Annals of Operations Research (2021) 302:533–562
https://doi.org/10.1007/s10479-021-04007-1

S.I. : PATAT 2018

A local search framework for industrial test laboratory scheduling

Florian Mischek[1] · Nysret Musliu[1]

Accepted: 23 February 2021 / Published online: 8 March 2021
© The Author(s) 2021

Abstract
In this paper we introduce a complex scheduling problem that arises in a real-world industrial test laboratory, where a large number of activities has to be performed using qualified personnel and specialized equipment, subject to time windows and several other constraints. The problem is an extension of the well-known Resource-Constrained Project Scheduling Problem and features multiple heterogeneous resources with very general availability restrictions, as well as a grouping phase, where the jobs have to be assembled from smaller units. We describe an instance generator for this problem and publicly available instance sets, both randomly generated and real-world data. Finally, we present and evaluate different metaheuristic approaches to solve the scheduling subproblem, where the assembled jobs are already provided. Our results show that Simulated Annealing can be used to achieve very good results, in particular for large instances, where it is able to consistently find better solutions than a state-of-the-art constraint programming solver within reasonable time.

Keywords RCPSP · Local search · Real-world · Simulated annealing

1 Introduction

Project scheduling problems appear in countless variations wherever multiple activities have to be scheduled and assigned resources of some kind, subject to various constraints. Examples include production and manufacturing environments, event management, software development, and many more. Since these problems can become quite large and include complex constraints in practical settings, there is an ever increasing need for automated solution approaches to produce high-quality solutions in acceptable time.

In this paper, we introduce a new real-world scheduling problem that arises in an industrial test laboratory of a large company. It is an extension to the well-known Resource-Constrained Project Scheduling Problem (RCPSP), on which it builds by adding various additional exten-

✉ Florian Mischek
fmischek@dbai.tuwien.ac.at

Nysret Musliu
musliu@dbai.tuwien.ac.at

[1] Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, DBAI, TU Wien, Favoritenstraße 9-11, 1040 Vienna, Austria

sions, both traditional and new, to capture the specific requirements of this, and other similar laboratories.

In the Test Laboratory Scheduling Problem (TLSP) tasks have to be grouped into larger units called jobs, which derive their properties from the tasks they contain. In general, tasks within a job are executed sequentially, but without any predefined order, and therefore jobs have to fulfill all requirements of their tasks over their whole duration. A detailed discussion of the properties of a job and the motivation behind it can be found in Sect. 3.2 of this paper.

Afterwards, the jobs have to be scheduled (i.e. assigned a mode, timeslots and resources), subject to various constraints. Besides several well-known features of (extensions of) the RCPSP from the literature, such as multiple execution modes and time windows, the TLSP also features additional constraints imposed by the real-world problem setting. In particular, RCPSP and its variants usually assume that units of a resource are identical and homogeneous and therefore can be used to cover the demand of all tasks. Due to this, an assignment of individual units to tasks is not necessary. Exceptions exist, e.g. by Dauzère-Pérès et al. (1998) or Bellenguez and Néron (2005), and sometimes an equivalent effect is achieved by introducing additional modes for each possible resource assignment (e.g. by Schwindt and Trautmann (2000) and Bartels and Zimmermann (2009)). However, this is practical only for a single resource with very few available values per task. In contrast, TLSP features multiple heterogeneous resources, with very general restrictions on which units can be used to perform a job, and potentially large demands.

In addition, jobs in TLSP can be *linked* to each other, indicating that these jobs must be performed by the same employees[1]. To the best of our knowledge, a similar concept exists only by Salewski et al. (1997) and Drexl et al. (2000), where the mode of several jobs is required to be the same.

In real-life practice, it is commonly the case that the grouping of tasks into jobs is already known and only a solution for the scheduling part of the problem is required. This gives rise to a restricted problem variant we denote as TLSP-S, which has a (fixed) list of jobs as additional input, but otherwise follows the same restrictions as TLSP.

The contributions of this article are as follows:

- We introduce TLSP, a new and complex real-world scheduling problem that is of direct practical relevance. For this new problem, we provide a formal definition and describe its subproblem TLSP-S. In addition, we prove the NP-hardness of both TLSP and TLSP-S via a reduction from RCPSP.
- We provide an instance generator for TLSP, which is capable of randomly generating instances based on real-world data with a wide variety of configuration options.
- Two sets of new generated instances for TLSP, plus three real-world instances taken directly from the laboratory of our industrial partner, are made publicly available for download.
- We developed a new local search framework for TLSP-S, and evaluate the performance of different meta-heuristics on the problem. We show that Simulated Annealing can be used to produce high-quality results that rival those of a state-of-the-art constraint programming model and consistently outperforms it on larger instances. The proposed algorithms have been used successfully in the daily scheduling in the laboratory of our industrial partner.

[1] This can be used to model multi-step procedures that require knowledge from previous steps, such as ensuring that documentation of completed tasks is prepared by the employees who actually performed the tasks.

This article is an extension of work presented at the conference on Practice and Theory in Automated Timetabling (PATAT) 2018 (see also the technical report Mischek and Musliu (2018)) where we first introduced the problem definition for TLSP.

The rest of the paper is structured as follows: Section 2 gives a short overview over related literature and variants of RCPSP that are particularly relevant for TLSP. Section 3 formally introduces the problem definition, input data and constraints. The last subsection also provides a reduction outline of RCPSP to TLSP, thereby showing the NP-hardness of the problem. Section 4 provides information about our instance generator and available data sets. Section 5 describes the local search framework and the algorithms used, followed by experimental results in Sect. 6. Finally, concluding remarks and future work are given in Sect. 7.

2 Related literature

2.1 Problem features

As the standard problem in the area of project scheduling, RCPSP has seen vast amounts of work over several decades. For surveys on literature regarding this problem and its many variants, we refer to surveys e.g. by Mika et al. (2015); Hartmann and Briskorn (2010) and Brucker et al. (1999). In particular, Hartmann and Briskorn (2010) provide a comprehensive overview regarding work dealing with extensions to RCPSP.

Many of these extensions contain problem features that are also found in the TLSP, or at least related concepts. Table 1 shows a selection of these features, as well as references from the literature for RCPSP variants that contain these particular features or at least some related concepts. While some aspects (e.g. multiple modes, setup times) are already well known, others have few (e.g. heterogeneous resources) or no (task grouping, linked tasks) direct correspondence in previously studied problems, to the best of our knowledge.

In the following, we will provide more details about these features and their treatment in the literature, describing both similarities and differences to TLSP where appropriate.

The first of these, and one of the most well-known, is multi-mode RCPSP (MRCPSP), originally formulated by Elmaghraby (1977). It allows each activity to be performed in one of several modes which can affect duration and resource requirements. A survey focused solely on MRCPSP formulations is provided by Węglarz et al. (2011).

Multiple separate projects, with project-specific constraints and objective functions, appear in the Resource Constrained Multi-Project Scheduling Problem (RCMPSP). Other works dealing with this problem are e.g. by Gonçalves et al. (2008) and Villafáñez et al. (2019). Wauters et al. (2016) introduce the Multi-Mode RCMPSP (MMRCMPSP), which combines both multiple modes and multiple projects and was used for the MISTA 2013 challenge.

RCMPSP also features multiple objectives, including one that is similar to the project completion time objective in TLSP. The main difference is that in RCMPSP, the completion times are normalized using the length of the critical path of each project. Also, the remaining objectives are completely unrelated to those that appear in TLSP. One of the main impacts of alternative (multi-)objective functions is that permutation schedules (Hartmann (1998)), which represent a solution as a permutation of activities and are used in many state-of-the-art works on RCPSP, are no longer guaranteed to yield optimal solutions.

Table 1 Main features of TLSP and their correspondence in other variants of RCPSP. The last column includes papers that contain similar concepts, which may not be directly applicable for TLSP

Problem feature	RCPSP extension	Related concept
Task grouping	–	Trautmann and Schwindt (2005); Wilson et al. (2012)
Setup times	Survey: Mika et al. (2006)	Batch scheduling, e.g. Schwindt and Trautmann (2000); Potts and Kovalyov (2000)
Multiple modes	MRCPSP, Survey: Węglarz et al. (2011), e.g. Wauters et al. (2016)	
Multiple projects	RCMPSP, e.g. Villafáñez et al. (2019), MMRCMPSP by Wauters et al. (2016)	
Heterogeneous resources	Dauzère-Pérès et al. (1998), MSPSP by Bellenguez and Néron (2005)	e.g. Schwindt and Trautmann (2000); Bartels and Zimmermann (2009)
Linked tasks	–	Salewski et al. (1997); Drexl et al. (2000)
Project completion time objective	Appears in RCMPSP, e.g. Villafáñez et al. (2019); Gonçalves et al. (2008)	

Setup times have been studied extensively in various forms, here we refer to the survey by Mika et al. (2006). Of note is that in particular sequence- or schedule-dependent setup times can be used to model batch processing, treated e.g. by Schwindt and Trautmann (2000) or Potts and Kovalyov (2000), which is related to the task grouping formalism in TLSP. Both approaches attempt to remove overheads due to setup times by grouping multiple smaller activities into larger units. The main difference is that in batch processing approaches, the batches arise implicitly from the completed schedule and the scheduled activities correspond to tasks. In contrast, in TLSP jobs are created explicitly from tasks, which are not directly scheduled at all.

The only paper using a similarly explicit approach to grouping small activities into larger units is by Trautmann and Schwindt (2005). They decompose a batch-scheduling problem into two independent subproblems, of which the first assembles the batches and the second produces a schedule for the previously created batches.

As mentioned in the previous section, the default assumption in RCPSP and most of its variants is that units of a resource are interchangeable, i.e. each unit can be used for each activity. A notable exception is by Dauzère-Pérès et al. (1998), who employ very general resource requirements which are even more flexible than those in TLSP. Unfortunately, the authors make heavy use of (a variant of) permutation schedules, which are unsuitable for TLSP.

A second example of heterogeneous resources can be found in the Multi-Skill RCPSP (MSPSP), introduced by Bellenguez and Néron (2005). In MSPSP, each resource unit has a number of skills, and resource requirements are given as a multi-set of required skills. Despite this different resource model, a distinct objective function and several other discrepancies, MSPSP is still quite closely related to TLSP-S. Instances for MSPSP contain up to 90 activities, compared to 300+ jobs for instances of practical size for TLSP-S.

Alternatively, heterogeneous resources like those in TLSP can sometimes be modeled using additional modes in MRCPSP (Bellenguez and Néron (2005)). Since each potential subset of assigned resources has to be encoded in a separate mode, this is only feasible for single resources and small numbers and therefore unsuitable for TLSP.

As far as we are aware, the concept of linked jobs, which need to have the same resource units assigned, has not been applied to any variant of RCPSP before. Salewski et al. (1997) and Drexl et al. (2000) deal with a somewhat related concept, where several activities have to be scheduled using the same mode, as described in Sect. 1. Of course, this is easier to add on top of RCPSP, since the assumption of homogeneous resources can be retained.

Finally, we would like to mention two problems dealing with scheduling activities in laboratories, due to the similarities in their overall setting with TLSP: Bartels and Zimmermann (2009) deal with scheduling tests of experimental vehicles. The described problem contains several aspects and constraints similar to TLSP. However, it uses a different resource model (in particular regarding destructive tests) and uses the number of employed vehicles as the main optimization criterion. Polo Mejia et al. (2017) developed an integer linear program for scheduling research activities for a nuclear laboratory, using a problem formulation derived from MSPSP, but with (limited) preemption of activities. Despite a similar setting, the requirements are unfortunately incompatible with those of TLSP.

Overall, to the best of our knowledge there is no variant of RCPSP that can fully model the requirements of TLSP(-S). Regarding TLSP-S, the closest related problem is probably MSPSP due to similar restrictions on the availability of resources to each activity. However, trying to model the very general restrictions of TLSP(-S) for MSPSP would result in a prohibitively large number of skills, not to speak of other features like linked jobs or objective criteria, which are not included in MSPSP at all. Therefore, a new approach is required to model and solve the TLSP(-S).

2.2 Solution approaches

As varied as the different extensions to RCPSP are also the solution approaches used to solve them. While initially the focus was on problem-specific heuristics, such as priority or dispatching rules, the last few decades have seen much work on metaheuristics, both local search-based and population-based, but also hybrid heuristic approaches combining two or multiple other techniques (Pellerin et al. (2020); Mika et al. (2015)).

Such a hybrid approach using a combination of memetic and hyperheuristic methods with Monte-Carlo tree search by Asta et al. (2016) won the 2013 MISTA challenge mentioned above, which dealt with MMRCMPSP. The same problem is also treated by Ahmeti and Musliu (2018), who provided several ideas that were useful in our solver implementation for TLSP-S.

Exact approaches have so far been mostly limited to smaller or more tightly constrained variants of RCPSP. However, recent years have seen some progress, in particular in the area of Constraint Programming (CP), e.g. by Szeredi and Schutt (2016), who developed a CP model for MRCPSP that is able to solve instances of realistic sizes.

Given its similarities to TLSP-S, successful solution approaches for MSPSP may be of particular interest. To the best of our knowledge, the best results so far for MSPSP have been achieved by Young et al. (2017), who also used a CP approach to solve the problem.

In general, metaheuristics seem to be the most promising approach to a new variant of RCPSP, such as TLSP-S, both due to their ease of implementation and their consistently high

performance on existing problem variants in the literature. Where exact solution approaches are desired, CP should certainly be considered, given its recent successes.

3 Problem description

In TLSP, a list of projects is given, which each contain several tasks. For each project, the tasks must be partitioned into a set of jobs, with some restrictions on the feasible partitions. Then, those jobs must each be assigned a mode, time slots and resources. The properties and feasible assignments for each job are calculated from the tasks contained within.

A solution of TLSP is a schedule consisting of the following parts:

– A list of jobs, composed of one or multiple similar tasks within the same project.
– For each job, an assigned mode, start and end time slots, the employees scheduled to work on the job, and an assignment to a workbench and equipment.

The quality of a schedule is judged according to an objective function that is the weighted sum of several soft constraints and should be minimized. Among others, these include the number of jobs and the total completion time (start of the first job until end of the last) of each project.

3.1 Input parameters

A TLSP instance can be split into three parts: The laboratory *environment*, including a list of resources, a list of *projects* containing the tasks that should be scheduled together with their properties and the current state of the *existing schedule*, which might be partially or completely empty.

3.1.1 Environment

In the laboratory, resources of different kinds are available that are required to perform tasks:

– *Employees* $e \in E = \{1, \dots, |E|\}$ who are qualified for different types of tasks.
– A number of *workbenches* $b \in B = \{1, \dots, |B|\}$ with different facilities. (These are comparable to machines in shop scheduling problems.)
– Various auxiliary lab *equipment* groups $G_g = \{1, \dots, |G_g|\}$, where g is the group index. These each represent a set of similar devices. The set of all equipment groups is called G^*.

The scheduling period is composed of discrete *time slots* $t \in T = \{0, \dots, |T| - 1\}$. Each time slot represents half a day of work.

Tasks are performed in one of several *modes* labeled $m \in M = \{1, \dots, |M|\}$. The chosen mode influences the following properties of tasks performed under it:

– The *speed factor* v_m, which will be applied to the task's original duration.
– The number of *required employees* e_m.

3.1.2 Projects and tasks

Given is a set P of *projects* labeled $p \in \{1, \dots, |P|\}$. Each project contains *tasks* $pa \in A_p$, with $a \in \{1, \dots, |A_p|\}$. The set of all tasks (over all projects) is $A^* = \bigcup_{p \in P} A_p$.

Each task pa has several properties:

- It has a *release date* α_{pa} and both a *due date* $\bar{\omega}_{pa}$ and a *deadline* ω_{pa}. The difference between the latter is that a due date violation only results in a penalty to the solution quality, while deadlines must be observed.
- $M_{pa} \subseteq M$ is the set of *available modes* for the task.
- The task's *duration* d_{pa} (in time slots, real-valued). Under any given mode $m \in M_{pa}$, this duration becomes $d_{pam} := d_{pa} * v_m$.
- Most tasks must be performed on a workbench. This is indicated by the boolean parameter $b_{pa} \in \{0, 1\}$. If required, this workbench must be chosen from the set of *available workbenches* $B_{pa} \subseteq B$.
- Similarly, pa requires *qualified employees* chosen from $E_{pa} \subseteq E$. The required number depends on the mode. A further subset $E_{pa}^{Pr} \subseteq E_{pa}$ is the set of *preferred employees*.
- Of each equipment group $g \in G^*$, the task requires r_{pag} devices, which must be taken from the set of *available devices* $G_{pag} \subseteq G_g$.
- A list of direct *predecessors* $\mathcal{P}_{pa} \subseteq A_p$, which must be completed before the task can start. Note that precedence constraints can only exist between tasks in the same project.

Each project's tasks are partitioned into *families* $F_{pf} \subseteq A_p$, where f is the family's index. For a given task pa, f_{pa} gives the task's family. Only tasks from the same family can be grouped into a single job.

Additionally, each family f is associated with a certain *setup time* s_{pf}, which is added to the duration of each job containing tasks of that family.

Finally, each project p may define *linked tasks*, which must be assigned the same employee(s). Linked tasks are given by the equivalence relation $L_p \subseteq A_p \times A_p$, where two tasks pa and pb are linked if and only if $(pa, pb) \in L_p$.

3.1.3 Initial schedule

All problem instances include an initial (or base) schedule, which may be completely or partially empty. This schedule can act both as an initial solution and as a baseline, placing limits on the schedules of employees and tasks, in particular by defining fixed assignments that must not be changed.

Provided is a set of jobs J^0, where each job $j \in J^0$ contains the following assignments:

- The tasks in the job: $\dot{A}_j$
  - A *fixed* subset of these tasks $\dot{A}_j^F \subseteq \dot{A}_j$. All fixed tasks of a job in the base schedule must also appear together in a single job in the solution.
- The mode assigned to the job: $\dot{m}_j$
- The start and completion times of the job: $\dot{t}_j^s$ resp. $\dot{t}_j^c$
- The resources assigned to the job:
  - Workbench: $\dot{b}_j$
  - Employees: $\dot{E}_j$
  - Equipment: $\dot{G}_{gj}$ for equipment group g

Except for the tasks, each individual assignment may or may not be present in any given job. Fixed tasks are assumed to be empty, if not given. In all other cases, missing assignments will be referred to using the value ϵ. Time slots and employees can only be assigned if also a mode assignment is given.

A subset of these jobs are the *started jobs* J^{0S}. A started job $j^s \in J^{0S}$ must fulfill the following conditions:

- It must contain at least one fixed task. It is assumed that the fixed tasks of a started job are currently being worked on.
- Its start time must be 0.
- It must contain resource assignments fulfilling all requirements.

A started job's duration does not include a setup time. In the solution, the job containing the fixed tasks of a started job must also start at time 0. Usually, the resources available to the fixed tasks of a started job are additionally restricted to those assigned to the job, to avoid interruptions of ongoing work in case of a rescheduling.

3.2 Jobs and grouping

For various operational reasons, tasks are not scheduled directly. Instead, they are first grouped into larger units called *jobs*.

A single job can only contain tasks from the same project and family.

Jobs have many of the same properties as tasks, which are computed from the tasks that make up a job. The general principle is that within a job, tasks are not explicitly ordered or scheduled; therefore the job must fulfill all requirements of each associated task during its whole duration.

The motivation behind this restriction, which deliberately overconstrains the schedule, is due to a combination of conditions in the laboratory of our industrial partner:

- Tasks of the same family usually have equivalent or very similar requirements in practice.
- Many tasks only cover a small fraction of a timeslot (e.g. half an hour out of a four-hour timeslot). Scheduling tasks directly to timeslots would therefore incur unacceptable overheads due to rounding. An alternative solution to this problem would be shorter timeslots, which would conflict with the flexible working times in the lab.
- Related formalisms, e.g. schedule-dependent setup times (Mika et al. 2006), would be difficult to apply since the actual setup time between two tasks may depend on multiple resources.
- Tasks frequently need to be reordered or delayed. The chosen formulation guarantees that this is always possible within a job, adding a measure of flexibility to the schedule. Results by Wilson et al. (2012) indicate that this flexibility can be useful in the presence of delays during the execution of a schedule.

Let $J = \{1, \ldots, |J|\}$ be the set of all jobs in a solution and $J_p \subseteq J$ be the set of jobs of a given project p. Then for a job $j \in J$, the set of tasks contained in j is $\dot{A}_j$. j has the following properties:

$$\tilde{p}_j \quad \text{and} \quad \tilde{f}_j$$

are the project and family of j.

$$\tilde{\alpha}_j := \max_{pa \in \dot{A}_j} \alpha_{pa}, \quad \tilde{\bar{\omega}}_j := \min_{pa \in \dot{A}_j} \bar{\omega}_{pa}, \quad \tilde{\omega}_j := \min_{pa \in \dot{A}_j} \omega_{pa}$$

are the release date, due date and deadline of j, respectively.

$$\tilde{M}_j := \bigcap_{pa \in \dot{A}_j} M_{pa}$$

is the set of available modes.

$$\tilde{d}_{jm} := \lceil (s_{p_j f_j} + \sum_{pa \in \dot{A}_j} d_{pa}) * v_m \rceil$$

is the (integer) duration of the job under mode m. The additional setup time is added to the total duration of the contained tasks.

$$\tilde{b_j} := \max_{pa \in \dot{A}_j} b_{pa}$$

is the required number of workbenches ($\tilde{b}_j \in \{0, 1\}$)

$$\tilde{B}_j := \bigcap_{pa \in \dot{A}_j} B_{pa}$$

are the available workbenches for j.

$$\tilde{E}_j := \bigcap_{pa \in \dot{A}_j} E_{pa}$$

are the employees qualified for j.

$$\tilde{E}_j^{Pr} := \bigcap_{pa \in \dot{A}_j} E_{pa}^{Pr}$$

are the preferred employees of j.

$$\tilde{r}_{jg} := \max_{pa \in \dot{A}_j} r_{pag}$$

are the required units of equipment group g.

$$\tilde{G}_{jg} := \bigcap_{pa \in \dot{A}_j} G_{pag}$$

are the available devices for equipment group g.

$$\tilde{\mathcal{P}}_j := \{k \in J \setminus \{j\} : \exists pa \in \dot{A}_j, pb \in \dot{A}_k \text{ s.t. } pb \in \mathcal{P}_{pa}\}$$

is the set of predecessor jobs of j. Finally,

$$\tilde{L}_p := \{(j, k) \in J \times J : j \neq k \wedge \exists pa \in \dot{A}_j, pb \in \dot{A}_k \text{ s.t. } (pa, pb) \in L_p\}$$

defines the linked jobs in project p.

In addition, a solution contains the following assignments for each job:

- $\dot{t}_j^s \in T$ the scheduled start time slot
- $\dot{t}_j^c \in T$ the scheduled completion time
- $\dot{m}_j \in M$ the mode in which the job should be performed
- $\dot{b}_j \in B$ the workbench assigned to the job (ϵ if no workbench is required)
- $\dot{E}_j \subseteq E$ the set of employees assigned to the job
- $\dot{G}_{jg} \subseteq G_g$ the set of assigned devices from equipment group g

3.3 Constraints

A solution is evaluated in terms of constraints that it should fulfill. *Hard constraints* must all be satisfied in any feasible schedule, while the number and degree of violations of *soft constraints* in a solution give a measure for its quality.

For the purpose of modeling, we introduce additional notation: The set of *active jobs* at time t is defined as $\mathcal{J}_t := \{j \in J : \dot{t}^s_j \leq t \wedge \dot{t}^c_j > t\}$.

3.3.1 Hard constraints

H1: Job assignment Each task must be assigned to exactly one job.

$$\begin{aligned} &\forall p \in P, pa \in A_p : \\ &\quad \exists! j \in J \text{ s.t. } pa \in \dot{A}_j \end{aligned}$$

H2: Job grouping All tasks contained in a job must be from the same project and family.

$$\begin{aligned} &\forall j \in J, pa \in \dot{A}_j : \\ &\quad p = \tilde{p}_j \\ &\quad f_{pa} = \tilde{f}_j \end{aligned}$$

H3: Fixed tasks Each group of tasks assigned to a fixed job in the base schedule must also be assigned to a single job in the solution.

$$\begin{aligned} &\forall j^0 \in J^0 : \\ &\quad \exists j \in J \text{ s.t. } \dot{A}^F_{j^0} \subseteq \dot{A}_j \end{aligned}$$

H4: Job duration The interval between start and completion of a job must match the job's duration.

$$\begin{aligned} &\forall j \in J : \\ &\quad \dot{t}^c_j - \dot{t}^s_j = \tilde{d}_{j\dot{m}_j} \end{aligned}$$

H5: Time Window Each job must lie completely within the time window from the release date to the deadline.

$$\begin{aligned} &\forall j \in J : \\ &\quad \dot{t}^s_j \geq \tilde{\alpha}_j \\ &\quad \dot{t}^c_j \leq \tilde{\omega}_j \end{aligned}$$

H6: Task precedence A job can start only after all prerequisite jobs have been completed.

$$\begin{aligned} &\forall j \in J, k \in \tilde{\mathcal{P}}_j : \\ &\quad \dot{t}^c_k \leq \dot{t}^s_j \end{aligned}$$

H7: Started jobs A job containing fixed tasks of a started job in the base schedule must start at time 0.

$$\begin{aligned} &\forall j \in J, j^s \in J^{0S} : \\ &\quad t^F_{j^s} = 1 \wedge \dot{A}^F_{j^s} \subseteq \dot{A}_j \implies \dot{t}^s_j = 0 \end{aligned}$$

H8: Single assignment At any one time, each workbench, employee and device can be assigned to at most one job.

$$\begin{aligned}&\forall b \in B, t \in T: \\ &\quad |\{j \in \mathcal{J}_t : \dot{b}_j = b\}| \leq 1 \\ &\forall e \in E, t \in T: \\ &\quad |\{j \in \mathcal{J}_t : e \in \dot{E}_j\}| \leq 1 \\ &\forall g \in G^*, d \in G_g, t \in T: \\ &\quad |\{j \in \mathcal{J}_t : d \in \dot{G}_{jg}\}| \leq 1\end{aligned}$$

H9a: Workbench requirements Each job requiring a workbench must have a workbench assigned.

$$\begin{aligned}&\forall j \in J: \\ &\quad \dot{b}_j = \epsilon \iff \tilde{b}_j = 0\end{aligned}$$

H9b: Employee requirements Each job must have enough employees assigned to cover the demand given by the selected mode.

$$\begin{aligned}&\forall j \in J: \\ &\quad |\dot{E}_j| = e_{\dot{m}_j}\end{aligned}$$

H9c: Equipment requirements] Each job must have enough devices of each equipment group assigned to cover the demand for that group.

$$\begin{aligned}&\forall j \in J, g \in G^*: \\ &\quad |\dot{G}_{jg}| = \tilde{r}_{jg}\end{aligned}$$

H10a: Workbench suitability The workbench assigned to a job must be suitable for all tasks contained in it.

$$\begin{aligned}&\forall j \in J: \\ &\quad \dot{b}_j = \epsilon \vee \dot{b}_j \in \tilde{B}_j\end{aligned}$$

H10b: Employee qualification All employees assigned to a job must be qualified for all tasks contained in it.

$$\begin{aligned}&\forall j \in J: \\ &\quad \dot{E}_j \subseteq \tilde{E}_j\end{aligned}$$

H10c: Equipment availability The devices assigned to a job must be taken from the set of available devices for each group.

$$\begin{aligned}&\forall j \in J, g \in G^*: \\ &\quad \dot{G}_{jg} \subseteq \tilde{G}_{jg}\end{aligned}$$

H11: Linked jobs Linked jobs must be assigned exactly the same employees.

$$\begin{aligned}&\forall p \in P, (j, k) \in \tilde{L}_p: \\ &\quad \dot{E}_j = \dot{E}_k\end{aligned}$$

3.3.2 Soft constraints

The following constraints can be used to evaluate the quality of a feasible solution. They arise from the business requirements of our industrial partner and have been formulated in close cooperation with them.

Each soft constraint violation induces a penalty on the solution quality, denoted as C^i, where i is the soft constraint violated.

S1: Number of jobs The number of jobs should be minimized.

$$:$$
$$C^{S1} := |J|$$

S2: Employee project preferences The employees assigned to a job should be taken from the set of preferred employees.

$$\forall j \in J :$$
$$C_j^{S2} := |\{e \in \dot{E}_j : e \notin \tilde{E}_j^{Pr}\}|$$

S3: Number of employees The number of employees assigned to each project should be minimized.

$$\forall p \in P :$$
$$C_p^{S3} := |\bigcup_{j \in J_p} \dot{E}_j|$$

S4: Due date The internal due date for each job should be observed.

$$\forall j \in J :$$
$$C_j^{S4} := \max(\dot{t}_j^c - \tilde{\bar{\omega}}_j, 0)$$

S5: Project completion time The total completion time (start of the first test to end of the last) of each project should be minimized.

$$\forall p \in P :$$
$$C_p^{S5} := \max_{j \in J_p} \dot{t}_j^c \; - \; \min_{j \in J_p} \dot{t}_j^s$$

Constraint S1 favors fewer, longer jobs over more fragmented solutions. This helps reducing overhead (fewer setup periods necessary, rounding of fractional durations) and increases the flexibility of the schedule, since tasks within a job can be freely reordered. Also, it reduces the complexity of the final schedule, both for the employees performing the actual tasks and any human planners in those cases where manual corrections or additions become necessary.

In practice in the lab of our industrial partner, it has proved efficient to have only few employees cover the tests of a single project, due to the presence of project-specific conditions and procedures as well as the need for continual coordination and communication with other parts of the laboratory as well as external clients. This is captured by constraint S3.

Constraint S4 makes the schedule more robust by encouraging jobs to be completed earlier than absolutely required, so they can still be finished on time in case of delays or other disturbances.

Finally, constraint S5 also helps reduce overheads, as longer timespans between the tests of a project would require additional effort to become familiar with project-specific test procedures, as well as storage space for the devices in between tests.

The overall solution quality will be determined as the weighted sum over all soft constraint violations.

The relative importance of these constraints (i.e. their weights) still needs to be defined. In practical applications, we expect them to be chosen interactively according to the current situation. For our evaluations, we have assumed a uniform weight of 1 for all soft constraints.

3.4 The TLSP-S problem

A practically relevant subproblem of the TLSP deals with the case where a grouping of tasks into jobs is already provided for each project, which cannot be changed by the solver. Thus, the goal is to find an assignment of a mode, time slot and resources to each (given) job, such that all constraints are fulfilled and the objective function is minimized. Since this variant focuses on the scheduling part of the problem, we denote it as *TLSP-S*.

In TLSP-S, the number and properties of jobs (see Sect. 3.2) are fixed and can be precomputed from the properties of the tasks they contain.

Without loss of generality, we assume that this fixed initial grouping is provided via the initial schedule. Thus, an instance for TLSP-S can also be given as input to a solver for TLSP and vice-versa, as long as the initial schedule contains a valid job grouping for all tasks.

For TLSP-S, constraints H1–H3 will always be trivially satisfied and can therefore be ignored. Similarly, the penalty induced by soft constraint S1 is constant, since the number of jobs cannot be modified. We still include this penalty in our results for comparability of the solutions with instances of TLSP.

3.5 Complexity analysis of TLSP

In this section, we provide a reduction from RCPSP to TLSP, to show that TLSP is indeed a generalization of RCPSP. Since RCPSP is known to be NP-hard (Blazewicz et al. (1983)), this also shows the NP-hardness of TLSP.

RCPSP can be defined as follows (summarized from the definition given by Hartmann and Briskorn (2010)): Input is a list of activities $A = \{1, \ldots, |A|\}$. Each activity a has a duration d_a, a list of predecessors $P_a \subset A$, which must be completed before a can start, and resource requirements r_{ak} for each resource $k \in K = \{1, \ldots, |K|\}$. Each resource has a capacity c_k, which denotes the number of available units per timeslot. The goal is to find a start time for each activity, such that precedence constraints are fulfilled, the capacity of each resource is not exceeded in any timeslot and the total makespan (i.e. the end of the last activity in the schedule) is minimal.

For this reduction, we use the decision variant of RCPSP (which was also proven NP-hard by Blazewicz et al. (1983)), which takes an integer s as additional input and asks whether a schedule with makespan $\leq s$ exists.

Given an arbitrary instance of RCPSP $\mathcal{I}$, we can then construct a corresponding instance of TLSP as follows:

- It contains a single project p_1.
- For each activity a in $\mathcal{I}$, we create a corresponding task in p_1 with the same duration as a. Each task has its own family (with a setup time of 0).
- Precedences between activities in $\mathcal{I}$ can be directly translated to the tasks of p_1.
- Each task has a release date of 0 and both a target date and a deadline of s.

- There is a single mode with speed factor 1 and no required employees. This mode is available for all tasks.
- For each resource k in $\mathcal{I}$, we create a new equipment group g_k, containing c_k devices.
- Equipment requirements for each task are determined as follows: A task requires r_{ak} devices of group g_k, where a is the activity in $\mathcal{I}$ that corresponds to the task. All devices in each group are available for all tasks.
- There are no workbenches or employees. Tasks do not require workbenches.
- There are no linked tasks.
- The base schedule is empty.

Since each family contains only a single task, the job grouping of each feasible schedule is uniquely determined: There is a separate job for each task, which has exactly the same properties (regarding duration, resource requirements, precedences, linked jobs) as the task it contains. For any activity a in $\mathcal{I}$, there must be exactly one corresponding job j_a, which is the one containing the task corresponding to a in the above construction.

Given a schedule (identified by the set of jobs J) that is a feasible solution for the TLSP instance constructed above, it is easy to see that it directly corresponds to a valid schedule $\mathcal{S}$ for the RCPSP instance $\mathcal{I}$, where each activity a starts at the same time as $j_a \in J$. Precedences are satisfied by construction and the capacity of each resource cannot be exceeded at any time, since otherwise at least one device would have to be assigned to two overlapping jobs in J at the same time (pidgeonhole principle). Further, all jobs in J must end before the deadline s. Since both start and duration of abilities and the corresponding jobs are equal, this also means that all activities in $\mathcal{S}$ must end before s, i.e. the makespan of $\mathcal{S}$ is at most s.

In the other direction, for any instance $\mathcal{I}$ of RCPSP, with a schedule S of makespan at most s, we can find a feasible schedule J for the corresponding instance of TLSP constructed as above. As before, each job $j_a \in J$ should start (and end) at the same time as the corresponding activity a in $\mathcal{S}$. Precedences and time windows are then satisfied by construction. Next, we will show that we can always find an equipment assignment for each job that fulfills both the requirements (constraint H9c) and the single assignment constraint (H8) are satisfied: We look at each job j in order of increasing start time. Let $J_{<j}$ be all jobs considered before j which overlap the start of j, $A_{<j}$ be the activities in $\mathcal{S}$ corresponding to those jobs and a be the activity corresponding to j. Since $\mathcal{S}$ is feasible, the activities in $A_{<j}$ can require a total of at most $c_k - r_{ak}$ units of each resource k (the remaining r_{ak} units are required for a). Therefore, there can be at most $c_k - r_{ak}$ devices assigned to jobs in $J_{<j}$. It follows that there are at least r_{ak} devices left to be assigned to j, which is enough to cover its requirements (H9c). Since jobs are considered in order of increasing start time, there can also be no other jobs not in $J_{<j}$ that overlap j and have those devices assigned. It follows that constraint H8 must be fulfilled at every step. All other constraints are trivially satisfied.

In conclusion, a schedule for the RCPSP instance with makespan $\leq s$ exists if and only if the corresponding TLSP instance is feasible. Since the reduction can be done in polynomial time, this proves that even the problem of finding any feasible solution for TLSP is already NP-hard.

If the base schedule is not empty, but instead the (unique) feasible job grouping is provided as additional input, we immediately arrive at an instance for TLSP-S. As before, each activity in $\mathcal{I}$ corresponds directly to both exactly one task and exactly one job in the constructed instance. Therefore, the same argument as for TLSP can be used to show the NP-hardness also of TLSP-S.

4 New instances for TLSP

In this section, we introduce an instance generator, which can be used to randomly generate instances for both TLSP and TLSP-S based on real-world data. We also propose two sets of new and publicly available instances for TLSP(-S) assembled using our instance generator. Finally, we describe three real-world instances taken from our industrial partner, which are also available online in anonymized form.

4.1 Instance generator

In order to be able to generate instances of specific size and complexity on demand, we developed an instance generator for TLSP. It randomly generates instances of various sizes that are based on the real-world data in the laboratory of our industrial partner and can be configured to produce instances of various sizes and properties. The generator was written in Java.

To generate a new instance, one has to specify the number of expected projects and the length of the scheduling period, plus optionally various configuration parameters that refine the desired properties of the instance. From this, an instance is generated as follows: First, the laboratory environment, including the available resources, is defined according to the desired number of projects and the length of the scheduling period. Then, the required number of projects is generated, together with a set of jobs for each project. This is used to populate the reference solution, which is guaranteed to be a feasible solution for the final problem instance. In the third step, task properties (resource requirements and availabilities, precedence constraints, time windows, ...) are defined in accordance with the reference schedule. Finally, the reference solution is modified to become the base schedule, which completes the instance generation.

4.1.1 Environment generation

From the number of projects given (together with their average total work) and the length of the scheduling period, a measure for the expected workload per time slot can be extracted. The number of employees and workbenches required to achieve a certain mean degree of utilization can be estimated from this measure.

Equipment is generated by a separate component, which can be passed to the instance generator. This component also handles equipment requirements of tasks in Step 4.1.2. Currently supported are two different implementations:

Lab equivalent mode generates devices for exactly those equipment groups that are relevant for planning at our industrial partner. The equipment requirements for tasks also closely corresponds to the distribution of requirements in the real-world laboratory.

General mode is more flexible and creates between 3 and 6 equipment groups, together with corresponding devices. Equipment requirements for each groups are selected to be either unitary (i.e. tasks require at most device of this group) or randomly generated for each project.

In both cases, the number of devices in each group depends on the same measure for workload as for the employees and workbenches, modified by the expected number of required devices per task.

In this step, also the task modes shown in Table 2 are generated.

Table 2 Task modes used by the instance generator

Mode	v_m	e_m
Single	1	1
Shift	0.6	2
External	1	0

4.1.2 Reference solution

Each project is assigned a certain total workload, which is taken from a distribution that is as close as possible to the real-world data. It is also assigned to a random interval of the scheduling period, where earlier intervals are slightly favored over later ones. Then a number of tasks are created, still without a duration. In general, the expected number of tasks grows with the project's workload. These tasks are then distributed into families and further into jobs. The families can either be taken from the real-world data or generated randomly, depending on the generator configuration.

Each generated job is then randomly assigned a preliminary duration (taken from the total workload of the project), a mode (with only a small probability for *External* mode), and equipment requirements according to the chosen equipment generation mode (see Step 4.1.1). Most jobs will also be set to require a workbench.

Once these parameters are determined, the job is placed into the reference schedule. This is done by first randomly choosing a seed point within the project's assigned interval. Starting at this point, the job is grown outward in both directions as long as any feasible resource assignment for it exists or until the desired duration is reached. If no feasible placement at the whole duration can be found, the position is still accepted as long as the duration is not much smaller than expected. Where this is not the case or no resource assignment exists even for the initial seed, the procedure is repeated for another random seed point, up to a certain maximum number of iterations. While this process cannot guarantee that a feasible schedule can be found, experiments so far have shown that this is sufficient in most cases.

Once a job is scheduled, it is assigned a feasible set of resources. Its final duration is split between its tasks, minus the setup time required for its family.

To model started jobs, the scheduling period is extended to the past by one month (about 40 time slots) in the beginning of this step. After all jobs have been scheduled, the scheduling period is reduced to its original duration. Jobs ending before the new first time slot are removed completely, including their contained tasks – it is assumed that those have been completed before the start of the scheduling period. Those overlapping the first time slot have their duration reduced accordingly (potentially including removing some of the contained tasks) and are defined as started jobs.

4.1.3 Task properties

After the reference schedule has been completed, the remaining properties of the tasks are finalized.

Release dates, deadlines and due dates for tasks of a project are set to the first start, respectively last end, of any job in the project. They are then extended outwards by an additional number of time slots (minimum of 0), which is usually smaller for the due date than for the deadline.

The available resources are set for each job (and all contained tasks) such that they include at least the assigned resources. The number of available resources is taken from the real-world data for employees (including preferred employees) and workbenches, and handled by the chosen equipment generation mode for equipment. A small subset of tasks may also have additional available resources beyond those of the other tasks in the job.

The available task modes are chosen such that tasks of a job with the *External* mode in the reference schedule must be performed in this mode, while all other tasks can be performed in *Single* mode, plus optionally in *Shift* mode. The latter is always the case if the task's job has *Shift* mode in the reference schedule.

Started jobs will always have their time window, available resources and modes set to exactly those values that they are assigned to, to ensure that they cannot be altered by the solver.

The generation of precedence constraints is again delegated to a separate component, which can be set in the configuration. Two implementations are currently supported, both start by building a maximum graph of possible dependencies according to the reference schedule for each project:

Ranked precedence constraints assign ranks to a subset of tasks such that tasks of higher rank have all tasks of lower rank as prerequisites.

General precedence constraints randomly choose arcs in the maximum dependency graph that will result in actual dependencies between tasks.

In both cases, the number of precedence constraints between tasks is rather low and most tasks don't have any prerequisite tasks at all. This circumstance is directly taken from the real-world data, where only few dependencies between tasks appear and contrasts with other project scheduling problems that include tighter constraints on the order of activities.

Finally, possible candidates for linked tasks are identified, both within and between jobs, and a small subset of those is chosen randomly.

4.1.4 Base schedule

The last step in the instance generation process is the derivation of a base schedule from the reference solution.

Again, there are several supported options for this, which the generator can be configured to use. Currently, there are two main implementations supported:

Delete mode Removes some assignments of the jobs in the reference schedule, but leaves the remaining assignments intact.

Random mode Replaces assignments of the jobs in the reference schedule by random values. Resource availability constraints and time windows are respected, but other constraints may be violated by these changes.

Either mode has a parameter that defines the strength of the perturbation (given as the percentage of jobs affected). Further, various flags can modify the behavior, including keeping certain types of assignments intact (e.g. the grouping of tasks into jobs).

Any perturbations do not affect started jobs as well as a number of jobs selected as *fixed*, which have their tasks fixed and either all or some assignments fixed to the current value by restricting the time window, available resources and/or modes to their assigned values.

Table 3 Instance generator configuration parameters for the two provided datasets

Parameter	LabStructure	General
Equipment	Lab equivalent	General
Task families	Lab equivalent	General
Precedence	Ranked	General
Base schedule mode	Delete	Delete
Base schedule perturbation	1.0	1.0
Base schedule options	Grouping constant	Grouping constant

4.2 Data sets

Currently, there are two sets of test data available, *General* and *LabStructure*, both with 60 instances each. These were generated for TLSP, but include a grouping of tasks into jobs in the (otherwise empty) base schedules that is guaranteed to have at least one feasible solution. Thus, they are directly usable also for TLSP-S. Feasible solutions for TLSP-S are guaranteed to also be feasible for TLSP under that grouping and have the same objective values.

Both data sets feature instances of various sizes, starting at 5 projects over a period of 88 time slots, up to 90 projects over 782 time slots. A single project contains an average of close to 4 jobs. There are no initial assignments except for the started jobs at the beginning of the scheduling period and a small number ($\approx 5\%$) of jobs whose assignments are fixed.

The difference between the two data sets is that for the LabStructure set, the instance generator was configured to produce instances that are as close as possible to the actual real-world data. In contrast, the General set uses the same distribution of work across projects and tasks, but is more flexible regarding several other structural features, such as equipment groups and job precedence. This distinction was made to obtain a diverse range of instances, including both those very similar to the real-world laboratory and those with different characteristics. The instance generator configurations used to create these two datasets are listed in Table 3.

On average, jobs have 5 available workbenches and 6 qualified employees (the different modes each require between 0 and 2 employees). The demand and availability of equipment is more varied and differs a lot between data sets and instances. Three different types of equipment demands appear: Of any given group, jobs require either a single specific device (these are usually project-specific), one out of a small list of options (< 10) or several (12 on average, but with a large variance) out of a large list (depends on the instance size, up to several hundred). The distribution of these types is heavily skewed towards the first two options. Further, the huge number of possible assignments for equipment demands of the third type is offset by the fact that most jobs do not require equipment of these groups and the number of devices per group is large enough that feasible equipment assignments can be found quite easily. These resource distributions were adapted from real-world data in our partner laboratory.

In addition to the two randomly generated data sets, we also provide three real-world instances taken directly from the laboratory of our industrial partner in anonymized form. These instances each cover a planning period of approximately one and a half years and contain between 59 and 74 projects. Table 4 lists some important parameters of these three instances.

Table 4 The three real-world instances. For each instance, the table lists the number of projects, jobs and the length of the scheduling period, followed by the number of employees, workbenches and equipment groups. The last columns contain the mean qualified employees and available workbenches per job, as well as the mean available devices per job and equipment group (only over those jobs that actually require at least one device of the group, about 10% of all jobs). *The generated instances also include tasks with multiple available devices per group. This discrepancy arises from the fact that several equipment groups were not yet considered for planning in practice at the time these instances were created

ID	$\|P\|$	$\|J\|$	h	$\|E\|$	$\|B\|$	$\|G^*\|$	$\overline{\|E_j\|}$	$\overline{\|B_j\|}$	$\overline{\|G_{gj}\|}$
2019-04	74	297	606	22	17	1	5.49	3.06	1*
2019-07	59	251	700	24	17	1	5.33	3.17	1*
2019-10	59	223	572	19	17	1	5.70	3.48	1*

All instances (including the real-world instances) are available for download at https://www.dbai.tuwien.ac.at/staff/fmischek/TLSP. Once additional data sets become available, they will also be added there.

5 Local search

For easy implementation of and comparison between different solution techniques, we have developed a solution framework that provides a unified workflow, common data structures and utility functionality for TLSP solver implementations.

While theoretically also applicable for other solution approaches, this framework is mainly intended for use with local search. Here, the basic building block is that of a *move*, which is a small change to a given solution, such as a replacement of a time slot or a resource assignment for a single job. Each move contains the necessary information to be applied to a schedule and to efficiently evaluate its effects on the solution quality. A basic set of move implementations are provided, which can be combined to form more complex changes. *Neighborhoods* define a set of moves available from a current schedule, and provide functionality to access and iterate over these moves. In addition they also allow for the selection of a random or the best move among those they contain. These neighborhoods are employed by *search heuristics*, which implement the strategies to choose a move that should be applied from among several neighborhoods in each step of the search.

5.1 Neighborhoods

For this paper, we have developed two different neighborhoods that are suitable to solve TLSP-S. Both are the combination of several smaller neighborhoods focused on specific types of moves, and contain the union of all these moves.

The first set, called *Simple*, contains neighborhoods that each affect a single aspect of a job's assignments. It consists of the following neighborhoods:

- *Mode change*: Switches the assigned mode of the job to a different value. The start time of the job is kept constant for all moves in this neighborhood, except where the new duration would conflict with time windows or precedence constraints. In these cases, the start time is adjusted to ensure that those constraints are satisfied. Also, the number of assigned employees is adjusted to match the new requirements.

- *Time slot change*: Moves the job to a new position in the schedule. As before, time windows and precedence constraints are respected by all possible moves.
- *Resource change*: Switches out a single assigned resource unit (workbench, employee or device) by a different unit of the same type (and group, for equipment).
- *Resource swap*: Swaps a unit of a resource assigned to this job with a different unit of the same type assigned to an overlapping job. A resource unit is considered for a swap only if it is suitable for its new job.

While moves from the first three neighborhoods are theoretically sufficient to reach an optimal solution from any schedule already satisfying time window and precedence constraints, the addition of resource swaps adds more options in situations where a single change would result in a prohibitively large number of conflicts.

The second variant is a larger neighborhood, with the main idea that a job is completely removed from the schedule and the neighborhood contains all possible combinations of mode, time slot and resource assignments for that job (as before, time windows and precedence constraints are respected by all moves).

However, the enormous number of potential equipment assignments per job in some instances made some adjustments necessary. For example, instance LabStructure/010 contains 88 devices in equipment group 1, of which 27 are required by job 45, for a total of $\binom{27}{88} \approx 3.3 \times 10^{22}$ different possible equipment assignments for this job alone.

Instead, we employed a reduced version of the neighborhood, which keeps the existing equipment assignments intact, and combined it with a Resource Change neighborhood limited to equipment changes. We refer to this combination of neighborhoods as *JobOpt*. To further increase the efficiency when the best move in the neighborhood is required, we utilize independencies between assigned resources to precompute and cache the effects of assigning individual units to the job.

Since the neighborhoods described above do not affect the task grouping, they are not suitable for solving TLSP, unless the are combined with additional neighborhoods covering the grouping aspect.

5.2 Search heuristics

The search heuristics make use of the available neighborhoods, which abstract away the problem specific details and can be dynamically changed. Thus, the heuristics are problem-agnostic and can be used for TLSP, TLSP-S and other related variants, as long as the employed neighborhoods are adjusted to fit. This makes implementing and modifying search heuristics very easy. For this paper, we have implemented and evaluated two well-known metaheuristics, Min-Conflict and Simulated Annealing, as an example of the kind of search strategies that are supported by the framework.

5.2.1 Min-conflict

The min-conflicts heuristic (Minton et al. (1992)) was originally developed to solve constraint satisfaction problems (including scheduling applications). It works by randomly selecting a variable appearing in at least one conflict (violated constraint) and choosing a value for it that minimizes the number of conflicts remaining.

This strategy can be adapted as follows for TLSP(-S) and our solver framework: Choose a job at random that violates at least one constraint, and pick a move from the neighborhood involving the chosen job that minimizes constraint violations (i.e. the best move).

Different selection strategies are possible, due to the distinction between hard and soft constraints. We have experimented with three different variants of Min-Conflict. The first selects jobs from among those violating hard constraints. Once the solution is feasible, also soft constraint violations are considered. The second variant considers both hard and soft constraint violations from the start. Finally, the third variant chooses from all jobs, regardless of their presence in any kind of constraint violation.

For TLSP(-S), the second and third variants are virtually equivalent due to soft constraints S3 (Number of employees) and S5 (Project completion time), whose presence entails that most, if not all, jobs are involved in at least one soft constraint violation[2]. Further, results did not differ in any meaningful way between the first and either of the latter variants.

For simplicity, and to avoid having to keep a running account of the jobs involved in constraint violations, the experiments in Sect. 6 have been done using variant three.

A weakness of MC is that it contains no mechanism to avoid repeating already visited solutions and thus might get stuck where several adjacent configurations for most or all jobs are locally optimal. A possible countermeasure is to inject additional randomness into the solution procedure. In our framework, we have also included a random walk (RW) algorithm, that randomly selects a job and performs a random move for the chosen job. RW is combined with MC in a hybrid heuristic that at each step calls RW with a low probability p^{RW}, and MC otherwise.

Also, the search automatically restarts from a new initial solution if no progress has been achieved within a certain number of moves.

5.2.2 Simulated annealing

Simulated Annealing (Kirkpatrick et al. (1983)) is a well-known metaheuristic that has been employed successfully to solve many NP-complete problems, including RCPSP (e.g. Bouleimen and Lecocq (2003); Laurent et al. (2017)).

In SA, the search starts from a randomly generated solution. In each step, a candidate move is chosen randomly from the available neighborhoods. The difference in objective value Δ due to the chosen move is calculated. If $\Delta < 0$ (for minimization problems), the move is applied. Otherwise, it is still accepted with probability $e^{-\Delta/T}$. Thus, the acceptance probability depends on Δ (smaller values have a larger probability to be accepted) and a parameter T called *temperature* (higher values result in a larger acceptance probability). The temperature starts at an initial value T^0 and is iteratively reduced by a *cooling factor* α, with $0 < \alpha < 1$, after a certain number of steps, until a minimum temperature T^{min} is reached. At this point, the search stops.

The choice of α is crucial because it determines the number of moves until the algorithm halts. In order to ensure that the available time is fully used, i.e. T^{min} is reached right at the timeout, we adjust α according to the number of moves applied per second. At a search speed of m moves per second, i iterations between successive cooling steps, current temperature T, T^{min} will be reached after $u = \frac{i}{m} \log_\alpha(\frac{T^{min}}{T^0})$ seconds. Conversely, for a given timeout of u seconds, we get the following for α:

$$\alpha = \left(\frac{T^{min}}{T^0}\right)^{\frac{i}{um}} \tag{1}$$

[2] This is exacerbated in TLSP, where soft constraint S1 (Number of jobs) trivially involves every single job in a soft constraint violation

Since search speed (the value of m) can vary between different instances, processors and also in the course of the search, we initially set α to 1, and periodically update it during the search, using the average measured search speed so far for m, the current temperature instead of T^0 and the remaining time for u.

We have also alternatively experimented with fixed cooling rates, where the search restarts when the minimum temperature is reached, either from a new initial solution or the best solution found so far (reheating). Neither of these options (at different cooling rates - and therefore different numbers of restarts or reheatings) improved upon the solution quality, with short cooling cycles producing infeasible solutions more often and worse penalties in most other cases.

Finally, a schedule for TLSP(-S) can contain both hard and soft constraint violations. This has to be taken into account when calculating a value for Δ. In our implementation, we have weighted each hard constraint violation by a factor w^H (chosen by the parameter tuner, see below).

6 Experimental results

For the experiments, a set of thirty instances of different sizes were chosen (15 each from the General and LabStructure sets described in Sect. 4.2 - one of each size, plus a second instance for the three smallest sizes), plus the three real-world instances available. The instances and some important properties are listed in Table 5.

Since we are solving TLSP-S, we use the task grouping provided in the reference solution (which is otherwise empty except for the assignments of started jobs).

The algorithms described in Section 5 were implemented in Java 8. All experiments were performed on a Lenovo ThinkPad University T480s with an Intel Core i7-8550U (1,8 GHz), using a single thread and a timeout of ten minutes.

6.1 Parameter configuration and tuning

For parameter tuning, we used SMAC3 (Hutter et al. (2011)), version 0.10.0. Tuning was performed on a set of 30 instances chosen in the same way as, but distinct from, the test data set. In each case, we allocated a budget of 500 target algorithm runs to SMAC.

6.1.1 Min-conflict

For the Min-Conflict heuristic, the restriction to a single job at each step means that both the Simple and the (larger) JobOpt neighborhoods can be explored in reasonable time. Over various solver runs, both neighborhoods achieved comparable results on the training set over several different configurations. For parameter tuning and the final evaluations, we decided to use the JobOpt neighborhood.

There are two construction heuristics that can be used to generate the initial solution. The first (*Greedy*) iterates over the jobs in order of ascending deadline and greedily assigns to each job the currently best values. The other (*Random*) uses random values for all assignments, respecting time windows, job precedence and resource availability constraints.

While MC itself does not include any parameters, there are still several possibilities for configuration. The parameters submitted to SMAC for tuning are listed on Table 6a and include the following: *Init* is the construction heuristic that should be used to build the

Table 5 The set of test instances used for the experiments. Shown are the data set the instance is taken from and the ID within that set. The following columns list the number of projects, jobs and the length of the scheduling period, followed by the number of employees, workbenches and equipment groups. The last columns contain the mean qualified employees and available workbenches per job, as well as the mean available devices per job and equipment group (only over those jobs that actually require at least one device of the group, about 10% of all jobs)

#	Data Set	ID	$\|P\|$	$\|J\|$	h	$\|E\|$	$\|B\|$	$\|G^*\|$	$\overline{\|E_j\|}$	$\overline{\|B_j\|}$	$\overline{\|G_{gj}\|}$
1	General	000	5	7	88	7	7	3	2.08	3.57	1.5
2	General	001	5	8	88	7	7	3	4.88	3.63	15.67
3	LabStructure	000	5	24	88	7	7	3	1.84	3.38	11.67
4	LabStructure	001	5	14	88	7	7	3	4.36	3.5	0.36
5	General	005	10	29	88	13	13	4	4.04	3.48	5.76
6	General	006	10	18	88	13	13	6	5.56	4.22	13.28
7	LabStructure	005	10	37	88	13	13	3	6.16	4.03	0.65
8	LabStructure	006	10	29	88	13	13	3	6.21	3.76	21.01
9	General	010	20	60	174	16	16	5	7.42	4.42	11.36
10	General	011	20	84	174	16	16	4	7.31	4.3	3.7
11	LabStructure	010	20	65	174	16	16	3	6.28	4.43	26.26
12	LabStructure	011	20	62	174	16	16	3	7.27	4.24	1.21
13	General	020	15	29	174	12	12	5	5.76	3.97	1.12
14	LabStructure	020	15	53	174	12	12	3	6.28	4.47	20.63
15	General	025	30	113	174	23	23	3	8.26	4.41	5.71
16	LabStructure	025	30	105	174	23	23	3	7.52	4.25	39.63
17	General	015	40	126	174	31	31	3	9.26	4.48	29.53
18	LabStructure	015	40	138	174	31	31	3	7.36	3.57	41.93
19	General	030	60	208	174	46	46	6	9.85	4.11	31.45
20	LabStructure	030	60	212	174	46	46	3	9.28	4.17	78.16
21	General	035	20	76	520	6	6	5	4.24	3.62	8.08
22	LabStructure	035	20	71	520	6	6	3	4.3	3.42	11.70
23	General	040	40	196	520	12	12	4	6.95	4.47	4.24
24	LabStructure	040	40	187	520	12	12	3	6.55	4.51	1.38
25	General	045	60	260	520	18	18	6	7.65	4.52	23.95
26	LabStructure	045	60	239	520	18	18	3	7.44	4.42	33.65
27	General	050	60	270	782	13	13	4	6.89	4.39	3.89
28	LabStructure	050	60	247	782	13	13	3	6.97	4.21	23.42
29	General	055	90	384	782	19	19	5	7.27	4.29	26.89
30	LabStructure	055	90	401	782	19	19	3	7.34	4.53	36.76

initial solution . Another parameter is the maximum number of moves without improvement (*MMWI*) before the search restarts from a new initial solution. Finally, p^{RW} denotes the probability at each step that a single move of random walk is performed instead of the min-conflict heuristic.

Table 6 Tuning parameters for Min-Conflict with Random Walk (a) and Simulated Annealing (b). The last column lists the parameter values for the best configuration found

Parameter	Value range	Best configuration
(a) Min-conflict + Random walk		
Init	{Random, Greedy}	Greedy
MMWI	{100 ... 10000}	131
p^{RW}	{0, 0.05, 0.1, 0.2}	0.1
(b)Simulated annealing		
T^0	{10 ... 100}	69
T^{min}	{0.001, 0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2, 5, 10}	0.1
w^H	{10, 20, 50, 100, 200}	10

6.1.2 Simulated annealing

The parameters to tune for SA are the initial temperature T^0, the minimum temperature T^{min} and the weight factor for hard constraint violations w^H, since the cooling rate is automatically determined from the available time and the number of moves applied per second. The parameters for SMAC and tuning results are listed in Table 6b.

As with the Min-Conflict heuristic, we saw comparable results for both the Simple and JobOpt neighborhoods and opted to continue the experiments with the JobOpt neighborhood for comparability.

6.2 Evaluation

Table 7 shows a comparison of results for Min-Conflict with Random Walk (MC+RW) and Simulated Annealing (SA), with the configuration described in the previous section. These results are compared to those of a Constraint Programming model (CP), written in MiniZinc (Nethercote et al. (2007)) and using the solver Chuffed (Chu (2011). Details for this CP model can be found in Geibinger et al. (2019).

Due to their non-deterministic behavior, both MC+RW and SA were run 10 times on each instance, with different seeds for the (pseudo-) random number generator. The table shows the best solution found over all runs, the number of feasible solutions found, and the average penalty among all feasible solutions.

From these results, it can be seen that instances can be split into two groups: For small instances with 20 projects or less (instances 1–14 and 21,22), good solutions could be found in most cases. This was much more difficult for larger instances with more than 20 projects.

MC+RW was unable to reliably find feasible solutions for most large instances within the given time. Moreover, even where it does find solutions without conflicts, the resulting penalty is often only slightly better than what was already achieved with the greedy construction heuristic.

SA performed much better and found feasible solutions in more than 97% of all runs. Also the solution quality is consistently better than with MC+RW, except for some small instances.

The CP model could find feasible solutions for all of the instances already within one minute. Within the full time limit, it could prove optimality for 12 of the 16 small instances.

Table 7 Comparison of results. Min-conflict with random walk and Simulated Annealing were both run 10 times each, with different seeds. Columns *Best* contain the best solution found, *#Feas.* the number of feasible solutions found (out of 10) and *Avg* the average penalty over all feasible solutions. Solutions for CP marked with "*" are optimal for the instance

#	MC+RW			SA			CP	
	Best	#Feas.	Avg	Best	#Feas.	Avg		
1	98	10	98	98	10	98	98	*
2	73	10	73	73	10	73	73	*
3	149	10	149.3	152	10	156.4	149	*
4	105	10	105.3	105	10	105	105	*
5	285	10	286.9	287	10	300.1	283	*
6	162	10	162.4	177	10	192.2	162	*
7	327	10	331.7	307	10	307.4	307	*
8	314	10	323.5	310	10	312	310	*
9	625	10	648.8	501	10	502.7	501	*
10	725	10	751	564	10	565.3	892	
11	993	10	1049.9	874	10	879	856	*
12	749	10	768.9	663	10	668	713	
13	340	10	341	352	10	352.1	340	*
14	450	10	457.9	422	10	425.7	420	*
15	1800	4	1841	1087	10	1090.6	1653	
16	1357	8	1429.8	1143	10	1155.2	1561	
17	1381	10	1410.9	1195	10	1234	1382	
18	1688	10	1760.7	1364	10	1375.3	1822	
19	2675	6	2735.3	2277	10	2337	2650	
20	2853	2	2898.5	2312	10	2360.6	2892	
21	908	10	1007.7	683	10	686.6	930	
22	1033	10	1079.7	767	10	771.9	839	
23	–	0	––	2393	6	2476.3	3531	
24	2484	2	2664	1808	10	1852.5	2454	
25	–	0	––	2908	8	3050.9	3281	
26	–	0	––	2724	10	2805	3899	
27	3372	3	3405.7	2176	10	2191.3	3146	
28	2585	10	2617.2	2367	10	2375.8	2569	
29	5334	2	5406.5	4208	9	4428.4	4548	
30	6453	10	6646.8	4828	9	4896.8	5905	

Compared to the results for SA, CP slightly outperforms SA on those instances where it could find optimal solutions. However, SA produced better results for every single other instance, sometimes by more than 30%. In particular, SA decisively outperformed CP on the large instances, i.e. those with more than 20 projects. These results can also be seen in Fig. 1.

The differences between the results for small and large instances indicates that the number of projects (and thus jobs) is the main factor in determining the time required to solve an instance. In contrast, neither the number of time slots in the scheduling period, nor the number of resources available for each job seem to have a decisive impact on the difficulty of an instance.

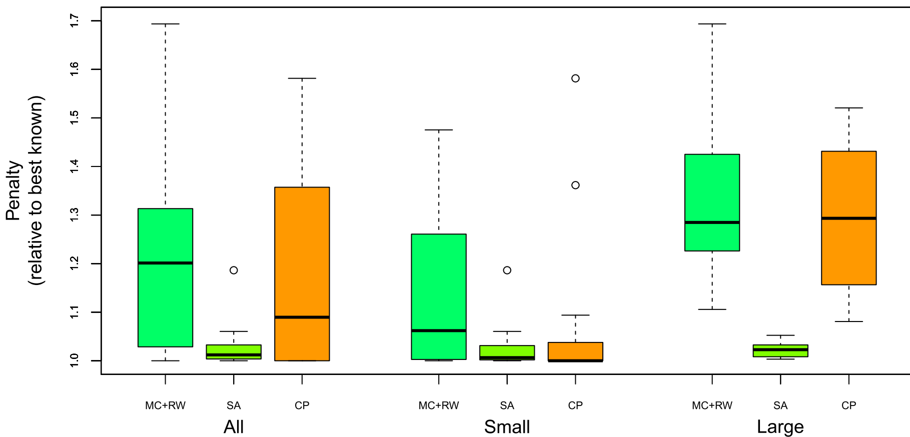

Fig. 1 Results for Min-Conflict with Random Walk (MC+RW), Simulated Annealing (SA) and Constraint Programming (CP), with a timeout of 10 minutes per instance. The center and right groups show results only for small ($\leq$ 20 projects) and large instances, respectively. Results were scaled by the best solution known for each instance

6.2.1 Additional runtime

We also repeated our experiments with the same configuration for Simulated Annealing with a longer timeout of one hour. Table 8 shows the results of these experiments, again compared with the results for the CP model (also with a timeout of one hour).

With the increased time budget, feasible solutions could be found for all instances. As with the shorter time limit, the solutions produced by SA for all instances where CP could not find optimal solutions are better than those found by CP, in some cases by more than 30%. In particular, this includes all large instances.

For those small instances, where optimal solutions have been found, SA achieved solutions with the same or very close penalties.

Compared to the results with a shorter time limit, the penalty for large instances (>20 projects) has improved by nearly 3.8% on average with SA. CP could find optimal solutions for three of the four remaining small instances. However, the results for large instances improved by less than 1% on average.

Figure 2 shows a comparison of the performance of the two solvers, both overall and separated into small and large instances. With the increased running time, CP finds the best known (indeed optimal) solution for all but one of the small instances, but is still outperformed by SA both on the larger instances and in overall quality.

Our results are in line with previous findings for RCPSP (e.g. by Pellerin et al. (2020), for a recent example) that exact approaches are suitable mostly for instances with fewer activities and that heuristics are needed to find good solutions for instances of larger sizes. A possible explanation for this may be that exact approaches in general have to examine a substantial part of the search space, which grows exponentially with the number of activities, while heuristics can often find more efficient paths through that search space, at the cost of losing any guarantees about the final solution reached.

Table 8 Comparison of results with a timeout of one hour. Simulated Annealing was run 5 times per instance with different seeds. Column *Best* contains the best solution found, *#Feas.* the number of feasible solutions found (out of 5) and *Avg* the average penalty over all feasible solutions. Solutions for CP marked with "*" are optimal for the instance

#	SA			CP	
	Best	#Feas.	Avg		
1	98	5	98	98	*
2	73	5	73	73	*
3	151	5	152.4	149	*
4	105	5	105.4	105	*
5	292	5	300.8	283	*
6	180	5	191	162	*
7	307	5	307	307	*
8	310	5	310.2	310	*
9	501	5	501	501	*
10	564	5	564.6	740	
11	872	5	873.4	856	*
12	663	5	664.8	656	*
13	352	5	353.6	340	*
14	422	5	422.2	420	*
15	1086	5	1086.8	1647	
16	1141	5	1142.2	1561	
17	1195	5	1206	1284	
18	1360	5	1361	1820	
19	2196	5	2233	2650	
20	2261	5	2284.6	2888	
21	683	5	683.6	679	*
22	765	5	766.6	765	*
23	2200	5	2276	3487	
24	1782	5	1799.4	2452	
25	2598	5	2737.8	3278	
26	2605	5	2633.4	3894	
27	2155	5	2159.8	3130	
28	2333	5	2341.2	2569	
29	4038	5	4204.8	4539	
30	4601	5	4636.6	5904	

7 Conclusions

In this article, we have introduced the new problem TLSP, and its subproblem TLSP-S, which are complex extensions to existing RCPSP variants based on real-world requirements. A complexity analysis via reduction from RCPSP shows that even finding a feasible solution for either of these two problems is already NP-hard. Our findings and comparison to related variants of RCPSP indicate that TLSP covers several aspects important in actual practice that cannot be efficiently modeled by existing formulations.

Our publicly available instance sets, first described in this paper, contain both instances randomly generated based on real-world requirements, using our configurable instance generator, and real-world instances. Our experimental evaluations indicate that these instances

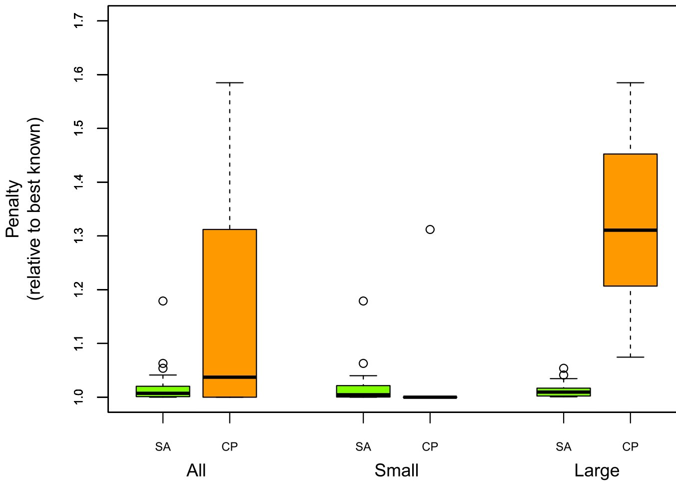

Fig. 2 Results for Simulated Annealing (SA) and Constraint Programming (CP), with a timeout of one hour per instance. The center and right groups show results for small ($\leq$ 20 projects) and large instances, respectively. Results were scaled by the best solution known for each instance

provide a challenging benchmark data set which can be further used by other researchers to work on this problem.

In addition, we have introduced a solver framework for solving these problems, which supports several metaheuristic solvers and provides multiple options for configuration and extensions. Using this framework, we have shown that Simulated Annealing using a suitable set of problem-specific neighborhoods can be used to provide high quality solutions for TLSP-S, and outperforms a state-of-the-art method based on Constraint Programming for larger and practical instances both under strict time limits and with longer runtimes. Our results also further support the general finding that for most scheduling problems based on RCPSP, exact methods work well only for smaller instances, and heuristics are necessary to solve instances of realistic sizes.

The methods proposed in this work are currently in use in the laboratory of our industrial partner, who successfully generate schedules for their long-term planning.

Regarding future work, we plan to investigate whether these results can also be transferred to the TLSP, which combines TLSP-S with an additional grouping stage. A promising direction of research also seems the combination of both local search and CP-based approaches, in the form of hybrid algorithms or large neighborhood search, to combine the advantages of both methods and further improve the results for TLSP-S.

Acknowledgements Open access funding provided by TU Wien (TUW). The financial support by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development is gratefully acknowledged.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory

regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

Ahmeti, A., & Musliu, N. (2018). Min-conflicts heuristic for multi-mode resource-constrained projects scheduling. In: Proceedings of the Genetic and Evolutionary Computation Conference, ACM, pp 237–244

Asta, S., Karapetyan, D., Kheiri, A., Özcan, E., & Parkes, A. J. (2016). Combining monte-carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. *Information Sciences, 373,* 476–498.

Bartels, J. H., & Zimmermann, J. (2009). Scheduling tests in automotive r&d projects. *European Journal of Operational Research, 193*(3), 805–819. https://doi.org/10.1016/j.ejor.2007.11.010.

Bellenguez, O., & Néron, E. (2005). Lower bounds for the multi-skill project scheduling problem with hierarchical levels of skills. In M. Trick & E. Burke (Eds.), *Practice and Theory of Automated Timetabling V* (pp. 229–243). Berlin: Springer Berlin Heidelberg.

Blazewicz, J., Lenstra, J. K., & Kan, A. R. (1983). Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics, 5*(1), 11–24.

Bouleimen K, Lecocq H (2003) A new efficient simulated annealing algorithm for the resource-constrained project scheduling problem and its multiple mode version. European Journal of Operational Research 149(2):268 – 281, https://doi.org/10.1016/S0377-2217(02)00761-0, http://www.sciencedirect.com/science/article/pii/S0377221702007610, sequencing and Scheduling

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

Chu G (2011) Improving combinatorial optimization. PhD thesis, University of Melbourne, Australia, http://hdl.handle.net/11343/36679

Dauzère-Pérès, S., Roux, W., & Lasserre, J. (1998). Multi-resource shop scheduling with resource flexibility. *European Journal of Operational Research, 107*(2), 289–305.

Drexl, A., Nissen, R., Patterson, J. H., & Salewski, F. (2000). Progen/πx -an instance generator for resource-constrained project scheduling problems with partially renewable resources and further extensions. *European Journal of Operational Research, 125*(1), 59–72.

Elmaghraby, S. E. (1977). *Activity networks: Project planning and control by network models*. New Jersey: Wiley.

Geibinger T, Mischek F, Musliu N (2019) Investigating constraint programming for real world industrial test laboratory scheduling. In: Proceedings of the Sixteenth International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2019)

Gonçalves, J., Mendes, J., & Resende, M. (2008). A genetic algorithm for the resource constrained multi-project scheduling problem. *European Journal of Operational Research, 189*(3), 1171–1190.

Hartmann, S. (1998). A competitive genetic algorithm for resource-constrained project scheduling. *Naval Research Logistics (NRL), 45*(7), 733–750.

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

Hutter F, Hoos HH, Leyton-Brown K (2011) Sequential model-based optimization for general algorithm configuration. In: International Conference on Learning and Intelligent Optimization, Springer, pp 507–523

Kirkpatrick, S., Gelatt, C. D., & Vecchi, M. P. (1983). Optimization by simulated annealing. *Science, 220*(4598), 671–680.

Laurent, A., Deroussi, L., Grangeon, N., & Norre, S. (2017). A new extension of the rcpsp in a multi-site context: Mathematical model and metaheuristics. *Computers and Industrial Engineering, 112,* 634–644.

Mika, M., Waligóra, G., & Węglarz, J. (2006). *Modelling setup times in project scheduling. Perspectives in modern project scheduling* (pp. 131–163). Boston: Springer.

Mika, M., Waligóra, G., & Węglarz, J. (2015). Overview and state of the art. In C. Schwindt & J. Zimmermann (Eds.), *Handbook on Project Management and Scheduling* (Vol. 1, pp. 445–490). Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-05443-8_21.

Minton, S., Johnston, M. D., Philips, A. B., & Laird, P. (1992). Minimizing conflicts: A heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence, 58,* 161–205.

Mischek F, Musliu N (2018) The test laboratory scheduling problem. Technical report, Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling, TU Wien, CD-TR 2018/1

Nethercote N, Stuckey PJ, Becket R, Brand S, Duck GJ, Tack G (2007) Minizinc: Towards a standard CP modelling language. In: Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings, pp 529–543, https://doi.org/10.1007/978-3-540-74970-7_38

Pellerin, R., Perrier, N., & Berthaut, F. (2020). A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research, 280*(2), 395–416.

Polo Mejia O, Anselmet MC, Artigues C, Lopez P (2017) A new RCPSP variant for scheduling research activities in a nuclear laboratory. In: 47th International Conference on Computers & Industrial Engineering (CIE47), Lisbonne, Portugal, p 8p., https://hal.laas.fr/hal-01630977

Potts, C. N., & Kovalyov, M. Y. (2000). Scheduling with batching: A review. *European Journal of Operational Research, 120*(2), 228–249.

Salewski, F., Schirmer, A., & Drexl, A. (1997). Project scheduling under resource and mode identity constraints: Model, complexity, methods, and application. *European Journal of Operational Research, 102*(1), 88–110.

Schwindt, C., & Trautmann, N. (2000). Batch scheduling in process industries: An application of resource-constrained project scheduling. *OR-Spektrum, 22*(4), 501–524.

Szeredi R, Schutt A (2016) Modelling and solving multi-mode resource-constrained project scheduling. In: Principles and Practice of Constraint Programming - 22nd International Conference, CP 2016, Toulouse, France, September 5-9, 2016, Proceedings, pp 483–492, https://doi.org/10.1007/978-3-319-44953-1_31

Trautmann, N., & Schwindt, C. (2005). *A minlp/rcpsp decomposition approach for the short-term planning of batch production. In: computer aided chemical engineering* (Vol. 20, pp. 1309–1314). Amsterdam: Elsevier.

Villafáñez, F., Poza, D., López-Paredes, A., Pajares, J., & del Olmo, R. (2019). A generic heuristic for multi-project scheduling problems with global and local resource constraints (rcmpsp). *Soft Computing, 23*(10), 3465–3479.

Wauters, T., Kinable, J., Smet, P., Vancroonenburg, W., Vanden Berghe, G., & Verstichel, J. (2016). The multi-mode resource-constrained multi-project scheduling problem. *Journal of Scheduling, 19*(3), 271–283.

Węglarz, J., Józefowska, J., Mika, M., & Waligóra, G. (2011). Project scheduling with finite or infinite number of activity processing modes - a survey. *European Journal of Operational Research, 208*(3), 177–205. https://doi.org/10.1016/j.ejor.2010.03.037.

Wilson M, Witteveen C, Huisman B (2012) Enhancing predictability of schedules by task grouping. In: BNAIC 2012: 24th Benelux Conference on Artificial Intelligence, Maastricht, The Netherlands, 25-26 October 2012, Citeseer

Young, K. D., Feydy, T., & Schutt, A. (2017). Constraint programming applied to the multi-skill project scheduling problem. In J. C. Beck (Ed.), *Principles and Practice of Constraint Programming* (pp. 308–317). Cham: Springer International Publishing.

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.