Constraint Logic Programming for Real-World Test Laboratory Scheduling

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
{tgeibing,fmischek,musliu}@dbai.tuwien.ac.at

Abstract

The Test Laboratory Scheduling Problem (TLSP) and its subproblem TLSP-S are real-world industrial scheduling problems that are extensions of the Resource-Constrained Project Scheduling Problem (RCPSP). Besides several additional constraints, TLSP includes a grouping phase where the jobs to be scheduled have to be assembled from smaller tasks and derive their properties from this grouping. For TLSP-S such a grouping is already part of the input.

In this work, we show how TLSP-S can be solved by Answer-set Programming extended with ideas from other constraint solving paradigms. We propose a novel and efficient encoding and apply an answer-set solver for constraint logic programs called clingcon. Additionally, we utilize our encoding in a Very Large Neighborhood Search framework and compare our methods with the state of the art approaches. Our approach provides new upper bounds and optimality proofs for several existing benchmark instances in the literature.

Introduction

In project scheduling problems, a large number of activities must be fit into a schedule and assigned resources, subject to several restrictions, such as precedence constraints or time windows. Such problems are highly relevant in practice and different variants appear in many real-world settings.

One such setting is that of an industrial test laboratory, where standardised tests of components have to be performed by qualified employees using diverse equipment. The Test Laboratory Scheduling Problem was first described in (Mischek and Musliu 2018b,a). In addition to several complex scheduling constraints, it also features a grouping phase, where activities (jobs) need to be assembled from smaller units (tasks) and derive their properties from them. In this work, we deal with a subproblem of TLSP, called TLSP-S, where the grouping of tasks into jobs is already provided as part of the input, in order to concentrate on the scheduling part of the problem.

TLSP(-S) is an extension to the well-known Resource-Constrained Project Scheduling Problem (RCPSP). Besides several additional features that are also included in many other variants of RCPSP in the literature, TLSP(-S) also contains new aspects that arise from the real-world situation. These aspects include heterogeneous resources, where only a subset of all units may be assigned to a job for each resource, linked jobs, which must be performed by the same employees, and non-standard objectives.

Solution methods that have been applied to TLSP-S include Simulated Annealing (Mischek and Musliu 2019), Constraint Programming (CP) (Geibinger, Mischek, and Musliu 2019b), and a hybrid Very Large Neighborhood Search (VLNS) based on the CP model (Geibinger, Mischek, and Musliu 2019a). Although these approaches provide good solutions for existing benchmark instances, for many instances the optimal solutions are still not known.

In this paper, we investigate Constraint Answer-set Programming (CASP) for solving TLSP-S. We introduce and describe a new CASP model for the problem and investigate/evaluate different formulations for the resource overlap constraint. Additionally, we incorporate this CASP model into a robust framework that is based on Very Large Neighborhood Search.

In our experiments, we show that CASP is very successful at providing good solutions for this large-scale project scheduling problem and even outperforms established solution approaches for project scheduling problems, such as CP, in some instances. These improvements extend to the Very Large Neighborhood Search framework, enabling us to find new upper bounds for multiple benchmark instances and prove the optimality of several solutions for the first time.

The rest of the paper is organised as follows. In the next Section we give an overview of related work, including other (C)ASP approaches to scheduling. We then provide a summary of the TLSP-S problem. A constraint model is given in the following section. Experimental results are presented and discussed next and the last section gives conclusions.

Literature Overview

Numerous project scheduling problems exist in the literature. Probably the most studied of those problems is the Resource-Constrained Project Scheduling Problem (RCPSP) (Brucker et al. 1999; Hartmann and Briskorn 2010; Mika, Waligóra, and Węglarz 2015) and its variants. Of those variants the ones of particular relevance for TLSP(-S) are the Multi-Mode RCPSP (MRCPSP) (Elmaghraby 1977; Węglarz et al. 2011; Hartmann and Briskorn 2010; Szeredi and Schutt 2016) and the Multi-Skill RCPSP (MSPSP) (Bellenguez and Néron

Copyright © 2021, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

2005; Young, Feydy, and Schutt 2017). MRCPSP allows for multiple execution modes just like TLSP(-S), whereas MSPSP 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). Furthermore, a solution approach for the full TLSP was provided in (Danzinger et al. 2020). All these approaches have been evaluated on publicly available real-life and randomly generated instances. Furthermore, according to the authors these methods have been deployed successfully to solve test laboratory scheduling problems on a daily basis. Although the reported results are good, many instances are still not solved to optimality. Therefore, the investigation of new solving paradigms for this challenging problem is an important research question.

Although other solving paradigms such as CP, SAT and SMT (Satisfiability Modulo Theories) have been used for related project scheduling problems (Young, Feydy, and Schutt 2017; Schutt et al. 2013; Bofill et al. 2020), to the best of our knowledge, neither Answer-set Programming nor hybrid extensions of ASP have been utilized for project scheduling problems. However, there are examples in the literature of those paradigms being used for other scheduling problems. In (Dodaro and Maratea 2017) and (Alviano, Dodaro, and Maratea 2017) the authors present ASP encodings for the Nurse Scheduling Problem. ASP solution methods for the Operation Room Scheduling Problem can be found in (Dodaro et al. 2018). Abseher et al. (Abseher et al. 2016) provide an ASP formulation for the Shift Design Problem. In the case of hybrid systems, (Balduccini 2011) and (Friedrich et al. 2016) employ an ASP and CP hybrid approach to solve industrial machine scheduling problems. An application of ASP combined with difference logic for a real-world train scheduling problem can be found in (Abels et al. 2020).

Constraint Answer-set Programming

First, we give a short introduction into Answer-set Programming (ASP) (Eiter, Ianni, and Krennwallner 2009).

Answer-set programs are defined over a vocabulary $\mathcal{V} = (\mathbb{P}, \mathbb{D})$, where $\mathbb{P}$ is a set of *predicates* and $\mathbb{D}$ is a set of *constants* (also referred to as the *domain* of $\mathcal{V}$). Each predicate in $\mathbb{P}$ has an *arity* $n \geq 0$. We also assume a set $\mathcal{A}$ of *variables*.[1]

An *atom* is defined as $p(t_1, \ldots, t_n)$, where $p \in \mathbb{P}$ and $t_i \in \mathbb{D} \cup \mathcal{A}$, for $1 \leq i \leq n$. We call an atom *ground* if no variable occurs in it.

A (*disjunctive*) *rule*, r, is an ordered pair of form

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m, \quad (1)$$

where $a_1, \ldots, a_n, b_1, \ldots, b_m$ are atoms, $n, m, k \geq 0$, and $n + m > 0$. Furthermore, "$\sim$" denotes *default negation* i.e. $\sim p$ is true if p is not derivable. The left-hand side of r is the *head* and the right-hand side is the *body* of r. For a program P, we define $H(P) = \bigcup_{r \in P} H(r)$ and $B(P) = \bigcup_{r \in P} B(r)$.

A rule r of form (1) is called (i) a *fact*, if $m = 0$ and $n = 1$; (ii) a *constraint*, if $n = 0$; (iii) *safe*, if each variable occurring in $H(r) \cup B^-(r)$ also occurs in $B^+(r)$; and (iv) *ground*, if all atoms in r are ground.

A *program* is a set of safe rules. We call a program ground if all of its rules are ground.

The set of all constants appearing in a program P is called the *Herbrand universe* of P, symbolically HU_P. If no constant appears in P, then HU_P contains an arbitrary constant.

Given a rule r and a set C of constants, we define $grd(r, C)$ as the set of all rules generated by replacing all variables of r with elements of C. For any program P, the *grounding of P with respect to C* is given by $grd(P, C) := \bigcup_{r \in P} grd(r, C)$. If P is a ground program, then $P = grd(P, C)$ for any C.

A set of ground atoms is called an *interpretation*. Following the answer-set semantics for logic programs as defined by Gelfond and Lifschitz (Gelfond and Lifschitz 1991), a ground rule r is *satisfied* by an interpretation I, denoted by $I \models r$, iff $H(r) \cap I \neq \emptyset$ whenever $B^+(r) \subseteq I$ and $B^-(r) \cap I = \emptyset$. For a ground program P, $I \models P$ iff each $r \in P$ is satisfied by I. The *Gelfond-Lifschitz reduct* (Gelfond and Lifschitz 1988) of a ground program P with respect to the interpretation I is given by

$$P^I := \{H(r) \leftarrow B^+(r) \mid r \in P, \; I \cap B^-(r) = \emptyset\}.$$

An interpretation I is an *answer set* of a non-ground program P iff I is a subset-minimal set satisfying $grd(P, HU_P)^I$.

Also, some ASP systems, for example *clingo* (Gebser et al. 2014), support *choice rules* i.e. rules of the form $l \; \{a_1 \; ; \; \ldots \; ; \; a_n \; ; \; \ldots \; ; \; \sim a_{n+1} \; ; \; \ldots \; ; \; \sim a_o\} \; u \leftarrow b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m$, where l and u are natural numbers specifying a *lower* and *upper bound*. An interpretation satisfies the head of such a rule if $l \leq (\sum_{1 \leq i \leq n} a_i \in I + \sum_{n+1 \leq i \leq o} a_i \notin I) \leq u$.

We are also going to use *conditional* choice rules where the head takes the form $l \; \{L_0 : L_1, \ldots, L_n\} \; u$ and L_i $(0 \leq i \leq n)$ are non-ground atoms or default negated non-ground atoms. Such a rule is essentially expanded during grounding into an unconditional choice rule containing an instantiation of L_0 whenever the corresponding instantiations of the atoms in $L_1, \ldots, \; L_n$ are present or respectively not present as facts.

Furthermore, we also make use of *aggregates*. Although different types of aggregates are supported by most ASP solvers, we exclusively employ *count aggregates*. Generally, a count aggregate has the form $\#count\{v_1, \ldots, v_n : L_1, \ldots, L_m\} = b$, where L_i $(i \leq m)$ are literals, v_j $(j \leq n)$ are variables occurring in these literals, and b is a natural number. An aggregate can be used in the positive body of a rule. During grounding a count aggregate is expanded to $\#count\{c_1^1, \ldots, c_n^1 : l_1^1, \ldots, l_m^1 \; ; \; \ldots \; ; \; c_1^k, \ldots, c_n^k : l_1^k, \ldots, l_m^k\} = b$, where $l_1^i, \ldots, l_m^i$ $(i \leq k)$ are ground instances of the literals and $c_1^i, \ldots, c_n^i$ are the respective constants instantiated for variables $v_1, \ldots, v_n$. An aggregate is satisfied by an interpretation I if the number of tuples $\langle c_1^i, \ldots, c_n^i \rangle$ where $l_1^k, \ldots, l_m^k$ is satisfied by I $(i \leq k)$, is exactly b.

For a more thorough reference for choice rules, aggregates and other ASP language features, we refer to the relevant literature (Eiter, Ianni, and Krennwallner 2009; Gebser et al. 2012, 2014).

[1] In our encodings, all variables are denoted by arbitrary uppercase letters.

Constraint Answer-set Programming (CASP) extends ASP with linear variables $V = \langle v_1, \ldots, v_n \rangle$ over domains $D = \langle D_1, \ldots, D_n \rangle$ and linear constraints. In *clingcon* (Banbara et al. 2017) – an extension of clingo and the CASP solver we concentrate on – these linear constraints can appear as body atoms or singular head atoms and the domains are sets of integers. An interpretation for a program as defined above is extended by a variable assignment $A = \langle a_1, \ldots, a_n \rangle$ of domain elements to the variables i.e. $a_i \in D_i$ $(i \leq n)$. A linear constraint is satisfied if the assignment satisfies it. Answer-sets for CASP can then be defined – *mutatis mutandis* – as for standard ASP above.

In clingcon, we can define the domains of linear variables via domain constraints of the form $\&dom\{l..u\} = v$, where l and u are integer constants, and v is a linear variable. The lower bound of the domain of v is represented by l and the upper bound by u. Linear constraints like $v_1 + \cdots + v_n \circ k$, for $\circ \in \{=, \leq, \geq, <, >\}$, can be expressed with the constraint atom $\&sum\{v_1 \;;\; \ldots \;;\; v_n\} \circ k$, where v_i $(i \leq n)$ are linear variables and k is an integer constant. For linear and domain constraints the constants can also be ASP variables and the linear variables can contain ASP variables as well. The latter leads to the generation of a linear variable for each corresponding constant during the grounding of the program.

We can also specify an optimization objective for a CASP program. In clingcon this can be done by adding a directive $\&minimize\{t : L_1, \ldots, L_n\}$, where t is a linear term over non-ground linear variables and L_i $(i \leq n)$ are non-ground literals. During grounding the directive is expanded to represent the sum of the corresponding grounded linear variables and the objective is to find an answer-set for which this sum is minimal. If multiple directives are contained in a program, then the overall objective is to minimize the total sum of all of them.

For a more detailed introduction to CASP and the input language of clingcon, we refer to the article by Banbara et al. (Banbara et al. 2017).

Problem Description

In this work, we deal with TLSP-S, which is a variant of TLSP. A complete and formal definition of both problems can be found in (Mischek and Musliu 2018b).

Each instance of TLSP-S consists of a set of *projects*, which contain *jobs* to be scheduled. Each job has several properties:

- A time window, given via a *release date* and a *deadline*. In addition, it has a *due date*, which is similar to the deadline, except that exceeding it is only a soft constraint violation.
- A set of *available modes* in which the job can be performed.
- A *duration* which is modified by the assigned mode.
- The *resource requirements* for the job. We distinguish between workbenches, employees and several equipment groups. For each of these resources, the job has a certain *demand* (at most one workbench, the number of employees depends on the assigned mode, demands for each equipment group can be arbitrary). The assigned resource units must be chosen from the set of *available units* for that job and resource. For employees, there is also a set of *preferred employees*, which should be chosen, if possible.
- The *predecessors* of the job, which must be completed before the job can start. Precedence relations will only occur between jobs of the same project.
- *Linked jobs* of this job, which must be performed by the same employee(s). As before, such links only occur between jobs of the same project.
- Optionally, the job may contain *initial assignments* of a mode, starting time slot and resources. Some or all of these assignments may be present for any given job.

Out of all jobs, a subset are *started jobs*, which will always start at time slot 0 and have initial mode and resource assignments that are available for the job. The initial assignments of a started job must not be changed in the solution.

The goal of TLSP-S is to find an assignment of a mode, a time slot interval, and resources to each job, such that all constraints are fulfilled and the objective function, the weighted sum of the violations of five soft constraints, is minimized. The weights of the soft constraints depend on the specific situation in the lab and as was done in previous work, we set them all to 1 for this paper. The hard and soft constraints that we consider for TLSP-S will be listed in the next section, where we will introduce the encoding as a constraint logic program. A complete description and formal definition of all constraints of the original model can be found in (Mischek and Musliu 2018b).

Constraint Answer-set Program

In this section we present our Constraint Answer-set Program for TLSP-S written in the input language of clingcon (Gebser, Ostrowski, and Schaub 2009; Banbara et al. 2017).

A more detailed description of the encoding can also be found in the master thesis of the first author (Geibinger 2020).

Solution Representation

Each answer-set of the encoding we are about to give will represent a solution of TLSP-S with respect to the given instance. More specifically, an answer-set will include the following facts for each job $j \in J$:

- $modeAssign(j, m)$ indicating that j is assigned mode m,
- $empAssign(j, e)$ expressing that employee e is assigned to j,
- $workbenchAssign(j, w)$ representing the assignment of workbench w to j, and
- $equipAssign(j, e)$ meaning that j is assigned equipment e.

Furthermore, an answer-set also contains a start time assignment $start(j)$ for each job j which is encoded as an integer variable.

Basic Hard Constraints

The encoding for the basic hard constraints can be found in Figure 1. The rules enforce the following constraints.

$$\&dom\{R..D\} = start(J) \leftarrow job(J), release(J,R), deadline(J,D) \quad (2)$$
$$\&dom\{R..D\} = end(J) \leftarrow job(J), release(J,R), deadline(J,D) \quad (3)$$
$$1\ \{modeAssign(J,M) : modeAvail(J,M)\}\ 1 \leftarrow job(J) \quad (4)$$
$$duration(J,T) \leftarrow job(J), modeAssign(J,M), durInMode(J,M,T) \quad (5)$$
$$\&sum\{end(J); -start(J)\} = T \leftarrow job(J), duration(J,T) \quad (6)$$
$$\&sum\{start(J)\} \geq end(K) \leftarrow job(J), job(K), precedence(J,K) \quad (7)$$
$$\&sum\{start(J)\} = 0 \leftarrow job(J), started(J) \quad (8)$$
$$1\ \{workbenchAssign(J,W) : workbenchAvail(J,W)\}\ 1 \leftarrow job(J), workbenchReq(J) \quad (9)$$
$$R\ \{empAssign(J,E) : empAvail(J,E)\}\ R \leftarrow job(J), modeAssign(J,M), reqEmployees(M,R) \quad (10)$$
$$R\ \{equipAssign(J,E) : equipAvail(J,E), inGroup(E,G)\}\ R \leftarrow job(J), group(G), reqEquip(J,G,R) \quad (11)$$
$$\leftarrow job(J), job(K), linked(J,K), empAssign(J,E), \sim empAssign(K,E) \quad (12)$$

Figure 1: Basic Hard Constraints

$$precedence(J,K) \lor precedence(K,J) \leftarrow job(J), job(K), workbenchAssign(J,W), workbenchAssign(K,W), J < K \quad (13)$$
$$precedence(J,K) \lor precedence(K,J) \leftarrow job(J), job(K), empAssign(J,E), empAssign(K,E), J < K \quad (14)$$
$$precedence(J,K) \lor precedence(K,J) \leftarrow job(J), job(K), equipAssign(J,E), equipAssign(K,E), J < K \quad (15)$$

Figure 2: No overlaps: Precedence formulation

Time windows For each job j, we have input facts $release(j,r)$ and $deadline(j,d)$ indicating that r is its release date and d is its deadline respectively. The rules (2) and (3) define the domains of the start and end times which are bounded by the release times and deadlines of a job. This ensures that every job is executed within its time window.

Job duration Rule (5) enforces that each job has the duration required by its mode and (6) links the start and end time to the duration. The duration t of a job j when performed in mode m is given by the input fact $durInMode(j,m,t)$. Additionally, rule (4) ensures that every job is assigned an available mode.

Precedences For each predecessor k of a job j , we have an input fact $precedence(j,k)$. Rule (7) ensures that a job's predecessors are completed before its own start.

Started Jobs If a job j is started then we have an input fact $started(j)$. Those jobs are required to start at the first available timeslot i.e. zero. This is enforced by rule (8).

Resource Requirements The resource requirements for each job are handled by rules (9), (10), and (11), ensuring that each job is assigned a workbench if required, is assigned as many employees as its mode requires, and the demands for each equipment group are met, respectively.

Linked Jobs We also need to ensure that any two linked jobs j and k (identified by input fact $linked(j,k)$) are performed by the same employees, this is accomplished by the constraint (12).

Unary Resource Constraints

In (Geibinger, Mischek, and Musliu 2019b), the authors expound on the difficulty of modelling *unary resource constraints* for TLSP-S i.e. constraints ensuring that no resource is used by multiple jobs simultaneously. Their best formulation of those constraints relied on a global cumulative constraint. clingcon has no such cumulative constraint and the disjunctive it does support only works with fixed durations, which we do not have because of the different possible execution modes. Hence, we had to decompose the constraints.

We propose two formulations for unary resource constraints. The first can be found in Figure 2.

Intuitively, those rules specify that whenever a resource is used by two different jobs $j,k \in J$, then either j has to precede k or k has to precede j. Note that with those rules it

could be possible that both jobs precede each other, but this cannot happen in any answer-set because of rule (7).

The second formulation is more straight-forward and is given in Figure 3. In this formulation, a fact $overlap(j, k)$ is derived for each pair of overlapping jobs $j, k \in J$ by rule (16). The constraints (17–19) then ensure that overlapping jobs cannot be assigned the same resources.

Soft Constraints

There are five soft constraints in the problem definition of TLSP (Mischek and Musliu 2018b). The objective function is the weighted sum over all soft constraint violations.

The first soft constraint counts the number of jobs. Since the job grouping is fixed in TLSP-S, this is reduced to a constant, which we can omit from the model (for comparability with previous work, we add this constant to the final objective value in post-processing). Hence, we are left with four soft constraints each of which we can specify in our encoding as a minimization objective. The complete encoding of all soft constraints can be found in Figure 4.

Employee project preferences The first soft constraint minimizes the number of employees assigned to each job despite not being preferred. This can easily be realised using a count aggregate as given in rule (20). For each job j, this aggregate counts the number of occurrences where an employee e is assigned to j but $empPref(j, e)$ (indicating that e is preferred for j) cannot be derived and is thus not part of the input. The objective (21) then encodes the necessary minimization.

Number of employees We also need to minimize the number of different employees in each project. This is achieved again by a simple count aggregate as formulated in (22) and (23). The input fact $projAssign(j, p)$ denotes that j is contained in project p.

Due dates Another soft constraint considered in (Mischek and Musliu 2018b) is minimizing tardiness i.e. we generally want a job j to end before is due date t given by input fact $due(j, t)$ or, if this is not possible, to minimize the delay. The encoding for this objective can be found in (24–26) where we introduce integer variables for the delay of each job. Those variables are then constrained to represent the difference between the end and the due date or zero if the job completes within its due date.

Project completion time The last soft constraint has the most complex formulation which is given in (27–35). The goal here is to minimize the completion time of each project i.e. the time between the project start and end. The reason why this objective is difficult to define is that we effectively need to determine the job with the earliest start in a project as well as the one with the latest end. We achieve this by guessing a first and last job for each project with the rules (29) and (32). For the selected first job, we ensure that no other job in the project has an earlier start. Similarly for the selected last job. We can then easily define the project start via rule (30) and the end with rule (33). The objective (35) then simply minimizes the sum of all completion times.

Very Large Neighborhood Search

Analogously to (Geibinger, Mischek, and Musliu 2019a), we employed our encodings in a Very Large Neighborhood Search framework. Just like Geibinger et al. we start with a feasible solution and repeatedly fix most of the schedule except for a small number of projects and then try to find an optimal solution for the unfixed projects.

The basic steps of the search are are the same as in (Geibinger, Mischek, and Musliu 2019a) but we do not *hot start* the solver as this functionality seems to be unsupported by clingcon. Instead, we only apply moves if they present an actual improvement of the current solution. Hence, we have the following algorithm:

1. *Find Initial Solution*
   In order for the Very Large Neighborhood Search to work, we need a feasible schedule for the instance. Hence, we use clingcon and our encoding to find such a schedule.
2. *Calculate lower bound for each project*
   In parallel to step one we determine a lower bound for each project. This is done by running clingcon on each project once without regard for the other projects. The optimal objective is then a lower bound for the penalty of the project in the original problem. Since this can take a long time for bigger projects we define a runtime limit of 30 seconds for each project, which is also going to be the move timeout later. If the limit is reached and no optimal solution could be found, we determine a heuristic lower bound for the project by adding up the number of jobs, the minimum number of different employees needed for the project, and the minimal duration of all jobs on the longest path in the job dependency graph.
3. *Fix all but k projects*
   After the initial solution is found we select at random a combination of k projects (initially, $k = 1$), with the following properties: All of them overlap in the current schedule (or, if there are no such combinations, have overlapping time windows) and at least one of the projects has potential for improvement i.e. the difference between the current penalty and the lower bound is bigger than zero.

   The projects which are not in the selected combination are then fixed by modifying the time windows and availabilities of each job contained in a fixed project.
4. *Perform move*
   After the preprocessing we try to find an optimal solution for the selected projects, where we set a runtime limit of 30 seconds. The best assignment found within this limit is then applied to the current incumbent schedule if it improves the current solution.

   The combination of projects used in this move is then saved in a list. Project combinations contained in this list (or subsets of a combination in the list) are not selected again unless there has been a change in a job overlapping the projects.

$$overlap(J,K) \leftarrow job(J), job(K), \&sum\{start(K)\} < end(J), \&sum\{end(K)\} > start(J), J < K \tag{16}$$

$$\leftarrow overlap(J,K), workbenchAssign(J,W), workbenchAssign(K,W), J < K \tag{17}$$

$$\leftarrow overlap(J,K), empAssign(J,E), empAssign(K,E), J < K \tag{18}$$

$$\leftarrow overlap(J,K), equipAssign(J,E), equipAssign(K,E), J < K \tag{19}$$

Figure 3: No overlaps: Direct formulation

$$\&sum\{unprefEmp(J)\} = N \leftarrow job(J), \#count\{E : empAssign(J,E), \sim empPref(J,E)\} = N \tag{20}$$

$$\&minimize\{unprefEmp(J) : job(J)\} \tag{21}$$

$$\&sum\{employees(P)\} = N \leftarrow project(P), \#count\{E : empAssign(J,E), projAssign(J,P)\} = N \tag{22}$$

$$\&minimize\{employees(P) : project(P)\} \tag{23}$$

$$\&sum\{delay(J); T\} = end(J) \leftarrow job(J), due(J,T), \&sum\{end(J); -T\} > 0 \tag{24}$$

$$\&sum\{delay(J)\} = 0 \leftarrow job(J), due(J,T), \&sum\{end(J); -T\} \leq 0 \tag{25}$$

$$\&minimize\{delay(J) : job(J)\} \tag{26}$$

$$\&dom\{0..H\} = projectStart(P) \leftarrow project(P), horizon(H) \tag{27}$$

$$\&dom\{0..H\} = projectEnd(P) \leftarrow project(P), horizon(H) \tag{28}$$

$$1 \; \{firstJob(J) : job(J), projAssign(J,P)\} \; 1 \leftarrow project(P) \tag{29}$$

$$\&sum\{projectStart(P)\} = start(J) \leftarrow firstJob(J), projAssign(J,P) \tag{30}$$

$$\&sum\{projectStart(P)\} \leq start(J) \leftarrow job(J), projAssign(J,P) \tag{31}$$

$$1 \; \{lastJob(J) : job(J), projAssign(J,P)\} \; 1 \leftarrow project(P) \tag{32}$$

$$\&sum\{projectEnd(P)\} = end(J) \leftarrow lastJob(J), projAssign(J,P) \tag{33}$$

$$\&sum\{projectEnd(P)\} \geq end(J) \leftarrow job(J), projAssign(J,P) \tag{34}$$

$$\&minimize\{projectEnd(P) - projectStart(P) : project(P)\} \tag{35}$$

Figure 4: Soft constraints

5. *Possibly change k*
   If k is bigger than 1 and the incumbent schedule has been improved by the last move, then we set k back to 1. Otherwise, if there are no more eligible combinations with size k, we increase k by one or – with probability 0.35 – by two. This probabilistic increase was also used in (Geibinger, Mischek, and Musliu 2019a). If there no more combinations for any k, we double the move timeout and perform all moves again which did run out of time before. If there are no such moves, we terminate.

   After this we check if the current solution is equal to the sum of all project lower bounds. Should that be the case, then we have found an optimal solution and can terminate. If not, we go back to step 3 or terminate if we have reached the time limit of the solver.

Experiments

We evaluate our model on the benchmark instances that were also used in (Geibinger, Mischek, and Musliu 2019b). This data set contains 30 generated instances containing between 7 and 401 jobs. In addition, we use the three real-world instances from (Danzinger et al. 2020). All instances are available online[2].

As our CASP solver we use `clingcon-5` (unpublished as of the writing of this article[3]).

Our experiments were performed on a benchmark server with 224GB RAM and two AMD Opteron 6272 Processors each with a frequency of 2.1GHz and 16 logical cores. Unless noted otherwise, we used single-threaded configurations and we usually executed two independent benchmarking runs in parallel. Each run had a time limit of 1800 seconds.

Unary Resource Constraints

As described above, our model contains two alternative formulations for the unary resource constraints. We evaluated

[2]dbai.tuwien.ac.at/staff/fmischek/TLSP

[3]github.com/potassco/clingcon

Configuration	#Feas	#Opt	#Best
bb,hier	30	18	28
bb,dec	30	18	30
usc,3	30	18	27
default	30	18	26

Table 1: Comparison of different clingcon optimization strategies. Shown are the number of feasible solutions found (#Feas), the number of proven optima (#Opt) and the number of instances for which the solution was the best among all four configurations (#Best).

both versions of our model, once with the direct formulation and once with the precedence formulation, on all 30 generated test instances. The direct formulation found a better solution on 5 instances, while the precedence formulation found a better solution on 11 instances (both models produced schedules with the same objective value for the remaining 14 instances). In addition, the direct formulation could not find any feasible solution for the two biggest instances.

For this reason, we used the model with the precedence formulation for all further experiments.

Clingcon Search Strategy

clingcon supports a large number of configuration options to modify its behavior. For our experiments in this paper, we mostly rely on its default configuration. However, we do include here a comparison between different optimization strategies (parameter `--opt-strategy`), inspired by the strategies used in (Abseher et al. 2016).

We evaluated the following four optimization strategies:

(i) Branch-and-bound in hierarchical order of priorities (`bb,hier`),

(ii) branch-and-bound with exponentially decreasing steps (`bb,dec`, not used in (Abseher et al. 2016)),

(iii) optimization based on unsatisfiable cores (`usc,3`), and

(iv) clingcon's default strategy.

Table 1 shows the results on the 30 generated test instances. All four strategies produce solutions of identical objective value on most instances. On the few instances where they differ, configuration (ii) consistently produced the best solution. The remaining experiments use this strategy.

Comparison

For our main evaluations, we used a timeout of 2 hours (7200 seconds), to be comparable with the results in (Geibinger, Mischek, and Musliu 2019a). That publication features Constraint Programming models solved by the solvers Chuffed and CP Optimizer (CPLEX was also used, but showed quite poor performance for this problem). Of these, Chuffed only supports single-threaded solving, whereas CP Optimizer was run with 8 threads in parallel. For better comparability, we evaluated our model with clingcon both in single-threaded mode and in multi-thread mode with 8 threads, using the portfolio solving option (`--configuration=many`).

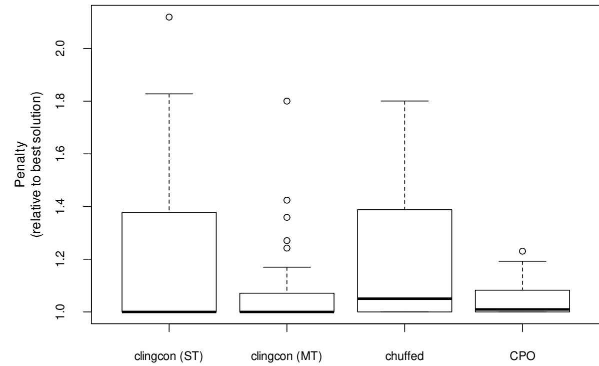

Figure 5: Performance comparison between clingcon in single-threaded (ST) or multi-threaded (MT) mode, Chuffed and CP Optimizer. The results were scaled by the best known penalty for each instance.

A comparison of the solvers' performance can be seen in Figure 5. All solvers found feasible solutions for all 33[4] instances. Both the single-threaded and the multi-threaded configuration of clingcon proved optimality for 19 instances, but the multi-threaded version found better solutions on all but one of the remaining instances. Compared to Chuffed, (single-threaded) clingcon found both more optimal solutions and slightly better penalties on average. Multi-threaded clingcon found the best solution among all four runs for 26 out of the 33 instances. On average, clingcon achieved comparable penalties to CP Optimizer, which found the best known solution for 16 instances and could only prove optimality for the four smallest instances.

Very Large Neighborhood Search

We also compare Very Large Neighborhood Search using clingcon (single-threaded) with the Very Large Neighborhood Search approach based on CP with Chuffed (Geibinger, Mischek, and Musliu 2019a). Due to the nondeterministic nature of Very Large Neighborhood Search, we performed five runs for each instance.

Our results show that while Very Large Neighborhood Search with either solver produces similar results, VLNS with clingcon finds equivalent or slightly better solutions on average for every single test instance (see Figure 6).

VLNS also produces significantly better solutions than clingcon alone, even when compared to the multi-threaded configuration. This matches the results of (Geibinger, Mischek, and Musliu 2019a).

Of particular interest is also that VLNS could prove optimality for 15 instances by finding solutions that match the precomputed lower bounds (Very Large Neighborhood Search with Chuffed proved optimality for 16 instances). However, optimality could also be shown for an additional five instances where the optimal solution is above the lower bounds. In four of those last cases, the proof of optimality

[4] Results for two of the real-world instances have not been published for CP Optimizer and are not included.

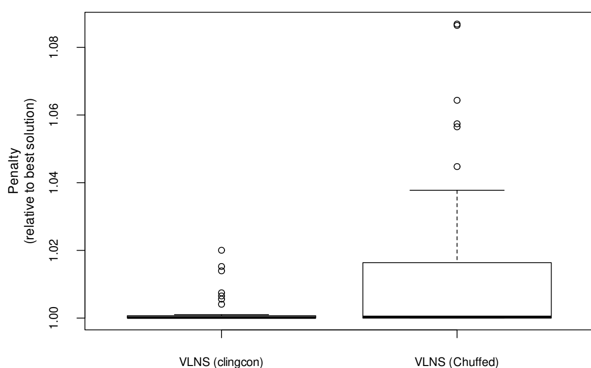

Figure 6: Performance comparison between Very Large Neighborhood Search using clingcon (single threaded) as the internal solver and Very Large Neighborhood Search using Chuffed. The figure shows average results over five runs, scaled by the best known penalty for each instance.

was obtained by increasing the neighborhood size to the total number of projects in the instance (effectively reducing the Very Large Neighborhood Search algorithm to the exact clingcon solver). The fifth instance (test instance number 22) can actually be separated in two independent subproblems, which were detected automatically and individually solved to completion by VLNS. In (Geibinger, Mischek, and Musliu 2019a), the Very Large Neighborhood Search approach based on Chuffed found the same solutions for these instances, but was unable to prove their optimality within the given time.

Exploration of such large neighborhoods became possible due to clingcons ability to quickly prove optimality of small subproblems and an aggressive strategy for increasing the neighborhood size.

Overall, we could find new best known solutions for 12 instances using Very Large Neighborhood Search based on clingcon, including for all three real-world instances, and prove the optimality of previously best known solutions for five further instances.

Conclusion

In this work, we provided an exact solution method for the real-world scheduling problem TLSP-S. We encoded the problem as a constraint answer-set program in the language of the CASP solver clingcon. To the best of our knowledge, this is the first CASP model for an extension of RCPSP in the literature.

We then experimentally compared different encoding methods and optimization strategies of the solver. In a further experiment we compared our clingcon encoding with existing CP based approaches for TLSP-S and showed that we achieve competitive results and can solve 19 out of 33 benchmark instances to optimality. Additionally, we also employed clingcon and our encoding in an existing Very Large Neighborhood Search framework and showed that this outperforms the existing best solution methods for TLSP-S (Very Large Neighborhood Search based on a CP solver). In fact, the clingcon based Very Large Neighborhood Search found new upper bounds for 12 of 33 benchmark instances, including the 3 available real-world instances.

For the future, we plan to investigate constraint answer-set programming for the full TLSP as well as utilizing the multi-shot solving capabilities of clingcon.

Acknowledgments

The financial support by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development and the Christian Doppler Research Association is gratefully acknowledged.

References

Abels, D.; Jordi, J.; Ostrowski, M.; Schaub, T.; Toletti, A.; and Wanko, P. 2020. Train Scheduling with Hybrid Answer Set Programming. *Theory and Practice of Logic Programming* 1–31.

Abseher, M.; Gebser, M.; Musliu, N.; Schaub, T.; and Woltran, S. 2016. Shift design with answer set programming. *Fundamenta Informaticae* 147(1): 1–25.

Alviano, M.; Dodaro, C.; and Maratea, M. 2017. An Advanced Answer Set Programming Encoding for Nurse Scheduling. In *Proceedings of the 16th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2017)*, volume 10640 of *LNCS*, 468–482. Springer.

Balduccini, M. 2011. Industrial-Size Scheduling with ASP+CP. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*, volume 6645 of *LNCS*, 284–296. Springer.

Banbara, M.; Kaufmann, B.; Ostrowski, M.; and Schaub, T. 2017. Clingcon: The next generation. *Theory and Practice of Logic Programming* 17(4): 408–461.

Bellenguez, O.; and Néron, E. 2005. Lower Bounds for the Multi-skill Project Scheduling Problem with Hierarchical Levels of Skills. In *Proceedings of the 5th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2005)*, volume 3616 of *LNCS*, 229–243. Springer.

Bofill, M.; Coll, J.; Suy, J.; and Villaret, M. 2020. SMT encodings for Resource-Constrained Project Scheduling Problems. *Computers & Industrial Engineering* 149: 106777.

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.

Danzinger, P.; Geibinger, T.; Mischek, F.; and Musliu, N. 2020. Solving the Test Laboratory Scheduling Problem with Variable Task Grouping. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*, 357–365. AAAI Press.

Dodaro, C.; Galatà, G.; Maratea, M.; and Porro, I. 2018. Operating Room Scheduling via Answer Set Programming. In *Proceedings of the 17th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2018)*, volume 11298 of *LNCS*, 445–459. Springer.

Dodaro, C.; and Maratea, M. 2017. Nurse Scheduling via Answer Set Programming. In *Proceedings of the 14th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2017)*, volume 10377 of *LNCS*, 301–307. Springer.

Eiter, T.; Ianni, G.; and Krennwallner, T. 2009. Answer Set Programming: A Primer. In Tessaris, S.; Franconi, E.; Eiter, T.; Gutiérrez, C.; Handschuh, S.; Rousset, M.; and Schmidt, R. A., eds., *Reasoning Web. Semantic Technologies for Information Systems*, 40–110. Springer.

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

Friedrich, G.; Frühstück, M.; Mersheeva, V.; Ryabokon, A.; Sander, M.; Starzacher, A.; and Teppan, E. 2016. Representing Production Scheduling with Constraint Answer Set Programming. In *Operations Research Proceedings 2014*, 159–165. Springer.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2012. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR* abs/1405.3694. URL arxiv.org/abs/1405.3694.

Gebser, M.; Ostrowski, M.; and Schaub, T. 2009. Constraint Answer Set Solving. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *LNCS*, 235–249. Springer.

Geibinger, T. 2020. *Investigating Constraint Programming and Hybrid Answer-set Solving for Industrial Test Lab oratory Scheduling*. Master's thesis, Technische Universität Wien, Institut for Logic and Computation.

Geibinger, T.; Mischek, F.; and Musliu, N. 2019a. Investigating Constraint Programming and Hybrid Methods for Real World Industrial Test Laboratory Scheduling. *CoRR* abs/1911.04766 (Submitted to journal). URL arxiv.org/abs/1911.04766.

Geibinger, T.; Mischek, F.; and Musliu, N. 2019b. Investigating Constraint Programming for Real World Industrial Test Laboratory Scheduling. In *Proceedings of the 16th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research (CPAIOR 2019)*, volume 11494 of *LNCS*, 304–319. Springer.

Gelfond, M.; and Lifschitz, V. 1988. The Stable Model Semantics For Logic Programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP 1988)*, 1070–1080. MIT Press.

Gelfond, M.; and Lifschitz, V. 1991. Classical negation in logic programs and disjunctive databases. *New Generation Computing* 9: 365–385.

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.

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*, 445–490. Springer.

Mischek, F.; and Musliu, N. 2018a. A Local Search Framework for Industrial Test Laboratory Scheduling. In *Proceedings of the 12th International Conference on the Practice and Theory of Automated Timetabling (PATAT 2018)*, 465–467.

Mischek, F.; and Musliu, N. 2018b. 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 Annals of Operations Research* URL dbai.tuwien.ac.at/staff/fmischek/TLSP/publications/Mischek2019AOR.pdf.

Schutt, A.; Feydy, T.; Stuckey, P. J.; and Wallace, M. G. 2013. Solving RCPSP/max by lazy clause generation. *Journal of Scheduling* 16(3): 273–289.

Szeredi, R.; and Schutt, A. 2016. Modelling and Solving Multi-mode Resource-Constrained Project Scheduling. In *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, volume 9892 of *LNCS*, 483–492. Springer.

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 *Proceedings of the 23rd International Conference on Principles and Practice of Constraint Programming (CP 2017)*, volume 10416 of *LNCS*, 308–317.