2.2 for bottleneck classification.

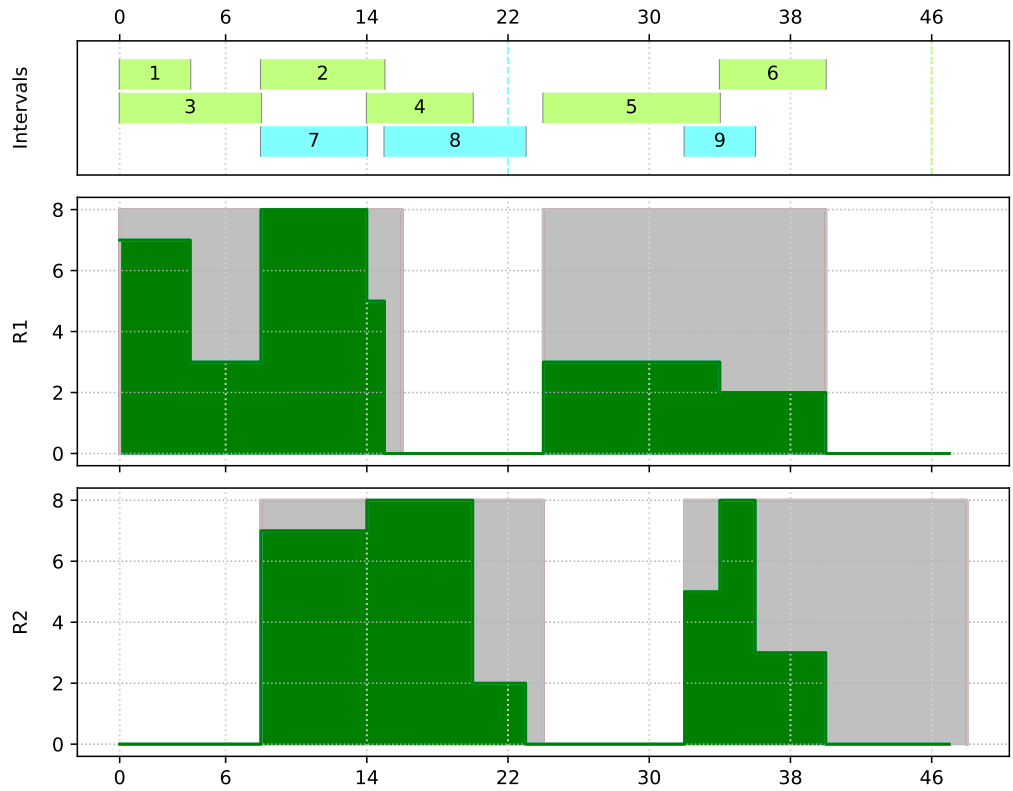
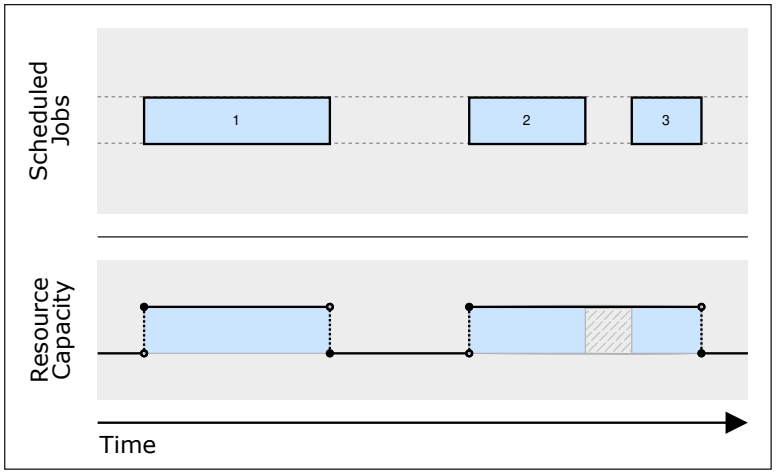


Figure 3.1 Computed example schedule. Job 9 is considered tardy as it is scheduled past its due date (time period 22). The figure consists of three panels: scheduled job intervals panel and two resource panels, each corresponding to a specific resource in the project. All panels share a common x-axis denoting time in time periods. The job interval panel illustrates the arrangement of jobs in the schedule, where their y-positions were chosen arbitrarily. In each resource panel, the availability of the corresponding resource is illustrated by a gray step function and the consuming load is illustrated by a green step function.

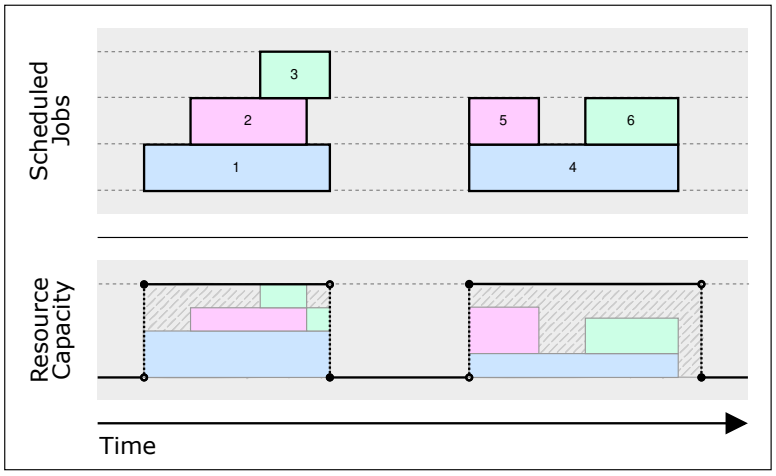
specified resource, the sequence of all uninterrupted execution periods is computed and the average length of those periods is considered the indicator value. An uninterrupted period is a sequence of jobs scheduled consecutively with no idle times between them. If there is an idle time period between two subsequent jobs, they belong to different uninterrupted periods.

Both identification indicators consider the relationship between the total duration of job executions on a resource and the duration for which the resource is idle. In a Job-Shop scheduling problem, this represents all the available information. This concept remains applicable in the RCPSP. However, due to the variability of machine load over time in the RCPSP, a binary "processing–idle" differentiation between machine states does not provide a sufficient machine-load indication. In the RCPSP, we have additional information available. By incorporating resource capacities and resource consumptions into the calculation, we can achieve a more precise result that better corresponds to the actual machine load. Figure 3.2 illustrates the difference in variability of the load between a Job-Shop machine (Figure 3.2a) and a RCPSP machine (Figure 3.2b).

We propose Machine Resource Utilization Rate (MRUR) as the adaptation of



(a) Job-Shop scheduling problem



(b) RCPSP

Figure 3.2 Examples of machine-load functions of the Job-Shop problem and the RCPSP problem. The variability in the possible machine loads during job execution between the Job-Shop problem and the RCPSP and the variability in machine loads during different periods in the RCPSP are demonstrated.

MUR and Average Uninterrupted Active Utilization (AUAU) as the adaptation of AUAD. For a resource k, the MRUR is defined as:

$$\mathrm{MRUR}_k \stackrel{\mathrm{def}}{=} \frac{\sum_{j \in \mathcal{J}} (p_j \cdot r_{jk})}{\sum_{t=1}^{C_{\max}} R_k^{(t)}},$$

where $C_{\max} \stackrel{\mathrm{def}}{=} \max_{j \in \mathcal{J}} C_j$[3]. For a resource k, the AUAU is defined as:

$$\mathrm{AUAU}_k \stackrel{\mathrm{def}}{=} \frac{\sum_{i=1}^{A_k} \mathrm{PRU}_k^{(i)}}{A_k},$$

where the Period Resource Utilization (PRU) of resource k during the uninter-

[3]In the scheduling literature, $C_{\max}$ is referred to as the *makespan* of the project. Minimizing project makespan is a common optimization goal in scheduling, and it is a simpler alternative to the total weighted tardiness we use.

rupted active period i is defined as

$$\mathrm{PRU}_k^{(i)} \stackrel{\text{def}}{=} \frac{\sum_{j \in \mathcal{J}_k^{UAP(i)}} p_j \cdot r_{jk}}{\sum_{t=a_{ki}^S}^{a_{ki}^E} R_k^{(t)}}.$$

For a resource k, $(a_{k1}^S, a_{k1}^E), \ldots, (a_{kA_k}^S, a_{kA_k}^E)$ is the the sequence of *uninterrupted active periods*, where $a_{ki}^S \in \{1, \ldots, \mathcal{T} - 1\}$ denotes the start of the period i and $a_{ki}^E \in \{a_{ki}^S + 1, \ldots, \mathcal{T}\}$ denotes the end of the period i. Similar to the definition of AUAD but with differences induced by non-unit capacities, an uninterrupted active period is a maximal set (maximal in terms of inclusion) of jobs scheduled consecutively or in parallel with no idle time occurring on the considered resource during the period. Two jobs are in the same uninterrupted active period, if and only if the execution intervals of the jobs overlap or the considered resource is not idle between the execution intervals of the jobs. The sequence of uninterrupted active periods of a resource is the transitive closure of this relation on jobs executed on the resource. Here, as opposed to the AUAD, we do not consider the duration of the periods, but the individual jobs executed during each of the periods, specifically their durations and consumptions of the evaluated resource.

In the formula for $\mathrm{PRU}_k^{(i)}$, $\mathcal{J}_k^{UAP(i)} \stackrel{\text{def}}{=} \{j \in \mathcal{J}_k : a_{ki}^S \leq S_j \leq a_{ki}^E\}$ is the set of jobs executed on resource k during the uninterrupted active period i. Recall from Section 2.2.3 that $\mathcal{J}_k = \{j \in \mathcal{J} : r_{jk} > 0\}$.

Having proposed bottleneck identification indicators for our RCPSP variant, we will formulate an algorithm utilizing those indicators in the following section.

3.1.2 Identification Indicator-based Relaxing Algorithm

In this section, we formulate the Identification Indicator-based Relaxing Algorithm (IIRA). IIRA employs a specified bottleneck identification indicator to identify bottleneck resources. It calculates the granular resource load function and uses its convolution with a suitably chosen kernel function to determine the improvement potential for granular periods. Finally, it relaxes capacity constraints in granular periods with the greatest improvement potential. Following this, a solution is obtained for the relaxed problem instance, and the proposed capacity relaxations are reduced to only include those utilized by the new solution. The Identification Indicator-based Relaxing Algorithm (IIRA) is given in Algorithm 1.

The algorithm consists of a main loop (lines 3–14), the input of which is a problem instance and its solution, the output of which is a modified problem instance and its solution. We describe the individual steps performed by the algorithm in the following outline.

1. First, the solution is evaluated using the given identification indicator and the bottleneck resource is identified by its maximal value of the identification indicator (lines 4 and 5).

2. The granular load of the bottleneck resource is computed and the improvement potentials for granular periods are computed using convolution (lines 6 and 7). The granular load of a resource indicates how is the resource being utilized during granular periods. High utilization could indicate a potential bottleneck. However, it is uncertain whether the bottleneck occurs in the

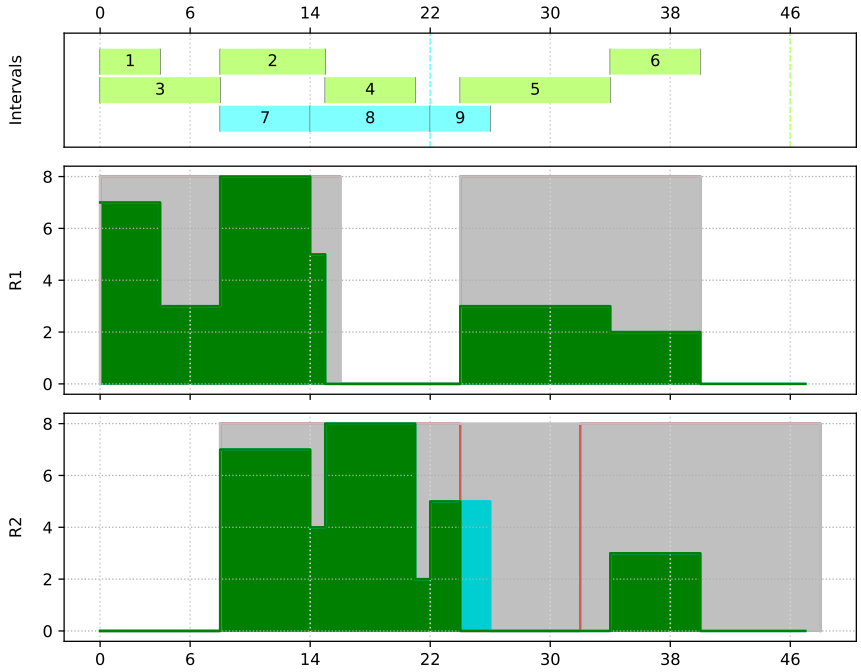


Figure 3.3 Computed example of an intermediate schedule obtained by the IIRA ($\mathrm{I} = \mathrm{AUAU}, \mathrm{G} = 8, \mathrm{C} = f_{\mathrm{around}}, I_{\max} = 1, P_{\max} = 1, \Delta = 8$). The capacity of resource 2 is increased during the granular period spanning over time periods $\{24, \ldots, 31\}$. Subsequently, job 9 can be scheduled earlier. The newly introduced capacity utilized by the job 9 is highlighted.

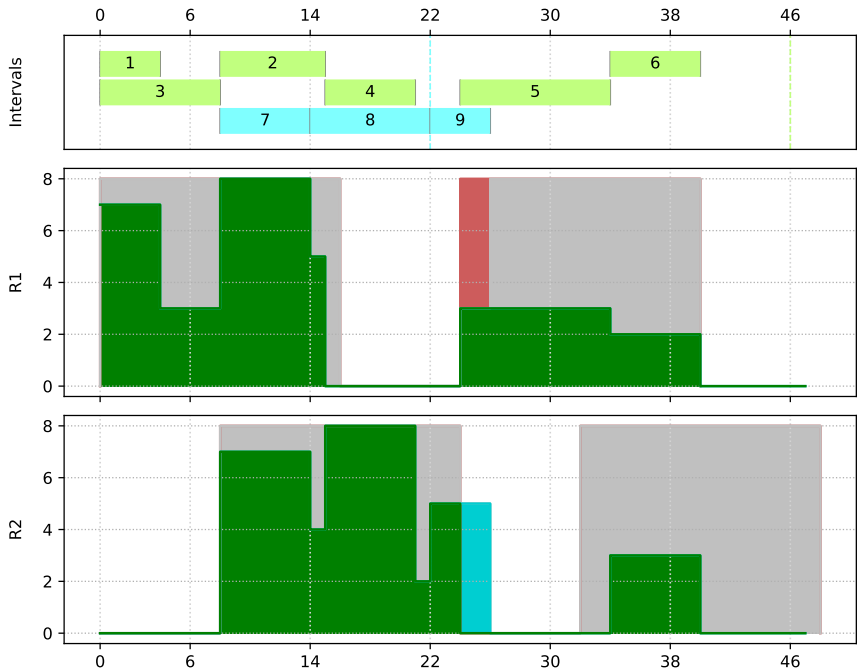


Figure 3.4 Computed example of a schedule with reduced resource capacity functions obtained by the IIRA ($\mathrm{I} = \mathrm{AUAU}, \mathrm{G} = 8, \mathrm{C} = f_{\mathrm{around}}, I_{\max} = 1, P_{\max} = 1, \Delta = 8$). The capacity of resource 2 is reduced during the granular period spanning over time periods $\{24, \ldots, 31\}$ to exclude unused capacity additions. The only additional capacity left is exactly the highlighted capacity consumed by the job 9.

Algorithm 1 Identification Indicator-based Relaxing Algorithm (IIRA)

Parameters: Identification indicator I, granularity G, convolution mask C, iterations limit $I_{\max}$, improvement periods limit $P_{\max}$, capacity improvement Δ

Input: Solution S to a problem instance $\mathcal{I}$

1: $PC \leftarrow \lceil \mathcal{T}/\mathrm{G} \rceil$ ▷ The number of granular periods

2: $\mathcal{I}^* \leftarrow \mathcal{I}$, $S^* \leftarrow S$ ▷ Modified instance and its solution, initially copies of the original instance and solution

3: **repeat:**

4: Evaluate S^* using I, obtaining: $\mathrm{I}_k \; \forall k \in \mathcal{R}$

5: Identify bottleneck resource: $k^* \leftarrow \mathrm{argmax}_k \, \mathrm{I}_k$

6: Compute granular resource load for k^*:
$L_{k^*} \leftarrow$ GranularResourceLoad(k^*, $\mathcal{I}^*$, S^*, PC)

7: Compute improvement potential of periods: $\Psi \leftarrow L_{k^*} * \mathrm{C}$

8: Find improvement periods:
$p_1, \ldots, p_{P_{\max}} \leftarrow$ periods with the highest potential $\Psi(i)$

9: **for** $i \in \{p_1, \ldots, p_{P_{\max}}\}$ **:**

10: $R^*_{k^*} \leftarrow$ IncreaseGranularPeriodCapacity(i, $R^*_{k^*}$, G, Δ)

11: Find solution S^* to the modified instance $\mathcal{I}^*$

12: $R^*_1, \ldots, R^*_m \leftarrow$ ReduceCapacityChanges($\mathcal{I}^*$, S^*, R_1, ..., R_m)

13: $\mathcal{A}^{\mathcal{I}^*}, \mathcal{M}^{\mathcal{I}^*} \leftarrow$ FindAdditionsAndMigrations($\mathcal{I}^*$, S^*)

14: **for** $I_{\max}$ **iterations**

Output: Modified instance $\mathcal{I}^*$ and its solution S^*, additions $\mathcal{A}^{\mathcal{I}^*}$, migrations $\mathcal{M}^{\mathcal{I}^*}$

Note: In the call to ReduceCapacityChanges (statement 12), R_{k^*} from the original instance is given as the original capacity function of the resource k^*.

highly-utilized granular period, or whether the high utilization is a consequence of a bottleneck in a preceding or a following granular period. This is why the granular load is convolved with a kernel function of choice, which propagates the information about high resource utilization to consecutive granular periods. The result of the convolution is trimmed to only contain "valid" values, i.e. values computed directly on the interval $\{1, \ldots, PC\}$.

3. A small subset of granular periods is selected for the increase in capacity. The granular periods with the highest improvement potentials are chosen (line 8). The capacity of the bottleneck resource is increased during the selected granular periods by a specified amount (line 10). The modified resource capacity function of the bottleneck resources forms, together with the unchanged capacity functions of other resources, a new modified problem instance. Note that the algorithm does not consider the target order o in any way when identifying bottlenecks (step 2.) nor when selecting granular periods for capacity constraints relaxation.

4. Finally, a new solution to the modified problem instance is found (line 11). An example of a modified solution is given in Figure 3.3. Based on this solution, reduced capacity functions are computed to exclude capacity

relaxations not utilized in the solution (line 12). An example of a schedule with reduced resource capacity functions is given in Figure 3.4. The reduction involves the capacity functions of all the resources, not only of the bottleneck resource. This is because, in subsequent iterations of the algorithm, the introduction of new relaxations often leads to changes in the solution schedule. Such changes might cause previous relaxations to no longer be necessary, so in turn, every resource capacity function is reduced. Note that the function ReduceCapacityChanges takes the original resource capacity functions as input and the reductions are made with respect to them. As a final step, capacity migrations are computed in the reduced capacity functions to best utilize the existing unused capacities (line 13). Any remaining capacity requirements are then fulfilled with capacity additions.

The algorithm utilizes multiple additional procedures and functions. For brevity, we exclude the detailed specifics of the procedures and instead offer short overviews of each procedure. Complete pseudocodes of the procedures are available in Appendix A.1.

- GranularResourceLoad computes a granular resource load for a given resource. The granular load is a function mapping granular periods to the cumulative sum of the load of the resource over the specified granular period. Example of a computed granular load can is illustrated in Figure 3.5.

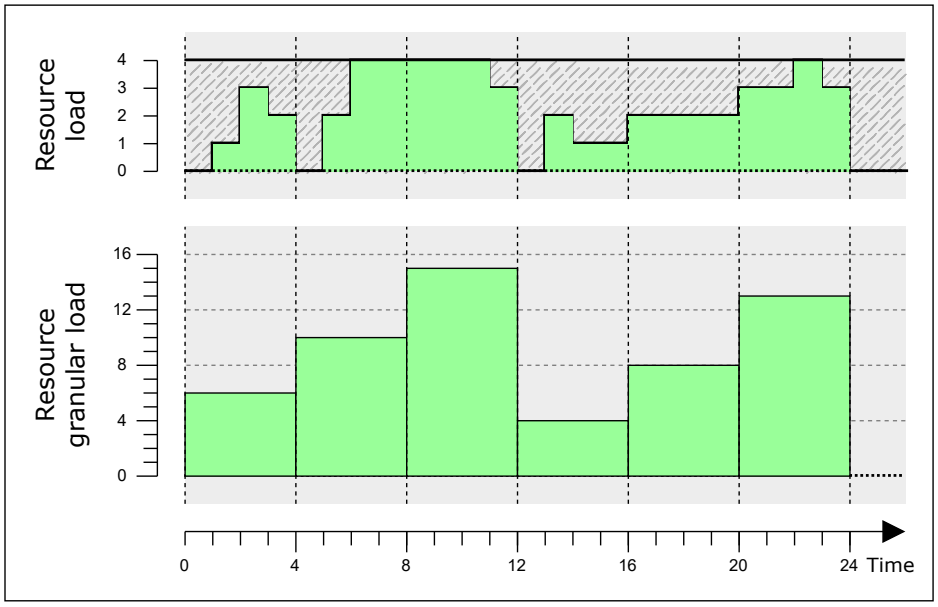


Figure 3.5 Resource granular load example. The first panel illustrates the resource load representing the consumption of jobs. The second panel illustrates the computed granular load with the granularity of 4 time periods.

- IncreaseGranularPeriodCapacity increases the values of the given capacity function during the specified granular period. The capacity is increased in each time period covered by the granular period, determined by the specified granularity.

- ReduceCapacityChanges constructs reduced capacity functions for the given problem instance based on the original resource functions and the actual load of the instance resources (computed from the given solution). This function reduces redundant capacity additions introduced by former relaxations so that the resource capacity functions do not contain capacity additions not utilized by the solution.

- FindAdditionsAndMigrations finds capacity additions and migrations, as defined in Section 1.3.3, for the resources of the given problem instance. In the same section, we discussed that in real-world production systems capacity migrations are preferred over capacity additions due to their comparatively small execution cost. Thus, we first find all possible migrations to utilize existing capacities. Then, when no other migrations are possible, the remaining capacity requirements are fulfilled by introducing capacity additions.

3.2 Extended solution

In this section, we present a novel method for detecting bottlenecks and relaxing related constraints in the RCPSP. The method is based on finding improvement intervals in partially relaxed versions of the given problem. A small subset of the improvement intervals is then selected and capacity constraints corresponding to the selected improvement intervals are relaxed. The primary goal is to identify relaxations that focus specifically on the target order. By doing so, we hope to achieve great improvements in the tardiness of the target order while maintaining low capacity modification costs and induced schedule differences.

3.2.1 Preliminaries

Before we formulate the Schedule Suffix Interval Relaxing Algorithm, we need to state a few definitions and ideas upon which the algorithm is designed. First, we define the *suffix-relaxed schedule* as a modification of an obtained schedule solution where the algorithm finds improvement intervals. We then define the *left-shift closure* as a tool for focusing the search for improvement intervals towards improving the tardiness of the target order.

Definition 4 (Suffix-relaxed schedule)**.** *Let $S = (S_1, \ldots, S_n)$ be a schedule to a problem instance $\mathcal{I}$. Given a time period $t \in \{1, \ldots, \mathcal{T}\}$, the* suffix-relaxed schedule *for the time period t is given by $\vec{S}^{(t)} = (\vec{S}_1^{(t)}, \ldots, \vec{S}_n^{(t)})$, where*

$$\vec{S}_j^{(t)} \stackrel{def}{=} \begin{cases} S_j & \text{if } S_j \leq t; \\ \max\left\{\vec{S}_i^{(t)} + p_i : i \to j \in \mathcal{P}\right\} & \text{otherwise.} \end{cases}$$

For a given job j, the value of $\vec{S}_j^{(t)}$ depends only on the values $\vec{S}_i^{(t)}$ of its precedence predecessors i, given by precedences $i \to j$. Since the precedence graph is directed and acyclic, all values of $\vec{S}^{(t)}$ are well-defined and the definition is correct.

The suffix-relaxed schedule for a time period t is a modification of the original schedule where the start times of jobs starting in time periods up to t remain unchanged, and the start times of jobs scheduled to start later can be shifted to earlier time periods, constrained only by precedence constraints. This essentially relaxes resource capacity constraints for all jobs that, in the original schedule, start after the time period t.

The idea is that a job scheduled during the later time periods could not be scheduled earlier due to the lack of remaining capacity on the required resources, assuming sufficient slack in precedence constraints. By fully relaxing the resource capacity constraints in the suffix of a schedule, we can compute potential starting times for jobs scheduled in that suffix, had they not been constrained by the resource capacity constraints. Following this, we can observe the potential improvements in the starting times and the consequent required resource capacity relaxations.

Definition 5 (Left-shift closure)**.** *Let $S = (S_1, \ldots, S_n)$ be a schedule to a problem instance $\mathcal{I}$. A* left-shift closure *of a job $j \in \mathcal{J}$ is the set $\mathcal{L}(j) \subseteq \mathcal{J}$, where:*

i) $j \in \mathcal{L}(j)$

ii) All precedence predecessors directly preceding in the schedule are included.

$$(\forall i \to j \in \mathcal{P}) : C_i = S_j \implies \mathcal{L}(i) \subset \mathcal{L}(j)$$

iii) All jobs consuming a common resource directly preceding in the schedule are included.

$$(\forall k \in \mathcal{R}, r_{jk} > 0)(\forall i \in \mathcal{J}_k) : C_i = S_j \implies \mathcal{L}(i) \subset \mathcal{L}(j)$$

iv) If j is scheduled exactly at the start of a working shift, all jobs scheduled at the end of the previous working shift are included.

$$(\forall k \in \mathcal{R}, r_{jk} > 0, R_k^{(S_j-1)} = 0)(\forall i \in \mathcal{J}_k) : \\ \mathrm{ps}_k(S_j) - p_j \leq C_i \leq \mathrm{ps}_k(S_j) \implies \mathcal{L}(i) \subset \mathcal{L}(j)$$

where $\mathrm{ps}_k(t) \stackrel{def}{=} \max\{t' \in \{1, \ldots, t-1\} : R_k^{(t')} > 0\}$.

The left-shift closure of a job j defines the set of all jobs preventing the job j from starting in an earlier time period. The only exception to this interpretation is the condition i), which simplifies the inductive definition.

Condition ii) states that a direct predecessor i of the job j, indicated by the precedence constraint $i \to j$, is included in $\mathcal{L}(j)$ if the jobs are scheduled consecutively. Jobs i and j are scheduled consecutively if the execution of the job i ends at the exact same time period where the execution of the job j starts, i.e. $C_i = S_j$.

Condition iii) involves all jobs scheduled consecutively with the job j, which share at least one required resource. Jobs can be executed on multiple resources, but sharing just one resource is sufficient for the jobs to influence each other. This resource requirement overlap can delay the job j if its predecessor job i's resource consumption prevents the job j from being scheduled earlier.

Algorithm 2 Schedule Suffix Interval Relaxing Algorithm (SSIRA)

Parameters: Iterations limit $I_{\max}$, improvement intervals limit $IT_{\max}$, interval sort key $\mathcal{K}$

Input: Solution S to a problem instance $\mathcal{I}$, target order o

1: $\mathcal{I}^* \leftarrow \mathcal{I}$, $S^* \leftarrow S$ ▷ Modified instance and its solution, initially copies of the original instance and solution

2: **repeat:**

3: $\quad \chi_1, \ldots, \chi_{IT_{\max}} \leftarrow$ FindIntervalsToRelax($\mathcal{I}^*$, S^*, $IT_{\max}$, $\mathcal{K}$, o)

4: $\quad R_1^*, \ldots, R_m^* \leftarrow$ ModifyResourceCapacities($\mathcal{I}^*$, $\chi_1, \ldots, \chi_{IT_{\max}}$)

5: $\quad$ Find solution S^* to the modified instance $\mathcal{I}^*$

6: $\quad R_1^*, \ldots, R_m^* \leftarrow$ ReduceCapacityChanges($\mathcal{I}^*$, S^*, R_1, ..., R_m)

7: $\quad \mathcal{A}^{\mathcal{I}^*}, \mathcal{M}^{\mathcal{I}^*} \leftarrow$ FindAdditionsAndMigrations($\mathcal{I}^*$, S^*)

8: **for** $I_{\max}$ **iterations**

Output: Modified instance $\mathcal{I}^*$ and its solution S^*, additions $\mathcal{A}^{\mathcal{I}^*}$, migrations $\mathcal{M}^{\mathcal{I}^*}$

Algorithm 3 FindIntervalsToRelax

Input: Problem instance $\mathcal{I}$, its solution S, improvement intervals limit $IT_{\max}$, interval sort key $\mathcal{K}$, target order o

1: **for** $t \in \{1, \ldots, \mathcal{T}\}$ **:** $\vec{S}^{(t)} \leftarrow$ ComputeSuffixRelaxedSchedule($\mathcal{I}$, S, t)

2: $\mathcal{L}(o) \leftarrow$ ComputeLeftShiftClosure($\mathcal{I}$, S, o)

3: $X \leftarrow \emptyset$

4: **for** $j \in \mathcal{L}(o)$ **:**

5: $\quad s \leftarrow \min_t \left\{ \vec{S}_j^{(t)} : \vec{S}_j^{(t)} < S_j \right\}$ ▷ Find the earliest improvement

6: $\quad X \leftarrow X \cup \{(j,\ s,\ s + p_j)\}$

7: $\chi_1, \ldots, \chi_{IT_{\max}} \leftarrow$ first $IT_{\max}$ intervals from X ordered by $\mathcal{K}$

Output: Improvement intervals $\chi_1, \ldots, \chi_{IT_{\max}}$, a set of 3-tuples $(j, s, e) \in \mathcal{J} \times \{1, \ldots, \mathcal{T}\}^2$

Lastly, condition iv) involves jobs at the end of previous working shifts. Assuming sufficient slack in precedence constraints, the job j starts exactly at the start of a working shift because it could not have been scheduled at the end of the previous working shift due to the lack of remaining capacities on its required resources. Jobs scheduled at the end of the previous working shift consume the required resources and thus prevent the job j from being scheduled there.

3.2.2 Schedule Suffix Interval Relaxing Algorithm

Having presented the main concepts in the previous section, we now formulate the Schedule Suffix Interval Relaxing Algorithm (SSIRA) in Algorithm 2. The formulation of the algorithm itself is quite simple. The core procedure is encapsulated in the FindIntervalsToRelax function, formulated in Algorithm 3.

The SSIRA consists of a main loop (lines 2–8), the input of which is a problem instance and its solution, the output of which is a modified problem instance and its solution. In each iteration of the main loop, the algorithm first finds

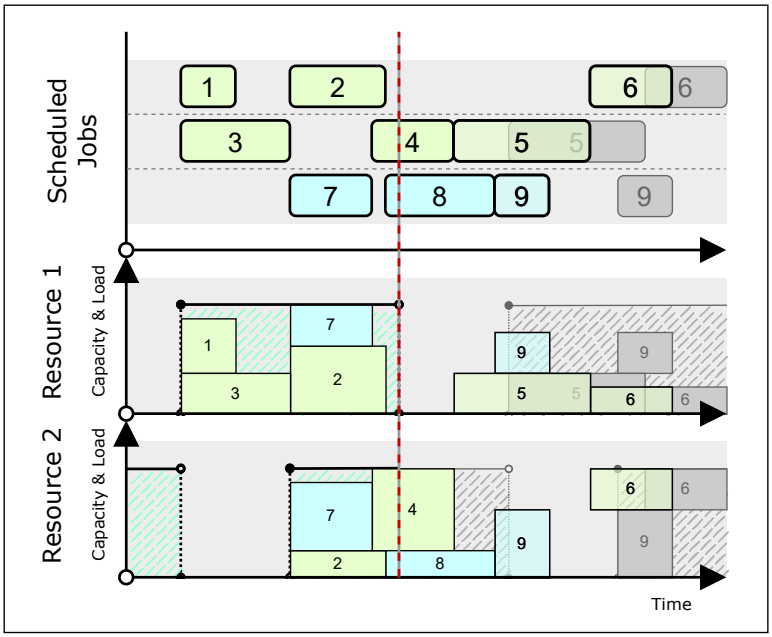


Figure 3.6 Example of a suffix-relaxed schedule based on the schedule given in Figure 1.2. The red vertical line represents the time period for which the suffix-relaxed schedule was computed. The resource capacity constraints were relaxed for jobs 5, 6, and 9 as they were scheduled later than the specified time period. We observe that in the relaxed schedule, job 9 would require only small capacity additions on both resources, should it be scheduled on them. Moreover, most of the requirements could be satisfied by migrating unused capacities between the resources.

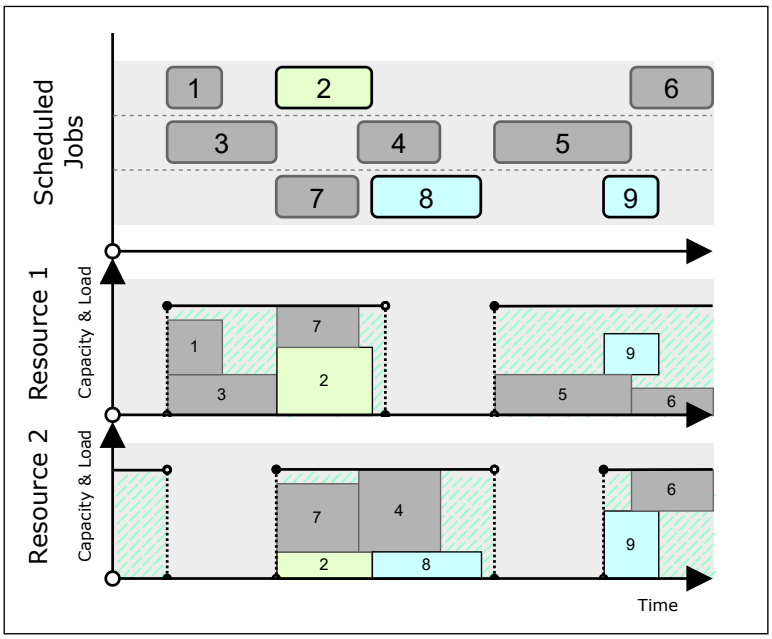


Figure 3.7 Example of a left-shift closure computed for the job 9 in the schedule given in Figure 1.2. Here, the left-shift closure of the job 9 is the set $\mathcal{L}(9) = \{2, 8, 9\}$. The included jobs are highlighted; jobs not included are dimmed. Job 9 is trivially included by condition i). Job 8 is included by condition iv) — pauses in resource working shifts. Job 2 is included by condition iii) — resource predecessor of the job 8 on resource 2. Conversely, job 3 ends at the same time period job 2 starts, however, they are not precedence predecessors and thus job 3 is not included.

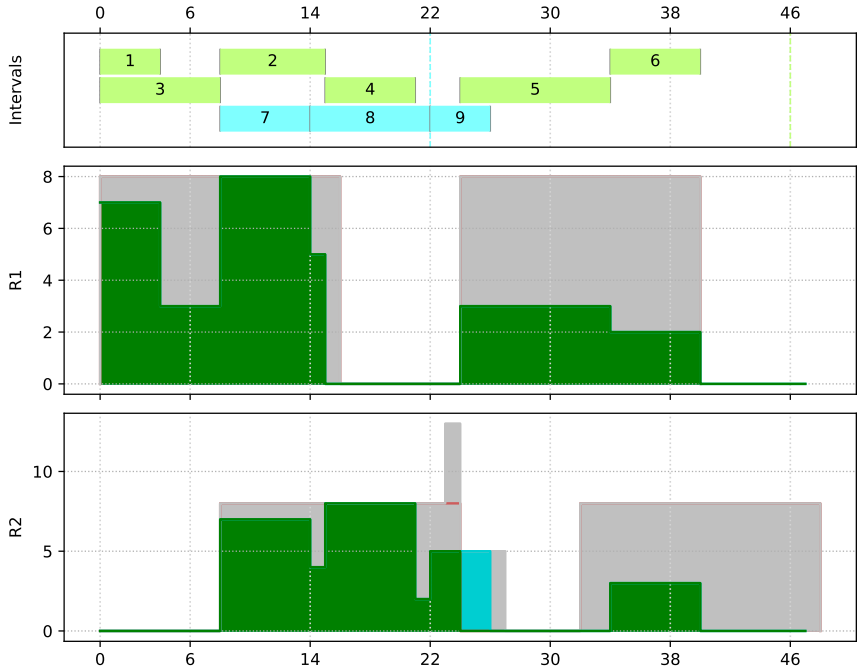


Figure 3.8 Computed example of an intermediate schedule obtained by the SSIRA ($I_{\max} = 1, IT_{\max} = 1, \mathcal{K} = \mathcal{K}_t$). The capacity of resource 2 is increased over the improvement interval spanning the time periods $\{22, \ldots, 27\}$. Subsequently, job 9 can be scheduled earlier. The newly introduced capacity utilized by the job 9 is highlighted. Note that the job 9 is scheduled earlier than anticipated by the improvement interval.

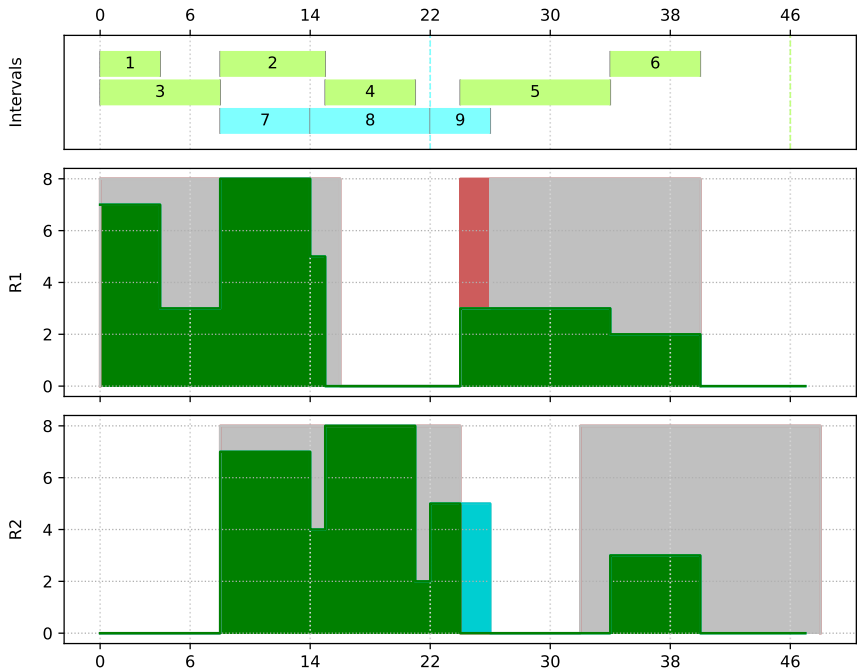


Figure 3.9 Computed example of a schedule with reduced resource capacity functions obtained by the SSIRA ($I_{\max} = 1, IT_{\max} = 1, \mathcal{K} = \mathcal{K}_t$). The capacity of resource 2 is reduced, removing unused capacities introduced in Figure 3.8. The only additional capacity left is the highlighted capacity consumed by the job 9 migrated from resource 1. Note that this schedule is identical to the schedule obtained by the IIRA in Figure 3.4.

improvement intervals (line 3). Then, resource capacity functions are modified based on these intervals (line 4). The subsequent steps (lines 5–7) mirror those of the IIRA, as stated in detail in Section 3.1.2. A new solution to the modified problem instance is found, the modified resource capacity functions are reduced based on that solution, and capacity migrations and additions are computed in the reduced capacity functions. An example of a schedule obtained in the modified instance is given in Figure 3.8 and that schedule with reduced capacity functions is given in Figure 3.9. Note that in this example, both the IIRA and the SSIRA obtained the same solution, given in Figure 3.4 and Figure 3.9 respectively.

The FindIntervalsToRelax function consists of initializations, the search for improvement intervals, and a preference-selection of intervals. We describe the individual steps performed in the function in the following outline.

1. Suffix-relaxed schedules are computed for each time period within the scheduling horizon (line 1). These schedules represent all possible job-interval relaxations, from which potential improvement intervals are subsequently identified and selected. Following this, the left-shift closure of the target order is computed (line 2). This closure represents the set of jobs considered for improvement.

2. Potential improvement intervals are identified iteratively for jobs within the left-shift closure of the target order (lines 3–6). For each job in the closure, the start of the potential improvement interval is determined as the earliest improving time for that job across all suffix-relaxed schedules (line 5). The start time of a job in a suffix-relaxed schedule is considered improving if it is earlier than the start time in the original schedule. Then, the potential improvement interval for the job is constructed (line 6), incorporating the identified earliest improving time and the job's execution duration. The constructed potential improvement interval is added to the set of potential improvement intervals for the subsequent selection of preferred improvement intervals.

3. Predefined number of improvement intervals is selected based on a specified sort key (line 7). The potential improvement intervals are ordered according to the sort key, and the first intervals from this ordering are selected as the proposed improvement intervals.

The SSIRA and the FindIntervalsToRelax function both utilize multiple additional procedures and functions. The ReduceCapacityChanges and FindAdditionsAndMigrations functions used by the SSIRA are the same functions as those used by the IIRA. Their overview was given in Section 3.1.2. For brevity, we exclude the detailed specifics of the remaining functions and instead offer their short overviews. Complete pseudocodes of the functions are available in Appendix A.1.

- ModifyResourceCapacities modifies the resource capacity functions of a given problem instance by increasing their capacities within the specified improvement intervals. Each improvement interval is associated with a particular job. For every resource required by that job, its resource capacity function is increased by the job's required resource consumption of that resource over the duration of the improvement interval.

- ComputeSuffixRelaxedSchedule computes the suffix-relaxed schedule for a specified time period, as defined in Definition 4. Initially, the topological order of the jobs in the instance is computed. Then, for each job considered in the topological order, its start time in the suffix-relaxed schedule is determined: if the job was scheduled up to the specified time period, its original start time is used; otherwise, the latest end time of its precedence predecessors is computed in the suffix-relaxed schedule, and this value is used. This maximum is well-defined due to the jobs being processed in topological ordering and thus, all values have been determined by the time they are first considered in the maxima.

- ComputeLeftShiftClosure computes the left-shift closure of a specified job, as defined in Definition 5. The precedence graph is traversed in a breadth-first manner, starting from the specified job. The traversal considers only the jobs that correspond to the defining conditions ii) to iv).

4 Numerical experiments

In this chapter, we evaluate the performances of the Identification Indicator-based Relaxing Algorithm and the Schedule Suffix Interval Relaxing Algorithm proposed in the previous chapter. We first design benchmark instances that model the addressed problem and choose ranges of parameters for each algorithm, creating evaluation parameter sets. Then, we conduct the experiments, make several observations about the outcomes, and discuss the achieved results.

4.1 Setup

In this section, we present benchmark instances used for evaluating the proposed methods and algorithms, we describe the modification process for creating problem instances suited to our studied problem, explain how we obtain (near) optimal solutions to the modeled problem using a constraint programming solver, and discuss the algorithms' parameters used in the following experiments.

All processing, including manipulating with problem instances, solving constraint programming models, and conducting experiments, all described in the following sections, was done in Python using a library developed specifically for the purposes of this thesis. More about this library and running the experiments can be found in Appendix A.2.

4.1.1 Problem instances

Kolisch and Sprecher (1997) created the PSPLIB[1] — set of benchmark instances for the RCPSP. This set has since been used to evaluate and compare many results in the literature, for example, see some recent papers from Bianco and Caramia (2011), Cheng et al. (2015), or Elsayed et al. (2017).

We use and modify specific instances from the PSPLIB single-mode instance set. Instances from this set model the standard RCPSP. They consist of 30, 60, 90, or 120 jobs, and 4 renewable resources with fixed capacities. Each job has an execution duration and consumption requirements for each of the resources defined. Precedences between jobs are stated, forming a single-component precedence graph. The goal when scheduling such problem instances is usually minimizing the project makespan, or minimizing the project tardiness with respect to a specified project due date. The PSPLIB instances were generated using diverse parameter settings. For each setting, 10 random seeds were used to create a 10-instance batch.

To accurately model our studied problem, we introduce several modifications to the original instances. Namely, we split the precedence graph to create individual order components, introduce job due dates, and introduce time-variable resource capacities. Additionally, to ensure feasibility when modeling specific production systems, we scale down job durations and resource consumptions. Following are the modification steps in more details.

Initially, to model a system with fewer than the original four resources, we remove the unwanted resources from the problem instance and optionally adjust

[1] Available online at `https://www.om-db.wi.tum.de/psplib/`

the resource consumptions of jobs. If removing all resources consumed by a job results in that job having no consumption requirements, the total removed consumption of that job is distributed among the remaining resources. We then proceed with modifications common for all instances.

First, we split the single-component precedence graph into disconnected components, creating an inforest. We do this by ordering the jobs topologically and selecting one of the last topological generations as seed jobs[2]. Incrementally, starting with the preceding generation and continuing in the reverse order of topological generations, each job from preceding generations selects a single successor precedence to connect to a successor job. Analogously, each job from succeeding generations selects a single predecessor precedence to connect to a predecessor job. After splitting the graph into an inforest, the sink-roots of the intrees are selected as orders $\mathcal{O}$.

Second, we limit the resource availabilities to simulate working shifts. We model three-shift production systems, i.e. with three possible 8-hours working shifts: the morning shift starting from 8 to 14, the afternoon shift starting from 14 to 22, and the night shift starting from 22 to 6 of the next day. We use periodical availability intervals, sub-intervals of $\{1, \ldots, 24\}$, to denote that a resource is available each day during the specified working shift. During those shifts, the resource capacity is set to its defined shift capacity $R_k^{(-)}$; outside those working shifts, the capacity is set to 0.

Third, we introduce job due dates. As stated in Sections 1.1 and 1.2, it is sufficient to set due dates for order jobs $j \in \mathcal{O}$ only. Those were set manually to simulate a continuous distribution of orders in time as they would appear in a real order-based manufacturing system.

Finally, if needed, the durations of jobs are scaled down appropriately. Previous modifications might have caused the problem instance to be infeasible — a solution to the problem instance could no longer be found. Such infeasibility can be introduced by limiting resource availabilities to shifts where the shifts of two resources consumed by a job do not overlap, or overlap for a time duration smaller than the required execution duration of the job. In such cases, the durations of jobs are scaled down appropriately. Given that preemption is not allowed and job resource consumptions have to be concurrent, the maximal duration $p_{\max}$ of a job in a problem instance is set to be the length of maximal overlap of all the resources' availabilities in the problem instance. Then, the durations of all jobs are scaled down as follows:

$$p_j \leftarrow \frac{p_j \cdot p_{\max}}{\max\{p_j : j \in \mathcal{J}\}}.$$

We propose 8 problem instance groups, each consisting of 5 individual instances. The premise is that instances within each group will share similar properties and that we will be able to analyze aggregated results of evaluations on each group and draw reasonable conclusions from those results. Table 4.1 contains an overview of the problem instance groups. Each of our instance groups is based on 5 instances from some PSPLIB 10-instance batch, where every instance in the batch has similar resource and precedence properties, as mentioned earlier. The 5 specific

[2]The depth of the selected generation influences the number of created components and the overall structure of the created intree forest.

instances were chosen based on similar precedence graphs that were formed by the precedence graph-splitting process described above.

4.1.2 Solving the constraint programming model

We use the IBM Decision Optimization CPLEX (DOcplex)[3] Python API for modeling the problems via constraint programming. We then utilize the IBM ILOG Constraint-Programming Optimizer (CP Optimizer)[4] for finding optimal solutions to the modeled problems.

Solver time limit was set to 10 seconds. Upon reaching the time limit without optimality verification, the best solution found so far was used. This follows from the argument that for the proposed methods to be applicable in real-world manufacturing systems, they need to be reasonably fast — not much time can be spent finding optimal solutions to the problem instances[5].

We use solver warm-starting to speed up consecutive solution finding of modified problem instances. Models of modified instances usually differ only slightly from the models of the original problem instance. This means, that an (optimal) solution to the original problem instance might remain a feasible (if not directly an optimal) solution to the modified version of the problem instance. If not, it is still probable that the desired feasible solution to the modified problem instance does not differ much from the initial solution to the original problem instance.

Warm-starting of the constraint programming solver utilizes this by starting the search not from a random initial solution, but from a particular given solution. With high probability, a feasible solution will be found shortly which consequently speeds up the process of finding an optimal one. In summary, consecutively finding solutions to modified problem instances will usually be faster than finding the initial solution to the original problem instance.

4.1.3 Algorithm parameters

For both evaluated algorithms, we choose multiple different combinations of parameters. Then, on each problem instance, the algorithms are evaluated using every parameter combination. As a result, for each instance and for each algorithm we will have a set of evaluations, each evaluation corresponding to a specific parameter combination. The parameters of the algorithms will be constructed from all possible combinations of the following parameter values, separate for each algorithm.

(i) *Identification Indicator-based Relaxing Algorithm*

- Bottleneck identification indicator $\mathrm{I} \in \{\mathrm{MRUR}, \mathrm{AUAU}\}$

[3]See online at `https://ibmdecisionoptimization.github.io/docplex-doc/cp/index.html`.

[4]See online at `https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer`. (IBM, 2024)

[5]Such an argument could lead to the preference of heuristic approaches for finding problem instance solutions. However, as discussed in Section 2.1.2, the benefits of utilizing an exact constraint programming solver outweigh the herein-mentioned drawbacks.

Instances	Basefiles	$\lvert\mathcal{J}\rvert$	$\lvert\mathcal{R}\rvert$	Resource shifts
instance01*	j3011_4.sm, j3011_2.sm, j3011_5.sm, j3011_6.sm, j3011_9.sm	30	4	R1: M | A | R2: M | A | R3: M | A | R4: M | A |
instance02*	j3010_2.sm, j3010_4.sm, j3010_5.sm, j3010_7.sm, j3010_8.sm	30	2	R1: M | A | N R2: M | A |
instance03*	j6010_7.sm, j6010_8.sm, j6010_9.sm, j6010_6.sm, j6010_2.sm	60	1	R1: M | A |
instance04*	j6010_7.sm, j6010_8.sm, j6010_9.sm, j6010_6.sm, j6010_2.sm	60	1	R1: M | A |
instance05*	j6011_10.sm, j6011_2.sm, j6011_3.sm, j6011_6.sm, j6011_7.sm	60	4	R1: M | A | R2: M | A | R3: | A | N R4: M | A | N
instance06*	j6013_6.sm, j6013_2.sm, j6013_3.sm, j6013_5.sm, j6013_10.sm	60	4	R1: | A | R2: | A | R3: | A | R4: | A |
instance07*	j1201_1.sm, j1201_3.sm, j1201_6.sm, j1201_7.sm, j1201_10.sm	120	4	R1: M | A | R2: M | A | R3: M | A | R4: M | A |
instance08*	j1205_1.sm, j1205_5.sm, j1205_6.sm, j1205_7.sm, j1205_9.sm	120	2	R1: M | A | R2: M | A |

Table 4.1 Overview of the experiment instances used. For each instance group, the "Basefiles" column contains names of the PSPLIB instances used to create the five instances from the group. Following are columns describing the instance properties of the instance group — the number of jobs, the number of resources, and the resource availability profiles. The resource availability profiles consist of three possible shifts — morning (M), afternoon (A), and night (N) — representing that the resource is available each day (period of 24 time periods) during the corresponding multiples of the time periods $\{7, \ldots, 14\}$, $\{15, \ldots, 22\}$, and $\{1, \ldots, 6\} \cup \{23, 24\}$ respectively.

- Granular period granularity G $\in \{4, 8\}$
- Improvement potential convolution kernel C $\in \{f_{\text{pre}}, f_{\text{around}}, f_{\text{post}}\}$
- Number of iterations $I_{\max} \in \{1, 2, 3\}$
- Number of improvement periods $P_{\max} \in \{1, 2, 3, 4\}$
- Capacity improvement $\Delta \in \{4, 10\}$

(ii) *Schedule Suffix Interval Relaxing Algorithm*

- Interval sort key $\mathcal{K} \in \{\mathcal{K}_t, \mathcal{K}_{\Delta S}\}$
- Number of iterations $I_{\max} \in \{1, 2, 3\}$
- Number of improvement intervals $IT_{\max} \in \{1, 2, 3, 4, 5, 6\}$

Regarding the possible improvement potential convolution kernels of the IIRA, see diagrams of the kernels in Figure 4.1. Regarding the SSIRA and its possible interval sort keys, the $\mathcal{K}_t$ is a sort key which orders the improvement intervals by the time periods during which the intervals are proposed in a descending order. Using this key, improvement intervals identified later in the schedule precede improvement intervals identified earlier in the schedule. The $\mathcal{K}_{\Delta S}$ orders the improvement intervals by the improvement in job start times they represent. Using this key, improvement intervals proposing large improvements of job start times precede improvement intervals which propose smaller improvements of job start times.

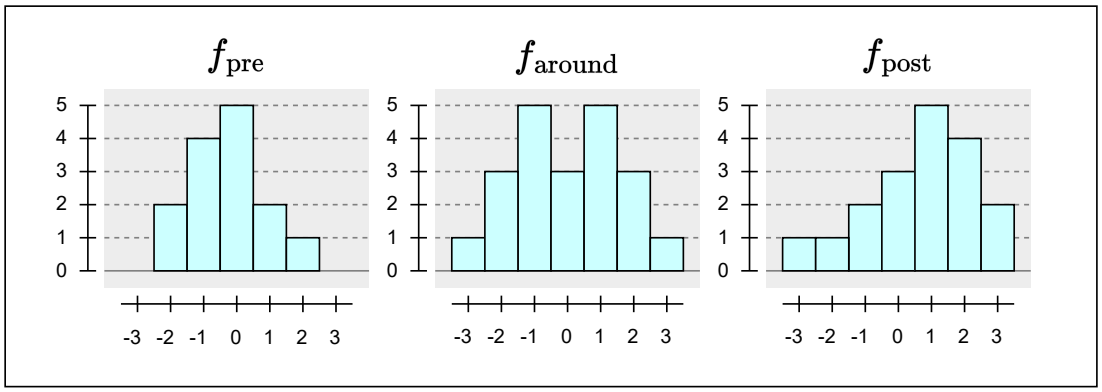


Figure 4.1 Improvement convolution kernels used in the IIRA algorithm.

The parameter value ranges were set based on prior testing of the algorithms. We do not rule out the possibility that a promising combination of parameters with great performance potential was not included in the constructed set of combinations, nor do we claim those ranges are exhaustive in terms of acceptable values. The value ranges were set to create a broad spectrum of algorithms with comparable performances.

When considering the number of iterations for example, the maximum value was set reasonably low as both algorithms tend to propose impractical relaxations after several iterations. For example, the algorithms would propose to disregard any planned resource working shift pauses or double the resource capacities throughout most of the scheduling horizon. While such proposals would certainly

achieve the desired improvement, they are not realistic within the context of real-world manufacturing systems.

The combinations of all algorithm parameters for the IIRA form a total of 288 different parameter combinations and for the SSIRA a total of 36 different parameter combinations. We justify this apparent difference in the parameter combinations totals simply by the fact that the IIRA is an algorithm with more parameters by design. To sufficiently evaluate the IIRA, we provide several values for each parameter. Reducing the choices would leave out interesting combinations of parameters. Conversely, additional value options for the parameters of the SSIRA would not introduce any more interesting combinations. We did not come up with other improvement interval sorting keys which would seem practical and we already discussed the impracticality of a larger number of algorithm iterations.

4.1.4 Methods of evaluation

To evaluate an algorithm, we follow the procedure stated in Section 1.4. We start with a given problem instance $\mathcal{I}$ for which we obtain a solution S. Then, we run the algorithm with specified parameters, the output of which is the modified instance $\mathcal{I}^*$ and its solution S^*. Finally, we compute the following metrics:

- Tardiness improvement
$$\Delta T_o \stackrel{\text{def}}{=} T_o - T_o^*.$$
This metric is arguably the most important, as it tells us how much the algorithm was able to reduce the tardiness of the target order o.

- Solution difference
$$\Delta S \stackrel{\text{def}}{=} \sum_{j \in J} \left| C_j - C_j^* \right|.$$
This metric shows how much the modified solution differs from the original solution. It hints about how challenging it would be to realize the proposed modifications. For instance, in manufacturing setups that employ human workers, working shifts may be planned weeks in advance. Therefore, significant changes to the work plan for the upcoming days may not be feasible.

- Instance modification cost
$$\text{cost} \stackrel{\text{def}}{=} \mathcal{C}_{\mathcal{A}} \left(\sum_{(k,s,e,c) \in \mathcal{A}^{\mathcal{I}^*}} (e-c)c \right) + \mathcal{C}_{\mathcal{M}} \left(\sum_{(k_\text{f},k_\text{t},s,e,c) \in \mathcal{M}^{\mathcal{I}^*}} (e-c)c \right),$$
where $\mathcal{C}_{\mathcal{A}}$ is a given capacity addition cost and $\mathcal{C}_{\mathcal{M}}$ is a given capacity migration cost.

- We also measure the algorithm computation time, denoted as time, which is the total run-time of the evaluated algorithm in seconds. This computation time will mostly consist of finding solutions to modified problems — the computation time of the CP Optimizer.

4.2 Comparative results

In this section, we present the experiment results. All experiments were conducted on a personal desktop computer equipped with an Intel® Core™ i5-8400 CPU and 16 GB of RAM.

The total evaluation time was 43 hours, 5 minutes, and 40 seconds. This time is the sum of the individual durations of each algorithm evaluation. As described in Section 4.1.4, the measured time does not include the experiment running overhead nor aggregate data manipulation, but such overheads are negligible in comparison to the experiment evaluation times.

Plots in this section, namely Figures 4.2 to 4.4, present the evaluation results split into 4 sets. We choose the arguably most important parameters for each of the algorithms, the bottleneck identification indicator for the IIRA and the improvement interval sort key for the SSIRA, for splitting the sets of evaluations to present how performances of the algorithms differ based on the choice of those parameters. For the IIRA, we split the evaluations by the bottleneck resource identification indicator parameter I, yielding two sets of evaluations for each of the two employed adapted identification indicators MRUR and AUAU. For the SSIRA, we split the evaluations by the sort key parameter $\mathcal{K}$, yielding two sets of evaluations for each of the two choices of the sort key parameter $\mathcal{K}_t$ and $\mathcal{K}_{\Delta S}$.

We could further split the larger parameter sets for the IIRA by the improvement potential convolution kernel parameter as this parameter also has a considerable influence on the algorithm functioning. However, such split would create 6 evaluation sets for the IIRA. Plotting results for this many different evaluation sets would negate the advantages of splitting the evaluations.

4.2.1 Observations

Table 4.2 provides a summary of achieved improvements on the individual instance groups. The IIRA found improvements for 72.5% of the instances, while the SSIRA found improvements for 87.5% of the instances. The IIRA found a better improvement than the SSIRA on 10 instances. Conversely, the SSIRA found a better improvement than the IIRA on 13 instances, and on 6 of these, the SSIRA was the only algorithm to find an improvement. There were no instances where the IIRA found an improvement and the SSIRA did not. Where it found an improvement, the IIRA found a solution with the overall best improvement for 75.8% of the instances, while the SSIRA did so for 71.4% of the instances.

Table 4.3 contains results concerning the achieved tardiness improvements. Instances for which no algorithm found any improvements were omitted. For each instance, the table lists the average cost per unit tardiness improvement (per time period) and the average tardiness improvement achieved per unit of evaluation time (per second). Out of the 29 instances where both algorithms found improving solutions, the IIRA had a lower average cost per unit tardiness improvement on 15 of them. The IIRA algorithm achieved a better average on all the instances within the instance group `instance08*`, except for `instance08_4`, where the only improvement was found by the SSIRA. No clear trend is observed in the average cost per unit tardiness improvement apart from this group.

Regarding the average improvement achieved per unit of evaluation time, the

	IIRA			SSIRA		
Instances	Improved	Unique	Best	Improved	Unique	Best
instance01*	0/5			3/5	3/3	3/3
instance02*	4/5		1/4	4/5		3/4
instance03*	5/5		5/5	5/5		5/5
instance04*	5/5		4/5	5/5		4/5
instance05*	5/5		4/5	5/5		2/5
instance06*	1/5		0/1	3/5	2/3	3/3
instance07*	5/5		5/5	5/5		1/5
instance08*	4/5		3/4	5/5	1/5	4/5
Total	29/40	0/29	22/29	35/40	6/29	25/35

Table 4.2 Number of improved solutions found in the instance groups. For each algorithm, the first column contains the number of instances within the specified instance group for which an improved solution was found. The second column contains the number of instances for which an improved solution was found and for which the other algorithm did not find an improvement. The third column contains the number of instances for which the solution with best improvement was found by this algorithm.

IIRA algorithm performed slightly better than the SSIRA algorithm. Out of the 29 instances improved by both, the IIRA achieved a greater improvement per unit time on 18 instances. This pattern is particularly evident on the instance groups `instance03*`, `instance05*`, and `instance07*`.

Table 4.4 contains results concerning induced schedule difference. Instances for which no algorithm found any improvements were omitted. For each instance, the table lists the average schedule difference induced in individual evaluations and the average difference across jobs within that instance. On smaller instances, such as in instance groups `instance02*`, `instance03*`, and `instance04*`, the IIRA proposes better solutions that differ from the original schedule less than those proposed by the SSIRA. Conversely, on larger instances, such as in instance groups `instance07*` or `instance08*`, the IIRA proposes solutions with a larger schedule impact than the SSIRA. We observe, that in terms of schedule difference the IIRA proposes better solutions on smaller instances while the SSIRA is more effective on larger instances. Exceptions exist, for example on the instances `instance03_4` or `instance05_2`, where the SSIRA proposes significantly less disruptive solutions. In the evaluations of the SSIRA algorithm on the instance group `instance07*`, we observe that while the IIRA has comparable results across all five instances, the SSIRA performs significantly better on instances `instance07_2` and `instance07_3`. A similar anomaly can be observed on the instance `instance05_2` within the instance group `instance05*`. Conversely, within the instance group `instance04*`, all evaluations of the SSIRA are consistent across the individual instances, but the performance of the IIRA on instance `instance04_2` is notably worse.

In Figure 4.2 we present the results concerning capacity changes cost related to tardiness improvement, aggregated over the individual instance groups. Note that the capacity changes costs (x-axis) are plotted on a logarithmic scale. We observe a consistent trend for the SSIRA with the $\mathcal{K}_t$ sort key: better improvements can

	IIRA		SSIRA	
Instance	cost $/\Delta T_o$	$\Delta T_o/$ time	cost $/\Delta T_o$	$\Delta T_o/$ time
instance01			33.29	15.29
instance01_1			9.19	111.23
instance01_4			12.68	88.23
instance02	25.36	5.51	35.27	10.29
instance02_1	2.50	27.52	6.74	34.52
instance02_3	2.55	43.98	10.40	28.78
instance02_4	87.75	4.10	78.49	7.78
instance03	5.34	326.54	4.85	264.55
instance03_1	7.47	246.88	5.03	177.59
instance03_2	5.01	131.22	5.20	80.98
instance03_3	12.00	9.64	11.29	18.15
instance03_4	5.04	510.35	3.26	303.34
instance04	6.64	307.15	6.46	118.71
instance04_1	15.48	4.44	5.89	5.19
instance04_2	5.27	5.34	7.59	3.65
instance04_3	21.10	2.03	15.78	2.10
instance04_4	6.79	244.79	2.64	315.92
instance05	0.69	333.11	0.87	140.57
instance05_1	0.39	120.89	0.80	97.22
instance05_2	1.17	109.79	0.65	91.95
instance05_3	0.66	119.20	0.56	163.56
instance05_4	1.47	75.03	1.78	65.27
instance06			11.49	2.45
instance06_1	0.53	3.48	14.65	3.20
instance06_2			27.25	1.78
instance07	6.60	4.97	9.44	3.67
instance07_1	6.01	1.62	4.74	0.47
instance07_2	2.77	1.77	2.54	1.07
instance07_3	14.19	0.89	8.20	0.52
instance07_4	7.28	0.93	11.32	0.53
instance08	16.01	0.85	20.45	0.85
instance08_1	11.44	1.19	21.61	0.73
instance08_2	10.43	0.87	15.69	1.11
instance08_3	27.83	0.39	76.92	0.40
instance08_4			14.94	1.19

Table 4.3 Results concerning the achieved improvement. For each algorithm, the first column represents the average cost per unit of improvement, the second column represents the average improvement per unit of computation time. Instances for which no algorithm found any improvements were omitted.

Instance	IIRA $\overline{\Delta S}$	IIRA $\overline{\Delta S}/n$	IIRA $\overline{\Delta S/\Delta T_o}$	SSIRA $\overline{\Delta S}$	SSIRA $\overline{\Delta S}/n$	SSIRA $\overline{\Delta S/\Delta T_o}$
instance01				295.50	9.23	1.11
instance01_1				357.94	11.19	1.11
instance01_4				281.93	8.81	1.57
instance02	184.96	5.78	0.58	187.88	5.87	0.95
instance02_1	316.48	9.89	1.01	412.35	12.89	1.61
instance02_3	188.19	5.88	1.65	268.50	8.39	3.28
instance02_4	326.41	10.20	0.57	442.77	13.84	1.28
instance03	423.85	6.84	1.06	423.29	6.83	1.66
instance03_1	326.67	5.27	1.29	446.78	7.21	4.64
instance03_2	541.73	8.74	1.31	677.58	10.93	1.18
instance03_3	314.54	5.07	1.35	602.22	9.71	6.41
instance03_4	514.60	8.30	1.21	233.61	3.77	1.20
instance04	678.00	10.94	1.99	765.14	12.34	2.51
instance04_1	783.50	12.64	1.22	730.15	11.78	0.25
instance04_2	1,220.55	19.69	1.03	974.38	15.72	1.11
instance04_3	705.52	11.38	0.66	645.53	10.41	1.72
instance04_4	651.43	10.51	1.15	432.61	6.98	1.08
instance05	575.40	9.28	0.94	312.78	5.04	1.01
instance05_1	699.94	11.29	1.01	318.81	5.14	0.99
instance05_2	446.49	7.20	1.07	64.19	1.04	1.11
instance05_3	585.67	9.45	1.03	197.36	3.18	0.97
instance05_4	653.69	10.54	0.94	288.33	4.65	1.06
instance06				1,754.09	28.29	1.22
instance06_1	2,791.00	45.02	1.58	1,512.48	24.39	1.62
instance06_2				1,892.41	30.52	1.71
instance07	2,243.23	18.39	1.31	1,926.06	15.79	0.45
instance07_1	2,956.84	24.24	1.81	2,996.11	24.56	1.00
instance07_2	2,343.17	19.21	1.02	701.47	5.75	1.00
instance07_3	2,243.60	18.39	1.19	869.78	7.13	2.55
instance07_4	2,965.11	24.30	0.96	2,550.06	20.90	1.00
instance08	1,538.20	12.61	1.44	1,437.97	11.79	1.77
instance08_1	1,601.69	13.13	1.09	1,165.19	9.55	1.29
instance08_2	1,614.05	13.23	0.75	1,715.33	14.06	1.65
instance08_3	1,319.86	10.82	1.40	1,174.39	9.63	1.31
instance08_4				1,262.40	10.35	0.45

Table 4.4 Results concerning the induced schedule difference. For each algorithm, the first column represents the average schedule difference found for the instance, the second column represents the average difference in start times per job in the instance, the third column represents the average schedule difference per unit of tardiness improvement. Instances for which no algorithm found any improvements were omitted.

be achieved with increasing costs. Smaller improvements require smaller costs, while seeking further improvements requires an increase in costs. A similar pattern is observed for the SSIRA with the $\mathcal{K}_{\Delta S}$ sort key; however, on certain smaller instance groups such as `instance01*` or `instance04*`, the improvement achieved for the same cost can vary significantly. An interesting observation is that on smaller instances, such as in instance groups `instance02*`, `instance03*`, and `instance04*`, the IIRA finds solutions with the same improvement to cost ratio using both bottleneck identification indicators. This is not the case on larger instances, where the evaluations employing the AUAU and MRUR indicators differ substantially. An important observation is that, particularly on larger instances, the IIRA is capable of finding significant improvements with relatively lows costs. This is apparent on instance groups `instance07*` and `instance08*`, where the best improvements for given costs are almost exclusively found by the IIRA, whereas the SSIRA only finds minor improvements.

In Figure 4.3 we present the results concerning tardiness improvement related to induced schedule difference, aggregated over the individual instance groups. We observe a consistent trend for the SSIRA with the $\mathcal{K}_t$ sort key: greater tardiness improvement corresponds with an increased schedule difference. This trend is also observable for other evaluations. Evaluations of the SSIRA with the $\mathcal{K}_{\Delta S}$ sort key are the most inconsistent, particularly on smaller instances. In general, the SSIRA utilizing the $\mathcal{K}_{\Delta S}$ sort key tends to propose the least favorable solutions in terms of the induced schedule difference. On larger instances, most evaluations follow the linear increasing trend. Notably, in instance groups `instance01*`, `instance04*`, or `instance06*`, the SSIRA utilizing the $\mathcal{K}_t$ finds solutions with improvement comparable to those proposed by the SSIRA utilizing the $\mathcal{K}_{\Delta S}$ sort key and the IIRA, but with a lower schedule difference. On larger instances, the SSIRA utilizing the $\mathcal{K}_t$ appears to be unable to find solutions with the overall best improvements. For the solutions with the highest improvement, the IIRA proposes marginally better solutions than the SSIRA, if any are found by the SSIRA.

In Figure 4.4 we present the results concerning evaluation computation time related to tardiness improvement, aggregated over the individual instance groups. These results are notably inconsistent. We observe that evaluation times of the SSIRA are more varied than those of the IIRA. This is particularly apparent on the instance groups `instance04*`, `instance07*`, and `instance08*`. Additionally, we observe that in some cases, most notably on instances from the instance groups `instance06*` and `instance08*`, the evaluations form linear clusters with identical evaluation times.

4.3 Discussion

The SSIRA found improvements for all instances for which the IIRA found improvements. Furthermore, the SSIRA found improvements for some instances where the IIRA did not. We can conclude that the SSIRA has a higher probability of finding an improvement than the IIRA. We believe this is because, when seeking improvements, the IIRA does not consider the target order. Instead, it identifies bottleneck resources with respect to the whole problem instance and its schedule. Our hypothesis is that this approach limits the IIRA in cases where the target

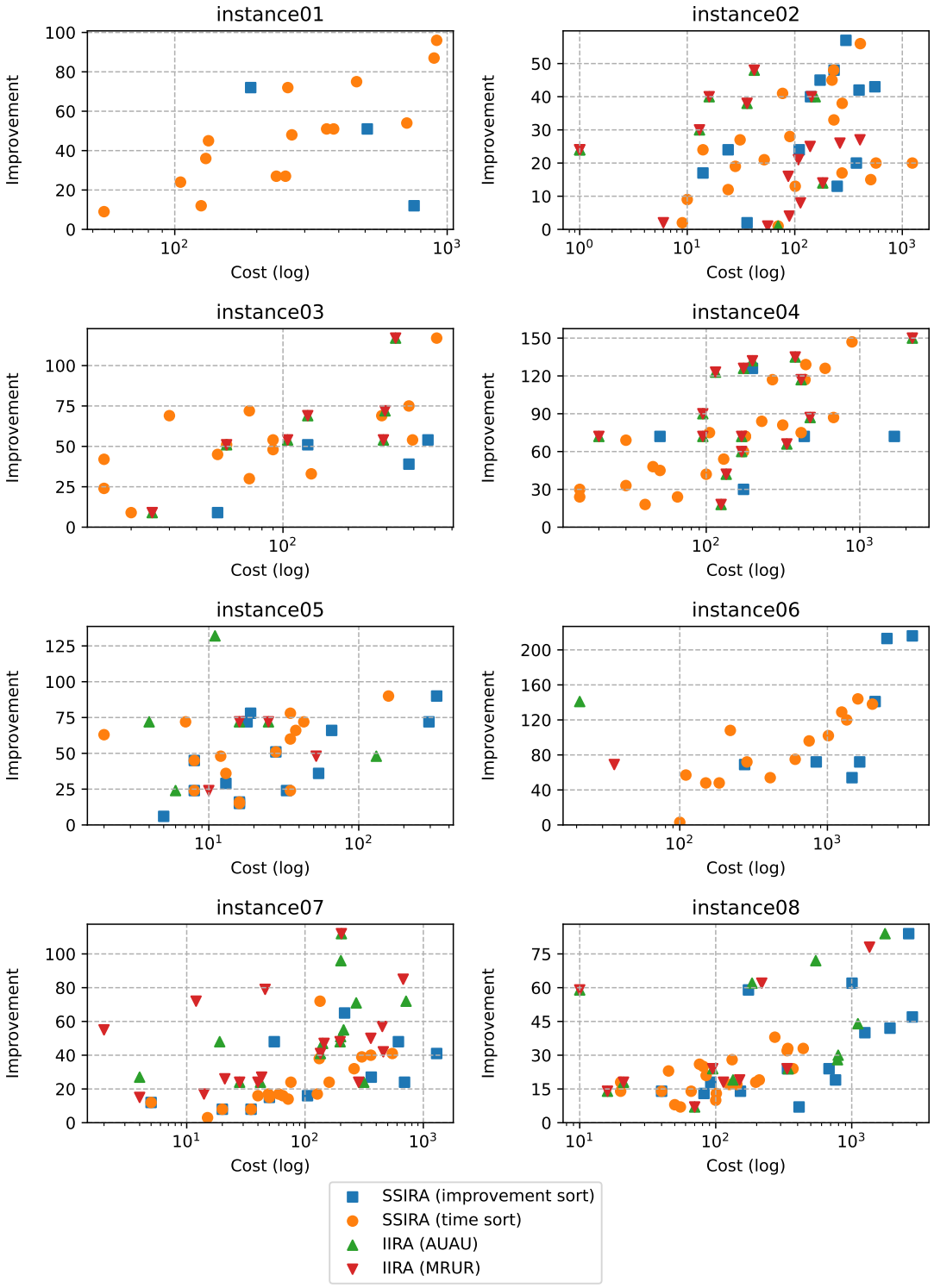


Figure 4.2 Aggregated plots of capacity changes costs (x-axis) to achieved improvement (y-axis). In each plot, every algorithm evaluation comprises of five Pareto fronts of evaluations on the five individual instances from the represented instance group. Full non-aggregated results for each individual instance can be found in Figure A.1.

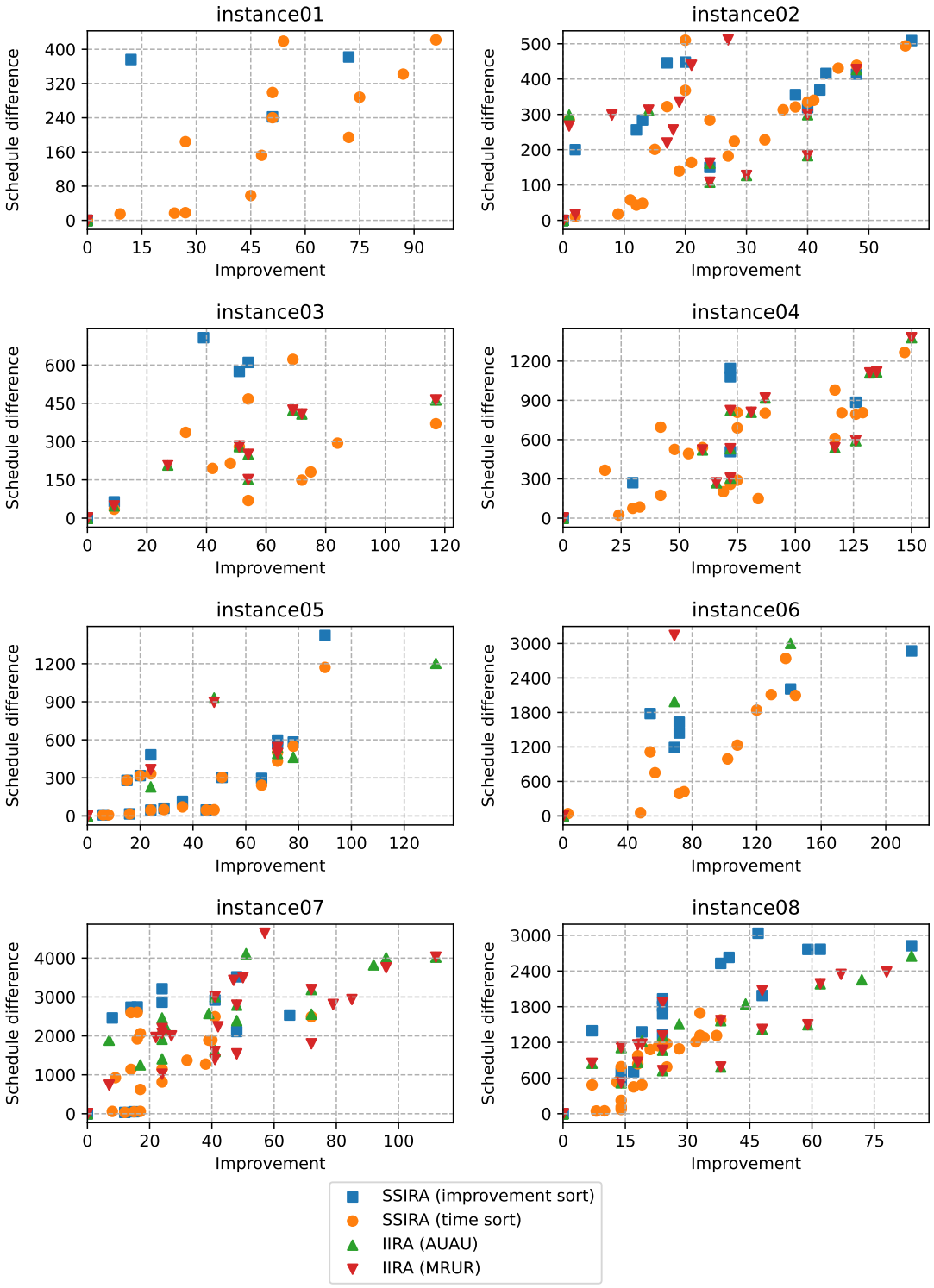


Figure 4.3 Aggregated plots of achieved improvement (x-axis) to induced schedule difference (y-axis). In each plot, every algorithm evaluation comprises of five Pareto fronts of evaluations on the five individual instances from the represented instance group. Full non-aggregated results for each individual instance can be found in Figure A.2.

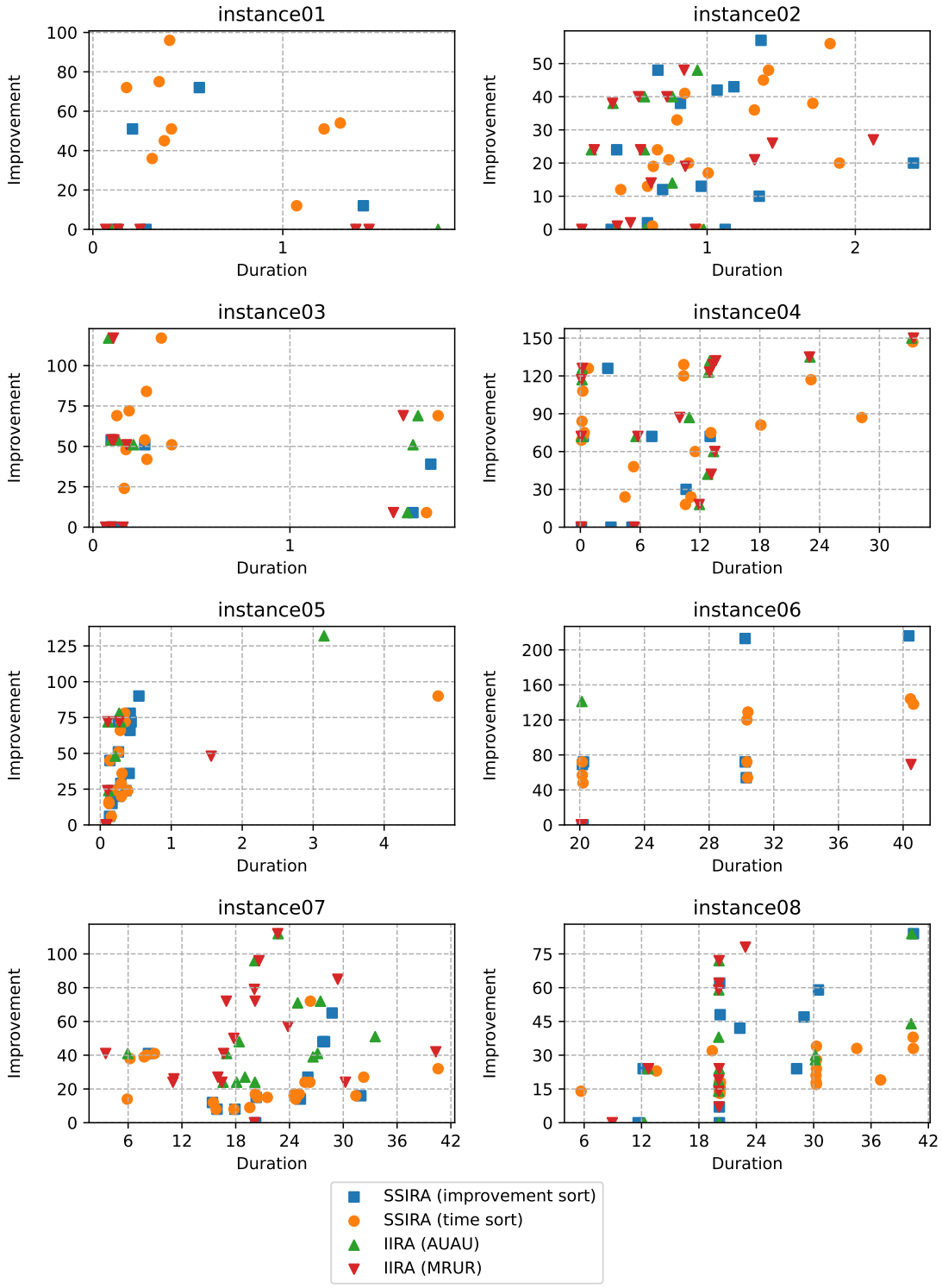


Figure 4.4 Aggregated plots of computation time (x-axis) to achieved improvement (y-axis). In each plot, every algorithm evaluation comprises of five Pareto fronts of evaluations on the five individual instances from the represented instance group. Full non-aggregated results for each individual instance can be found in Figure A.3.

order is not directly affected by execution bottlenecks but is instead only partially constrained by the impact of the execution bottlenecks on the schedule. In such cases, relaxations focused on the bottlenecks of the whole instance do not directly improve the schedule with respect to the target order. This is where the SSIRA benefits from its focused relaxations, as all of its relaxations focus specifically on the target order.

As stated, the IIRA does not consider the target order when finding improvements. Nonetheless, it is able to find improvements at relatively low costs compared to the solutions achieving the same improvements found by the SSIRA. Furthermore, on larger instances, the IIRA finds solutions with greater improvements and overall better quality than the SSIRA. This is an unexpected result, as the initial assumption was that relaxations focusing specifically on the target order would achieve better improvements than general relaxations. We speculate that targeted relaxations of the SSIRA might be too specific, not providing sufficient slack in the modified constraints and thus making the model too sensitive to minor variations when finding modified solutions. Another possibility is that the SSIRA often executes multiple relaxations simultaneously, assuming their independence. This assumption is inherent to the algorithm through the process it identifies potential relaxations. When the presumed independent relaxations are implemented and the solver seeks a modified solution, the combined relaxations might direct the solver towards a solution that differs from what the algorithm intended.

Regarding the schedule difference induced by the proposed modifications, the IIRA is able to find better solutions that the SSIRA on smaller instances. On larger instances, the SSIRA typically surpasses the IIRA in performance. This is likely because the IIRA proposes general relaxations whereas the SSIRA proposes relaxations focusing on the target order. On smaller instances, general relaxations usually affect most of the schedule, but on larger instances, the impact of these relaxations becomes more local. On larger instances, the systematic approach of the SSIRA utilizing the focused relaxations is able to find specific relaxations that are necessary to improve the target order. Moreover, even on smaller instances, the SSIRA is sometimes able to find solutions with smaller impact on the schedule difference than the solutions of the IIRA. However, the general relaxing approach of the IIRA is still viable on most instances.

We observed that on several instances, distinct linear clusters of evaluations with identical evaluation times are formed. In those clusters, the evaluations achieve varying improvements in the same amount of computation time. Our theory is that the formation of those clusters is due to the set solver time limit, which interrupts an evaluation if a solution has not been found in the specified time limit. We can conclude that on such instances, extending the solver time limit might result in better achieved improvements. The introduction of the solver time limit represents a trade-off between solution quality and computation time, particularly for more difficult instances. Nonetheless, since the evaluations on most instances were not affected by the set time limit, we conclude that the time limit was set appropriately in accordance with the difficulty of our problem.

We summarize the results in the following key points:

- The SSIRA tends to find improvements more consistently than the IIRA, possibly because the SSIRA proposes relaxations focused specifically on the

target order, while the IIRA proposes only general relaxations.

- Despite not directly considering the target order, the IIRA tends to find improvements at lower costs and achieves solutions with greater improvements on larger instances compared to the SSIRA. We theorize that this may be due to the SSIRA's specific relaxations being overly restrictive and that its execution of multiple relaxations simultaneously sometimes leads to unintended solutions.
- Regarding induced schedule difference, the IIRA proposes better solutions on smaller instances, while the SSIRA outperforms the IIRA on larger instances. The SSIRA sometimes achieves significantly better solutions than the IIRA even on smaller instances, suggesting its viability across various problem sizes. However, the IIRA proposes adequate solutions with acceptable solution differences more consistently.

Conclusion

In this thesis, we addressed the problem of reducing the tardiness of a selected manufacturing order. First, we formulated an extension of the standard RCPSP to model the problem. We then focused on manufacturing bottlenecks, specifically execution-level bottlenecks in obtained schedules. Following the identification of such bottlenecks, we proposed relaxations for related resource capacity constraints.

Contribution

The subject of identifying execution bottlenecks and subsequently relaxing related constraints has not been studied on the variant of the RCPSP used in this thesis to model the problem at hand. We proposed two methods to address this problem; the IIRA, utilizing adaptations of methods that address this problem in the Job-Shop scheduling problem, and the SSIRA, designed specifically for this problem. We conducted numerical experiments to analyze the capabilities of the proposed methods. For that purpose, we designed a wide variety of problem instances modeling the extension of the RCPSP used in this thesis. The performances of the two methods were studied on the presented problem instances. We observed that the SSIRA is more consistent in finding improving solutions than the IIRA. However, on many instances, the IIRA is able to find great improvements with low modification costs where the SSIRA finds the same improvements with greater costs.

Further work

The SSIRA algorithm selects improvement intervals and proposes relaxations of related resource capacity constraints. However, it currently assumes independence among relaxations, which does not correspond to the actual complex dependencies in the solved models. The relaxations could alternatively be modeled as an optimization problem, which would better capture the dependencies.

Additionally, the SSIRA finds improvement intervals only for jobs included in the left-shift closure of the target manufacturing order. This approach has proven effective at limiting the number of jobs considered for improvement. However, different heuristics could be used to focus the search on the targeted manufacturing order. Alternatively, useful information could be extracted from the constraint programming solvers used to solve the problem instance models, as the solvers usually provide details about the solving process, such as statistics about variable and constraint conflicts.

In the conducted experiments, we measure the schedule difference between the original schedule and the proposed modified schedule as the total sum off job-start differences between the schedules. This simple metric was sufficient to compare the two presented algorithms, however, the proposed changes could be studied further to evaluate their impact on the system. For example, by following the work of Lawrence and Buss (1994), the bottleneck "shiftiness" could be studied to evaluate how the proposed changes affect the bottleneck identification.

Bibliography

Adams, Joseph; Balas, Egon; Zawack, Daniel, 1988. The Shifting Bottleneck Procedure for job shop scheduling. *Manage. Sci.* Vol. 34, no. 3, pp. 391–401.

Arkhipov, Dmitry I.; Battaïa, Olga; Lazarev, Alexander A., 2017. Long-term production planning problem: scheduling, makespan estimation and bottleneck analysis. *IFAC-PapersOnLine*. Vol. 50, no. 1, pp. 7970–7974. ISSN 2405-8963. Available from DOI: https://doi.org/10.1016/j.ifacol.2017.08.991. 20th IFAC World Congress.

Betterton, C.E.; Silver, S.J., 2012. Detecting bottlenecks in serial production lines – a focus on interdeparture time variance. *International Journal of Production Research*. Vol. 50, no. 15, pp. 4158–4174. Available from DOI: 10.1080/00207543.2011.596847.

Bianco, Lucio; Caramia, Massimiliano, 2011. A new formulation for the project scheduling problem under limited resources. *Flexible Services and Manufacturing Journal*. Vol. 25, no. 1–2, pp. 6–24. ISSN 1936-6590. Available from DOI: 10.1007/s10696-011-9127-y.

Biller, Stephan; Li, Jingshan; Marin, Samuel P.; Meerkov, Semyon M.; Zhang, Liang, 2010. Bottlenecks in Bernoulli Serial Lines With Rework. *IEEE Transactions on Automation Science and Engineering*. Vol. 7, no. 2, pp. 208–217. Available from DOI: 10.1109/TASE.2009.2023463.

Blazewicz, J.; Lenstra, J.K.; Kan, A.H.G.Rinnooy, 1983. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics*. Vol. 5, no. 1, pp. 11–24. ISSN 0166-218X. Available from DOI: 10.1016/0166-218x(83)90012-4.

Brucker, Peter; Drexl, Andreas; Möhring, Rolf; Neumann, Klaus; Pesch, Erwin, 1999. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*. Vol. 112, no. 1, pp. 3–41. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/S0377-2217(98)00204-5.

Cheng, Junzilan; Fowler, John; Kempf, Karl; Mason, Scott, 2015. Multi-mode resource-constrained project scheduling problems with non-preemptive activity splitting. *Computers & Operations Research*. Vol. 53, pp. 275–287. ISSN 0305-0548. Available from DOI: 10.1016/j.cor.2014.04.018.

Chiang, S.-Y.; Kuo, C.-T.; Meerkov, S.M., 2001. c-bottlenecks in serial production lines: identification and application. In: *Proceedings of the 38th IEEE Conference on Decision and Control (Cat. No.99CH36304)*. IEEE. CDC-99. Available from DOI: 10.1109/cdc.1999.832820.

Cooper, Dale F., 1976. Heuristics for Scheduling Resource-Constrained Projects: An Experimental Investigation. *Management Science*. Vol. 22, no. 11, pp. 1186–1194. Available from DOI: 10.1287/mnsc.22.11.1186.

Demeulemeester, Erik; Herroelen, Willy, 1992. A Branch-and-Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem. *Management Science*. Vol. 38, no. 12, pp. 1803–1818. ISSN 1526-5501. Available from DOI: 10.1287/mnsc.38.12.1803.

Elsayed, Saber; Sarker, Ruhul; Ray, Tapabrata; Coello, Carlos Coello, 2017. Consolidated optimization algorithm for resource-constrained project scheduling problems. *Information Sciences*. Vol. 418–419, pp. 346–362. ISSN 0020-0255. Available from DOI: 10.1016/j.ins.2017.08.023.

Franck, Birger; Neumann, Klaus; Schwindt, Christoph, 2001. Project scheduling with calendars. *OR-Spektrum*. Vol. 23, no. 3, pp. 325–334. ISSN 1436-6304. Available from DOI: 10.1007/pl00013355.

Ganian, Robert; Hamm, Thekla; Mescoff, Guillaume, 2021. The complexity landscape of resource-constrained scheduling. In: *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*. Yokohama, Yokohama, Japan. IJCAI'20. ISBN 9780999241165.

Hartmann, Sönke; Briskorn, Dirk, 2010. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*. Vol. 207, no. 1, pp. 1–14. ISSN 0377-2217. Available from DOI: 10.1016/j.ejor.2009.11.005.

Hartmann, Sönke; Briskorn, Dirk, 2022. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*. Vol. 297, no. 1, pp. 1–14. ISSN 0377-2217. Available from DOI: 10.1016/j.ejor.2021.05.004.

IBM, 2024. *Constraint program solvers* [online]. [visited on 2024-02-10]. Available from: https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer.

Klein, Robert, 2000. Project scheduling with time-varying resource constraints. *International Journal of Production Research*. Vol. 38, no. 16, pp. 3937–3952. ISSN 1366-588X. Available from DOI: 10.1080/00207540050176094.

Kolisch, Rainer; Hartmann, Sönke, 1999. Heuristic Algorithms for the Resource-Constrained Project Scheduling Problem: Classification and Computational Analysis. In: *Project Scheduling*. Springer US, pp. 147–178. ISBN 9781461555339. ISSN 0884-8289. Available from DOI: 10.1007/978-1-4615-5533-9_7.

Kolisch, Rainer; Sprecher, Arno, 1997. PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program. *European Journal of Operational Research*. Vol. 96, no. 1, pp. 205–216. ISSN 0377-2217. Available from DOI: https://doi.org/10.1016/S0377-2217(96)00170-1.

Kuo, C.-T.; Lim, J.-T.; Meerkov, S. M., 1996. Bottlenecks in serial production lines: A system-theoretic approach. *Mathematical Problems in Engineering*. Vol. 2, no. 3, pp. 233–276. ISSN 1563-5147. Available from DOI: 10.1155/s1024123x96000348.

Lawrence, Stephen R.; Buss, Arnold H., 1994. Shifting production bottlenecks: causes, cures, and conundrums. *Production and Operations Management*. Vol. 3, no. 1, pp. 21–37. Available from DOI: 10.1111/j.1937-5956.1994.tb00107.x.

Li, Lin, 2009. Bottleneck detection of complex manufacturing systems using a data-driven method. *International Journal of Production Research*. Vol. 47, no. 24, pp. 6929–6940. ISSN 1366-588X. Available from DOI: 10.1080/00207540802427894.

Luo, Jingyu; Vanhoucke, Mario; Coelho, José, 2023. Automated design of priority rules for resource-constrained project scheduling problem using surrogate-assisted genetic programming. *Swarm and Evolutionary Computation*. Vol. 81, p. 101339. ISSN 2210-6502. Available from DOI: https://doi.org/10.1016/j.swevo.2023.101339.

Mönch, Lars; Zimmermann, Jens, 2010. A computational study of a shifting bottleneck heuristic for multi-product complex job shops. *Production Planning & Control*. Vol. 22, no. 1, pp. 25–40. ISSN 1366-5871. Available from DOI: 10.1080/09537287.2010.490015.

Nonobe, Koji; Ibaraki, Toshihide, 2002. Formulation and Tabu Search Algorithm for the Resource Constrained Project Scheduling Problem. In: *Essays and Surveys in Metaheuristics*. Springer US, pp. 557–588. ISBN 9781461515074. ISSN 1387-666X. Available from DOI: 10.1007/978-1-4615-1507-4_25.

Patterson, James H., 1976. Project scheduling: The effects of problem structure on heuristic performance. *Naval Research Logistics Quarterly*. Vol. 23, no. 1, pp. 95–123. Available from DOI: https://doi.org/10.1002/nav.3800230110.

Pellerin, Robert; Perrier, Nathalie; Berthaut, François, 2020. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*. Vol. 280, no. 2, pp. 395–416. ISSN 0377-2217. Available from DOI: 10.1016/j.ejor.2019.01.063.

Roh, P.; Kunz, A.; Netland, T., 2018. Data-driven detection of moving bottlenecks in multi-variant production lines. *IFAC-PapersOnLine*. Vol. 51, no. 11, pp. 158–163. ISSN 2405-8963. Available from DOI: 10.1016/j.ifacol.2018.08.251.

Roser, C.; Nakano, M.; Tanaka, M., 2001. A practical bottleneck detection method. In: *Proceeding of the 2001 Winter Simulation Conference (Cat. No.01CH37304)*. IEEE. WSC-01. Available from DOI: 10.1109/wsc.2001.977398.

Schnell, Alexander; Hartl, Richard F., 2015. On the efficient modeling and solution of the multi-mode resource-constrained project scheduling problem with generalized precedence relations. *OR Spectrum*. Vol. 38, no. 2, pp. 283–303. ISSN 1436-6304. Available from DOI: 10.1007/s00291-015-0419-6.

Subramaniyan, M.; Skoogh, A.; Gopalakrishnan, M.; Hanna, A., 2016. Real-time data-driven average active period method for bottleneck detection. *International Journal of Design & Nature and Ecodynamics*. Vol. 11, no. 3, pp. 428–437. ISSN 1755-7445. Available from DOI: 10.2495/dne-v11-n3-428-437.

Subramaniyan, Mukund; Skoogh, Anders; Bokrantz, Jon; Sheikh, Muhammad Azam; Thürer, Matthias; Chang, Qing, 2021. Artificial intelligence for throughput bottleneck analysis – State-of-the-art and future directions. *Journal of Manufacturing Systems*. Vol. 60, pp. 734–751. ISSN 0278-6125. Available from DOI: 10.1016/j.jmsy.2021.07.021.

Vanhoucke, Mario; Demeulemeester, Erik; Herroelen, Willy, 2001. An Exact Procedure for the Resource-Constrained Weighted Earliness–Tardiness Project Scheduling Problem. *Annals of Operations Research*. Vol. 102, no. 1/4, pp. 179–196. ISSN 0254-5330. Available from DOI: 10.1023/a:1010958200070.

Wang, Jun-Qiang; Chen, Jian; Zhang, Yingqian; Huang, George Q., 2016. Schedule-based execution bottleneck identification in a job shop. *Computers & Industrial Engineering*. Vol. 98, pp. 308–322. ISSN 0360-8352. Available from DOI: https://doi.org/10.1016/j.cie.2016.05.039.

Zhang, Rui; Wu, Cheng, 2008a. Bottleneck identification procedures for the job shop scheduling problem with applications to genetic algorithms. *The International Journal of Advanced Manufacturing Technology*. Vol. 42, no. 11–12, pp. 1153–1164. ISSN 1433-3015. Available from DOI: 10.1007/s00170-008-1664-5.

Zhang, Rui; Wu, Cheng, 2008b. Bottleneck machine identification based on optimization for the job shop scheduling problem. *ICIC Express Letters*. Vol. 2. Available also from: https://www.researchgate.net/publication/228918308_Bottleneck_machine_identification_based_on_optimization_for_the_job_shop_scheduling_problem.

Zhang, Rui; Wu, Cheng, 2012. Bottleneck machine identification method based on constraint transformation for job shop scheduling with genetic algorithm. *Information Sciences*. Vol. 188, pp. 236–252. ISSN 0020-0255. Available from DOI: 10.1016/j.ins.2011.11.013.

Notation

Symbol	Definition
Problem instances	
$\mathcal{I} = (\mathcal{J}, \mathcal{P}, \mathcal{R}, \mathcal{T})$	Problem instance
$\mathcal{J} = \{1, \ldots, n\}$	Set of jobs
p_j	Processing time of job j
d_j	Deadline for a job j
w_j	Tardiness weight for a job j
$i \rightarrow j$ or (i, j)	Precedence between jobs i and j
$\mathcal{P}$	Set of precedences
$G = (\mathcal{J}, \mathcal{P})$	Precedence graph
$\mathcal{R} = \{1, \ldots, m\}$	Set of resources
R_k	Capacity function of resource k
$R_k^{(t)}$	Capacity of resource k in time period t
$R_k^{(-)}$	Shift capacity of a resource k
r_{jk}	Per-period consumption of a resource k by a job j
$\mathcal{T}$	Time horizon
$1, \ldots, \mathcal{T}$	Time periods
$\mathcal{J}_k$	The set of jobs executed on the resource k
$\mathcal{J}_k^{(t)}$	The set of jobs executed on the resource k during the time period t
$\mathcal{J}_k^{UAP(i)}$	The set of jobs executed on the resource k during the uninterrupted active period i
Schedules, solutions	
S_j	Start time of job j
$S = (S_1, \ldots, S_n)$	Schedule
$C_j = S_j + p_j$	Completion time of job j
$C = (C_1, \ldots, C_n)$	All completion times of jobs
Identification indicators	
MUR_k	Machine Utilization Rate
AUAD_k	Average Uninterrupted Active Duration
RS_k, RC_k	Resource Strength, Resource Constrainedness
MRUR_k	Machine Resource Utilization Rate
AUAU_k	Average Uninterrupted Active Utilization
$\mathrm{PRU}_k^{(i)}$	Period Resource Utilization

Note: supper-scripting a value with a schedule symbol — for example C_j^S — relates that value to the specified schedule.

Symbol	Definition
IIRA parameters	
I	Identification indicator
G	Granular period granularity
C	Improvement potential convolution kernel
$f_{\text{pre}}, f_{\text{around}}, f_{\text{post}}$	Various improvement potential convolution kernels
$I_{\max}$	Iterations limit
$P_{\max}$	Improvement periods limit
Δ	Capacity improvement
SSIRA parameters	
$I_{\max}$	Iterations limit
$IT_{\max}$	Improvement intervals limit
$\mathcal{K}$	Interval sort key
$\mathcal{K}_t$	Interval sort key sorting by interval time periods
$\mathcal{K}_{\Delta S}$	Interval sort key sorting by interval proposed improvement

A Attachments

A.1 Algorithms, Functions, and Procedures

Function GranularResourceLoad(k, $\mathcal{I}$, S, PC)

$L : \{1, \ldots, PC\} \to \mathbb{N}_0$ ▷ Period load function
for $j \in \mathcal{J}_k$ **:**
    $i_l \leftarrow \lfloor S_j / \mathrm{G} \rfloor$, ▷ First overlapping period
    $i_h \leftarrow \lfloor C_j / \mathrm{G} \rfloor$ ▷ Last overlapping period
    if $i_l = i_h$ **:** ▷ If the job overlaps with a single period...
        $L(i_l) \leftarrow L(i_l) + p_j r_{jk}$
    else: ▷ ...the job overlaps with multiple periods
        $L(i_l) \leftarrow L(i_l) + (\mathrm{G}(i_l + 1) - S_j) \cdot r_{jk}$
        for $i \in \{i_l + 1, \ldots, i_h - 1\}$ **:**
            $L(i) \leftarrow L(i) + \mathrm{G} \cdot r_{jk}$
        $L(i_h) \leftarrow L(i_h) + (C_j - \mathrm{G}(i_h - 1)) \cdot c$
return L

Function IncreaseGranularPeriodCapacity(i, R_k, G, Δ)

$t_l \leftarrow 1 + (i - 1)\mathrm{G}$ ▷ First time period covered
$t_h \leftarrow i\mathrm{G}$ ▷ Last time period covered
for $t \in \{t_l, \ldots, t_h\}$ **:**
    $R_k^{(t)} \leftarrow R_k^{(t)} + \Delta$
return $R_k^{(t)}$

Function ReduceCapacityChanges($\mathcal{I}$, S, R_1^{orig}, ..., R_m^{orig})

$R'_1, \ldots, R'_m : \{1, \ldots, \mathcal{T}\} \to \mathbb{N}_0$ ▷ Reduced capacity functions
for $k \in \mathcal{R}$ **:**
    $L \leftarrow$ ResourceLoad(k, $\mathcal{I}$, S)
    for $t \in \{1, \ldots, \mathcal{T}\}$ **:**
        $R_k'^{(t)} \leftarrow \max(R_k^{\mathrm{orig}(t)}, L(t))$
return $R'_1, \ldots, R'_m$

Function ResourceLoad(k, $\mathcal{I}$, S)

$L : \{1, \ldots, \mathcal{T}\} \to \mathbb{N}_0$
for $t \in \{1, \ldots, \mathcal{T}\}$ **:**
    $L(t) \leftarrow \sum_{j \in \mathcal{J}_k^{(t)}} r_{jk}$ ▷ $\mathcal{J}_k^{(t)}$ with respect to given schedule S
return L

Function FindAdditionsAndMigrations($\mathcal{I}$, S)

$\mathcal{M} \leftarrow \emptyset$
for $(\forall k \in \mathcal{R})$ **:** $L_k \leftarrow$ ResourceLoad$(k, \mathcal{I}, S)$
for $(\forall k \in \mathcal{R})$ **:** $R_k^{\oplus} \leftarrow R_k - L_k$ ▷ Capacity surpluses
REQ $\leftarrow$ set of all additional non-shift capacities
for $(k, s, e, c) \in$ REQ **:**
    while $c > 0$ **:**
        $c_1, \ldots, c_m \leftarrow$ maximal continuous surpluses overlapping $\{s, \ldots, e\}$
        $k_{\text{from}} \leftarrow \text{argmax}_k\, c_k$
        if $c_{k_{\text{from}}} = 0$ **:** ▷ If no further migrations are possible
            break ▷ Remaining c will be fulfilled by capacity additions
        $c_{\text{mig}} \leftarrow \min(c, c_{k_{\text{from}}})$
        Reduce surplus $R^{\oplus}_{k_{\text{from}}}$ by c_{mig} during time periods $\{s, \ldots, e-1\}$
        $\mathcal{M} \leftarrow \mathcal{M} \cup \{(k_{\text{from}}, k, s, e, c_{\text{mig}})\}$
$\mathcal{A} \leftarrow \{(k, s, e, c) \in$ REQ $: c > 0\}$
return $\mathcal{A}$, $\mathcal{M}$

Function ModifyResourceCapacities($\mathcal{I}$, χ_1, ..., χ_c)

$R_1^*, \ldots, R_m^* \leftarrow R_1, \ldots, R_m$ ▷ Copy the original capacity functions
for $(j, s, e) \in \{\chi_1, \ldots, \chi_c\}$ **:**
    for $k \in \mathcal{R} : r_{jk} > 0$ **:**
        for $t \in \{s, \ldots, e-1\}$ **:**
            $R_k^{*(t)} \leftarrow R_k^{*(t)} + r_{jk}$
return $R_1^*, \ldots, R_m^*$

Function ComputeSuffixRelaxedSchedule($\mathcal{I}$, S, t)

$T \leftarrow$ Topological ordering of $\mathcal{J}$ on the precedence graph G
for $j \in T$ **:**
    if $S_j \leq t$ **:**
        $\vec{S}_j^{(t)} \leftarrow S_j$
    else:
        $\vec{S}_j^{(t)} \leftarrow \max\left\{\vec{S}_i^{(t)} + p_i : i \to j \in \mathcal{P}\right\}$
return $\vec{S}^{(t)}$

Function ComputeLeftShiftClosure($\mathcal{I}$, S, j)

$\mathcal{L}(j) \leftarrow \emptyset$
$Q \leftarrow \{j\}$ ▷ Queue of jobs to process
while Q not empty **:**
    $i \leftarrow$ pop Q
    $\mathcal{L}(j) \leftarrow \mathcal{L}(j) \cup \{i\}$
    $Q \leftarrow Q \cup \{p : p \rightarrow i, C_p = S_i\}$ ▷ Precedence predecessors
    $Q \leftarrow Q \cup \bigcup_{k:r_{ik}>0}\{p \in \mathcal{J}_k : C_p = S_i\}$ ▷ Resource predecessors
    for $k \in \mathcal{R}, r_{ik} > 0, R_k^{(S_i-1)} = 0$ **:** ▷ Resource-pause predecessors
        $\mathrm{ps}_k(S_i) \leftarrow \max\{t' \in \{1, \ldots, S_i - 1\} : R_k^{(t')} > 0\}$
        $Q \leftarrow Q \cup \{p \in \mathcal{J}_k : \mathrm{ps}_k(S_i) - p_i \leq C_p \leq \mathrm{ps}_k(S_i)\}$
return $\mathcal{L}(j)$

Note: We assume, that when the value of $\mathrm{ps}_k(S_i)$ is undefined, the following set of predecessors is trivially empty and we continue with a different resource.

A.2 Documentation

This section contains the description of the project implementing the algorithms and running the experiments presented in this thesis. The full project is attached in the `RCPSPSandbox.zip` file. The repository of the project can be found online at `https://github.com/Krtiiik/RCPSPSandbox`.

A.2.1 Requirements

All implementations are written in Python, utilizing several external libraries. To run scripts, the following is required:

- Python 3.11 with installed packages listed in the `requirements.txt` file.
- Configured IBM ILOG Constraint-Programming Optimizer. IBM ILOG Constraint-Programming Optimizer is part of the IBM ILOG CPLEX Optimization Studio commercial software package. Community and academic editions are available. See online at `https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer`.

Python version at least 3.11 is required as we utilize several functionalities introduced in that version. However, note that the `docplex` library does not fully support this version. We had no issues with this partial incompatibility.

A.2.2 Running scripts

A simple example demonstrating the two algorithms presented in this thesis is contained in the `rcpsp_sandbox/example.py` script file. Input and resulting data of this example is located in the `example` directory.

All experiments are run using the `experiments.py` Python script file, located in the `rcpsp_sandbox` source directory. Following is the invocation help:

```
usage: experiments.py [-h] [--save_plots] [--addition ADDITION]
                      [--migration MIGRATION]

options:
  -h, --help             show this help message and exit
  --save_plots           Determines whether to save plots to files
  --addition ADDITION    Cost of capacity addition (default is 5)
  --migration MIGRATION  Cost of capacity migration (default is 1)
```

Running this script computes the experiments. The script evaluates the algorithms (or loads computed evaluations from data files), computes statistics, which are then saved to data files, and finally creates plots of the results.

We provide the computed results in the `data` directory. In this directory, base instances, modified instances, and computed evaluations and evaluation KPIs are stored in directories named accordingly. Before running the experiments, the `modified_instances` directory has to be extracted from a zip file named `modified_instances.zip`.

A.2.3 Project overview

The project is divided into the following sub-packages, each containing several modules:

- `rcpsp_sandbox.instances` — modules for manipulating problem instances.
- `rcpsp_sandbox.solver` — modules for solving the problem instances.
- `rcpsp_sandbox.bottlenecks` — modules implementing the presented algorithms and hosting experiment evaluations.

The `rcpsp_sandbox.instances` sub-package contains several modules for manipulating with problem instances. Those modules are used in the rest of the project, forming a core data infrastructure. In the `problem_instance` module contains the definition of the `ProblemInstance` class, representing the problem instance defined in Definition 1. The `io` module is used for parsing and serializing problem instance object, be it in the original PSPLIB file format, or in JSON. The `problem_modified` module provides the `modify_instance` function, which for a given problem instance returns a `ProblemModifier` object. The interface of the object allows the user to modify all aspects of the problem instance. Most important, it implements the modifications described in Section 1.1, namely, splitting the precedence graph, introducing time-variable resource capacities, assigning job due dates. The `algorithms` module implements several algorithms regarding problem instances, mostly various precedence graph traversals.

The `rcpsp_sandbox.solver` sub-package contains, among others, the `solver` module. This module facilitates the solving of problem instances via the `Solver` class. The `Solver` class contains a single `solve` method, which takes in either a problem instance, of a built model to solve. The function utilizes the `docplex` library to call the IBM ILOG Constraint-Programming Optimizer solver, which finds (optimal) solutions to given models. If a problem instance is given to the `Solver.solve` method, the solver builds a standard model described in Section 1.2.

For a finer control over the model, the `model_builder` module provides the `build_model` function. This function, for a given problem instance, return an initialized `ModelBuilder` object. The interface of this object allows for the creation of specific models, introducing only selected constraints, restraining job intervals, or choosing alternate optimization goals. The `solution` module contains the definition of the `Solution` abstract class, along with several implementing derived classes and utility functions concerning solutions to the problem instance models.

The `rcpsp_sandbox.bottlenecks` sub-package contains the implementations of the Identification Indicator-based Relaxing Algorithm and Schedule Suffix Interval Relaxing Algorithm, and a framework for evaluating the algorithms on problem instances. The `improvements` module contains the algorithms' implementations together with implementations of helper functions from Appendix A.1. The `evaluations` module contains the `evaluate_algorithms` and the `compute_evaluation_kpis` functions. Those are central for the experiments: the former runs the algorithms with specified parameters on a given problem instance and return the sets of computed evaluations, the latter computes the experiment KPIs of the evaluations.

Following is a minimal working example of evaluating both algorithms utilizing the `evaluate_algorithms` function on a problem instance parsed from the `instance.json` file. The set of all algorithm parameters is the exact same set used for the experiments conducted in Chapter 4.

```
import rcpsp_sandbox.instances.io as iio
from rcpsp_sandbox.bottlenecks.evaluations \
    import evaluate_algorithms

instance = iio.parse_json("instance.json", is_extended=True)
evaluations = evaluate_algorithms(instance, [
    (ScheduleSuffixIntervalRelaxingAlgorithm(), {
        "max_iterations": [1, 2, 3],
        "relax_granularity": [1],
        "max_improvement_intervals": [1, 2, 3, 4, 5, 6],
        "interval_sort": ["improvement", "time"]}),
    (IdentificationIndicatorRelaxingAlgorithm(), {
        "metric": ["auau", "mrur"],
        "granularity": [4, 8],
        "convolution_mask": ["pre1", "around", "post"],
        "max_iterations": [1, 2, 3],
        "max_improvement_intervals": [1, 2, 3, 4],
        "capacity_addition": [4, 10]}),
])
```

The `rcpsp_sandbox.manager` module contains the `ExperimentManager` class, which manages loading and saving of experiment evaluations, KPIs, and problem instances. It can be passed to the `evaluate_algorithms` function to attempt loading existing evaluations from files before computing them anew.

A.3 Full instance plots

Figure A.1 Capacity changes cost (x-axis) to achieved improvement (y-axis) for every experiment instance.

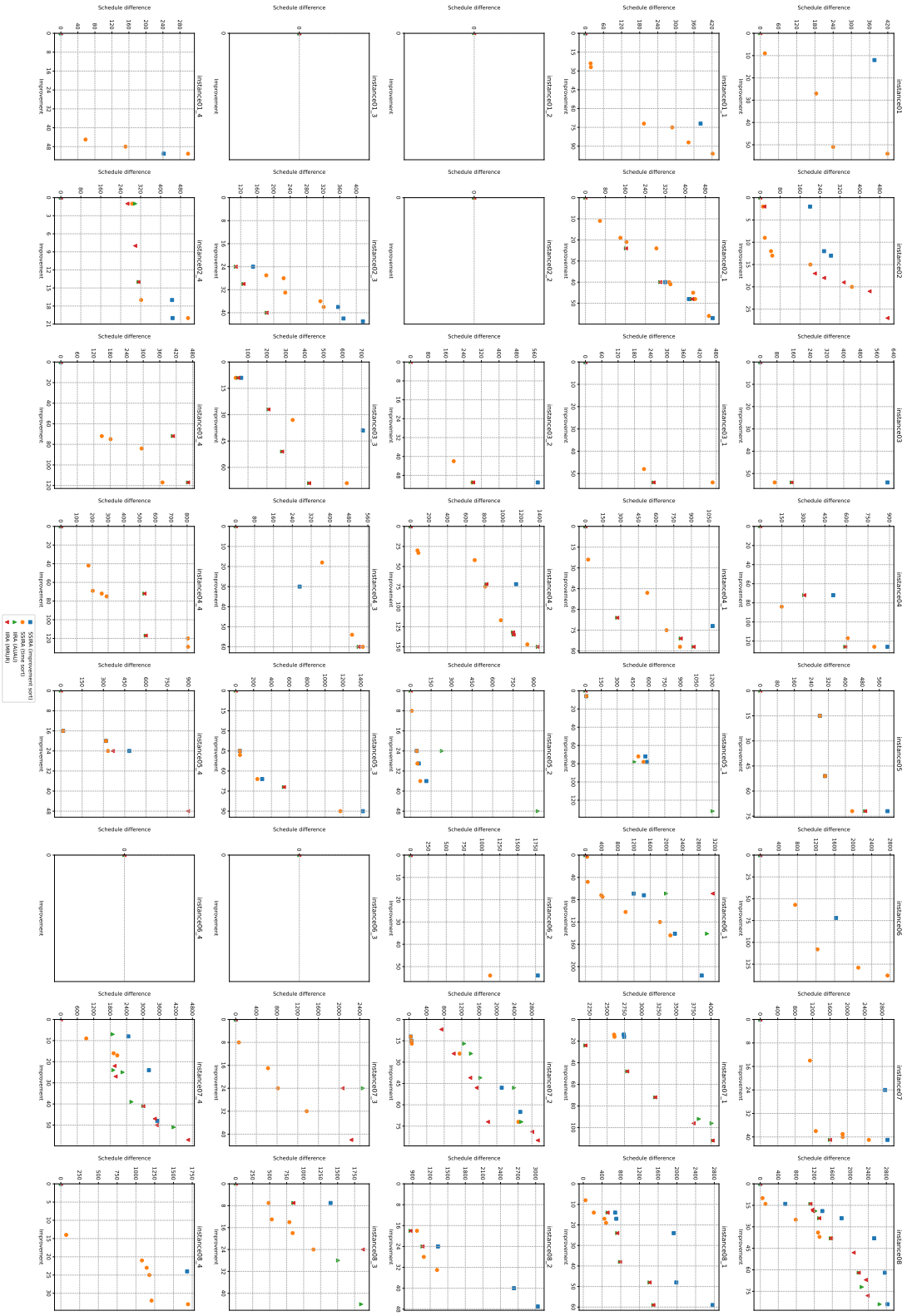


Figure A.2 Achieved improvement (x-axis) to schedule difference (y-axis) for every experiment instance.

Figure A.3 Evaluation duration (x-axis) to achieved improvement (y-axis) for every experiment instance.