Informatics

Investigating Constraint Programming and Hybrid Answer-set Solving for Industrial Test Laboratory Scheduling

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Logic and Computation

eingereicht von

Tobias Geibinger, BSc
Matrikelnummer 01427138

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Priv.-Doz. Dr. Nysret Musliu

Wien, 26. November 2020

Tobias Geibinger Nysret Musliu

Informatics

Investigating Constraint Programming and Hybrid Answer-set Solving for Industrial Test Laboratory Scheduling

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Logic and Computation

by

Tobias Geibinger, BSc
Registration Number 01427138

to the Faculty of Informatics

at the TU Wien

Advisor: Priv.-Doz. Dr. Nysret Musliu

Vienna, 26th November, 2020

Tobias Geibinger	Nysret Musliu

Erklärung zur Verfassung der Arbeit

Tobias Geibinger, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 26. November 2020

Tobias Geibinger

Acknowledgements

First, I would like to thank my advisor Priv.Doz. Dr. Nysret Musliu for his guidance through this thesis and giving me the opportunity to do this kind of research. I would also like to express my gratitude to Dipl. Ing. Florian Mischek, who was my colleague during my work on this thesis and whose input and insight was invaluable. Finally, I also want to acknowledge the support of my family, who enabled my education and fostered my desire to do academic research.

The work was supported by the Austrian Federal Ministry for Digital and Economic Affairs and the National Foundation for Research, Technology and Development through the Christian Doppler Laboratory for Artificial Intelligence and Optimization for Planning and Scheduling.

Kurzfassung

In dieser Diplomarbeit betrachten wir ein real auftretendes, komplexes Projektplanungsproblem. Dieses Planungsproblem ist eine Erweiterung des bekannten Resource-Constrained Project Scheduling Problem (RCPSP) und tritt in industriellen Testlaboren auf, bei denen eine große Anzahl an Tests durch qualifiziertes Personal und unter Einhaltung von Fristen und anderen Einschränkungen durchgeführt werden müssen.

Wir untersuchen exakte Methoden für dieses Problem basierend auf Constraint Programming und Answer-set Solving. Unter anderem schlagen wir verschiede Constraint Programming Modelle vor, welche zum Teil auf bestehenden Ideen aus der Literatur aufbauen, aber für die komplexen Constraints unseres Problems erweitert wurden. Außerdem, stellen wir neuartige redundante Constraints vor und nutzen verschiedene Suchstrategien.

Darüber hinaus zeigen wir, wie dieses Projektplanungsproblem mit Hybrid Answer-set Programming gelöst werden kann. Im Speziellen stellen wir ein Encoding für den Constraint Answer-set Solver clingcon vor und untersuchen verschiedene Modellierungsoptionen und Konfigurationen.

Wir liefern außerdem einen Very Large Neighborhood Search (VLNS) Ansatz, welcher auf unseren exakten Methoden basiert und die Qualität, der von diesen Methoden gefundenen Lösungen, drastisch verbessert.

Unsere Lösungsansätze werden im Anschluss empirisch evaluiert, sowohl mit echten Daten als auch mit existierenden, zufällig generierten Benchmark-Instanzen von verschiedenen Größen. Zusätzlich vergleichen wir unsere Methoden mit einem state-of-the-art metaheuristischen Ansatz für dieses Problem und einem Mixed-Integer Programming Solver. Unsere Methoden konnten gültige Lösungen für alle 31 Testinstanzen finden, und für 22 sogar optimale Lösungen. Wir zeigen außerdem, dass VLNS die momentan besten Ergebnisse für dieses Problem liefert. Zusätzlich zeigen wir, dass VLNS für 30 der 31 Instanzen Lösungen findet, welche im Bereich von bis zu 5% des Optimums liegen.

Abstract

In this thesis we deal with a complex real-world scheduling problem closely related to the well-known Resource-Constrained Project Scheduling Problem (RCPSP). The problem concerns industrial test laboratories in which a large number of tests has to be performed by qualified personnel using specialized equipment, while respecting deadlines and other constraints.

We investigate exact approaches based on Constraint Programming and Answer-set Programming for this problem. First, we propose various Constraint Programming models based on existing approaches for the related problems and introduce new encodings for the complex constraints of the problem. Additionally, we provide novel redundant constraints and analyze the impact of various search strategies.

Furthermore, we show how we can solve this problem by Answer-set Programming extended with ideas from constraint solving. We propose an innovative and efficient encoding, apply an answer-set solver for constraint logic programs called clingcon, and investigate different modeling options and solver configurations.

We also propose a Very Large Neighborhood Search (VLNS) approach which exploits our exact methods and manages to drastically improve the quality of the solutions found by these methods.

Our solution approaches are empirically evaluated both on real-world data and existing randomly generated benchmark instances of different sizes. Additionally, we compare our methods with a state-of-the-art metaheuristic approach for this problem and a Mixed-Integer Programming solver. Our approaches provide feasible solutions for all 31 instances and for 22 we could find optimal solutions. The VLNS approach currently yields the best known solutions for this problem outperforming existing metaheuristic approaches. In fact, we show that VLNS finds solutions which are within 5% of the optimum for 30 out of the 31 instances.

Finally, the solution methods described in this thesis are currently in use in a real-world industrial test laboratory.

Contents

CHAPTER 1

Introduction

Project scheduling includes various problems of high practical relevance. Such problems arise in many areas and include different constraints and objectives. Usually project scheduling problems require scheduling of a set of project activities over a period of time and assignment of resources to these activities. Typical constraints include time windows for activities, precedence constraints between the activities, assignment of appropriate resources etc. The aim is to find feasible schedules that optimize several criteria such as the minimization of total completion time.

In this thesis, we investigate solving a real-world project scheduling problem that arises in an industrial test laboratory of a large company. This problem, Industrial Test Laboratory Scheduling (TLSP), which is an extension of the well known Resource-Constrained Project Scheduling Problem (RCPSP), was originally described in [MM18b, MM18a]. It consists of a grouping stage, where smaller activities (tasks) are grouped into larger jobs, and a scheduling stage, where these jobs are scheduled and have resources assigned to them. Here, we deal with the second stage and assume that a grouping of tasks into jobs is already provided. Since we focus on the scheduling part, we denote the resulting problem as TLSP-S.

The investigated problem has several features of previous project scheduling problems in the literature, but also includes some specific features imposed by the real-world situation, which have rarely been studied before. Among others, those include heterogeneous resources with availability restrictions on the activities each unit of a resource can perform. While work using similar restrictions exists ([DPRL98, YFS17]), most problem formulations either assume homogeneous, identical units of each resource or introduce additional activity modes for each feasible assignment, which quickly becomes impractical for higher resource requirements and multiple resources. Another specific feature of TLSP(-S) is that of linked activities, which require identical assignments on a subset of the resources. To the best of our knowledge, a similar concept appears only in [SSD97], where modes should be identical over subsets of all activities. We also deal with several non-standard objectives instead of the usual makespan minimization, which arise from various business objectives of our industrial partner. Most notably, we try to minimize the total completion time of each project, i.e. the time between the start of the first and the end of the last job in the project.

Mischek and Musliu [MM19] introduced a local search framework for TLSP-S. They investigate several metaheuristics out of which Simulated Annealing (SA) is shown to perform best. However, no exact methods for TLSP-S are known yet. Moreover, for many instances optimal solutions are not known and finding them has immense impact in practice.

In practice, exact solutions for this problem are desired especially in situations where it is necessary to check if a feasible solution exists at all. In the application that we consider, checking quickly if activities of additional projects can be added on top of an existing schedule is very important. Although it is known from previous papers [SS16, YFS17] that constraint programming techniques can give good results for related project scheduling problems, it is an interesting research question if Constraint Programming (CP) techniques can also solve TLSP-S that includes additional features and larger instances. We also investigate Constraint Answer-set Programming (CASP) for solving TLSP-S, which has so far not been used in project scheduling.

The existence of exact methods clears the path for the development of a Very Large Neighborhood Search (VLNS) approach.

1.1 Aims of the thesis

The goals of this work are the following:

- Investigate Constraint Programming and Constraint Answer-set Programming for TLSP-S and thus provide novel exact methods for this problem.
- Utilize these exact methods in a Very Large Neighborhood Search in order to provide high quality solutions for large problem instances.
- Empirically evaluate our proposed solution methods and compare them to existing approaches on randomly generated benchmark instances and real-world data.

1.2 Contributions

The main contributions of this thesis are:

- We give a basic CP model of the hard constraints of TLSP-S, review different options to model the most difficult constraints, and provide novel redundant constraints which improve solving performance.

- We propose an alternative CP model based on interval variables.
- We provide a Constraint Answer-set Solving approach for TLSP-S and again provide various encoding options.
- We introduce a Very Large Neighborhood search utilizing the provided exact methods.
- An evaluation of the methods proposed in this thesis is given using 30 randomly generated instances and 1 real-life instance. The solution approaches discussed in this thesis achieve the currently best known results for these instances.
- The solution methods described in this work are currently being employed in a real-world industrial test laboratory.

Some of the result of this thesis have been already published at the CPAIOR conference [GMM19].

1.3 Structure of the thesis

The rest of this thesis is structured as follows: In Chapter 2 we describe the theoretical background of the solution approaches used in this work. A formal definition of TLSP-S, basic complexity analysis, and a review of related work is given in Chapter 3. Afterwards, we give different Constraint Programming formulations for TLSP-S in Chapter 4 and a Constraint Answer-set Programming encoding in Chapter 5. Our Very Large Neighborhood Search approach is described in Chapter 6, while Chapter 7 provides the empirical evaluation and comparison of the solutions methods developed in this thesis. Finally, we discuss our findings in Chapter 8.

CHAPTER 2

Theoretical Background

In this chapter, an overview of the solving paradigms applied in this thesis will be given. In particular, the main concepts of Constraint Programming and Constraint Answer-set Programming will be discussed. Furthermore, we will give a short introduction into Very Large Neighborhood Search techniques.

2.1 Constraint Programming

Constraint Programming (CP) is a solving paradigm which allows for the specification of constraints in a declarative manner and thus independent from underlying solving mechanisms. CP is used for solving *constraint satisfaction problems* or CSP for short. They were originally introduced by Montanari [Mon74] and Mackworth [Mac77] and are formally defined as follows [RvBW06].

A CSP is a triple $\mathcal{P} = \langle V, D, C\rangle$ where $V = \langle v_1, \dots, v_n\rangle$ is an n-tuple of *variables* (also called decision variables), $D = \langle D_1, \dots, D_n\rangle$ is an n-tuple of sets called *domains*, and C is a set of *constraints*. Each constraint $c \in C$ has a $scope(c)$ which is a tuple of variables from V. Furthermore, a constraint also consists of a $relation(c)$ over the domains of the variables in its scope. A *variable assignment* for a CSP is an n-tuple $A = \langle d_1, \dots, d_n\rangle$ where $d_i \in D_i$. We say that an assignment satisfies a constraint $c \in C$ if the projection of A onto $scope(c)$ is an element of $relation(c)$. An assignment that satisfies all constraints is called a *solution*. We often use variables and their respective assignment synonymously if it is clear from context.

The problem of checking whether or not a given CSP has a solution is well-known to be NP-complete. However, depending on the use case we might also want to find all solutions of a problem or a solution which is deemed optimal. *Constraint optimization problems* (COP) allow us to formulate such optimization problems. A COP $\mathcal{P}$ is a CSP which was extended with an *objective function* $\sigma : sol(\mathcal{P}) \to \mathbb{R}$ where $sol(\mathcal{P})$ is the set of solutions of $\mathcal{P}$. If $\mathcal{P}$ is a *minimization problem*, then a solution S of $\mathcal{P}$ is called *optimal* whenever for any other solution S', $\sigma(S') \geq \sigma(S)$. Optimality for *maximization problems* is defined analogously.

CSPs and COPs can, for example, be solved using constraint propagation and backtracking search. In a nutshell, the approach works by picking a variable v_i and assigning it a domain value $d_i \in D_i$. This can be seen as changing the domain D_i of v_i and for each constraint $c \in C$ with v_i is its scope, we can propagate this change to the domains of the other variables in *scope*(c). One way to propagate these changes would be to remove all values that do not correspond to tuples in the relations. However, different types of propagation exist. Whenever the domain of a variable becomes empty, we have found a partial assignment which cannot be extended to a solution. In such a case we backtrack our variable assignments. In practice, there exist sophisticated heuristics for variable and value selection and also for backtracking strategy. For a more comprehensive introduction to CP solving we refer to the Handbook of Constraint Programming [RvBW06].

Later on in our CP models, most of the constraints will be boolean or arithmetical, and their meaning should be clear. However, we will also use so called *global constraints* [vHK06]. Such constraints specify a certain relation over the variables and are usually (but not always) reducible to basic boolean and arithmetic relations. The motivation behind using global constraints is having a shorthand for complex constraints but most solvers implement specialized propagation strategies for global constraints. These propagators help reduce the search space and can speed up the solution process significantly.

In Chapter 4 we are going to give models in two distinct CP modeling languages. One of them is the solver-independent modeling language *MiniZinc* [NSB$^+$07]. This language supports basic boolean and arithmetical constraints as well as integer and boolean decision variables. Furthermore, it supports a variety of global constraints. We are going to utilize *cumulative* [AB93] and *global cardinality* [Ŕ96] constraints. Formally, a cumulative constraint is of the form `cumulative`(S, D, R, b) where S, D, and R are n-tuples of integer decision variables and b is an integer constant. To illustrate the semantics of the constraint suppose we have n activities with start times $S = \langle s_1, \ldots, s_n \rangle$, durations $D = \langle d_1, \ldots, d_n \rangle$, and resource usages $R = \langle r_1, \ldots, r_n \rangle$. Let $\mathcal{A}_t = \{i \in \mathbb{N}^+ \mid i \leq n,\ s_i \leq t \leq s_i + d_i\}$ be the set of activities active at time point t. We say that the cumulative constraint above is satisfied, if for every time point t it holds that $\sum_{i \in \mathcal{A}_t} r_i \leq b$.

There are a number of global cardinality constraints in MiniZinc but we are going use ones of the form `gcc_low_up`(V, c, l, u) where $V = \langle v_1, \ldots, v_n \rangle$ is an n-tuple of decision variables, c is a domain element, and l, u are constant non-negative integers. Let $o = |\{i \in \mathbb{N}^+ \mid i \leq n,\ v_i = c\}|$ be the number of occurrences of the domain value c in the assignment of variables V, then the constraint is satisfied if $l \leq o \leq u$.

The other CP language we are going to use, is the one of *IBM CP Optimizer* [IC17a]. In our CP Optimizer model, all the decision variables are (possibly optional) intervals. Internally, intervals contain a start and an end time and when declaring an interval we can directly constrain their domains. The keywords `startOf`(a) and `endOf`(a) can be used to obtain the start and respectively end of an interval a and `lengthOf`(a) returns its length. Furthermore, it is also possible to fix the size of an interval variable at declaration. Optional intervals are not necessarily present in a solution unless some constraints require it. Intervals which are not present are ignored by other constraints. For example, the constraint `noOverlap`(I), where I is a tuple of intervals, ensures that only the present intervals of I do not overlap. The presence (or non-presence) of an interval a can be enforced using the boolean function `presenceOf`(a) which is true whenever a is present and an appropriate constraint. The constraint `alternative`(a, I, n), where a is an interval, I a tuple of intervals and n is a natural number, can also affect the presence of intervals. This constraint is true if whenever a is present, also exactly n intervals from I are present as well. Further CP Optimizer constraints we use are `endBeforeStart`(a_1, a_2) enforcing that interval a_1 ends before the start of interval a_2, and `span`(a, I) which constrains the interval a to start at the earliest start and end at the lastest end of any interval in I.

We can also formulate cumulative constraints in CP Optimizer in the following way: $\sum_{i\leq n}$ `pulse`$(a_i, r_i) \leq b$, where $a_i, \ldots, a_n$ are intervals, $r_i, \ldots, r_n$ their resource usages, and b is the resource bound.

For a complete formal definition of the CP Optimizer language and all the supported constraints, we refer to the article by Laborie et al. [LRSV18].

2.2 Constraint Answer-set Programming

First, we give a short introduction into Answer-set Programming (ASP) [EIK09].

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*.

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

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, \sim b_{k+1}, \ldots, \sim b_m, \tag{2.1}$$

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 (2.1) 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 [GL91], 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* [GL88] 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* [GKKS14], support *choice rules* i.e. rules of the form $l \ \{a_1 \ ; \ \dots \ ; \ a_n \ ; \ \dots \ ; \ \sim a_{n+1} \ ; \ \dots \ ; \ \sim a_o\} \ u \leftarrow b_1, \dots, b_k, \sim b_{k+1}, \dots, \sim b_m$, 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 $l \ \{L_0 : L_1, \dots, L_n\} \ u$ 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 $L_1, \dots,$ 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 $\#count\{v_1, \dots, v_n : L_1, \dots, L_m\} = b$, 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 $\#count\{c^1_1, \dots, c^1_n : l^1_1, \dots, l^1_m \ ; \ \dots \ ; \ c^k_1, \dots, c^k_n : l^k_1, \dots, l^k_m\} = b$, where $l^i_1, \dots, l^i_m$ $(i \leq k)$ are ground instances of the literals and $c^i_1, \dots, c^i_n$ are the respective constants instantiated for variables $v_1, \dots, v_n$. An aggregate is satisfied by an interpretation I if the number of tuples $\langle c^i_1, \dots, c^i_n \rangle$ where $l^k_1, \dots, l^k_m$ 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 [EIK09, GKKS12, GKKS14].

Constraint Answer-set Programming (CASP) extends ASP with linear variables $V = \langle v_1, \dots, v_n \rangle$ over domains $D = \langle D_1, \dots, D_n \rangle$ and linear constraints. In *clingcon* [BKOS17] – 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 as defined for CSPs in the previous section. 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.

Similarly to COPs, 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 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. [BKOS17].

2.3 Very Large Neighborhood Search

Very Large Neighborhood Search (VLNS) [PR10] or sometimes just *Large Neighborhood Search* is a metaheuristic [GP$^+$10] introduced by Shaw [Sha98]. The main idea behind VLNS is a continuous *destroy and repair* approach. We begin with an initial solution. The destroy method then removes part of this solution and the repair method completes the solution again. The repairing should aim to fix the destroyed part in the optimal way i.e. out of all possible repairs it should select the best. Most of the time, this process is repeated until some criterion is met. Which parts of the solution get destroyed is of course highly problem specific. Often, the repair method is implemented through a general purpose solving paradigm like MIP or CP.

CHAPTER 3

Problem Definition and Related Work

In this chapter, we are going to give a formal definition of the problem investigated in this thesis and an overview of related work.

3.1 Problem Definition

As mentioned in the introduction, we deal with a variant of TLSP which was recently introduced by Mischek and Musliu [MM18b, MM18a, MM19], where we assume that a grouping of tasks into jobs is already provided for each project, and focus on the scheduling part of the problem instead (TLSP-S). Thus, the goal is to find an assignment of a mode, time slot and resources to each (given) job, such that all hard constraints are fulfilled and the number of violations of soft constraints is minimized.

The problem description below captures TLSP-S and is based on one of TLSP [MM18b, MM18a, MM19].

3.1.1 Parameters

Each instance consists of a scheduling period of h discrete *time slots*. The set of all time slots is denoted by $T = \{0, \ldots, h\}$. Further, it lists resources of different kinds:

- *Employees* $E = \{1, \ldots, |E|\}$ who are qualified for different types of jobs.
- A number of *workbenches* $B = \{1, \ldots, |B|\}$ with different facilities.
- Various auxiliary lab *equipment* groups $G_g = \{1, \ldots, |G_g|\}$, where g is the group index. These each represent a set of similar devices. The set of all equipment groups is denoted G^*.

Furthermore, we have given the set of *projects* labeled $P = \{1, \dots, |P|\}$, and the set of *jobs* to be scheduled $J = \{1, \dots, |J|\}$. For a project p, the jobs of this project are given as $J_p \subseteq J$.

Each job j has several properties[1]:

- A time window, given via a *release date* α_j and a *deadline* ω_j. In addition, it has a *due date* $\bar{\omega}_j$, which is similar to the deadline, except that exceeding it is only a soft constraint violation.
- A set of *available modes* $M_j \subseteq M$, where M is the set of all modes.
- A *duration* d_{mj} for each available mode $m \in M_j$.
- The resource requirements for the job:
  - The number of *required employees* r_m^{Em} depends on the mode $m \in M_j$. Each of these employees must be chosen from the set of *qualified employees* $E_j \subseteq E$. Additionally, there is also a set of *preferred employees* $E_j^{Pr} \subseteq E_j$.
  - The number of *required workbenches* $r_j^{Wb} \in \{0, 1\}$. If a workbench is required, it must be chosen from the *available workbenches* $B_j \subseteq B$.
  - For each equipment group $g \in G^*$, the job requires r_{gj}^{Eq} devices, which must be taken from the set of *available devices* $G_{gj} \subseteq G_g$ for the group.
- The *predecessors* $\mathcal{P}_j$ 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* L_j of this job. All linked jobs must be performed by the same employee(s). As before, such links only occur between jobs of the same project.

Out of all jobs, a subset are *started jobs* $J^S \subseteq J$, which are considered already being worked on.

3.1.2 Solution

In general, a solution for an instance of TLSP-S needs to contain the following assignments for each job $j \in J$:

- $\dot{t}_j^s \in T$ its starting timeslot,
- $\dot{t}_j^c \in T$ its completion time,
- $\dot{m}_j \in M$ the mode which it is assigned,

[1]In TLSP, these are derived from the tasks contained within a job. Since we assume the distribution of tasks into jobs to be fixed, they can be given directly as part of the input for TLSP-S.

- $\dot{b}_j \in B$ the assigned workbench or 0 if none is assigned,
- $\dot{E}_j \subseteq E$ its set of assigned employees, and
- for each equipment group $g \in G^*$, the set $\dot{G}_{jg} \subseteq G_g$ denotes the assigned equipment to j from group g.

3.1.3 Constraints

We now give a formal account of all hard and soft constraints contained in TLSP-S. For a full account of all constraints of TLSP we refer to the technical report by Mischek and Musliu [MM18b]. To simplify notation, we introduce the set of *active jobs* at timeslot t denoted by $\mathcal{J}_t = \{j \in J \mid \dot{t}^s_j \leq s,\ \dot{t}^c_j > t\}$.

Hard Constraints

Job duration The difference between a jobs start time and completion time has to match its duration.

$$\dot{t}^s_j - \dot{t}^c_j = d_{\dot{m}_j j} \qquad j \in J \qquad (3.1)$$

Time window Each job has to be performed within its time window i.e. it has to start after its release time and complete before its deadline.

$$\alpha_j \leq \dot{t}^s_j \ \wedge\ \dot{t}^c_j \leq \omega_j \qquad j \in J \qquad (3.2)$$

Job precedence The predecessors of a job have to be completed before it can start.

$$\dot{t}^c_k \leq \dot{t}^s_j \qquad j \in J,\ k \in \mathcal{P}_j \qquad (3.3)$$

Started jobs Jobs which are already started have to start at the first timeslot.

$$\dot{t}^s_j = 0 \qquad j \in J^S \qquad (3.4)$$

Single assignment Any resource i.e. workbench, employee or equipment cannot be used two or more jobs simultaneously.

$$|\{j \in \mathcal{J}_t \mid \dot{b}_j = b\}| \leq 1 \qquad b \in B,\ t \in T \qquad (3.5)$$
$$|\{j \in \mathcal{J}_t \mid e \in \dot{E}_j\}| \leq 1 \qquad e \in E,\ t \in T \qquad (3.6)$$
$$|\{j \in \mathcal{J}_t \mid d \in \dot{G}_{jg}\}| \leq 1 \qquad g \in G^*,\ d \in G_g,\ t \in T \qquad (3.7)$$

Workbench requirements Each job which requires a workbench has one assigned.

$$r^{Wb}_j = 1 \ \leftrightarrow\ \dot{b}_j \neq 0 \qquad j \in J \qquad (3.8)$$

Employee requirements A job has exactly as many employees assigned as its mode requires.

$$|\dot{E}_j| = r^{Em}_{\dot{m}_j} \qquad j \in J \tag{3.9}$$

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

$$|\dot{G}_{jg}| = r^{Eq}_{gj} \qquad j \in J,\ g \in G^* \tag{3.10}$$

Workbench availability The assigned workbench of a job has to be available for said job.

$$\dot{b}_j \neq 0 \rightarrow \dot{b}_j \in B_j \qquad j \in J \tag{3.11}$$

Employee qualification The employees assigned to a job need to be qualified.

$$\dot{E}_j \subseteq E_j \qquad j \in J \tag{3.12}$$

Equipment availability The equipment assigned to a job has to be available for said job.

$$\dot{G}_{jg} \subseteq G_{jg} \qquad j \in J,\ g \in G^* \tag{3.13}$$

Linked jobs Jobs which are linked have to be performed by the same employees.

$$\dot{E}_j = \dot{E}_k \qquad j \in J,\ k \in L_j \tag{3.14}$$

Soft Constraints

Solutions of TLSP-S which satisfy the hard constraints defined above are further compared by their quality. The quality of a solution is determined by the penalty induced by the following soft constraints where lower penalty is better. Each soft constraint is a sum expressing the number of violations of this constraint. The total penalty of a solution is then simply the weighted sum of all soft constraints. The weights for each soft constraint are currently being determined in correspondence with a real-world test laboratory. These weights may also be subject to change depending on the currents needs of the laboratory or the people involved in the planning. For this work, all weights $w_1, \ldots, w_5$ are considered to be 1 as they were in previous work [MM19].

In TLSP, the first soft constraint depends on the number of jobs. Since we consider the job grouping fixed in TLSP-S, the number of jobs is fixed as well and we consider $w_1 \cdot |J|$ a constant penalty in the objective value.

Employee project preferences Each employee assigned to job while not being preferred will lead to a penalty.

$$w_2 \cdot \sum_{j \in J} |\{e \in \dot{E}_j \mid e \notin E_j^{Pr}\}| \tag{3.15}$$

Number of employees Each employee assigned to a project will lead to a penalty. Hence, we want to minimize the number of different employees in each project.

$$w_3 \cdot \sum_{p \in P} |\bigcup_{j \in J_p} \dot{E}_j| \tag{3.16}$$

Due date For each job, the amount which it exceeds its due date will be registered as a penalty.

$$w_4 \cdot \sum_{j \in J} max(\dot{t}_j^c - \bar{\omega}_j, 0) \tag{3.17}$$

Project completion time The project completion time for each project i.e. the time between the earliest start time and the latest end time of all jobs in the project should be kept as minimal as possible.

$$w_5 \cdot \sum_{p \in P} max(\{\dot{t}_j^c \mid j \in J_p\}) - min(\{\dot{t}_j^s \mid j \in J_p\}) \tag{3.18}$$

3.2 Complexity

We are now going to present complexity results for the problem considered in this thesis. Similar results were already shown for TLSP in the article by Mischek and Musliu [MM19] where they also gave the outline of the proofs formalized below.

TLSP-S as described above is of course an optimization problem. In order to analyze its complexity we specify a corresponding decision problem i.e. a problem with a yes or no answer. In general, the decision variants for optimization problems are obtained by extending the problem with a parameterized bound on the objective value. The decision problem would then be whether or not the optimization problem has a solution with an objective value less or equal to the given bound.

Using this approach the corresponding decision problem for TLSP-S can be obtained by adding a parameter B – which specifies the bound on the objective – and a constraint ensuring that the total penalty is less or equal to B. We shall denote this decision problem as TSLP-S(D).

The first result we want to show regards the hardness of TLSP-S(D). We will show that the problem is NP-hard. We will achieve this by showing that the subproblem of finding a feasible solution for a TLSP-S instance is already NP-hard.

Theorem 1. *Checking whether an instance of TLSP-S is feasible i.e. has a solution is an NP-hard problem.*

Proof. We will show our claim by reduction from the decision variant of the Resource Constrained Project Scheduling Problem (RCPSP) which is known to be NP-hard [GJ79].

Formally, an instance of RCPSP is defined as follows. We have n *activities* represented by the set $V = \{0, \dots, n+1\}$ which also includes auxiliary start and end activities. Each activity $a \in V$ has a *duration* p_a ($p_0 = p_{n+1} = 0$). We also have a binary relation $Q \subseteq V \times V$ which defines *precedences* among activities. Note that $Q(a, 0)$ and $Q(n+1, a)$ do not hold for any $a \in V$, but $Q(0, a)$ and $Q(a, n+1)$ hold for all $a \in V \setminus \{0, n+1\}$. Furthermore, an RCPSP instance also includes m *renewable resources* represented by the set $R = \{1, \dots, m\}$ each $r \in R$ with an *availability* c_r. For each activity $a \in V$ and resource $r \in R$, we have a *demand* b_{ar}. Lastly, we have a *makespan* C_{max}.

A solution to an RCPSP instance as defined above is given by a start time assignment s_i ($0 \le i \le n+1$) such that the following constraints are satisified, where $\mathcal{A}_t = \{a \in V \mid s_a \le t < s_a + p_a\}$ is the set of active activities at time slot t.

$$s_0 = 0 \tag{3.19}$$

$$s_j \ge s_i + p_i \qquad (i, j) \in Q \tag{3.20}$$

$$\sum_{a \in \mathcal{A}_t} b_{ar} \le c_r \qquad r \in R,\ 0 \le t \le C_{max} \tag{3.21}$$

$$s_{n+1} = C_{max} \tag{3.22}$$

The first constraint (3.19) ensures that the auxiliary start activity begins at the first timeslot zero. The second constraint (3.20) enforces the given precedence relation and constraint (3.21) the resource availabilities. The last constraint (3.22) requires the end activity to start – and thus end – at the makespan. We say that an instance of RCPSP is a positive instance if it has a solution.

Now, we can define a many-one reduction f which takes an instance A of RCPSP and transforms it into an instance of TLSP-S which is feasible iff A is a positive instance of RCPSP.

Let A be an arbitrary instance of RCPSP with the components described above. We want $f(A)$ to be an instance of TLSP-S. The set of time slots will simply be $T = \{0, \dots, C_{max}\}$, the set of jobs $J = V$, the projects are a singleton set $P = \{1\}$ and $J_1 = J$, and the sets of employees and workbenches are the empty set $E = W = \emptyset$. The set of modes of $f(A)$ is also going to be a singleton set $M = \{1\}$, $M_j = M$ for all $j \in J$, and $r_1^{Em} = 0$. Furthermore, the equipment groups correspond the resources in RCPSP. Hence, $G^* = R$ and $G_g = \{1, \dots, c_r\}$ for each $r \in G^* = R$.

Now, we come to the job properties. The duration of a job j in the only possible mode $m = 1$ is $d_{mj} = p_j$. The predecessors of a job j are given by $\mathcal{P}_j = \{k \in V \mid (k, j) \in Q\}$. For each equipment group g, we set the requirements of a job j to $r_{gj}^{Eq} = b_{gj}$ and $G_{gj} = G_g$.

For any job j except the auxiliary end job, we set its release date α_j to 0 and its deadline ω_j and due date $\bar{\omega}_j$ to C_{max}. For the end job we set all three values to C_{max}. The set of linked jobs, preferred employees, available employees and workbenches are all the empty set for each job j. Additionally, $r_j^{Wb} = 0$ for each job j. Lastly, $J^S = \{0\}$ is the set of started jobs. It can easily be seen that the reduction f only needs polynomial time.

We need to show that A is a positive instance of RCPSP iff $f(A)$ is a feasible instance of TLSP-S. We start with the only-if direction. Hence, we will prove that if A is a positive instance, then $f(A)$ is feasible. Since A is a positive instance, there is a start time assignment s_a for each activity $a \in V$ such that the constraints (3.19–3.22) are satisfied. We are going to transform this RCPSP solution into a feasible solution of the TLSP-S instance $f(A)$.

By construction, we have $J = V$. Hence, we are given the starting timeslot of each job $j \in J$ by $\dot{t}_j^s = s_j$. Since there only is one mode, $\dot{m}_j = 1$ for each $j \in J$ and thus also $\dot{t}_j^c = s_j + d_{1j}$. The assigned workbench and employees of a job j are simply $\dot{b}_j = 0$ and $\dot{E}_j = \emptyset$.

Now, the equipment assignment for a group $g \in G^*$ is a bit more involved. We are going to construct the assignment in m stages where $m = |J| = |V|$. Each stage will add the equipment assignment for a job and we will go through them in the ascending order of their start times $\dot{t}_j^s$. We going to use the auxiliary set $\mathcal{E}_t^g = \{e \in G_g \mid j \in \mathcal{J}_t, e \notin G_{jg}\}$ representing the yet unassigned equipment at time slot t.

Now, at each stage j we get the equipment assignment G_{jg} by taking the first $r_{gj}^{Eq} = b_{gj}$ elements of $\mathcal{E}_{\dot{t}_j^s}^g$. Suppose that at some stage j, the construction fails because $|\mathcal{E}_{\dot{t}_j^s}^g| < r_{gj}^{Eq}$. Then there exists a time slot $t = \dot{t}_j^s$ such that $\sum_{j \in \mathcal{J}_t} r_{gj}^{Eq} > |G_g|$ which implies $\sum_{j \in \mathcal{A}_t} b_{rj} > c_r$ where $r = g$ and $r \in R$. The latter in turn implies that constraint (3.21) is violated contradicting that A is a positive instance of RCPSP. Hence, this case cannot occur.

It is not hard so see that the above construction ensures that TLSP-S single assignment constraint for equipment (3.7) is satisfied as well the requirement constraint (3.10). The satisfaction of job precedence constraint (3.3) is implied by RCPSP constraint (3.20) and the remaining constraints hold by construction. Hence, we have a feasible solution for TLSP-S instance $f(A)$.

Remains to show the if-direction i.e. whenever $f(A)$ is a feasible TLSP-S instance, then A is a positive instance of RCPSP. We again show this by describing how a solution to $f(A)$ can be transformed to a solution of A. The RCPSP solution will simply be $s_a = \dot{t}_a^s$ for each activity/job $a \in J = V$. Clearly, since TLSP-S constraint (3.4) is satisfied, the RCPSP constraint (3.19) holds as well. Similarly, (3.22) has to be satisfied as for the end activity $n+1$, $\alpha_{n+1} = \omega_{n+1} = C_{max}$, $d_{1(n+1)} = p_{n+1} = 0$, and TLSP-S constraint (3.2) holds.

A violation of RCPSP constraint (3.20) would imply a violation of the corresponding TLSP-S constraint (3.3), so this cannot be the case. Lastly, the resource bounds (3.21) cannot be exceeded as the total number of equipment $|G_g|$ for some resource $r = g$, is exactly c_r by construction.

Therefore, A is a positive instance of RCPSP iff $f(A)$ is a feasible instance of TLSP-S. Finding a feasible solution for a TLSP-S instance is thus NP-hard. □

The following corollary trivially follows from the theorem above.

Corollary 1. *TLSP-S(D) is NP-hard.*

Now that we have a lower bound for the complexity of TLSP-S(D), we would like to determine an upper bound.

Theorem 2. *TLSP-S(D) is in NP.*

Proof. We show NP-membership of TLSP-S(D) by presenting a guess and check algorithm. The guess part consists of guessing $\dot{t}^s_j$, $\dot{t}^c_j$, $\dot{m}_j$, $\dot{b}_j$, $\dot{E}_j$, and $\dot{G}_{jp}$ for each job j.

We can then easily check the constraints (3.1–3.4) and (3.8–3.14) for each job and constraints (3.5–3.7) for each time slot. Similarly, we check whether the total objective is below the given bound B. Obviously, the complete check only needs polynomial time. Hence, TLSP-S(D) is in NP. □

Since we both have NP-hardness and membership, the next result follows immediately.

Corollary 2. *TLSP-S(D) is NP-complete.*

Unless P = NP, it is thus not possible to solve TLSP-S in polynomial time. This further motivates the solution approaches discussed in this thesis as they effectively reduce the problem to other formalisms which are all at least NP-hard.

3.3 Related Work

The Test Laboratory Scheduling Problem (TLSP) as well as the restricted problem TLSP-S were introduced by Mischek and Musliu [MM18b, MM18a, MM19]. They also offered a local search framework for TLSP-S [MM19] and compared different metaheuristics. Their experiments show that Simulated Annealing offers the best results. Aside from their work, there exist no other published solution approaches for the problem considered in this work.

The Resource-Constrained Project Scheduling Problem (RCPSP) has been investigated by numerous researchers over the last decades and can be seen as the standard problem in the field of project scheduling. For a comprehensive overview over publications dealing with this problem and its many variants, we refer to surveys e.g. by Brucker et al. [BDM+99], Hartmann and Briskorn [HB10], or Mika et al. [MWW15].

Of particular interest for the problem treated in this work are various extensions to the classical RCPSP.

Multi-Mode RCPSP (MRCPSP) formulations allow for activities that can be scheduled in one of several modes. This variant has been extensively studied since 1977 [Elm77], we refer to the surveys by Węglarz et al. [WJMW11] and Hartmann and Briskorn [HB10]. A good example of a CP-Model for the MRCPSP was given by Szeredi and Schutt [SS16].

Many formulations, including TLSP, make use of release dates, due dates, deadlines, or combinations of those. An example of this can be found in [DB08]. Further relevant extensions deal with multi-project formulations, including alternative objective functions (e.g. [PWW69]). The Resource Constrained Multi-Project Scheduling Problem (RCMPSP) offers multiple projects, with project-specific constraints and objective functions [GMR08, VPLP+19]. RCMPSP was also extended with execution modes yielding Multi-Mode RCMPSP (MMRCMPSP) which is also subject of the MISTA challenge and has been considered in several publications [AKK+16, WKS+16, AM18].

Usually, the objective in (variants of) RCPSP is the minimization of the total makespan [HB10]. However, also other objective values have been considered. Of particular relevance to TLSP are objectives based on total completion time and multi-objective formulations (both appear in e.g. [NR97]). Salewski et al.[SSD97] include constraints that require several activities to be performed in the same mode. This is similar to the concept of linked jobs introduced in the TLSP.

RCPSP itself and most variants assume that individual units of each resource are identical and interchangeable, which is one of the main differences between them and TLSP-S. In difference to that in Multi-Skill RCPSP (MSPSP), first introduced by Bellenguez and Néron [BN05], each resource unit possesses certain skills, and an activity can only have these resources with the required skills assigned to it. This is similar to the availability restrictions on resources that appear in TLSP-S. Just like for our problem, they also deal with the problem that while availability restrictions could be modeled via additional activity modes corresponding to feasible resource assignments (e.g. in [BZ09, PWW69, ST00]), this is intractable due to the large number of modes that would have to be generated [BN05]. The best results for the MSPSP problem have been achieved by Young et al. [YFS17], who use a CP model to solve the problem.

Also variants of VLNS have been used to solve the RCPSP or extensions of it, such as in [PAM04]. One of the main challenges in these approaches is the choice of a subset of activities that is selected for optimization at each step. In TLSP-S, we have multiple projects over which most of the constraints are evaluated. This makes single or multiple projects a natural choice for this subset, which we exploit in our heuristic.

Bartels and Zimmermann [BZ09] describe a problem for scheduling tests of experimental vehicles. It contains several constraints that also appear in similar form in TLSP-S, but includes a different resource model and minimises the number of experimental vehicles used.

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 [DM17] and [ADM17] the authors present ASP encodings for the Nurse Scheduling Problem. ASP solution methods for the Operation Room Scheduling Problem can be found in [DGMP18]. Also, Abseher et al. [AGM$^+$16] provide an ASP formulation for the Shift Design Problem. In the case of hybrid systems, [Bal11] and [FFM$^+$16] employ an ASP and CP hybrid approach to solve industrial machine scheduling problems. An application of ASP combined with difference logic for a real-world train scheduling problem can be found in [AJO$^+$20].

CHAPTER 4

Constraint Programming Models

We initially developed our constraint programming model using the solver-independent modeling language MiniZinc [NSB+07]. Using MiniZinc we can easily compare different CP solvers and even MIP solvers.

We provide a CP model for our problem by exploiting some previous ideas for similar problems [SS16, FGS+17] and extend them to model the additional features of TLSP-S. This includes, for example, the handling of the problem specific differences discussed above but also new redundant constraints as well as search procedures tailored to the problem.

In order to provide an alternative CP formulation, we also modeled our problem with the IBM ILOG CP Optimizer [IC17a]. This model uses a different formulation of constraints and decision variables than the MiniZinc model and is described in Chapter 4.2.

4.1 MiniZinc Formulation

4.1.1 Solution Representation

In order to represent a solution for the scheduling problem we use the following decision variables. The start time variable s_j assigns a start time to each job j. Similarly, for each job j, mode variable m_j assigns it a mode. For resource assignments we need the following variables: For each job j, the variable a^{Em}_{ej} is set to 1 if employee e is assigned to j and 0 otherwise, the variable a^{Wb}_{bj} is 1 if j is performed on workbench b and 0 otherwise, and the variable a^{Eq}_{dj} is 1 if device d is used by j and 0 otherwise.

4.1.2 Basic Hard Constraints

The following constraints follow directly from the problem definition.

$$s_j \geq \alpha_j \ \wedge \ (s_j + d_{m_j j}) \leq \omega_j \qquad j \in J \tag{4.1}$$
$$s_j \geq (s_k + d_{m_k k}) \qquad j \in J,\ k \in \mathcal{P}_j \tag{4.2}$$
$$m_j \in M_j \qquad j \in J \tag{4.3}$$
$$a^{Em}_{ej} = 1 \ \rightarrow \ e \in E_j \qquad j \in J,\ e \in E \tag{4.4}$$
$$a^{Wb}_{bj} = 1 \ \rightarrow \ b \in B_j \qquad j \in J,\ b \in B \tag{4.5}$$
$$a^{Eq}_{dj} = 1 \ \rightarrow \ d \in G_{gj} \qquad j \in J,\ g \in G^*,\ d \in G_g \tag{4.6}$$
$$\sum_{e \in E} a^{Em}_{ej} \ = \ r^{Em}_{m_j} \qquad j \in J \tag{4.7}$$
$$\sum_{b \in B} a^{Wb}_{bj} \ = \ r^{Wb}_{j} \qquad j \in J \tag{4.8}$$
$$\sum_{d \in G_g} a^{Eq}_{dj} \ = \ r^{Eq}_{gj} \qquad j \in J,\ g \in G^* \tag{4.9}$$
$$a^{Em}_{ej} = a^{Em}_{ek} \qquad j \in J,\ k \in L_j,\ e \in E \tag{4.10}$$
$$s_j = 0 \qquad j \in J^S \tag{4.11}$$

The job duration constraint (3.1) is enforced by (4.1). The adherence to job precedences as required by (3.1) is given by constraint (4.2). The required availabilities of resources as formulated in hard constraints (3.11), (3.12) and (3.13) is ensured by the corresponding constraints (4.4), (4.5) and (4.6). In order to make sure that each job has exactly as many resources as required i.e. satisfying (3.8), (3.9) and (3.10) from the problem definition, we have constraints (4.7), (4.8) and (4.9). Furthermore, we need (4.10) to make sure that linked jobs are assigned to the same employees as required in (3.14) and constraint (4.11) to fix the start time of jobs which already started i.e. to satisfy hard constraint (3.4).

The above set of constraints is however not enough to ensure a valid solution. Additionally, we have to consider hard constraints (3.6), (3.5), and (3.7) which enforce that no resource (employee, workbench, or equipment) is assigned to two or more jobs at the same time. Like it was the case with MSPSP [YFS17], the constraints used for modeling these *unary resource requirements* have a tremendous impact on the practicability of the model and in the next subsection we will present different options for modeling such constraints.

4.1.3 Unary Resource Constraints

We will now present three different approaches for modeling unary resource constraints in MiniZinc, each of which is designed with CP solvers in mind. Two of these three quickly proved to be impractical for TLSP-S.

Time-indexed approach

The probably most straightforward way to model the non-overuse of any resource at any given time is captured by the following constraints.

$$\sum_{j \in J, s_j \leq t < (s_j + d_{m_j j})} a_{ej}^{Em} \leq 1 \qquad e \in E,\ t \in T \qquad (4.12)$$

$$\sum_{j \in J, s_j \leq t < (s_j + d_{m_j j})} a_{bj}^{Wb} \leq 1 \qquad b \in B,\ t \in T \qquad (4.13)$$

$$\sum_{j \in J, s_j \leq t < (s_j + d_{m_j j})} a_{dj}^{Eq} \leq 1 \qquad g \in G^*,\ d \in G_g,\ t \in T \qquad (4.14)$$

The number of constraints generated by MiniZinc based on (4.12–4.14) is of course directly dependent on the planning horizon h and the total number of resources. Because of the long compilation time and the high computer resource consumption, it quickly became immanent that for our larger instances the time-indexed approach is not efficient. This is of course not surprising since Young et al. [YFS17] came to a similar conclusion for MSPSP. Hence, we discarded this option after some preliminary testing.

Overlap constraint

For MSPSP, Young et al. [YFS17] achieved their best results using a so-called *order constraint*. This constraint basically enforces that two activities cannot overlap in their execution when they use a common resource. During the initial modeling phase we tried a very similar approach. First, we introduced the new predicate `overlap`:

$$\texttt{overlap}(j,k) := s_k < (s_j + d_{m_j j}) \ \wedge \ (s_k + d_{m_k k}) > s_j$$

In MSPSP, resources are assigned to activities with respect to the needed skill of the activity. For the overlap constraint it is not important which skill requirement the resource contributes to, so Young et al. [YFS17] had to introduce an auxiliary variable to express that a resource is used by an activity. We on the other hand assign the resources directly and thus can model our *overlap constraint* without any new variables.

$$\texttt{overlap}(j,k) \rightarrow \Big(\bigwedge_{e \in E} (\neg a_{ej}^{Em} \vee \neg a_{ek}^{Em}) \bigwedge_{b \in B} (\neg a_{bj}^{Wb} \vee \neg a_{bk}^{Wb}) \bigwedge_{g \in G^*, d \in G_g} (\neg a_{dj}^{Eq} \vee \neg a_{dk}^{Eq}) \Big)$$
$$j,k \in J,\ j \neq k,\ \alpha_k < \omega_j \wedge \omega_k > \alpha_j \qquad (4.15)$$

Just like with the time-indexed approach, it turned out that the overlap constraint produced too many constraints and was thus impractical for larger instances. This is interesting because Young et al. had no such problems, but their biggest instances only had 60 resources and 42 activities, whereas we have instances with more than 300 resources and jobs, respectively. It should however be noted that Young et al. [YFS17] reduced the number of generated constraints by considering only *unrelated* activity pairs, i.e. activities which do not depend on the execution of each other via precedence constraints (related activities can obviously never overlap). We on the other hand generate constraints for all pairs of jobs which are allowed to overlap based on their release dates and deadlines. Comparing only unrelated jobs requires the computation of the transitive closure of the job precedence relation and because our instances have a lot of unrelated jobs, we don't expect any significant improvement.

Cumulative constraints

Another way to model the unary resource constraints is to use a global constraint like `cumulative`. The `cumulative` constraint takes as input the start times, durations and resource requirements of a list of jobs and ensures that their resource assignments never exceed a given bound. This is of course a perfect way to enforce non overload of any resource and both MSPSP and MRCPSP have efficient models which make use of `cumulative` in some way [SS16, YFS17]. In order to enforce the non-overload of any resource we need three constraints (one for each resource type).

$$\texttt{cumulative}((s_j)_{j\in J},\ (d_{m_j j})_{j\in J},\ (a^{Em}_{ej})_{j\in J},\ 1) \qquad e \in E \qquad (4.16)$$

$$\texttt{cumulative}((s_j)_{j\in J},\ (d_{m_j j})_{j\in J},\ (a^{Wb}_{bj})_{j\in J},\ 1) \qquad b \in B \qquad (4.17)$$

$$\texttt{cumulative}((s_j)_{j\in J},\ (d_{m_j j})_{j\in J},\ (a^{Eq}_{dj})_{j\in J},\ 1) \qquad g \in G^*, d \in G_g \qquad (4.18)$$

In difference to our first two modeling approaches, this one turned out to scale well. Since the others performed so poorly on large instances, the rest of our experiments were performed with the `cumulative` unary resource constraints.

4.1.4 Soft Constraints

We have given several soft constraints in Chapter 3.

MiniZinc has no direct support for soft constraints, hence we define them as a function which should be minimised. This function given as follows.

As stated in Section 3.1.3, the first soft constraint of TLSP is the number of jobs and we want to maintain comparability with TLSP. Hence, we consider the term $s_1 = w_1 \cdot |J|$ part of the objective function.

According to soft constraint (3.15), the solutions where the assigned employees of a job are taken from the set of preferred employees are preferred:

$$s_2 = w_2 \cdot \sum_{j\in J} \sum_{e\in(E\setminus E^{Pr}_j)} a^{Em}_{ej}$$

For each project, the total number of employees assigned to it should be minimised (soft constraint (3.16)):

$$s_3 = w_3 \cdot \sum_{p\in P} \sum_{e\in E} ((\sum_{j\in J_p} a^{Em}_{ej}) > 0)$$

For each job, due date violation should be avoided (soft constraint (3.17)):

$$s_4 = w_4 \cdot \sum_{j \in J} max(s_j + d_{m_j j} - \bar{\omega}_j, 0)$$

Lastly, there is a soft constraint (3.18) stating that project durations should be as small as possible:

$$s_5 = w_5 \cdot \sum_{p \in P} (max_{j \in J_p}(s_j + d_{m_j j}) - min_{j \in J_p}(s_j))$$

The complete objective function is then given by $min\ \sum_{1 \leq i \leq 5} s_i$.

4.1.5 Redundant Constraints

Finding good redundant constraints for our problem proved to be very hard since the search space is usually very big and at the beginning of the search there is little knowledge about the final duration of the jobs. To deal with this issue we introduced a relaxed `cumulative` constraint enforcing a global resource bound.

$$\begin{aligned}\texttt{cumulative}((s_j)_{j \in J},\ & \\ & (min_{m \in M}(d_{m_j j}))_{j \in J}, \\ & (min_{m \in M_j}(r^{Em}_m) + r^{Wb}_j + \sum_{g \in G^*} r^{Eq}_{gj}\ )_{j \in J}, \\ & |E| + |B| + \sum_{g \in G^*} |G_g| \qquad) \end{aligned} \tag{4.19}$$

This enables the search to discard scheduling options which are impossible regardless of the chosen modes early on.

On top of that, we can also formulate more straightforward `cumulative` constraints which enforce the global resource bounds for each resource at any point in time.

$$\texttt{cumulative}((s_j)_{j \in J},\ (d_{m_j j})_{j \in J},\ (r^{Em}_{m_j})_{j \in J},\ |E|) \tag{4.20}$$

$$\texttt{cumulative}((s_j)_{j \in J},\ (d_{m_j j})_{j \in J},\ (r^{Wb}_j)_{j \in J},\ |B|) \tag{4.21}$$

$$\texttt{cumulative}((s_j)_{j \in J},\ (d_{m_j j})_{j \in J},\ (r^{Eq}_{gj})_{j \in J},\ |G|) \qquad g \in G^* \tag{4.22}$$

Given the large search space, trying to restrict the scope of the decision variables seems like a worthwhile idea. We achieve this by using *global cardinality constraints*. These constraints allow us to give tight bounds for the total number of resources which should be used.

$$\texttt{gcc\_low\_up}((a^{Em}_{ej})_{e \in E, j \in J},\ 1,\ \sum_{j \in J} min_{m \in M_j}(r^{Em}_m), \sum_{j \in J} max_{m \in M_j}(r^{Em}_m)) \tag{4.23}$$

$$\texttt{gcc\_low\_up}((a_{bj}^{Wb})_{b\in B, j\in J},\ 1,\ \sum_{j\in J} r_j^{Wb},\ \sum_{j\in J} r_j^{Wb}) \tag{4.24}$$

$$\texttt{gcc\_low\_up}((a_{dj}^{Eq})_{g\in G^*, d\in G_g, j\in J},\ 1,\ \sum_{j\in J}\sum_{g\in G^*} r_{gj}^{Eq},\ \sum_{j\in J}\sum_{g\in G^*} r_{gj}^{Eq}) \tag{4.25}$$

Constraint (4.23) enforces that no more employees can be assigned than the sum of the highest possible employee requirements and no less than the sum of the minimum requirements. The other two constraints analogously ensure that the number of assigned workbenches and equipment is tightly bounded by the cumulative requirement of all jobs.

4.1.6 Search Strategies

During initial testing it quickly became immanent that the default search strategy of Chuffed (or Gecode) was not even able to find feasible solutions for most instances. This was not surprising since Young et al. [YFS17] already had a similar issue with MSPSP. However, they were able to improve their results drastically by employing a new MiniZinc search annotation called `priority_search` which is supported by Chuffed [FGS+17]. Based on their research we have experimented with four slightly different versions of `priority_search`:

(i) `ps_startFirst_aff`

(ii) `ps_startFirst_ff`

(iii) `ps_modeFirst_aff`

(iv) `ps_modeFirst_ff`

All four search strategies branch over the jobs and their resource assignments. The order of the branching is the same for all strategies and is determined by the smallest possible start times of the jobs in ascending order. For each branch, searches (i) and (ii) initially assign the smallest start time to the selected job followed by assigning it the mode which minimises the job duration. Search procedures (iii) and (iv) start with the mode assignment and then assign the start time. Once the start time and the mode have been assigned for the selected job, all of the search strategies make resource assignments for the job. Searches (i) and (iii) start by assigning those resources to the job which are available and have the biggest domain i.e. those which are not fixed to assigned/unassigend, whereas (ii) and (iv) start with assignments which are either unavailable or have only one value in their domain.

4.2 Interval-based Model

We also modelled our problem with CP Optimizer [IC17a]. This solver uses a different modeling paradigm than MiniZinc. We gave formal definitions for the variables and constraints we are going to use in Chapter 2.1.

In the CP Optimizer model the decision variables are given by the following interval variables:

$$
\begin{array}{lr}
\texttt{interval } a_j \subseteq [\alpha_j, \omega_j) & j \in J \\
\texttt{interval } b_p & p \in P \\
\texttt{interval } a_{ij} \texttt{ optional size } d_{ij} & j \in J, i \in M_j \\
\texttt{interval } a^{EmM}_{eij} \texttt{ optional} & j \in J, i \in M_j, e \in E_j \\
\texttt{interval } a^{Em}_{ej} \texttt{ optional} & j \in J, e \in E_j \\
\texttt{interval } a^{Wb}_{bj} \texttt{ optional} & j \in J, b \in B_j \\
\texttt{interval } a^{Eq}_{gdj} \texttt{ optional} & j \in J, g \in G^*, d \in G_{gj}
\end{array}
$$

The intervals a_j represents the jobs and are constrained to the time windows of the respective job. The second set of intervals b_p are auxiliary variables representing the total duration of the projects. These intervals enable an easy formulation of the project duration soft constraint. The next intervals a_{ij} are optional and the presence of such an interval indicates that job j is performed in mode i. For a job j several employee allocations are possible depending on its mode. The presence of an optional interval a^{EmM}_{eij} represents the allocation of employee e to perform job j in mode i. The last three sets of optional intervals are used to indicate resource allocation.

The hard constraints of the problem are encoded as follows:

$$\texttt{span}(b_p, [a_j]_{j \in J_p}) \qquad p \in P \tag{4.26}$$
$$\texttt{endBeforeStart}(a_k, a_j) \qquad k \in P_j \tag{4.27}$$
$$\texttt{alternative}(a_j, (a_{ij})_{i \in M_j}, 1) \qquad j \in J \tag{4.28}$$
$$\texttt{alternative}(a_{ij}, (a^{EmM}_{eij})_{e \in E_j}, r^{Em}_{ij}) \qquad j \in J, i \in M_j \tag{4.29}$$
$$\texttt{alternative}(a^{Em}_{ej}, (a^{EmM}_{eij})_{i \in M_j}, 1) \qquad j \in J, e \in E_j \tag{4.30}$$
$$\texttt{noOverlap}((a^{EmM}_{eij})_{j \in J, i \in M_j, e \in E_j}) \qquad e \in E \tag{4.31}$$
$$\texttt{noOverlap}((a^{Em}_{ej})_{j \in J, e \in E_j}) \qquad e \in E \tag{4.32}$$
$$\texttt{alternative}(a_j, (a^{Wb}_{bj})_{b \in B_j}, r^{Wb}_j) \qquad j \in J \tag{4.33}$$
$$\texttt{noOverlap}((a^{Wb}_{bj})_{j \in J, b \in B_j}) \qquad b \in B \tag{4.34}$$
$$\texttt{alternative}(a_j, (a^{Eq}_{gdj})_{d \in G_{gj}}, r^{Eq}_{gj}) \qquad j \in J, g \in G^* \tag{4.35}$$
$$\texttt{noOverlap}((a^{Eq}_{gdj})_{j \in J, d \in G_{gj}}) \qquad g \in G^*, d \in G_g \tag{4.36}$$
$$\neg\texttt{presenceOf}(a_{ji}) \qquad j \in J, i \in (M \setminus M_j) \tag{4.37}$$
$$\neg\texttt{presenceOf}(a^{Em}_{ej}) \qquad j \in J, b \in (E \setminus E_j) \tag{4.38}$$
$$\neg\texttt{presenceOf}(a^{Wb}_{bj}) \qquad j \in J, b \in (B \setminus B_j) \tag{4.39}$$
$$\neg\texttt{presenceOf}(a^{Eq}_{dj}) \qquad j \in J,\ g \in G^*,\ d \in (G_g \setminus G_{gj}) \tag{4.40}$$

$$\texttt{presenceOf}(a_{ej}^{Em}) = \texttt{presenceOf}(a_{ek}^{Em}) \qquad j \in J, k \in L_i, e \in E \tag{4.41}$$

$$\texttt{startOf}(a_j) = 1 \qquad j \in J^S \tag{4.42}$$

Constraint (4.26) allows us to easily formulate the soft constraints which depend on the completion time of a project. The job precedences as defined in (3.3) are enforced through (4.27). The alternative constraints (4.28–4.30) make sure that the employee requirements for each job (3.10) are satisfied, while (4.31–4.36) express the single assignment constraints (3.5–3.7). In order to make sure that the resource availability requirements (3.11–3.13) are satisfied, we need constraints (4.31–4.36). Lastly, linked jobs (3.14) are modelled using (4.41) and started jobs (3.4) are modelled with (4.42).

Using pulse constraints we can also formulate redundant constraints which are similar to the `cumulative` constraints defined in Section 4.1.5.

$$\sum_{j \in J} \sum_{m \in M_j} \texttt{pulse}(a_{jm}, r_{jm}^{Em}) \leq |E| \tag{4.43}$$

$$\sum_{j \in J} \sum_{m \in M_j} \texttt{pulse}(a_{jm}, r_{jm}^{Wb}) \leq |B| \tag{4.44}$$

$$\sum_{j \in J} \sum_{m \in M_j} \texttt{pulse}(a_{jm}, r_{jm}^{Eq}) \leq \sum_{g \in G^*} |G_g| \tag{4.45}$$

Finally, the objective is to minimize the following formula which is just the weighted sum of all violations of soft constraints defined in Chapter 3.1.3.

$$\begin{aligned} min \quad & w_1 \cdot |J| \\ & + w_2 \cdot \sum_{j \in J} \max(0, \texttt{endOf}(a_j) - \varpi_j) \\ & + w_3 \cdot \sum_{j \in J} \sum_{e \in (E \setminus E_j^{Pr})} \texttt{presenceOf}(a_{ej}^{Em}) \\ & + w_4 \cdot \sum_{p \in P} \sum_{e \in E} (0 < \sum_{j \in J_p} \texttt{presenceOf}(a_{ej}^{Em})) \\ & + w_5 \cdot \sum_{p \in P} \texttt{lengthOf}(b_p) \end{aligned}$$

CHAPTER 5

Constraint Answer-set Programming Model

In this chapter we investigate a hybrid Answer-set solving approach for TLSP-S. We present an encoding of the problem in the input language of the Constraint Answer-set Programming (CASP) solver clingcon [GOS09, BKOS17] and give alternative formulations for unary resource constraints. To the best of our knowledge, this is the first time CASP has been employed for a project scheduling problem.

5.1 Input Facts

In order to solve a TLSP-S instance with clingcon we need to encode its parameters described in Chapter 3.1.1 as facts.

For each job $j \in J$ we have the following input facts:

- $\mathit{job}(j)$,
- $\mathit{release}(j, \alpha_j)$,
- $\mathit{deadline}(j, \omega_j)$,
- $\mathit{due}(j, \bar{\omega}_j)$,
- $\mathit{durInMode}(j, m, d_{mj})$ for each mode $m \in M$,
- $\mathit{precedence}(k, j)$ for each $k \in \mathcal{P}_j$,
- $\mathit{linked}(j, k)$ for each $k \in L_j$,
- $\mathit{modeAvail}(j, m)$ for each available mode $m \in M_j$,

- $\mathit{empAvail}(j, e)$ for each qualified employee $e \in E_j$,
- $\mathit{wbAvail}(j, w)$ for each available workbench $w \in W_j$,
- $\mathit{equipAvail}(j, e)$ for each equipment group $g \in G^*$ and each available device $e \in G_{jg}$,
- $\mathit{reqEquip}(j, g, r_{gj}^{Eq})$ for each equipment group $g \in G^*$,
- $\mathit{empPref}(j, e)$ for each preferred employee $e \in E_j^{Pr}$, and
- $\mathit{projAssign}(j, p)$ where $p \in P$ and $j \in J_p$.

Additionally, if $j \in J^S$ then we have the fact $\mathit{started}(j)$ and also $\mathit{wbReq}(j)$ if $r_j^{Wb} = 1$.

Furthermore, for each mode $m \in M$ we also have a fact $\mathit{reqEmp}(m, r_m^{Em})$, for each equipment group $g \in G^*$ we have $\mathit{group}(g)$ and for each project $p \in P$ we have the fact $\mathit{project}(p)$. Lastly, we have the fact $\mathit{horizon}(h)$ denoting the scheduling horizon.

5.2 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 $j \in J$:

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

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

5.3 Basic Hard Constraints

We will now give the clingcon encoding of the basic hard constraints of TLSP-S as described in the problem definition.

$$\begin{aligned} \&\mathit{dom}\{R..D\} = \mathit{start}(J) \;&\leftarrow \mathit{job}(J), \mathit{release}(J, R), \\ &\quad \mathit{deadline}(J, D) \qquad (5.1) \\ \&\mathit{dom}\{R..D\} = \mathit{end}(J) \;&\leftarrow \mathit{job}(J), \mathit{release}(J, R), \\ &\quad \mathit{deadline}(J, D) \qquad (5.2) \end{aligned}$$

$$
\begin{aligned}
1\ \{modeAssign(J,M) : modeAvail(J,M)\}\ 1 &\leftarrow job(J) && (5.3)\\
duration(J,T) &\leftarrow job(J), modeAssign(J,M), \\
&\quad\ durInMode(J,M,T) && (5.4)\\
\&sum\{end(J); -start(J)\} = T &\leftarrow job(J), duration(J,\ T) && (5.5)\\
\&sum\{start(J)\} \geq end(K) &\leftarrow job(J), job(K), precedence(J,K) && (5.6)\\
\&sum\{start(J)\} = 0 &\leftarrow job(J), started(J) && (5.7)\\
1\ \{wbAssign(J,W) : wbAvail(J,W)\}\ 1 &\leftarrow job(J), wbReq(J) && (5.8)\\
R\ \{empAssign(J,E) : empAvail(J,E)\}\ R &\leftarrow job(J), modeAssign(J,M), \\
&\quad\ reqEmployees(M,R) && (5.9)\\
R\ \{equipAssign(J,E) : equipAvail(J,E), \\
group(E,G)\}\ R &\leftarrow job(J), group(\_,G), \\
&\quad\ reqEquip(J,G,R) && (5.10)\\
&\leftarrow job(J), job(K), linked(J,K), \\
&\quad\ empAssign(J,E), \\
&\quad\ \sim empAssign(K,E) && (5.11)
\end{aligned}
$$

The first two rules (5.1) and (5.2) define the domains of the start and end times which are bounded by the release time and deadline of a job thus enforcing the time window constraint (3.2). Rule (5.3) ensures that each job is assigned exactly one available mode. Rule (5.4) enforces that each job has the duration required by its mode and rule (5.5) links the start and end time to the duration. Rule (5.6) ensures that the job precedence constraint (3.3) is satisfied. Started jobs are enforced to start at the first timeslot by rule (5.7) thus ensuring constraint (3.4). The resource requirement constraints (3.8–3.10) are handled by rules (5.8), (5.9), and (5.10). Finally, the constraint rule (5.11) ensures that the linked jobs constraint (3.14) is satisfied.

The rules presented above do not enforce the single assignment constraints (3.6), (3.5), and (3.7). In the next section we will introduce 2 ways of encoding these constraints in clingcon.

5.4 Unary Resource Constraints

In Section 4.1.3, we expounded on the difficulty of modelling unary resource constraints for TLSP-S i.e. constraints ensuring that no resource is used by multiple jobs simultaneously in CP. Our best formulation of these 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 need to decompose the constraints.

We propose two such decompositions. The first is given by the following rules.

$$\begin{aligned}
\mathit{precedence}(J,K) \lor \mathit{precedence}(K,J) &\leftarrow \mathit{job}(J), \mathit{job}(K), \mathit{wbAssign}(J,W), \\
&\quad \mathit{wbAssign}(K,W), J < K \qquad (5.12)\\
\mathit{precedence}(J,K) \lor \mathit{precedence}(K,J) &\leftarrow \mathit{job}(J), \mathit{job}(K), \mathit{empAssign}(J,E), \\
&\quad \mathit{empAssign}(K,E), J < K \qquad (5.13)\\
\mathit{precedence}(J,K) \lor \mathit{precedence}(K,J) &\leftarrow \mathit{job}(J), \mathit{job}(K), \mathit{equipAssign}(J,E), \\
&\quad \mathit{equipAssign}(K,E), J < K \qquad (5.14)
\end{aligned}$$

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.

The second is more straight-forward and is given as follows.

$$\begin{aligned}
\mathit{overlap}(J,K) &\leftarrow \mathit{job}(J), \mathit{job}(K), \&\mathit{sum}\{\mathit{start}(K)\} < \mathit{end}(J), \\
&\quad \&\mathit{sum}\{\mathit{end}(K)\} > \mathit{start}(J), J < K \qquad (5.15)\\
&\leftarrow \mathit{overlap}(J,K), \mathit{wbAssign}(J,W), \mathit{wbAssign}(K,E), J < K \qquad (5.16)\\
&\leftarrow \mathit{overlap}(J,K), \mathit{empAssign}(J,E), \mathit{empAssign}(K,E), J < K \qquad (5.17)\\
&\leftarrow \mathit{overlap}(J,K), \mathit{equipAssign}(J,E), \mathit{equipAssign}(K,E), J < K \qquad (5.18)
\end{aligned}$$

In this formulation, a fact $\mathit{overlap}(j,k)$ is derived for each pair of overlapping jobs $j, k \in J$ by rule (5.15). The constraints (5.16–5.18) then ensure that overlapping jobs cannot be assigned the same resources.

An empirical evaluation of both formulations is given below in Chapter 7.4.1.

5.5 Soft Constraints

We now give the encodings for the the soft constraints given in Chapter 3.

$$\begin{aligned}
\&\mathit{sum}\{\mathit{unprefEmp}(J)\} = N &\leftarrow \mathit{job}(J), \\
&\quad \#\mathit{count}\{E : \mathit{empAssign}(J,E), \\
&\quad \sim \mathit{empPref}(J,E)\} = N \qquad (5.19)\\
\&\mathit{minimize}\{w_2 \cdot \mathit{unprefEmp}(J) : \mathit{job}(J)\} & \qquad (5.20)
\end{aligned}$$

The first soft constraint (3.15) 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 (5.19). For each job j, this aggregate counts the number of occurrences where an employee e is assigned to j but $\mathit{empPref}(j, e)$ (indicating that e is preferred for j) cannot be derived and is thus not part of the input. The objective (5.20) then encodes the necessary minimization.

$$\begin{aligned}
\&\mathit{sum}\{\mathit{employees}(P)\} = N \ &\leftarrow \mathit{project}(P), \\
&\quad \#\mathit{count}\{E : \mathit{empAssign}(J, E), \\
&\quad \mathit{projAssign}(J, P)\} = N && (5.21) \\
\&\mathit{minimize}\{w_3 \cdot \mathit{employees}(P) : \mathit{project}(P)\} & && (5.22)
\end{aligned}$$

According to soft constraint (3.16), 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 (5.21) and (5.22). The input fact $\mathit{projAssign}(j, p)$ denotes that j is contained in project p.

$$\begin{aligned}
\&\mathit{sum}\{\mathit{delay}(J); T\} = \mathit{end}(J) \ &\leftarrow \mathit{job}(J), \mathit{due}(J, T), \\
&\quad \&\mathit{sum}\{\mathit{end}(J); -T\} > 0 && (5.23) \\
\&\mathit{sum}\{\mathit{delay}(J)\} = 0 \ &\leftarrow \mathit{job}(J), \mathit{due}(J, T), \\
&\quad \&\mathit{sum}\{\mathit{end}(J); -T\} \leq 0 && (5.24) \\
\&\mathit{minimize}\{w_4 \cdot \mathit{delay}(J) : \mathit{job}(J)\} & && (5.25)
\end{aligned}$$

Now we come to soft constraint (3.17) i.e. we generally want a job j to end before is due date t given by input fact $\mathit{due}(j, t)$ or, if this is not possible, to minimize the delay. The encoding for this objective can be found in (5.23–5.25) where we introduce integer variables for the delay of each job. These 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.

$$\begin{aligned}
\&\mathit{dom}\{0..H\} = \mathit{projectStart}(P) \ &\leftarrow \mathit{project}(P), \mathit{horizon}(H) && (5.26) \\
\&\mathit{dom}\{0..H\} = \mathit{projectEnd}(P) \ &\leftarrow \mathit{project}(P), \mathit{horizon}(H) && (5.27) \\
1\ \{\mathit{firstJob}(J) : \mathit{job}(J), \mathit{projAssign}(J, P)\}\ 1 \ &\leftarrow \mathit{project}(P) && (5.28) \\
\&\mathit{sum}\{\mathit{projectStart}(P)\} = \mathit{start}(J) \ &\leftarrow \mathit{firstJob}(J), \mathit{projAssign}(J, P) && (5.29) \\
\&\mathit{sum}\{\mathit{projectStart}(P)\} \leq \mathit{start}(J) \ &\leftarrow \mathit{job}(J), \mathit{projAssign}(J, P) && (5.30) \\
1\ \{\mathit{lastJob}(J) : \mathit{job}(J), \mathit{projAssign}(J, P)\}\ 1 \ &\leftarrow \mathit{project}(P) && (5.31) \\
\&\mathit{sum}\{\mathit{projectEnd}(P)\} = \mathit{end}(J) \ &\leftarrow \mathit{lastJob}(J), \mathit{projAssign}(J, P) && (5.32) \\
\&\mathit{sum}\{\mathit{projectEnd}(P)\} \geq \mathit{end}(J) \ &\leftarrow \mathit{job}(J), \mathit{projAssign}(J, P) && (5.33) \\
\&\mathit{minimize}\{w_5 \cdot \mathit{projectEnd}(P) - w_5 \cdot \mathit{projectStart}(P) :\ \mathit{project}(P)\} & && (5.34)
\end{aligned}$$

The last soft constraint (3.18) has the most complex formulation which is given in (5.26–5.34). 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 (5.28) and (5.31). 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 (5.29) and the end with rule (5.32). The objective (5.34) then simply minimizes the sum of all completion times.

Not given is the encoding for the fixed penalty $w_1 \cdot |J|$. However, this can easily be formulated using a count aggregate.

CHAPTER 6

Very Large Neighborhood Search

Utilizing the exact methods described in Chapters 4 and 5 we propose a *Very Large Neighborhood Search* (VLNS) approach. The basic idea is to start with a feasible but suboptimal solution and repeatedly fix most of the schedule except for a small number of projects and then try to find an optimal solution for those projects.

Furthermore, we also use our exact approaches to determine a lower bound of the objective function. This idea will be described in the next section, afterwards we give the details of the Very Large Neighborhood Search.

6.1 Lower Bound Calculation

In order to provide the VLNS with knowledge of which projects should be rescheduled to improve the objective value, we calculate a lower bound for the introduced penalty of each project. If a project induces a penalty equal to its lower bound in the current solution, rescheduling said project cannot improve the objective and hence should be avoided.

Algorithm 6.1 gives the details of the lower bound calculation in pseudo code. The algorithms loops over all projects in the instance and solves each project independently with a given timeout. This is achieved by creating a new problem instance where all other projects are removed. If the solution found for this instance is optimal, we add its objective value as a lower bound for the project. If the limit is reached and no optimal solution could be found, we determine a heuristic lower bound for the project as follows: We sum up the number of jobs (to maintain comparability with TLSP), the minimum number of different employees needed for the project (3.16), and the minimal duration of all jobs on the longest path in the job dependency graph (3.18). For soft constraints (3.15) and (3.17), zero is used as the lower bound.

It should be noted that this algorithm works independently of the exact method used.

Algorithm 6.1: Lower Bound Calculation

```
Data: the problem $instance$ and a $timeout$
Result: a map with a project as key and as value its lower bound
1  initialize empty map $lowerBounds$;
2  $projects \leftarrow getProjects(instance)$;
3  for $p \in projects$ do
4      $projectInstance \leftarrow deleteAllExcept(instance, p)$;
5      $schedule \leftarrow solveOptimal(projectInstance, timeout)$;
6      if $isOptimal(schedule)$ then
7          $lowerBounds(p) \leftarrow getPenalty(schedule)$;
8      else
9          $lowerBounds(p) \leftarrow getHeuristicBound(projectInstance)$;
10     end
11 end
12 return $lowerBounds$
```

6.2 Search Algorithm

As already mentioned, VLNS starts with a feasible solution and then applies the exact solver repeatedly on parts of the current solution. This process is described in detail in Algorithm 6.2.

The basic steps of the search are the following:

1. *Find Initial Solution*
   In order for the VLNS to work, we need a feasible schedule for the instance. Since our exact approaches find feasible solutions within a couple of minutes, we used them to provide an initial solution.

2. *Calculate lower bound for each project*
   In parallel to step one we calculate the lower bounds as described above.

3. *Fix all but k projects*
   Once we have an initial solution and the lower bounds we can start the actual heuristic. We start by selecting at random a combination of k projects (initially, $k = 1$), with the following properties: If $k > 1$, then all of the projects 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 larger than zero.

   The projects which are not in the selected combination are then fixed. To achieve this, we modify the time windows and availabilities of each job contained in a fixed project. The release and due dates are changed to correspond to the current start and end dates, the available modes and resources are changed to only include the

Algorithm 6.2: Very Large Neighborhood Search

```
Data: the problem $instance$, $globalTimeout$, $moveTimeout$ and $jumpProb$
Result: a solution schedule
$currentSchedule \leftarrow solveFeasible(instance)$;
$lowerBounds \leftarrow calculateLowerBounds(instance, moveTimeout)$;
$lowerBound \leftarrow$ sum $lowerBounds$;
$noChangeList \leftarrow \emptyset$;
$projects \leftarrow getProjects(instance)$;
$k \leftarrow 1$;
while $not\ globalTimeout\ reached$ do
    $availableProjects \leftarrow \emptyset$;
    for $p \in projects$ do
        if $lowerBounds(p) < getPenalty(currentSchedule, p)$ then
            $availableProjects \leftarrow availableProjects \cup \{p\}$;
        end
    end
    $selectedProjects \leftarrow$
      $getRandomCombination(availableProjects, noChangeList, k)$;
    if $selectedProjects = \emptyset$ then
        $k = k + 1$;
        with probability $jumpProb$ do $k = k + 1$;
        $selectedProjects \leftarrow$
          $getRandomCombination(availableProjects, noChangeList, k)$;
        if $selectedProjects = \emptyset$ then
            abort;
        end
    end
    $tmpInstance \leftarrow fixAllExcept(currentSchedule, selectedProjects)$;
    $tmpSchedule \leftarrow solveOptimal(tmpInstance)$;
    if $getPenalty(tmpSchedule) < getPenalty(currentSchedule)$ then
        $currentSchedule \leftarrow tmpSchedule$;
        $noChangeList \leftarrow$
          $removeOverlaps(noChangeList, currentSchedule, selectedProjects)$;
        $noChangeList \leftarrow noChangeList \cup \{selectedProjects\}$;
        if $lowerBound = getPenalty(currentSchedule)$ then
            abort;
        end
    end
end
return $schedule$
```

current assignments, and finally the resource requirements are restricted to equal the assignments.

The resulting instance is then further reduced by cutting away the fixed jobs outside of the merged time window of the selected projects. These removed jobs cannot be changed or influence the result and reducing the size of the instance improves compilation/grounding time.

4. *Perform move*
After the preprocessing we try to find an optimal solution for the selected projects, where we again set a runtime limit of *moveTimeout*. The best assignment found within this limit is then applied to the current incumbent schedule if it decreases the penalty of the solution.

   After we have performed the move, we save the selected combination in a list. 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 combination.

5. *Possibly change k*
If k is bigger than 1 and the incumbent schedule has been changed 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 *jumpProb* – by two. If there no more combinations for any k, 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.

6.3 Modifications for Chuffed

Initial experiments showed that using VLNS with our MiniZinc model described in Chapter 4.1 and the CP solver Chuffed falls into local optima quite often. Hence, it was necessary to increase diversification. One way to do this was by modifying line 24 of Algorithm 6.2 to allow the acceptance of moves which neither increase nor decrease the current penalty. However, this yielded situations where the search would loop by doing and undoing the same moves repeatedly. To combat this we extended the algorithm further. With a parameterised probability *hotStartProb* we hot start the CP solver. This means that the current assignment of the selected projects is given to the solver as an initial solution. The hot start functionality is based on work by Demirovic et. al [DCS18] and has been integrated by us into the current version of the solver Chuffed. The hot starts get the search to reject nonimproving solutions while also increasing the performance of the underlying CP solver. Additionally, if we do not hot start the solver, we additionally set up `priority_search` to assign resources not in input order but randomly to enhance diversification.

CHAPTER 7

Experiments and Comparison

In this chapter we evaluate our solutions methods on a set of benchmark instances and real-life data. We also provide a comparison of our solution approaches with Simulated Annealing which was introduced by Mischek and Musliu [MM19]. This approach was shown to be the best performing metaheuristic for TLSP-S.

7.1 Experimental Setup

We ran our experiments on a benchmark server with 224GB RAM and two AMD Opteron 6272 Processors each with max. 2.1GHz and 16 logical cores. Since all of the solvers we experimented with are single threaded, we usually ran two independent sets of benchmarks in parallel. For the VLNS parameter experiments we used a Lenovo ThinkPad University T480s with an Intel Core i7-8550U (1,8 GHz). We used MiniZinc 2.2.3 [NSB^{+}07] with Chuffed 0.10.3 [Chu11] and CPLEX 12.8.0 [IC17b]. Our VLNS which is described in Chapter 6 was implemented in Java 8. Furthermore, we also experimented with the ILOG CP Optimizer 12.8.0 [IC17a] which was not run from MiniZinc but with the ILOG Java API. We have also tested Gecode [SLT18] as an additional CP solver included in MiniZinc, but it quickly proved to be inferior to Chuffed on this model even when run with multiple threads. As CASP solver we used clingcon 5 [GOS09, BKOS17] (unpublished as of the writing of this thesis but available on github[1]).

7.2 Instances

We use a total of 30 randomly generated instances (based on real-life situations) of different sizes for our experiments [MM18b]. In addition, we also include one real-life instance (Lab) taken directly from the lab of our industrial partner (in anonymized

[1] `github.com/potassco/clingcon`

form). This instance covers a scheduling period of over one year, at a granularity of two timeslots per working day. It includes a reference schedule that is the manually created schedule actually used in the lab at the time the instance was created. Those instances are listed in Table 7.1. The instances all have three modes: a *single* mode requiring only one employee, a *shift* mode which requires two employees but has a reduced duration, and an *external* mode that requires no employees at all. In general, jobs can be done in single mode or optionally in shift mode. Some instances however also include jobs which can only be performed in external mode. Initial assignments appear only for jobs which are already started or are fixed to their current value via availability restrictions and time windows.

While the the 30 benchmark instances were generated randomly, they are still modeled after real-world scenarios. Half of the instances are modeled very closely to a real-world laboratory, whereas the other half is more general and makes full use of the problem features. The details of how this generation works as well as the exact differences between the laboratory instances and general instances are given in [MM18b]. Furthermore, our 30 instances are a selection from a total of 120 instances given in the report. We chose the first two instances of each size (scheduling horizon and number of projects) and two additional instances for the 3 smallest sizes. This selection was necessary, because of the long time it would have taken to experiment with all 120 instances and the same selection was made in previous work [MM19]. Those 120 instances as well as the 30 we selected for this paper can be found at `https://www.dbai.tuwien.ac.at/staff/fmischek/TLSP/`.

Since those instances were generated with the full TLSP in mind and in TLSP-S we take the initial job grouping as fixed and unchangeable, the instances had to be converted. This is achieved by viewing the jobs as the smallest planning unit and assigning the job parameters – which are defined by the tasks contained in the job in TLSP – directly to the jobs.

7.3 Constraint Programming Experiments

In Chapter 4.1.6 we gave several different search strategies for our MiniZinc model and the CP solver Chuffed. Table 7.2 shows the comparison of the search procedures described. The column *#feas* shows for how many instances (out of the 30 generated instances) the model-search combination found feasible solutions, *#opt* contains the number of instances solved to optimality. Furthermore, the values in the column *#best* show the number of instances where the respective configuration found the best solution w.r.t all six configurations.

Each model was run using Chuffed with free search enabled. Free search alternates between user-defined and activity-based search on each restart. The time limit was set to 30 minutes for each instance. It can be easily seen that any version of `priority_search` is vastly superior to the default search of Chuffed.
`priority_search` strategies solve more instances to optimality and also found feasible solution for every instance. It should be noted that the fourteen optimally solved instances

#	**Data Set**	**ID**	$\lvert P\rvert$	$\lvert J\rvert$	h	$\lvert E\rvert$	$\lvert B\rvert$	$\lvert G^*\rvert$	$\overline{\lvert E_j\rvert}$	$\overline{\lvert B_j\rvert}$	$\overline{\lvert G_{gj}\rvert}$
1	General	000	5	7	88	7	7	3	2.08	3.57	1.5
2	General	001	5	8	88	7	7	3	4.88	3.63	15.67
3	LabStructure	000	5	24	88	7	7	3	1.84	3.38	11.67
4	LabStructure	001	5	14	88	7	7	3	4.36	3.5	0.36
5	General	005	10	29	88	13	13	4	4.04	3.48	5.76
6	General	006	10	18	88	13	13	6	5.56	4.22	13.28
7	LabStructure	005	10	37	88	13	13	3	6.16	4.03	0.65
8	LabStructure	006	10	29	88	13	13	3	6.21	3.76	21.01
9	General	010	20	60	174	16	16	5	7.42	4.42	11.36
10	General	011	20	84	174	16	16	4	7.31	4.3	3.7
11	LabStructure	010	20	65	174	16	16	3	6.28	4.43	26.26
12	LabStructure	011	20	62	174	16	16	3	7.27	4.24	1.21
13	General	020	15	29	174	12	12	5	5.76	3.97	1.12
14	LabStructure	020	15	53	174	12	12	3	6.28	4.47	20.63
15	General	025	30	113	174	23	23	3	8.26	4.41	5.71
16	LabStructure	025	30	105	174	23	23	3	7.52	4.25	39.63
17	General	015	40	126	174	31	31	3	9.26	4.48	29.53
18	LabStructure	015	40	138	174	31	31	3	7.36	3.57	41.93
19	General	030	60	208	174	46	46	6	9.85	4.11	31.45
20	LabStructure	030	60	212	174	46	46	3	9.28	4.17	78.16
21	General	035	20	76	520	6	6	5	4.24	3.62	8.08
22	LabStructure	035	20	71	520	6	6	3	4.3	3.42	11.70
23	General	040	40	196	520	12	12	4	6.95	4.47	4.24
24	LabStructure	040	40	187	520	12	12	3	6.55	4.51	1.38
25	General	045	60	260	520	18	18	6	7.65	4.52	23.95
26	LabStructure	045	60	239	520	18	18	3	7.44	4.42	33.65
27	General	050	60	270	782	13	13	4	6.89	4.39	3.89
28	LabStructure	050	60	247	782	13	13	3	6.97	4.21	23.42
29	General	055	90	384	782	19	19	5	7.27	4.29	26.89
30	LabStructure	055	90	401	782	19	19	3	7.34	4.53	36.76
Lab	-	-	74	297	606	22	17	3	6.02	5.36	1*

Table 7.1: The set of test instances used for the experiments. Shown are the data set the instance is taken from and the ID within that set. The following columns list the number of projects, jobs and the length of the scheduling period, followed by the number of employees, workbenches and equipment groups. The last columns contain the mean qualified employees and available workbenches per job, as well as the mean available devices per job and equipment group (only over those jobs that actually require at least one device of the group, about 10% of all jobs).
*The discrepancy compared to the generated instances arises from the fact that several equipment groups were not yet considered for planning at the time this instance was created.

Red. Constraints	**Search**	**#sat**	**#opt**	**#best**
(4.19–4.25)	Default	13	8	8
(4.19–4.25)	`ps_modeFirst_ff`	**30**	**14**	18
(4.19–4.25)	`ps_modeFirst_aff`	**30**	**14**	18
(4.19–4.25)	`ps_startFirst_ff`	**30**	**14**	21
(4.19–4.25)	`ps_startFirst_aff`	**30**	**14**	22
(4.20–4.25)	`ps_startFirst_aff`	**30**	**14**	**25**

Table 7.2: Priority Search Experiments (Runtime 30m)

are the same over all search configurations. They all have less than or equal to 20 projects. The search strategy `ps_startFirst_aff` achieved the best cumulative objective value, which is why this strategy – albeit slightly modified to support random value selection – is also used in the VLNS.

While initial experiments showed that redundant constraints (4.20–4.25) have a high impact on the search, Table 7.2 shows that constraint (4.19) has no positive impact on the solution quality. Hence, we decided to drop the constraint for the comparison with the other solvers and the VLNS experiments.

7.4 Constraint Answer-set Programming Experiments

For CASP we present two further experiments. The first concerns different encodings for a particular constraint, whereas the second is a comparison of built-in search options of clingcon. The solver supports a multitude of further configuration options, but we mostly rely on its default settings.

For each of the experiments in this section, clingcon was given a time limit of 30 minutes per instance. Unless stated otherwise we used the default configuration and a single solving thread.

7.4.1 Unary Resource Constraints

As described in Chapter 5.4, our model contains two alternative formulations for the unary resource constraints. We evaluated both versions of our model, once with the direct formulation and once with the precedence formulation, on all 30 generated test

Formulation	**Unary Resource Constraints**	**#sat**	**#opt**	**#best**
precedence	(5.12–5.14)	**30**	**18**	**25**
direct/overlap	(5.15–5.18)	28	15	19

Table 7.3: clingcon Unary Resource Constraints Experiments (Runtime 30m)

instances. A summary of this evaluation is given in Table 7.3. 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 quality 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 and VLNS.

7.4.2 clingcon search strategy

Now, we offer a comparison between different optimization strategies of clingcon (parameter $--$ `opt` $-$ `strategy`), inspired by the strategies used in [AGM+16].

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 [AGM+16]),

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

(iv) clingcon's default strategy.

Table 7.4 shows the results on the 30 generated test instances. All four strategies produce solutions of identical quality on most instances. On the few instances where they differ, configuration (ii) consistently produced the best solution. The remaining experiments use this strategy.

Configuration	**#feas**	**#opt**	**#best**
`bb`, `hier`	**30**	**18**	28
`bb`, `dec`	**30**	**18**	**30**
`usc`, `3`	**30**	**18**	27
default	**30**	**18**	26

Table 7.4: clingcon Optimization Strategy Experiments (Runtime 30m)

7.5 Comparison of Exact Approaches

Table 7.5 shows the results of our experimental evaluation of the exact solution methods mentioned in this these. Hence, it is a comparison between clingcon, Chuffed, the MIP solver CPLEX and the CP solver CP Optimizer. The time limit was set to 2 hours for each instance. The CASP solver clingcon was run both with a single thread and with 8 solving threads. Chuffed does not support multithreading, so it was only run in with one thread.

#	clingcon		Chuffed	CP Optimizer	CPLEX
	1 thread	8 threads			
1	**98** *	**98** *	**98** *	**98** *	**98** *
2	**73** *	**73** *	**73** *	**73** *	**73**
3	**149** *	**149** *	**149** *	**149** *	**149**
4	**105** *	**105** *	**105** *	**105** *	**105** *
5	**283** *	**283** *	**283** *	**283**	287
6	**162** *	**162** *	**162** *	**162**	**162**
7	**307** *	**307** *	**307** *	**307**	–
8	**310** *	**310** *	**310** *	**310**	317
9	**501** *	**501** *	**501** *	502	–
10	**564** *	**564** *	574	566	–
11	**856** *	**856** *	**856** *	859	–
12	**656** *	**656** *	**656** *	661	–
13	**340** *	**340** *	**340** *	**340**	370
14	**420** *	**420** *	**420** *	424	–
15	**1084** *	**1084** *	1646	1104	–
16	**1138** *	**1138** *	1560	1193	–
17	**1194** *	**1194** *	1254	1203	1242
18	1555	**1358**	1817	1404	–
19	2617	2241	2650	**2099**	–
20	2663	**2140**	2887	2284	–
21	**679** *	**679** *	**679** *	722	–
22	**765** *	**765** *	**765** *	770	–
23	2619	2757	3486	**2207**	–
24	2456	**1859**	2445	1863	–
25	3347	**2425**	3278	2551	–
26	3856	**2600**	3896	2799	–
27	3169	2923	3095	**2338**	–
28	4244	2423	2569	**2402**	–
29	6607	5613	4539	**3718**	–
30	6062	5590	5904	**4995**	–
Lab	4477	3969	5188	**3444**	–

Table 7.5: Comparison of Exact Solution Approaches (Runtime 2h)

CP Optimizer was run with 8 threads and the parameter *FailureDirectedSearchEmphasis* was set to 4. Numbers in boldface indicate that the solution is the best found in this experiment, whereas the star next to an objective value signifies that the respective solver could prove optimality for this instance.

It can easily be seen that CPLEX performed very poorly in comparison to the rest, although it should be noted that our model was developed with CP solvers in mind and thus might not be a perfect fit for MIP solvers. Also, CPLEX was only run in single-threaded mode.

The other solvers on the contrary could find solutions for all instances. However, they differ in the number of optimally solved instances, where clingcon is the clear winner both single and multi-threaded with 19 instances.

In terms of solution quality, CP Optimizer dominates for the larger instances and the real-life instance, but overall clingcon finds the best solutions for most instances with 19 in single and 23 in multi-threaded mode.

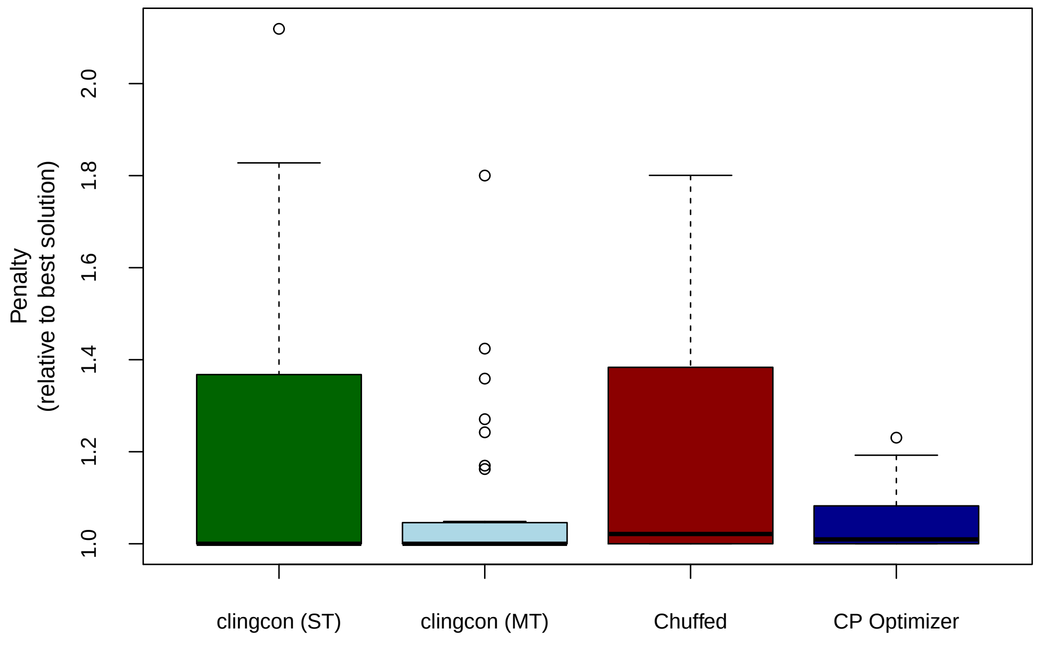

Figure 7.1: Solution Quality Comparison Between Exact Methods

Figure 7.1 further shows the quality of the solutions found by each solver relative to the best known objective values. We omit the results for CPLEX here since – in difference to the other solvers – it did not find feasible solutions for all instances.

7.6 VLNS Parameter Configuration

As described in Chapter 6, we have two basic parameters for VLNS called *jumpProb* and *moveTimeout*. For VLNS using Chuffed we additionally have the parameter *hotStartProb*. The latter and *jumpProb* Both represent probabilities and are thus reals ranging from 0 to 1, whereas *moveTimeout* is given in seconds.

Parameter	**Value Range**	**Chosen Value**
jumpProb	$[0.0, 1.0]$	0.35
moveTimeout	$[0, 300]$	30
hotStartProb	$[0.0, 1.0]$	0.8

Table 7.6: The chosen VLNS parameter configuration

We manually tested different configurations and the trend in the experiments was that a high *hotStartProb* makes the search more stable and thus less reliant on randomness. However, by setting it to 1.0 the search would get stuck in local optima very often. A good compromise between those two extremes turned out to be 0.8. In other words, we hot start from the previously best solution 80% of the moves and in 20% of the time we do not hot start and start the search from a random assignment of timeslots and resources.

Similarly, a very low *jumpProb* can lead to local optima out of which the search cannot escape, but if we set it too high then we might waste a lot of time in larger neighborhoods. In the end, the experiments showed that a *jumpProb* of 35% performed well.

The parameter *moveTimeout* was set to 30 seconds. This timeout was also used for each project in the lower bound calculation as described in Chapter 6.1.

Table 7.6 shows the chosen configuration for the all further experiments.

7.7 VLNS Experiments

In our experiments for our VLNS described in Chapter 6 we tested both clingcon (single threaded) and Chuffed as internal solvers. We used the basic algorithm for clingcon and the modified version given in Chapter 6.3 for Chuffed. We did not use multithreading for clingcon, since our intention was to keep VLNS portable to workstations with limited resources. Furthermore, we did not experiment with CP Optimizer here since it already employs its own VLNS internally [LG07] and in difference to the other solvers it showed difficulty in proving optimality. The latter is rather inconvenient for VLNS as it affects the average time a move needs in a negative way.

The summary of computational results for VLNS are given in Table 7.7. Shown are best and average results out of five runs (2h runtime each) and instances marked with a star are again those which were proven optimal. It can be seen that VLNS using clingcon generally achieves the best results except for one instance, but even where the difference in the solution quality is very small. Furthermore, the same can be said about the average solution quality.

Of particular interest is that Very Large Neighborhood Search with clingcon could prove optimality for 15 instances by finding solutions that match the precomputed lower bounds (see Chapter 6.1) while VLNS with Chuffed proved optimality for 16 instances with the

#	**VLNS with clingcon** **Best**		**Avg**	**VLNS with Chuffed** **Best**		**Avg**
1	**98**	*	98.0	**98**	*	98.0
2	**73**	*	73.0	**73**	*	73.0
3	**149**	*	149.0	**149**	*	149.0
4	**105**	*	105.0	**105**	*	105.0
5	**283**	*	283.0	283		283.0
6	**162**	*	162.0	**162**	*	162.0
7	**307**	*	307.0	307		307.0
8	**310**	*	310.0	310		310.0
9	**501**	*	501.0	**501**	*	501.0
10	**564**	*	564.0	**564**	*	564.0
11	**856**	*	856.0	856		856.4
12	**656**	*	656.0	**656**	*	656.0
13	**340**	*	340.0	**340**	*	340.0
14	**420**	*	420.0	**420**	*	420.0
15	**1084**	*	1084.6	**1084**	*	1085.0
16	**1138**	*	1138.8	**1138**	*	1142.6
17	**1194**	*	1194.2	**1194**	*	1194.4
18	**1356**		1356.4	1359		1378.2
19	**1928**		1942.4	1964		2000.8
20	**2080**		2081.4	2114		2140.8
21	**679**	*	679.0	**679**	*	679.2
22	**765**	*	765.0	765		765.0
23	**1936**		1974.8	2029		2104.2
24	**1773**		1774.8	1776		1798.6
25	**2073**		2104.6	2112		2165.8
26	**2543**		2553.4	2558		2573.8
27	2152		2152.0	**2151**	*	2151.8
28	**2322**	*	2323.2	**2322**	*	2322.6
29	**3118**		3138.4	3233		3297.0
30	**4399**		4401.6	4736		4779.4
Lab	**3195**		3239.6	3334		3400.6

Table 7.7: VLNS Comparison (Runtime 2h)

same method. However, VLNS with clingcon could also show optimality for an additional 5 instances where the optimal solution is above the lower bounds. In four of these last cases, the proof of optimality 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 Very Large Neighborhood Search. The VLNS 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 clingcon's ability to quickly prove optimality of small subproblems and the aggressive strategy for increasing the neighborhood size.

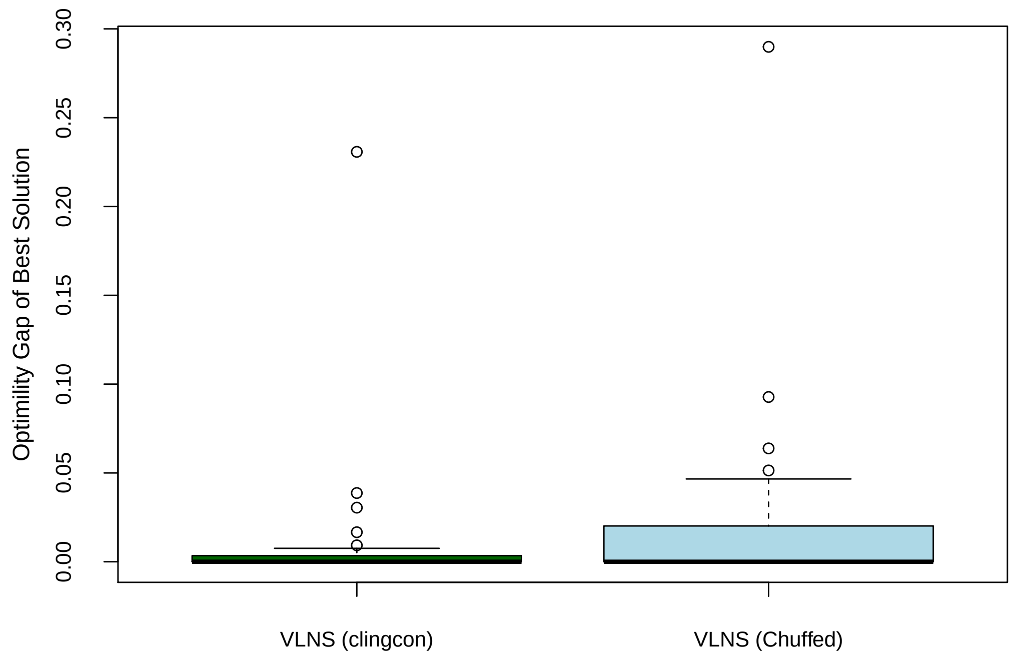

Figure 7.2: VLNS Optimality Gaps of Best Solutions Found

Using the lower bounds, VLNS can not only prove some instances optimal but also give optimality gaps for each best solution found. Figure 7.2 shows the respective gaps of VLNS with clingcon and Chuffed. For instances which were proven optimal despite having a lower bound less than the optimal solution, we also list the gap as zero. It can be seen in the figure that the best solutions found with the clingcon based VLNS are within 5% of the optimum for all instances except one. Similarly, for VLNS with Chuffed we are within 10% of the optimum for all but one instance.

Finally, Figure 7.3 shows the quality of the solutions found by clingcon and Chuffed based VLNS. It can easily be seen that the results for VLNS with clingcon are much better in general.

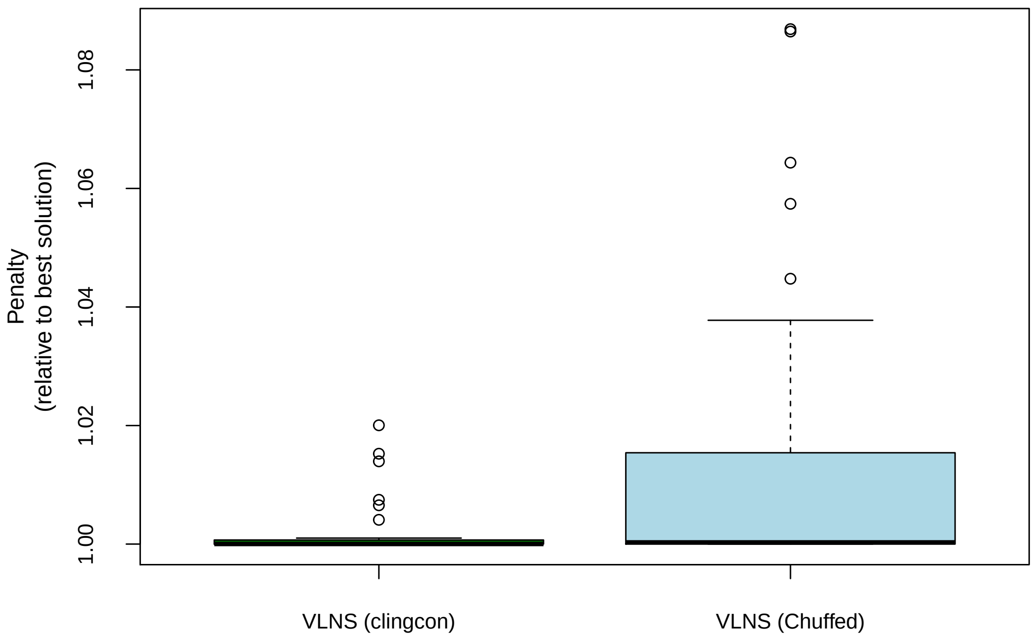

Figure 7.3: VLNS Average Solution Quality Comparison

7.8 Comparison with the State-of-the-Art

We also compared the methods described in this thesis with a metaheuristic method. Namely, a Simulated Annealing (SA) approach introduced by Mischek and Musliu [MM19], which has given very good results for the existing benchmark instances. We ran SA with the same parameter configuration as given by the authors. Furthermore, since the method is nondeterministic, we ran SA five times for each instance with different seeds and in accordance with our other experiments, a time limit of 2h was set for each run. Figure 7.4 shows the full comparison of all of our methods and SA. For VLNS and SA, the numbers are the average over the five respective runs. The detailed results for SA are given in Table 7.8. While SA mostly outperforms all exact approaches, VLNS with Chuffed yields better results for all but one instance and VLNS with clingcon has the best results in general. Also, it should be noted that SA fails to find feasible solutions in a total of 2 out of 155 runs.

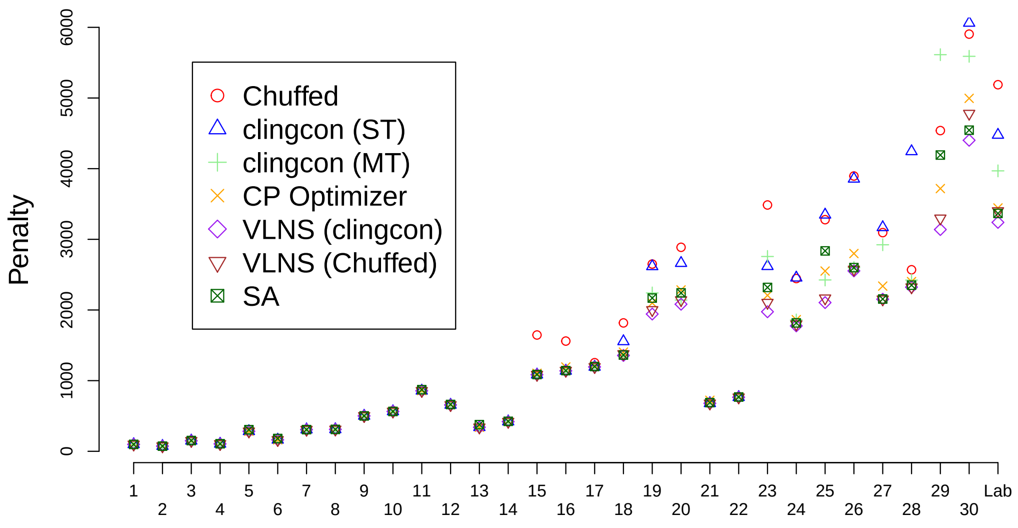

Figure 7.4: Comparison of Solution Approaches with SA

#	SA		
	Best	**#Feas.**	**Avg**
1	98	5	98
2	73	5	73
3	152	5	152.6
4	106	5	107
5	287	5	306.4
6	180	5	181.4
7	307	5	307
8	310	5	310
9	501	5	501
10	564	5	564
11	872	5	872.8
12	660	5	660
13	352	5	377.4
14	423	5	424.2
15	1085	5	1086.8
16	1141	5	1142
17	1195	5	1200.2
18	1359	5	1362.2
19	2127	5	2168.6
20	2228	5	2242.2
21	685	5	689
22	766	5	768.4
23	2209	4	2319.25
24	1780	5	1817.2
25	2701	5	2836
26	2579	5	2599.8
27	2153	5	2155.2
28	2338	5	2351
29	3996	5	4192.6
30	4478	4	4544.75
Lab	3338	4	3366.75

Table 7.8: Simulated Annealing Experiments (Runtime 2h)

CHAPTER 8

Conclusion

In this thesis we investigated different ways to model and solve a complex project scheduling problem appearing in an industrial test laboratory. After giving a formal definition of all hard and soft constraints of the problem, as well as providing basic complexity analysis, we modeled the problem using Constraint Programming (CP) and Constraint Answer-set Programming (CASP).

For CP we reviewed and extended existing modeling techniques from the literature which were used for related project scheduling problems. To deal with this more complex problem and larger instances we introduced several extensions in modeling, including novel redundant constraints and search strategies. In order to compare with another state-of-the-art CP solver, we also proposed an interval-based CP model.

Our CASP encoding is – to the best of our knowledge – the first use of this solving paradigm for project scheduling. Besides giving a basic encoding of the problem, we also provide an alternative formulation for unary resource constraints, which tend to have a big impact on solving.

Furthermore, we introduced a Very Large Neighborhood Search (VLNS) metaheuristic for the problem considered in this thesis. This approach utilizes the exact solving methods and manages to find high quality solutions.

We evaluated our different modeling approaches, search strategies, and solver configurations on 30 randomly generated benchmark instances. Additionally, we provided a comparison of all approaches on these 30 instances as well as 1 real-life instance.

For VLNS, we experimented with different exact repair methods on the same instances. We showed that our approach managed to find the best known solutions for 31 instances, where we could show that for 30 instances the found solution is within 5% of the optimum. Lastly, we also compared VLNS to an existing Simulated Annealing metaheuristic and showed that while exact methods fall behind SA, VLNS is generally superior.

For future work, it would be interesting to extend our CASP models to the full TLSP problem i.e. also do the grouping dynamically. Similar work has already been done for our CP models and VLNS [DGMM20]. Additionally, investigating a combination of VLNS and Simulated Annealing would be a worthwhile endeavor.

Bibliography

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[ADM17] Mario Alviano, Carmine Dodaro, and Marco Maratea. 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*, pages 468–482. Springer, 2017.

[AGM$^+$16] Michael Abseher, Martin Gebser, Nysret Musliu, Torsten Schaub, and Stefan Woltran. Shift design with answer set programming. *Fundamenta Informaticae*, 147(1):1–25, 2016.

[AJO$^+$20] Dirk Abels, Julian Jordi, Max Ostrowski, Torsten Schaub, Ambra Toletti, and Philipp Wanko. Train scheduling with hybrid answer set programming. *Theory and Practice of Logic Programming*, pages 1–31, 2020.

[AKK$^+$16] Shahriar Asta, Daniel Karapetyan, Ahmed Kheiri, Ender Özcan, and Andrew J. Parkes. Combining monte-carlo and hyper-heuristic methods for the multi-mode resource-constrained multi-project scheduling problem. *Information Sciences*, 373:476–498, 2016.

[AM18] Arben Ahmeti and Nysret Musliu. Min-conflicts heuristic for multi-mode resource-constrained projects scheduling. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO 2018)*, pages 237–244. ACM, 2018.

[Bal11] Marcello Balduccini. 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*, pages 284–296. Springer, 2011.

[BDM$^+$99] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

[BKOS17] Mutsunori Banbara, Benjamin Kaufmann, Max Ostrowski, and Torsten Schaub. Clingcon: The next generation. *Theory and Practice of Logic Programming*, 17(4):408–461, 2017.

[BN05] Odile Bellenguez and Emmanuel Néron. 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*, pages 229–243. Springer, 2005.

[BZ09] J.-H. Bartels and J. Zimmermann. Scheduling tests in automotive R&D projects. *European Journal of Operational Research*, 193(3):805–819, 2009.

[Chu11] Geoffrey Chu. *Improving combinatorial optimization*. PhD thesis, University of Melbourne, Australia, 2011.

[DB08] L.-E. Drezet and J.-C. Billaut. A project scheduling problem with labour constraints and time-dependent activities requirements. *International Journal of Production Economics*, 112(1):217–225, 2008. Special Section on Recent Developments in the Design, Control, Planning and Scheduling of Productive Systems.

[DCS18] Emir Demirovic, Geoffrey Chu, and Peter J. Stuckey. Solution-based phase saving for CP: A value-selection heuristic to simulate local search behavior in complete solvers. In *Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP 2018)*, volume 11008 of *LNCS*, pages 99–108. Springer, 2018.

[DGMM20] Philipp Danzinger, Tobias Geibinger, Florian Mischek, and Nysret Musliu. Solving the test laboratory scheduling problem with variable task grouping. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS 2020)*, pages 357–365. AAAI Press, 2020.

[DGMP18] Carmine Dodaro, Giuseppe Galatà, Marco Maratea, and Ivan Porro. 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*, pages 445–459. Springer, 2018.

[DM17] Carmine Dodaro and Marco Maratea. 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*, pages 301–307. Springer, 2017.

[DPRL98] S. Dauzère-Pérès, W. Roux, and J.B. Lasserre. Multi-resource shop scheduling with resource flexibility. *European Journal of Operational Research*, 107(2):289–305, 1998.

[EIK09] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. Answer set programming: A primer. In Sergio Tessaris, Enrico Franconi, Thomas Eiter, Claudio Gutiérrez, Siegfried Handschuh, Marie-Christine Rousset, and Renate A. Schmidt, editors, *Reasoning Web. Semantic Technologies for Information Systems*, pages 40–110. Springer, 2009.

[Elm77] Salah Eldin Elmaghraby. *Activity networks: Project planning and control by network models*. John Wiley & Sons, 1977.

[FFM+16] Gerhard Friedrich, Melanie Frühstück, Vera Mersheeva, Anna Ryabokon, Maria Sander, Andreas Starzacher, and Erich Teppan. Representing production scheduling with constraint answer set programming. In *Operations Research Proceedings 2014*, pages 159–165. Springer, 2016.

[FGS+17] Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter J. Stuckey, and Kenneth D. Young. Priority search with minizinc. In *Proceedings of ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation at CP 2017*, 2017.

[GJ79] Michael R Garey and David S Johnson. *Computers and intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979.

[GKKS12] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[GKKS14] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, and Torsten Schaub. Clingo = ASP + control: Preliminary report. `https://arxiv.org/abs/1405.3694`, 2014.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference on Logic Programming (ICLP 1988)*, pages 1070–1080. MIT Press, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GMM19] Tobias Geibinger, Florian Mischek, and Nysret Musliu. 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*, pages 304–319. Springer, 2019.

[GMR08] J.F. Gonçalves, J.J.M. Mendes, and M.G.C. Resende. A genetic algorithm for the resource constrained multi-project scheduling problem. *European Journal of Operational Research*, 189(3):1171–1190, 2008.

[GOS09] Martin Gebser, Max Ostrowski, and Torsten Schaub. Constraint answer set solving. In *Proceedings of the 25th International Conference on Logic Programming (ICLP 2009)*, volume 5649 of *LNCS*, pages 235–249. Springer, 2009.

[GP$^+$10] Michel Gendreau, Jean-Yves Potvin, et al. *Handbook of metaheuristics*, volume 2. Springer, 2010.

[HB10] Sönke Hartmann and Dirk Briskorn. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010.

[IC17a] IBM and CPLEX. 12.8.0 IBM ILOG CPLEX Optimization Studio CP Optimizer user's manual. `https://www.ibm.com/analytics/cplex-cp-optimizer`, 2017.

[IC17b] IBM and CPLEX. 12.8.0 IBM ILOG CPLEX Optimization Studio CPLEX user's manual. `https://www.ibm.com/analytics/cplex-optimizer`, 2017.

[LG07] Philippe Laborie and Daniel Godard. Self-adapting large neighborhood search: Application to single-mode scheduling problems. *In Proceedings of the 3rd Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2007)*, 2007.

[LRSV18] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. IBM ILOG CP Optimizer for scheduling. *Constraints*, 23(2):210–250, 2018.

[Mac77] Alan K. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977.

[MM18a] Florian Mischek and Nysret Musliu. 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)*, pages 465–467, 2018.

[MM18b] Florian Mischek and Nysret Musliu. 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, 2018.

[MM19] Florian Mischek and Nysret Musliu. A local search framework for industrial test laboratory scheduling. *Submitted to Annals of Operations Research*, 2019.

[Mon74] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Sciences*, 7:95–132, 1974.

[MWW15] Marek Mika, Grzegorz Waligóra, and Jan Węglarz. Overview and state of the art. In Christoph Schwindt and Jürgen Zimmermann, editors, *Handbook on Project Management and Scheduling Vol.1*, pages 445–490. Springer, 2015.

[NR97] Nudtapon Nudtasomboon and Sabah U. Randhawa. Resource-constrained project scheduling with renewable and non-renewable resources and time-resource tradeoffs. *Computers & Industrial Engineering*, 32(1):227–242, 1997.

[NSB$^+$07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007.

[PAM04] Mireille Palpant, Christian Artigues, and Philippe Michelon. Lssper: Solving the resource-constrained project scheduling problem with large neighbourhood search. *Annals of Operations Research*, 131(1):237–257, 2004.

[PR10] David Pisinger and Stefan Ropke. Large neighborhood search. In *Handbook of Metaheuristics*, pages 399–419. Springer, 2010.

[PWW69] A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, 16(1):93–108, 1969.

[R9́6] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI 1996)*, pages 209–215. AAAI Press, 1996.

[RvBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, editors. *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*. Elsevier, 2006.

[Sha98] Paul Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming (CP 1998)*, volume 1520 of *LNCS*, pages 417–431. Springer, 1998.

[SLT18] Christian Schulte, Mikael Lagerkvist, and Guido Tack. Gecode 6.10 reference documentation. `https://www.gecode.org`, 2018.

[SS16] Ria Szeredi and Andreas Schutt. 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*, pages 483–492. Springer, 2016.

[SSD97] Frank Salewski, Andreas Schirmer, and Andreas Drexl. Project scheduling under resource and mode identity constraints: Model, complexity, methods, and application. *European Journal of Operational Research*, 102(1):88–110, 1997.

[ST00] Christoph Schwindt and Norbert Trautmann. Batch scheduling in process industries: an application of resource–constrained project scheduling. *OR-Spektrum*, 22(4):501–524, 2000.

[vHK06] Willem-Jan van Hoeve and Irit Katriel. Global constraints. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 169–208. Elsevier, 2006.

[VPLP$^+$19] Félix Villafáñez, David Poza, Adolfo López-Paredes, Javier Pajares, and Ricardo del Olmo. A generic heuristic for multi-project scheduling problems with global and local resource constraints (RCMPSP). *Soft Computing*, 23(10):3465–3479, 2019.

[WJMW11] Jan Węglarz, Joanna Józefowska, Marek Mika, and Grzegorz Waligóra. Project scheduling with finite or infinite number of activity processing modes – a survey. *European Journal of Operational Research*, 208(3):177–205, 2011.

[WKS$^+$16] Tony Wauters, Joris Kinable, Pieter Smet, Wim Vancroonenburg, Greet Vanden Berghe, and Jannes Verstichel. The multi-mode resource-constrained multi-project scheduling problem. *Journal of Scheduling*, 19(3):271–283, 2016.

[YFS17] Kenneth D. Young, Thibaut Feydy, and Andreas Schutt. 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*, pages 308–317, 2017.