Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS 2020)

Solving the Test Laboratory Scheduling Problem with Variable Task Grouping

Philipp Danzinger, Tobias Geibinger, Florian Mischek, Nysret Musliu
Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling
DBAI, TU Wien, Karlsplatz 13, 1040 Vienna, Austria
{pdanzing, tgeibing, fmischek, musliu}@dbai.tuwien.ac.at

Abstract

The Test Laboratory Scheduling Problem (TLSP) is an extension of the Resource-Constrained Project Scheduling Problem (RCPSP). Besides several additional constraints, it includes a grouping phase where the jobs to be scheduled have to be assembled from smaller tasks and derive their properties from this grouping. Previous solution approaches for TLSP have focused primarily on the scheduling subproblem (TLSP-S), for which it is assumed that a suitable grouping is already given as part of the input. In this paper, we provide for the first time a solution approach that encompasses the full problem including grouping. We propose both a Constraint Programming model for TLSP and a Very Large Neighborhood Search algorithm based on that model. Furthermore, we apply our algorithms to real-world instances as well as randomly generated ones and compare our results to the best existing solutions. Experimental results show that our solution methods consistently outperform those for TLSP-S when both are initialised with a good grouping and in many cases even when this grouping is provided only to the latter.

Introduction

Project scheduling problems appear in many different settings where activities have to performed using limited resources. This includes, but is not limited to, factories and other manufacturing processes as well as project management applications. Solving these problems manually usually requires expert knowledge, large amounts of time and is prone to potentially expensive errors and suboptimal solutions.

The Test Laboratory Scheduling Problem (TLSP) is one such problem that arises in an industrial test laboratory, where a large number of tests have to be performed by qualified employees, using specialised equipment. At the same time, several constraints such as release dates, deadlines and precedences between tests have to be considered. TLSP is an extension of RCPSP. It introduces several new constraints compared to RCPSP, including general availability constraints on resources, which limit which resources can be assigned to individual activities.

Most importantly, the jobs to be scheduled in TLSP are not indivisible atoms in the schedule, but actually composed of smaller components, called tasks. The properties of each job (such as duration, time windows and resource requirements) are completely determined by the tasks it contains. Solvers need to both find such a grouping of tasks into jobs and assign time slots and resources to each job.

Existing solution approaches to TLSP only deal with a subproblem focusing on the scheduling part, which assumes that a feasible grouping of tasks into jobs is already provided (TLSP-S). This grouping is then taken as-is and not modified during the search process: In (Mischek and Musliu 2019), TLSP-S is solved using metaheuristics, whereas in (Geibinger, Mischek, and Musliu 2019b; 2019a), a CP model and a Very Large Neighborhood Search (VLNS) are proposed. These solution approaches are limited to scenarios where such an initial grouping is known or can be easily generated. For the more general case, grouping has to be considered as part of the solution process. In order to deal with this issue, we developed an innovative constraint programming (CP) model for TLSP that is able to handle a dynamic number of jobs. An optimization we included in this model is also applicable for the CP model for TLSP-S described in (Geibinger, Mischek, and Musliu 2019a) and we show that it further improves the performance of that model.

The main contributions of this paper are:

- We provide for the first time a CP model for the full TLSP.
- We extend the VLNS used in (Geibinger, Mischek, and Musliu 2019a) making it suitable as a solver for TLSP.
- We show that our approaches achieve very good results and outperform previous methods even compared to TLSP-S solvers that start out from a good initial grouping.

Our solution approaches are currently employed successfully for scheduling in an industrial test laboratory.

This paper is structured as follows: The next section contains a review of related literature, followed by a section containing a formal description of TLSP. In the main part of the paper, we describe our CP model and our VLNS algorithm in detail. We provide experimental results and a short discussion in the section after that. Finally, the last section contains our conclusions and an outline of future work.

Literature Overview

Project scheduling problems have been investigated extensively in the literature. The most studied variants of these problems include the Resource-Constrained Project Scheduling Problem (RCPSP) (Brucker et al. 1999; Hartmann and Briskorn 2010; Mika, Waligóra, and Węglarz 2015) and its Multi-Mode version (MRCPSP) (Elmaghraby 1977; Węglarz et al. 2011; Hartmann and Briskorn 2010; Szeredi and Schutt 2016). Of particular relevance for TLSP(-S) is the Multi-Skill RCPSP (MSPSP) (Bellenguez and Néron 2005; Young, Feydy, and Schutt 2017), which features similar resource availability constraints.

TLSP-S was investigated by (Mischek and Musliu 2019), (Geibinger, Mischek, and Musliu 2019b) and (Geibinger, Mischek, and Musliu 2019a). It arises in a real-life situation and included several extensions compared to other project scheduling problems. TLSP generalizes TLSP-S, in which a feasible grouping of tasks into jobs is already provided. Solving of both task grouping and TLSP-S simultaneously increases drastically the size and the search space. Our solution techniques for TLSP proposed in this paper are based on some initial ideas from (Geibinger, Mischek, and Musliu 2019a), but they include significant extensions regarding the constraint programming model and the large neighborhood search algorithm.

Aspects similar to the grouping mechanism in TLSP appear in other works in the form of batching (e.g. (Schwindt and Trautmann 2000; Potts and Kovalyov 2000)) or schedule-dependent setup times (e.g. (Mika, Waligóra, and Węglarz 2006; 2008)), although they are typically handled implicitly, i.e. the batches arise from the finished schedule, instead of the other way round.

Problem Description

This section formally describes TLSP which was introduced in (Mischek and Musliu 2018). We mostly follow their notation, albeit with some slight changes.

Input Parameters

Each instance encompasses a laboratory *environment*, containing the number of time slots available as well as the available resources, a list of *projects* containing tasks to schedule and their requirements, and finally optionally information about an *existing schedule*.

Environment The planning horizon consists of discrete *time slots* $t \in T = \{1, \ldots, |T|\}$. The laboratory environment provides different kinds of resources used to perform tasks:

- *Employees* $e \in E = \{1, \ldots, |E|\}$ who are able to perform some tasks.
- *Workbenches* $b \in B = \{1, \ldots, |B|\}$ which tasks may be performed on.
- *Equipment groups* $G_g = \{1, \ldots, |G_g|\}$ where g is the group's index. Each group represents similar individual devices $e \in G_g$. The set of all equipment groups is G^*.

Finally, tasks must be performed in one of several modes $m \in M = \{1, \ldots, |M|\}$. The assigned mode determines the number of required employees, given by e_m, as well a *speed factor* v_m by which the duration of the task is multiplied.

Projects and Tasks The tasks to be performed are partitioned into *projects* $p \in P = \{1, \ldots, |P|\}$. Individual tasks for a project p are denoted by $a \in A_p$. We also use p_a to refer to the project that task a belongs to. The set of all tasks is denoted by $A^* = \bigcup_{p \in P} A_p$. Each task a has the following properties:

- It has a *release date* α_a and a *due date* $\bar{\omega}_a$ as well as a *deadline* ω_a. Violating the deadline is forbidden, while violating the due date only results in a penalty.
- Each task has a set of *available modes* $M_a \subseteq M$.
- The real-valued *duration* of the task is denoted by d_a and measured in time slots. When tasks are scheduled, this duration must be multiplied by the speed factor v_m of the mode the task is performed in. The duration for task a performed in mode m then becomes $d_{am} := d_a \cdot v_m$.
- Tasks may need to be performed on a workbench, which is indicated by $b_a \in \{0, 1\}$. If a workbench is required, the assigned workbench must be part of the *available workbenches* $B_a \subseteq B$.
- Similarly, employees assigned to a task must be chosen from its set of *qualified employees* denoted by $E_a \subseteq E$. The number of required employees solely depends on the assigned mode. Additionally, the set $E_a^{Pr} \subseteq E_a$ is the set of *preferred employees*.
- From each equipment group $g \in G^*$, a task requires r_{ag} devices, which must be taken from the set of *available devices* $G_{ag} \subseteq G_g$.
- Each task may also have *predecessor tasks* $\mathcal{P}_a \subseteq A_{p_a}$, which must be completed before a starts. Predecessors must belong to the same project.

Each project's tasks are further partitioned into *families*, where f_a denotes the family of task a and $F_f \subseteq A^*$ are the tasks contained in family f. In addition to the syntax from (Mischek and Musliu 2018), we also use F^* to refer to the set of all families. Jobs must be formed from tasks of the same family only.

Each family f further has a *setup time* s_f. The setup time gets added to the duration of each job containing tasks from family f alongside the tasks' durations themselves. When a setup time is added to a job, it gets scaled with the speed factor of the job's assigned mode, just like task durations. Hence, $s_{fm} = s_f \cdot v_m$ denotes the setup time for jobs formed from tasks from family f and performed in mode m.

Finally, sometimes it is required that different tasks are performed by exactly the same employees. This is ensured by specifying *linked tasks*. The linked tasks of project p are described by the equivalence relation $L_p \subseteq A_p \times A_p$, where two tasks a and b, both from project p, are linked if and only if $(a, b) \in L_p$.

Existing Schedule All instances specify a base schedule, containing fixed assignments that may not be changed.[1]

[1]In some scenarios, the base schedule may also contain other assignments that are not fixed, but we do not cover this aspect in

Compared to the original definition, we omit the fixed time slot, mode and resource assignments given in the base schedule of the instance. This makes it possible to specify the subsequent CP model more succinctly without sacrificing much flexibility, since the possible assignments can also be restricted by narrowing down task properties such as release time, and available resources in a preprocessing step.

Thus the base schedule is given by the set J^0 of *base jobs*, where each job $j \in J^0$ contains a set of fixed tasks $\dot{A}_j^F$ that must appear together in a single job in the solution. Since only tasks of the same family can be combined into a job, also all tasks in $\dot{A}_j^F$ must belong to a single family.

A subset of the base jobs J^0 are the *started jobs* $J^{0S} \subseteq J^0$. Any job in the solution containing at least one task of a started job must start at time slot 1 and has no setup time added to its duration. The reason for this is that it is assumed those tasks are already being worked on.

Jobs and Grouping

Before any time slots or resources are assigned, *tasks* must first be grouped into larger units called *jobs*. Among other things, this helps reuse test setups (as modelled by the setup time) and reduces the rounding error when converting durations to full time slots. In addition, it reduces the operational and mental overhead for employees as well as the schedule's complexity for human planners.

Each job may only contain tasks from the same task family, and by extension from the same project. Jobs have similar properties to tasks, which are computed from the tasks that make up the job. Within a job, tasks are executed sequentially, but without a defined order. For this reason, a job's assignment must fulfill all of its tasks' requirements for its full duration. For example, a job's start time must be greater or equal to all its tasks' release times, and its end time smaller or equal to all its tasks' deadlines. A job must only be assigned modes and resources available to all of its tasks and must be assigned exactly enough resources to cover its most demanding task. Likewise, the set of preferred employees is equal to the intersection of all its tasks' preferred employees. Finally, tasks between which a precedence relation exists must either be part of the same job or their jobs must obey an equivalent precedence relation in the final schedule.

The duration of a job j is calculated by taking the sum of the durations d_{am} of its tasks, under the mode m assigned to j. Assuming j does not contain any started tasks, the setup time s_{fm} of the family f containing the tasks in j is added, otherwise it is assumed that the setup is already complete and no setup time is added. The final duration is then obtained by rounding up that sum to the next full time slot.

The choice of this formulation, which deliberately overconstrains schedules, was made due to a combination of several operational aspects present in the laboratory of our industrial partner.

- Working hours for employees are quite flexible, which makes schedules with a finer granularity than half a day difficult to implement in practice (also with regards to labor law). Since many tasks have a duration smaller than this, any formulation directly scheduling tasks must necessarily incur substantial overheads when rounding those tasks to full time slots.
- Tasks within a family usually share many properties, such as resource requirements and availabilities. This lessens the impact of the overconstrained nature of a job's properties. Also in difference to RCPSP, precedence constraints play a rather small role in typical TLSP instances, with only a few percent of tasks having prerequisites, and many of these are shared by all tasks of a family.
- In practice, there are often last-minute changes in the order of performed tasks due to minor delays, unclear specifications, or other external factors. This formulation guarantees that at least within a job, such changes can always be performed without conflicts.

this paper and ignore all such assignments where present.

Solution Description

A solution to a TLSP instance consists of a set of jobs which encompasses all tasks, as well as assignments to those jobs to satisfy all requirements. In particular, each job must be assigned a start time slot and completion time, the mode in which the job should be executed, the workbench on which the job is performed (if required), a set of employees, and the required number of devices for each equipment group g.

For a more formal treatment of how a job's properties are calculated as well as a comprehensive list of all constraints present in TLSP we refer to (Mischek and Musliu 2018).

Constraint Programming Model

In this section, we propose a CP model for the full TLSP. Our implementation is written in the solver-independent modeling language MiniZinc (Nethercote et al. 2007).

One major challenge in creating a model for the full TLSP was finding an efficient representation for the task groupings. Such a representation should not only be symmetry-free, but it must also allow to schedule and assign resources to a varying number of jobs whose requirements may change depending on the tasks assigned to them. The approach used in our model is to treat each task as a potential job. Each job is identified by a representative task, which is used to assign time slots and resources to jobs. To this end, we introduce an array of decision variables $\xi(a)$ that assign to each task the representative task of the job it belongs to. To break the symmetry in choosing the representative tasks, we assign each task an (arbitrary) id such that the set of tasks is well-ordered and require the representative task of each job to be its task with the least id.

For convenience, we define the set J of tasks that act as representatives for a job: $J = \{a \in A^* \mid \xi(a) = a\}$. Since the elements of this set depend on the chosen task grouping, they are, of course, decided on by the solver during runtime.

As mentioned previously, the model must be able to calculate job durations ad-hoc based on task durations, which are real-valued fractions of time slots. Because support for float variables is limited across different MiniZinc solvers, we opted to approximate this calculation using integers. To that end, all durations and setup times are scaled up by a factor $\mathcal{M}$ and then rounded up to the next integer during pre-processing. Because time requirements are always over-estimated, this transformation does not lead to any invalid schedules, but it may make some valid ones appear to be infeasible. The impact of this rounding can, of course, be lessened by increasing $\mathcal{M}$. In our experiments, the choice of $\mathcal{M}$ (between 100 and 10000) did not affect the quality of the produced solutions, but values above 1000 led to a drastic increase in memory usage. For this reason, we decided to use $\mathcal{M} = 1000$ for our final evaluations.

The variables s_a and n_a assign start and end times to jobs, respectively. They are set to valid time slots for all tasks $a \in J$, and set to 0 for all other tasks. Together, they functionally define the duration $d_a = n_a - s_a$.

In the same manner as before, m_a assigns a mode to each job. Resource assignments are described as follows: The variable a^{Em}_{ea} is set to 1 if employee e is assigned to job a and 0 otherwise, the variable a^{Wb}_{ba} is set to 1 if a is performed on workbench b and 0 otherwise, and finally the variable a^{Eq}_{ea} is set to 1 if a uses device e, and 0 otherwise. Similar to time slots, all resource assignnments are set to 0 for all $a \notin J$.

Hard Constraints

$$\xi(\xi(a)) = \xi(a) \qquad a \in A^* \tag{1}$$

$$p_a = p_{\xi(a)} \wedge f_a = f_{\xi(a)} \qquad a \in A^* \tag{2}$$

$$\texttt{all\_equal}(\{\xi(a) \mid a \in \dot{A}^F_j\}) \qquad j \in J^0 \tag{3}$$

$$\xi(a) \le a \qquad a \in A^* \tag{4}$$

Constraints (1–3) ensure that $\xi(a)$ describes a valid grouping of tasks. (1) enforces that representative tasks point at themselves, (2) ensures that only tasks from the same project and family can be combined, and (3) ensures that tasks inside a fixed set are assigned to the same job. Finally, (4) serves symmetry-breaking purposes, enforcing that the task with the least id from each job are chosen as the representative.

$$s_{\xi(a)} \ge \alpha_a \wedge n_{\xi(a)} \le \omega_a \qquad a \in A^* \tag{5}$$

$$d_a = n_a - s_a \qquad a \in J \tag{6}$$

Constraint (5) ensures that each task's release time and deadline are compatible with its job's assigned start and end times. Equation (6) defines a job's assigned duration to be the difference between its assigned end and start time.

$$d_a \cdot \mathcal{M} \ge \sum_{\substack{a' \in A^* \\ \text{s.t. } \xi(a')=a}} d_{a' m_a} + st(a) \qquad a \in J \tag{7}$$

$$(d_a - 1) \cdot \mathcal{M} < \sum_{\substack{a' \in A^* \\ \text{s.t. } \xi(a')=a}} d_{a' m_a} + st(a) \qquad a \in J \tag{8}$$

Constraints (7) and (8) calculate, scale down and round up the job durations. Each job's duration is the sum of the durations of its assigned tasks, plus its setup time $st(a)$, which is 0 for jobs containing a started task and $s_{f_a m_a}$ otherwise.

$$\xi(a) = \xi(a') \vee n_{\xi(a')} \le s_{\xi(a)} \qquad a \in A^*, a' \in \mathcal{P}_a \tag{9}$$

$$s_{\xi(a)} = 1 \qquad j \in J^{0S}, a \in \dot{A}_j \tag{10}$$

Constraint (9) ensures that prerequisite tasks are either part of the same job, or are completed before the job containing their successor is started. Constraint (10) enforces that jobs containing started tasks are assigned the start time 1.

$$\texttt{cumulative}((s_a)_{a \in A^*}, (d_{m_a a})_{a \in A^*}, (a^{Em}_{ea})_{a \in A^*}, 1) \qquad e \in E \tag{11}$$

Constraint (11) models the unary resource constraints for employees, ensuring that employees are not used by multiple jobs at the same time. The constraints for workbenches and devices (for each equipment group) are modeled in the same way.

$$e_{m_a} = \sum_{e \in E} a^{Em}_{ea} \qquad a \in J \tag{12}$$

$$b_a = \sum_{b \in B} a^{Wb}_{b\xi(a)} \qquad a \in A^* \tag{13}$$

$$\max_{\substack{a' \in A^* \\ \text{s.t. } \xi(a')=a}} r_{a'g} = \sum_{e \in G_g} a^{Eq}_{ea} \qquad a \in J, g \in G^* \tag{14}$$

Although constraints (12–14) look very different at first glance, they serve a similar purpose in making sure that jobs are assigned the correct number of employees, workbenches and equipment, respectively. The employee constraint is easiest to model because the required number of employees is only dependent on the mode assigned to the job. Workbenches are still straight-forward because at most one of them can be assigned to each job. Constraining the assigned equipment is most complicated because there is no fixed upper bound like for workbenches: instead, the constraint needs to compute the exact maximum equipment requirements over all tasks assigned to the job.

$$a^{Em}_{e\xi(a)} = 1 \rightarrow e \in E_a \qquad a \in A^*, e \in E \tag{15}$$

$$m_{\xi(t)} \in M_a \qquad a \in A^* \tag{16}$$

$$a^{Em}_{e\xi(a)} = a^{Em}_{e\xi(a')} \qquad e \in E, p \in P, (a, a') \in L_p \tag{17}$$

Constraints (15–16) ensure that all employees and modes assigned to a job are available to all of its tasks. Availability constraints for equipment and workbenches are modeled analogously. (17) enforces that linked tasks, or more precisely their jobs, must be assigned the same employees.

Soft Constraints

TLSP contains several soft constraints according to (Mischek and Musliu 2018).

As MiniZinc has no direct support for soft constraints, they are defined as a function that should be minimized. The minimization target for this model is the weighted sum of several soft constraints s_1 through s_5. For the purposes of the benchmarks presented here, all weights $w_i (1 \le i \le 5)$ are set to 1. We realise that setting all weights to 1 is not ideal and our industrial partner is currently tasked with tuning the those values.

The individual soft constraints are formulated as follows:

$$s_1 = w_1 \cdot \sum_{j \in J} 1 \tag{18}$$

$$s_2 = w_2 \cdot \sum_{j \in J} \sum_{e \in (E \setminus E_j^{Pr})} a_{ej}^{Em} \tag{19}$$

$$s_3 = w_3 \cdot \sum_{p \in P} \sum_{e \in E} ((\sum_{a \in A_p} a_{ea}^{Em}) > 0) \tag{20}$$

$$s_4 = w_4 \cdot \sum_{j \in J} \max(0, n_j - \min_{a \in A^* \text{ s.t. } \xi(a)=j} (\bar{\omega}_a)) \tag{21}$$

$$s_5 = w_5 \cdot \sum_{p \in P} (\max_{a \in A_p}(n_a) - \min_{a \in A_p \text{ s.t. } \xi(a)=a} (s_a)) \tag{22}$$

First, we want to minimize the number of jobs with (18). Next, (19) penalizes employee assignments that deviate from the preferred employees, where E_j^{Pr} is the set of preferred employees of job j. (20) minimizes the number of different employees assigned to each project. Further, due date violations are penalized with (21) and finally, the project durations should be minimized with (22). The minimization objective is simply the sum $\sum_{1 \le i \le 5} s_i$.

Optimizing for Identical Resources

One major factor that sets TLSP apart from RCPSP is how resource requirements are described. In RCPSP each activity can require some quantity of each resource and resources themselves are limited but replenishable. TLSP further differentiates between individual units of each resource. A task's requirements may require assigning only specific individual resources. As a result, handling resource units individually increases the size of TLSP models significantly and slows down search. Equipment is particularly problematic here due to the additional breakdown into different equipment groups.

Fortunately, there is a way to alleviate this: One pattern present in real-world data as well as the test instances is that some equipment units are completely interchangeable. In a sense, this means they behave similarly to classical RCPSP resources. This is the case for two units of equipment e_1 and e_2 if they both belong to group g and are available to exactly the same tasks.

To exploit the symmetry introduced by this equivalence relation, we developed a problem transformation. The individual pieces of equipment in the input are replaced by equipment (equivalence) classes. Then, instead of assigning to a job individual pieces of equipment using binary decision variables, the model uses integer variables to decide how many members of each equipment class are assigned.

Formally, this means replacing G_g by new sets C_g, adding corresponding sets of available devices $C_{ag} \subseteq C_g$ to tasks and introducing an array q_c to store the quantity of pieces in each equipment class, such that for each device, a class exists that is available to exactly the same tasks, and vice versa. Furthermore, we ensure that all equipment classes from the same group are different regarding task availabilities and that the quantity q_c of an equipment class c is equal to the number of devices available to the same tasks. Constraints (11), (14) and (15) need to be adapted accordingly.

Applying this transformation improved the efficiency of the CP model in all domains, including compile time, run time and memory usage. At the same time, we saw no performance slowdowns even with an artificial instance that only contains distinct equipment. We also applied this optimisation to the existing CP model for TLSP-S (Geibinger, Mischek, and Musliu 2019a) and found similar improvements.

Other Optimizations

There were two smaller optimizations that provided a significant speed-up.

$$\xi(\min_{a \in F_f} a) = \min_{a \in F_f} a \qquad f \in F^* \tag{23}$$

First, the redundant constraint (23) explicitly states that the smallest task of each family must be a representative task and therefore represent a job. Even though this easily follows from hard constraints (1) and (2), the constraint provided a large improvement to compile and search times, reducing total run times by more than a third.

The second optimization regards the formulation of the soft constraints. Adding decision variables for each project's violations of each soft constraint reduced the run time of the search significantly. Additionally, introducing very primitive bounds that can be resolved when the model is compiled had further positive effect. As an example for such a bound, each project's duration (soft constraint (22)) must be at least as long as its longest task plus (except for projects containing started tasks) the smallest possible setup time of its family. An obvious upper bound is the interval between the smallest release date and the largest deadline.

Unfortunately, using redundant global `cumulative` and `global_cardinality` constraints as in (Geibinger, Mischek, and Musliu 2019a) did not translate well to TLSP. In the first case, an efficient formulation is not possible, because it would require prior knowledge about the resource requirements of jobs. On the other hand, `global_cardinality` constraints can be formulated easily but offer only loose bounds that did not result in any improvements.

Search Strategies

The previous CP model for TLSP-S (Geibinger, Mischek, and Musliu 2019b) very successfully employed the `priority_search` annotation in MiniZinc (Feydy et al. 2017). It ordered the jobs by their earliest possible start time and then schedule them one by one, fully assigning time slots, a mode, and resources to each job one after another. This search can easily be transferred to the new model for the most part. However, there are multiple ways to include task grouping. We closely investigated two approaches:

- *Grouping before Scheduling*

  The first part of the search is to decide on a job grouping. After some initial experimentation the most promising approach to this was assigning the variables in $\xi(a)$ family by family, starting with the largest family. The values were assigned in the way which creates the fewest number of jobs. After grouping all tasks, scheduling is described by a `priority_search` annotation. Jobs are scheduled one after another in ascending order of their lower bound on the starting time. This means assigning each job its earliest starting time, followed by a mode (preferring the shortest execution time), and then employees, workbenches and equipment (using `first_fail` and `indomain_max` turned out most beneficial).

- *Grouping while Scheduling*

  In contrast to the first approach, this search starts with the `priority_search` right away. Because the grouping is not decided upon at this point, the `priority_search` searches over all tasks, again in ascending order of the lower bound on the starting time. The first step when scheduling a task a is to fix $\xi(a)$. Afterward, time slots and resources are assigned like in the previous strategy.

In the end, *grouping while scheduling* turned out to be significantly better when it came to finding feasible solutions. Not only was it faster in almost all cases, but it could also solve one more benchmark instance. *Grouping before scheduling* appeared to be slightly better at closing small instances, including benchmark instances of up to 5 projects and those arising during VLNS. However, the differences are much less pronounced here. While there were some aggregate differences over a large number of runs, the typical run-to-run variance usually played a much larger factor. This is why in the experimental evaluation of the CP model we use the *grouping while scheduling* approach.

Very Large Neighborhood Search

Based on the proposed CP model, we implemented a *Very Large Neighborhood Search* (VLNS). The algorithm and implementation are based on the existing VLNS algorithm for TLSP-S (Geibinger, Mischek, and Musliu 2019a), with several extensions to incorporate the new CP model.

Given an initial feasible solution, the algorithm repeatedly fixes the schedule (including the task grouping, as well as assigned time slots and resources) for all but a small number of projects and uses a CP model to try to find an optimal schedule for the unfixed projects.

Although being able to modify the task grouping is, in principle, a big advantage and allows for better solutions, this comes at the cost of much longer run times. To alleviate this, we employ both our new CP model and the model for TLSP-S used by the existing VLNS, and switch between them randomly – having some moves only change the schedule and others also alter the task grouping.

1. *Generate initial solution*

   Our VLNS requires a feasible schedule to operate on. To solve the full TLSP and generate solutions without knowing a feasible task grouping a priori, we use our new CP model. As soon as a feasible solution is found, the algorithm continues. There is no time limit for this stage apart from the total time available.

2. *Decide which CP Model to use*

   Each move utilizes either the CP model for TLSP-S proposed in (Geibinger, Mischek, and Musliu 2019b) or the CP model for the full TLSP described above. One of those models is selected randomly and independently for each move. The full TLSP model is chosen with a probability given by the parameter *regroupProb*.

3. *Fix all but k projects*

   Now, we generate an instance for the move. First, a random combination of k_X projects is selected to be rescheduled, where X is either ”fixed” or ”variable”, depending on the chosen CP model. Both variables are initially set to 1 and updated at a later step in the algorithm. The projects are chosen in such a way that all of them overlap in the current schedule (or, if that is not possible, could overlap based on their release and due times).Once some projects have been chosen, an instance is created by fixing the assignments of all jobs contained in the other projects and cutting away all irrelevant information. In order to further optimise the instance, the grouping and all assignments of tasks contained in fixed projects are assigned directly. This significantly reduces compilation time.

4. *Perform move*

   Once the instance has been prepared, we execute a CP solver to (ideally) solve it to optimality. This changes the time and resource assignments of the selected projects. The best assignments found by the solver are then applied to the current schedule unless doing so would increase the penalty. To prevent the algorithm from spending too much time on individual hard instances, the CP solver is executed with a time limit, which is passed as a parameter to the algorithm. We differentiate between the two CP models, introducing the parameters *fixedMzTimeout* and *variableMzTimeout*.

   Similarly to (Geibinger, Mischek, and Musliu 2019a) we also *hot start* the CP solver with a parameterized probability given by *variableHotStartProb*or *fixedHotStartProb*, again differentiating between the two CP models. Since hot starting allows the solver to start from a known feasible solution, it can speed up the search significantly, albeit with the drawback that the solver never returns different solutions of the same quality, hence why a probability is used. Also as in (Geibinger, Mischek, and Musliu 2019a) if hot starting is not used, we modify our `priority_search` to assign resources randomly, for further diversification.

5. *Change k and save combination*

   Depending on which model was used during the move this step operates on either k_{fixed}or k_{variable}, hereafter called *k*. If $k \geq 1$ and the move changed the schedule, *k* is reset to 1. If there are no valid combinations of projects left for *k*, it is increased by one or – with probability *jumpProb* – by two. If there are no valid combinations of projects left for any *k* and for any CP model, the algorithm terminates. If there are no combinations left for any *k* for the fixed CP model only, the TLSP model is used for the next move. Additionally we save the combinations of projects which have already been scheduled once and do not select them again until there has been a change in the schedule overlapping their time window.

6. *Repeat*

Until the algorithm's time limit is reached, we go back to step 2 and perform another move.

Experiments

For our evaluations we used a data set containing 33 TLSP instances of varying size (ranging from 5 to 90 projects and from 13 to around 1500 tasks) and scheduling period length (88 to 782 time slots), taken from https://www.dbai.tuwien.ac.at/staff/fmischek/TLSP/. 30 of those instances were randomly generated and the remaining three were taken directly from a real-world laboratory. This is the same set of instances as was used in (Geibinger, Mischek, and Musliu 2019a) and we refer to that paper for a comprehensive description of these instances, and to (Mischek and Musliu 2018) for details on the instance generation process for the randomly generated instances. The exceptions to this are the second and third real-world instances, which are introduced for the first time in this paper. Those two instances are anonymized snapshots taken directly from our industrial partner at different dates.

We conducted our experiments on a benchmark server with 224GB RAM and two AMD Opteron 6272 Processors each with a frequency of 2.1GHz and 16 logical cores. As was done in (Geibinger, Mischek, and Musliu 2019a), we usually executed two independent benchmarking runs in parallel, since all of our solution approaches are single-threaded. Each run had a time limit of two hours. As our backend CP solver, we used Chuffed (Chu 2011).

Parameter Configuration

As described earlier, there are a total of 6 parameters for VLNS. First, there is the probability *regroupProb* of using the TLSP model, as opposed to the model TLSP-S preserving the task grouping. Then, there are the timeouts for single moves based on the chosen model, *fixedMzTimeout* and *variableMzTimeout*. Each move is hot-started with probability *fixedHotStartProb* or *variableHotStartProb*. Finally, when the algorithm is forced to increase the number of projects re-scheduled simultaneously, *jumpProb* is the probability of increasing this number by two instead of one.

For parameter tuning, we employed SMAC3 (Hutter, Hoos, and Leyton-Brown 2011), version 0.11.0. Tuning was performed on a set of 30 generated instances which were distinct from, but chosen in the same way as our test set. We used a budget of 1200 algorithm runs, performing four trials in parallel. In the end, SMAC recommended setting *regroupProb* to 10%, and *fixedMzTimeout* and *variableMzTimeout* to $20s$ and $40s$, respectively. Further, *fixedHotStartProb* and *variableHotStartProb* were both set to 80%, reflecting the results from (Geibinger, Mischek, and Musliu 2019a). Finally, the recommended value for *jumpProb* was a low probability of 3% and stands in stark contrast to the 35% used in (Geibinger, Mischek, and Musliu 2019b). This seems plausible given that, in the previous VLNS for TLSP-S, the *jumpProb* parameter mainly helped the algorithm explore a larger neighborhood. This kept it from getting stuck in local optima. Incorporating the new CP model with variable grouping, which can explore a much larger neighborhood by itself, could thus lessen the importance of the parameter. In fact, it might even hurt the performance of the TLSP model by increasing the neighborhood to a size that cannot be efficiently explored by the solver in the given timelimit for each move.

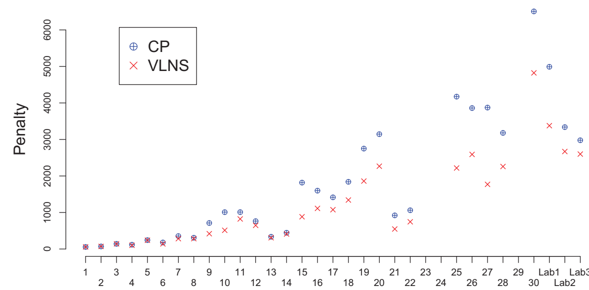

Figure 1: Results for the CP model and for VLNS. The y-axis shows the the total objective value for each instance, i.e. the sum of equations (18–22). No feasible solution was found within the time limit for instances 23, 24 and 29.

Results

We evaluated the performance of both our CP model and our VLNS approach. Since VLNS is non-deterministic, we performed five runs of it for each instance with different seeds. The results given for VLNS in this section are averages of those five runs, unless noted otherwise.

Figure 1 shows the results for the CP approach and VLNS. It can be immediately seen that CP could find feasible solutions for 30 of the 33 instances (including all real-life instance), although optimality of the solutions could be proved only for the two smallest instances. Since the CP solver was also used to provide an initial feasible solution for VLNS, the same three instances as with CP alone remained unsolved. For the remaining instances, the solutions found using VLNS were at least as good as with CP alone, and better in all cases but those where CP could already find optimal solutions. In some cases, the solutions produced by VLNS were improved by more than 50% compared to CP.

When comparing our results to those reported in (Geibinger, Mischek, and Musliu 2019a) (see Figure 2), one has to keep in mind that the problem solved in that paper is actually not TLSP, but TLSP-S, which requires that a (feasible) grouping of tasks into jobs is already provided and cannot be changed. On the one hand, this means that for any given instance, the optimal solution for TLSP is at least as good as the one for TLSP-S for any provided grouping. On the other hand, not having to include grouping allows for much simpler and more efficient models, including, but not limited to precomputed constants for the number and properties of jobs. As long as the given grouping is good enough, it is easier to find good solutions within limited time than for a model that simultaneously has to build up and dynamically adjust such a grouping.

This effect could also be seen in our results: For small instances with up to 10 projects, we could consistently improve upon the best known results for TLSP-S using the initially given grouping. However, as the instances get larger,

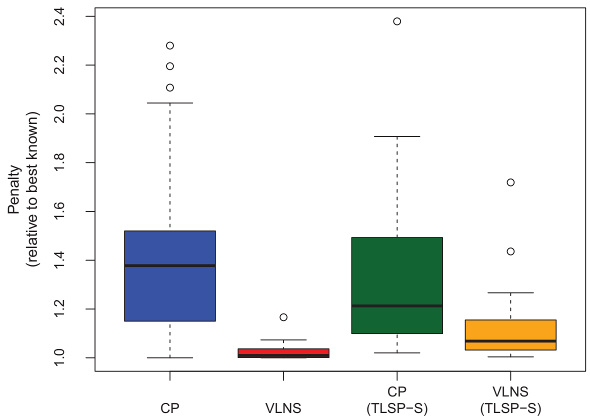

Figure 2: Results for CP and VLNS are compared to existing solutions for TLSP-S. To normalize over different instance sizes, the penalty for each run was divided by the penalty of the best known solution for the instance.

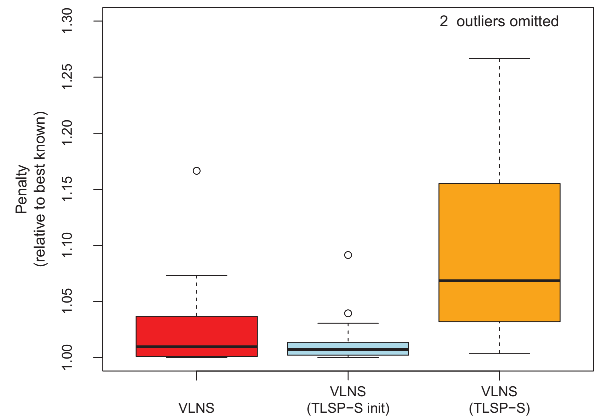

Figure 3: Comparison of VLNS and VLNS for TLSP-S (also depicted in Figure 2) with a variant of VLNS that generates its initial solution using the CP model for TLSP-S (center). Penalties are normalized by the best known solution for each instance.

this is no longer always the case, in particular for CP alone, which falls behind its counterpart for TLSP-S. VLNS fares much better, presumably due to the fact that the instances to evaluate at each move are consistently small, and we report several new best-known solutions even for large instances. The initial solution in the first step of the algorithm was found within one minute for 18 of the 30 generated instances, and within 7 minutes for further 8 instances. For three instances, no feasible solution could be found at all within two hours.

In order to provide a fair comparison between the VLNS for TLSP-S and our approach for TLSP, we also evaluated a variant of our VLNS algorithm where the initial solution is generated by the CP model for TLSP-S, using initially provided grouping. The rest of the solution process was performed as described in the section above. Figure 3 shows the results for this experiment. Here, the inclusion of regrouping moves resulted in improved solutions for almost every single instance, by up to a third of the original best known penalty.

On the other hand, any solution approach for TLSP-S can be applied to TLSP, if it is combined with a mechanism to generate an initial grouping. For this purpose, we created a greedy construction heuristic. This heuristic iteratively assigns tasks to existing jobs as long as this does not introduce a local infeasibility for that job (e.g. by having less resources available than required or by creating a cycle in the precedence graph). A new job is created whenever this assignment is not possible for a task. The only exception in this process are fixed tasks, which are assigned first to ensure that they end up in the same job. As an example of a TLSP-S solver, we used the CP model from (Geibinger, Mischek, and Musliu 2019a). For better comparability with our new model for TLSP, we also included the optimization regarding equipment classes described in a previous section. We then ran the TLSP-S model with the grouping obtained by our construction heuristic. This grouping turned out to be infeasible for 10 of the 33 instances, including all three real-world instances. For the remaining instances, the TLSP-S solver managed to find feasible solutions that are comparable to those achieved by our TLSP-S model. This shows that while solution approaches for TLSP-S can find good solutions in some scenarios, approaches such as ours have to be used in those cases where a feasible grouping is not known and cannot be easily found.

Conclusion

In this work we successfully modeled the real-world scheduling problem TLSP, which so far has only been studied in its restricted case TLSP-S. Besides utilising and adapting existing approaches for formalising scheduling problems from the literature and earlier work, we found a novel way to model the task grouping aspect of the problem. Furthermore, we investigated several optimisations for our approach and managed to further improve the performance for larger instances. In order to improve the quality of solutions for large instances, we developed a Very Large Neighborhood Search based on our exact method and an existing VLNS for TLSP-S. We evaluated our methods with 30 randomly generated benchmark instances and 3 real-world examples. With our CP model we could prove optimality for the two smallest benchmark instances and found feasible solutions for all but 3 instances in total. Furthermore, VLNS was able to reduce the penalty of the solutions for every instance where the TLSP model found a feasible solution. We also experimented with an approach which takes a feasible TLSP-S solution and uses VLNS to improve the penalty and showed that this generally achieves better results than using VLNS with our TLSP model.

The methods discussed in this paper are currently being deployed for real-world use and show very good results. For the future we plan to improve VLNS by making it less reliant on a feasible initial solution, which is currently its main bottleneck for larger instances. We also plan to investigate even more general constraint formulations that will allow us to deploy our models in other settings with similar requirements.

Acknowledgments

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.

References

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

Brucker, P.; Drexl, A.; Möhring, R.; Neumann, K.; and 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*. Ph.D. Dissertation, University of Melbourne, Australia.

Elmaghraby, S. E. 1977. *Activity networks: Project planning and control by network models*. John Wiley & Sons.

Feydy, T.; Goldwaser, A.; Schutt, A.; Stuckey, P. J.; and Young, K. D. 2017. Priority search with minizinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation at CP2017*.

Geibinger, T.; Mischek, F.; and Musliu, N. 2019a. Investigating constraint programming and hybrid methods for real world industrial test laboratory scheduling. *Submitted to journal, preprint at arXiv:1911.04766*.

Geibinger, T.; Mischek, F.; and Musliu, N. 2019b. Investigating constraint programming for real world industrial test laboratory scheduling. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 16th International Conference, CPAIOR 2019, Thessaloniki, Greece, June 4-7, 2019, Proceedings*, 304–319.

Hartmann, S., and 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, H. H.; and Leyton-Brown, K. 2011. Sequential model-based optimization for general algorithm configuration. In Coello, C. A. C., ed., *Learning and Intelligent Optimization*, 507–523. Berlin, Heidelberg: Springer Berlin Heidelberg.

Mika, M.; Waligóra, G.; and Węglarz, J. 2006. Modelling setup times in project scheduling. *Perspectives in modern project scheduling* 131–163.

Mika, M.; Waligóra, G.; and Węglarz, J. 2008. Tabu search for multi-mode resource-constrained project scheduling with schedule-dependent setup times. *European Journal of Operational Research* 187(3):1238 – 1250.

Mika, M.; Waligóra, G.; and Węglarz, J. 2015. Overview and state of the art. In Schwindt, C., and Zimmermann, J., eds., *Handbook on Project Management and Scheduling Vol.1*. Cham: Springer International Publishing. 445–490.

Mischek, F., and 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.

Mischek, F., and Musliu, N. 2019. A local search framework for industrial test laboratory scheduling. *Submitted to journal, preprint at dbai.tuwien.ac.at/staff/fmischek/TLSP/*.

Nethercote, N.; Stuckey, P. J.; Becket, R.; Brand, S.; Duck, G. J.; and 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*, 529–543.

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

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

Szeredi, R., and 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*, 483–492.

Węglarz, J.; Józefowska, J.; Mika, M.; and 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.

Young, K. D.; Feydy, T.; and Schutt, A. 2017. Constraint programming applied to the multi-skill project scheduling problem. In *Principles and Practice of Constraint Programming - 23rd International Conference, CP 2017, Melbourne, VIC, Australia, August 28 - September 1, 2017, Proceedings*, 308–317.