Constraints
https://doi.org/10.1007/s10601-021-09321-w

Cable tree wiring - benchmarking solvers on a real-world scheduling problem with a variety of precedence constraints

Jana Koehler[1,2] · Josef Bürgler[3] · Urs Fontana[3] · Etienne Fux[3] · Florian Herzog[3] · Marc Pouly[3] · Sophia Saller[2] · Anastasia Salyaeva[2] · Peter Scheiblechner[3] · Kai Waelti[3]

Accepted: 30 March 2021 Published online: 15 June 2021
© The Author(s) 2021

Abstract
Cable trees are used in industrial products to transmit energy and information between different product parts. To this date, they are mostly assembled by humans and only few automated manufacturing solutions exist using complex robotic machines. For these machines, the wiring plan has to be translated into a wiring sequence of cable plugging operations to be followed by the machine. In this paper, we study and formalize the problem of deriving the optimal wiring sequence for a given layout of a cable tree. We summarize our investigations to model this cable tree wiring problem (CTW) as a traveling salesman problem with atomic, soft atomic, and disjunctive precedence constraints as well as tour-dependent edge costs such that it can be solved by state-of-the-art constraint programming (CP), Optimization Modulo Theories (OMT), and mixed-integer programming (MIP) solvers. It is further shown, how the CTW problem can be viewed as a soft version of the coupled tasks scheduling problem. We discuss various modeling variants for the problem, prove its NP-hardness, and empirically compare CP, OMT, and MIP solvers on a benchmark set of 278 instances. The complete benchmark set with all models and instance data is available on github and was included in the MiniZinc challenge 2020.

Keywords TSP with precedence constraints · Tour-dependent edge costs · Soft coupled task scheduling · Constraint optimization · Benchmarking CP-SAT · MIP · OMT solvers

✉ Sophia Saller
sophia.saller@dfki.de

[1] Universität des Saarlandes, Saarland Informatics Campus, Saarbrücken, Germany

[2] Deutsches Forschungszentrum für Künstliche Intelligenz (DFKI), Saarland Informatics Campus, Saarbrücken, Germany

[3] Lucerne University of Applied Sciences and Arts, Lucerne, Switzerland

1 Introduction

Cable trees are widely used in industrial products to transmit energy and information between different product parts. For example, cable trees are needed in cars to automate many previously mechanical functions such as moving seats or opening windows and to add new functions such as a voice-controlled navigation or an onboard entertainment system. It is thus not surprising that for example a car like the VW Golf 7 contains 14 cable trees with a total of 1633 cables.

The manufacturing of cable trees usually relies on cheap manual labor performed in low-cost countries where humans plug cables into harnesses following a wiring plan. Only few automated manufacturing solutions exist, which rely on complex robotic machines. These machines execute a sequence of wiring operations that highly qualified technicians develop by analyzing the wiring plan. With the continuing tendency towards customer-specific and resource-efficient just-in-time manufacturing, smaller batch sizes of cable trees need to be manufactured requiring a frequent change of wiring plans, for which wiring sequences should be derived instantly. Scaling up human expertise to such frequent changes is simply impossible, which explains a growing interest in the intelligent automated manufacturing of cable trees. This interest is also nourished by a further miniaturization of cable harnesses, which will make their manual manufacturing impossible.

To wire a cable tree on an automatic cable wiring machine,[1] two problems have to be solved by experienced human technicians today:

1. Layout: At which positions do cable harnesses have to be mounted on the palette such that the robot arm can insert cables into the designated harness cavities?
2. Insertion order: Given a layout of harnesses on the palette, in which order are the cable ends to be inserted into harness cavities such that a robot arm of the machine can fast and safely produce the desired wiring?

This paper presents a solution to the problem of finding an optimal insertion order for a given and fixed layout, called the problem of cable tree wiring CTW. Figure 1 illustrates a wiring situation for a given layout. The large rectangle represents the palette surface. On the palette, we see a number of small rectangles that represent the positions of the cable harnesses (the specified layout) on the mounting palette. Each rectangle contains several light and dark gray numbered rectangular fields. These are the harness cavities into which the cables need to be inserted. Dark gray cavities are filled with cables, whereas light gray cavities are left unused. These unused cavities might be required for other variants of a cable tree. Note that the figure shows a 2D-model of the palette, harnesses, and the desired wiring of cavities abstracting from 3D-geometric information. In reality, cable harnesses and cavities can occur in many different shapes (rectangular, oval, or round). The wiring of the cable tree is illustrated by Bezier curves showing the connections between cavities. Note that these curves do not represent the real geometric dimension nor physical behavior of the cables—cables can be up to several meters long and can have very different diameters and physical properties.

[1] Our problem is motivated by the automatic manufacturing of cable trees on a Zeta machine by Komax AG, Switzerland. This machine is a highly sophisticated robot that prepares and cuts cables, adds contacts, and then inserts the cables into cable harnesses mounted on a palette. A video of such a machine wiring a cable tree can be watched at https://www.youtube.com/watch?v=cvfnb0thjXA.

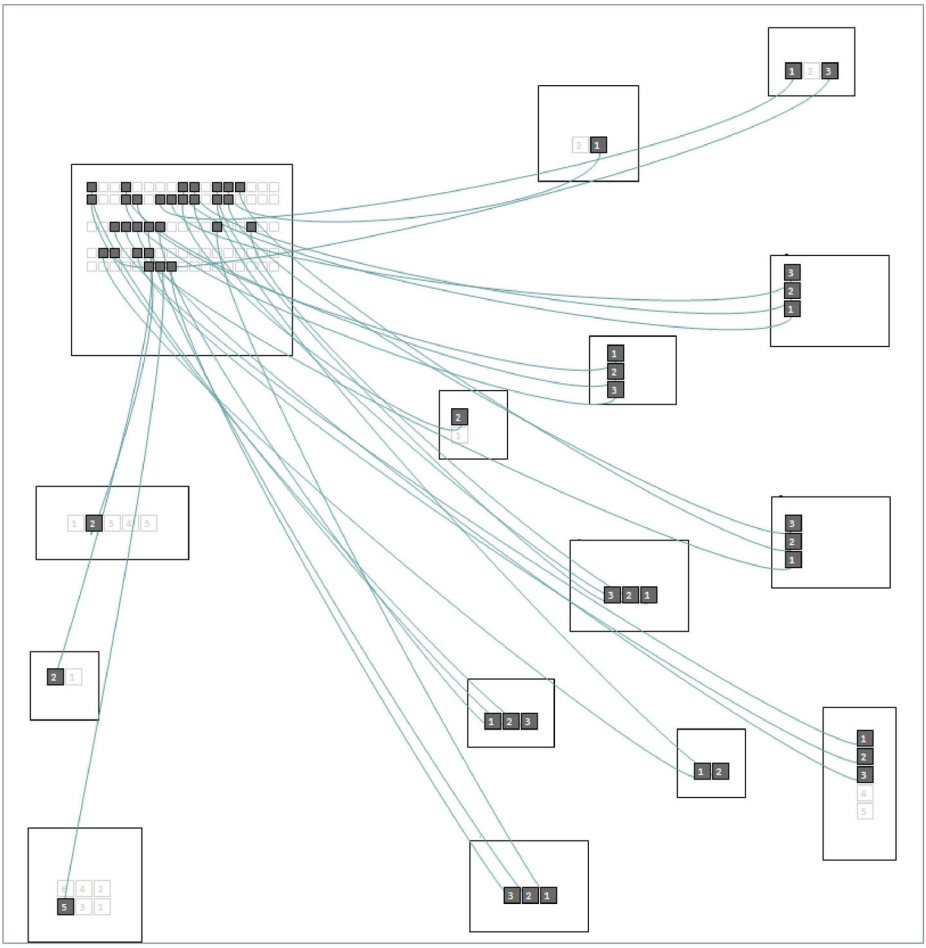

Fig. 1 Layout of a cable tree on a palette and desired wiring

For a given layout, we are looking for an enumeration of the dark gray cavities—a robust and fast sequential order (a permutation) of insertion operations that plug all cables into their designated cavities. Constraints restrict the positions that cavities can take in the permutation and are derived from an analysis of the cable behavior and properties of the machine. For example, a constraint can express that a cable end needs to be inserted into a cavity A after another cable end is inserted into a cavity B as otherwise the robot arm of the machine might not be able to approach cavity B, which can be occluded by the cable plugged into A. We distinguish between constraints and soft constraints, where the latter do not necessarily have to be satisfied, but their violation incurs a penalty. The computation of these constraints is beyond the scope of this paper.

The paper is organized as follows: Section 2 reviews related work. Section 3 formalizes the CTW problem, proves its NP-hardness and identifies subclasses of the problem that can

be solved in polynomial time. Section 4 introduces the CTW benchmark set of 278 real-world and artificial instances used in the MiniZinc challenge 2020. Section 5 summarizes modeling variants for the CTW problem and presents a comprehensive tool chain supporting a benchmarking of constraint, mixed-integer, and optimization modulo theories solvers. Section 6 summarizes our findings from experiments with the following solvers:

- CP-SAT solvers
  - IBM Cplex CP Optimizer 12.10 (C# API) [31]
  - Google OR-Tools CP-SAT Solver 7.5.7466 (Windows Executable) [48]
  - Chuffed 0.10.3 (Windows Batch File/Executable) [15]
- MIP solvers
  - IBM Cplex MIP solver 12.10 (C# API and MiniZinc command line tool)[31]
  - Gurobi 9.0.1 (C# API and MiniZinc command line tool) [26]
- OMT solvers
  - Microsoft Z3 4.8.7 (Windows Executable) [20]
  - OptiMathSAT 1.5.1 (C API) [58]

Section 7 discusses our empirical findings and interesting open research problems. Section 8 concludes the paper.

2 Related work

Permutations represent an abstract characterization of many manufacturing problems where an optimal sequential ordering of a given set of manufacturing steps has to be computed. They have for example been subject to intense research in the context of routing, scheduling, or assignment problems [5, 23, 36, 57, 66]. Frequently, boundary conditions lead to various forms of constraints, notably precedence constraints, which can also occur as disjunctions and which add additional complexity to a problem formulation [1, 4, 50]. To optimally solve such a problem, a constraint satisfaction or mixed-integer programming approach can be followed, but also a number of heuristic approaches have been popular in the literature, e.g., greedy or tabu search [25, 55], ant colony and swarm particle algorithms [24, 59, 61], or simulated annealing [49].

In Section 3, we position our problem as a traveling salesperson problem (TSP) with (disjunctive) precedence constraints (TSPPC) and tour-dependent edge costs. In traditional TSP variants, only static edge costs occur, which are constant and independent of time. Tour-dependent edge costs depend on the tour leading to the edge, i.e., which other cities (cavities for wiring) have or have not been visited before. They are a variant of time-dependent edge costs, which depend only on the time at which the edge is traversed (TDTSP) [51, 62, 63]. The CTW problem belongs to the TDTSPPC class of TSP problems. A related variant of such a problem with time windows is for example studied in [22], variants of vehicle routing problems with time-dependent travel times are studied for example in [21, 28]. Note that time-dependent edge costs can be considered as soft time window constraints.

Precedence constraints also occur in many other settings. For example in vehicle routing problems, when customer visits have to happen in a specific order or within a time window

or when vehicles have to meet. An example for these type of problems is described in [9], however their constraints are different from ours. We were not able to find work that exactly addresses the unique combination of precedence constraints and tour-dependent edge costs as it occurs in the CTW problem.

The CTW problem can also be seen as a variant of the coupled task scheduling problem where a set of jobs consisting of two operations with processing times is given, which should be scheduled on a single machine observing a given time-lag [16, 47]. In the CTW problem, the coupled tasks represent the two operations that insert the ends of a cable. As we discuss in Section 3, we wish the two ends of a cable to be plugged right after one another. In contrast to the coupled task scheduling problem, we do not enforce a time-lag smaller than a given constant for any coupled insertion operations, but we incur two different penalties. One counts the number of decoupled insertion operations and the other depends on the amount of lateness by which the two insertion operations are interrupted. This makes our problem a soft version of the coupled task scheduling problem.

Different approaches to the formulation of permutation problems as CSP problems have been systematically studied in [30, 65], whose results influenced the modeling approach that we present in Section 5. The work in [30, 65] also demonstrated that there is no model that is best suited for all problems, which motivated the comparison of different non-dual and dual modeling variants, which we present in this paper. Another variant, which we use in the context of MIP solvers, is based on a so-called big-M reformulation [10, 55] to effectively rewrite disjunctive constraints. Permutation problems with disjunctive constraints lead to AND/OR constraint graphs for which first analysis techniques have been discussed in [3, 39]. The impact of very large sets of disjunctive constraints on the difficulty of permutation problems has not been studied very widely. Our experiments show that in particular modern constraint solvers handle problems with disjunctive constraints very effectively. An early study of disjunctive precedence constraints is described in [13].

A similar study to ours that empirically compares different MIP solvers on various models for the standard job shop scheduling problem is presented in [34]. These models also contain precedence and disjunctive constraints. Whereas the authors use a well-known problem to investigate the progress made by MIP solvers, we are introducing a novel benchmark set that combines two very different and practically relevant TSP variants in an interesting way and which also seems to be the first known representative of the soft coupled task scheduling problem. Furthermore, our empirical analysis spans over three different classes of solvers, for which we developed an elaborate tool chain supporting the conversation of models and data. Cross-approach comparisons of solvers seem to be rather infrequent. The two solvers that perform best on our benchmark set, the IBM Cplex CP and the Google OR-Tools CP-SAT solver, are compared in [19] on instances of the job shop scheduling problem. Another paper, which compares MIP and CP solvers, is [38], but it focuses on the hospitals/residents problem.

3 Formalization and complexity of the cable tree wiring problem

In the following, we formally define the CTW problem and define valid and optimal solutions of a CTW instance. We investigate the complexity of the problem and prove it to be NP-hard, but also show that specific subclasses of the problem can be solved in polynomial time. Finally, we discuss the relationship to the TSP problem more formally.

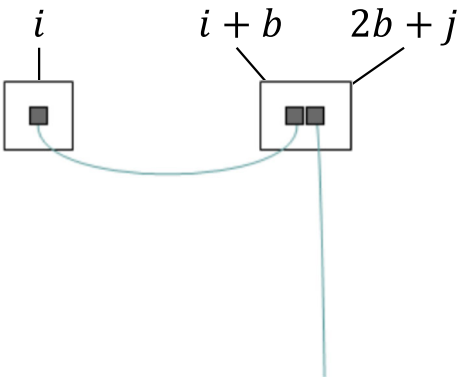

Fig. 2 Labeling of cable ends for a job pair $\langle c_i, c_{i+b} \rangle$ and a one-sided job c_{2b+j}

3.1 Formalization of the cable tree wiring problem

The cable tree wiring problem CTW can be considered as a scheduling problem where an optimal schedule for inserting each given cable end into its designated cavity needs to be determined. Note that two different cable ends can never be inserted into the same cavity and one cable end cannot be inserted into two different cavities. As this matching of cable ends to cavities is given initially, the insertion of cable end i in its designated cavity hence uniquely determines a job c_i, which we formally define as follows:

Definition 1 (Job) A job c_i is defined as the task of inserting a cable end i into its corresponding cavity.

The CTW problem can be considered as the problem of finding an optimal schedule for executing these jobs while satisfying certain constraints. The schedule is described by the permutation sequence of insertions on a single machine executing the jobs as fast as possible. For each insertion operation, the robot arm has to pick up the cable from a fixed position in the storage, then travel to the cavity on the palette, plug the cable, and then travel back to the storage to pick up the other cable end or the next cable. The travel times of the robot arm are identical for all valid permutation sequences, because the layout is given and fixed and thus, they do not need to be considered in the problem formalization.[2]

We consider two types of cables, which we denote as one-sided and two-sided cables. For two-sided cables, both ends must be inserted into cavities. For one-sided cables only one of their ends needs to be inserted into a cavity. We define the following convention for labeling the corresponding jobs, which is illustrated in Fig. 2:

Definition 2 (Labeling of jobs, job pair $\langle c_i, c_j \rangle$) Let b be the number of two-sided cables in a CTW instance. We label the two ends of a two-sided cable i and $j = b + i$, where $i = 1, \ldots, b$. Every two-sided cable hence defines a job pair $\langle c_i, c_j \rangle$ where $i \in \{1, \ldots, b\}$ and $j = b + i$. Let n be the number of one-sided cables in a CTW instance labeled with $i = 2b + 1, \ldots, 2b + n$. The one-sided jobs are hence the jobs c_i where $i = 2b + 1, \ldots, 2b + n$.

[2] When computing the layout of the harnesses on the palette, we indeed minimize travel costs for the robot arm using another optimization solution. The layout optimization problem is out of scope for this paper.

A solution of a CTW instance with $k = 2b + n$ jobs is described by a permutation $\mathcal{P}$ of length k. This permutation sequence is an ordered set in which each job occurs exactly once.

Definition 3 (Solution $\mathscr{P}$) Let $p(c_i)$ be the position of a job c_i in $\mathscr{P}$. $\mathscr{P}$ is a solution of the CTW problem if p is an invertible function from $\{c_i\}_{i\in\{1,\ldots,k\}}$ to $\{1,\ldots,k\}$ such that

$$\forall x \in \{1,\ldots,k\}\ \exists!\, i \in \{1,\ldots,k\} \text{ s.t. } p(c_i) = x \tag{1a}$$

$$\forall i, j \in \{1,\ldots,k\} \text{ with } i \neq j : p(c_i) \neq p(c_j)\,, \tag{1b}$$

where $\exists!$ means that there exists a unique one.

As mentioned in the introduction, the position of a job in the permutation sequence is restricted by the behavior of cables and the machine. To capture these restrictions, constraints are formulated over the jobs. These constraints occur in three different forms:

Definition 4 (Atomic Precedence Constraint, sets $\mathscr{A}$ and $\mathscr{A}_s$) Let c_i and c_j be two jobs in a CTW instance. An atomic precedence constraint $c_i \lhd c_j$ with $i, j \in \{1,\ldots,k\}$ and $i \neq j$ specifies that job c_i must be executed before job c_j. The set of all atomic precedence constraints in a CTW instance is denoted by $\mathscr{A}$. We further define another set of soft atomic precedence constraints $\mathscr{A}_s$, which do not necessarily have to be satisfied by a valid solution permutation $\mathscr{P}$, but will lead to a penalty when violated. The two sets are disjoint, i.e., $\mathscr{A} \cap \mathscr{A}_s = \emptyset$. A solution $\mathscr{P}$ satisfies a (soft) atomic precedence constraint $c_i \lhd c_j \in \mathscr{A} \cup \mathscr{A}_s$ if and only if $\mathscr{P} \vdash p(c_i) < p(c_j)$.

Given any arbitrary pair of jobs c_i and c_j, an atomic precedence constraint $c_i \lhd c_j$ or $c_j \lhd c_i$ may or may not occur in a problem instance. Sometimes, even both constraints can occur, which renders a problem instance unsatisfiable, because for example the layout of harnesses was chosen in an unfortunate way.

Disjunctive precedence constraints $\mathscr{D}$ combine two atomic precedence constraints in a disjunction and occur in a limited syntactic form in CTW.

Definition 5 (Disjunctive Precedence Constraint, set $\mathscr{D}$) Given a job pair $\langle c_i, c_j\rangle$ of a two-sided cable and a job c_l from another one-sided or two-sided cable, two syntactic forms of disjunctive precedence constraints can occur in a CTW instance:

1. $\mathscr{D}_1 : c_l \lhd c_i \vee c_l \lhd c_j$
2. $\mathscr{D}_2 : c_l \lhd c_i \vee c_j \lhd c_l$ or switching i and j: $c_l \lhd c_j \vee c_i \lhd c_l$

For $l \in \{1,\ldots,k\}$ and a job pair $\langle c_i, c_j\rangle$ with $i \in \{1,\ldots,b\}$ and $j = i + b$, a solution $\mathscr{P}$ satisfies the constraint

- $(c_l \lhd c_i \vee c_l \lhd c_j) \in \mathscr{D}_1$ if and only if $\mathscr{P} \vdash p(c_l) < p(c_i)$ or $\mathscr{P} \vdash p(c_l) < p(c_j)$
- $(c_l \lhd c_i \vee c_j \lhd c_l) \in \mathscr{D}_2$ if and only if $\mathscr{P} \vdash p(c_l) < p(c_i)$ or $\mathscr{P} \vdash p(c_j) < p(c_l)$.

Note that two constraints of the form $c_l \lhd c_i \vee c_l \lhd c_j$ and $c_i \lhd c_l \vee c_l \lhd c_j$ (c_l and c_i flipped) can never occur in the set $\mathscr{D}_1$ for the same problem instance, because a cable has at most two ends, i.e., two job pairs $\langle c_i, c_j\rangle$ and $\langle c_l, c_j\rangle$ cannot exist together in the same instance. Furthermore, note that whenever a disjunctive constraint for three jobs c_i, c_j, c_l occurs in $\mathscr{D}_1$ with $\langle c_i, c_j\rangle$ being a job pair, a constraint in $\mathscr{D}_2$ can never be present for the same three jobs and vice versa. For the set $\mathscr{D}_2$, both variants of the constraint with c_i and c_j being flipped can occur within the same instance.

Direct successor constraints formalize coupled tasks in a CTW instance. A Zeta machine can insert the end i of one cable into a cavity, put the other end of the same cable j into storage for later insertion, and continue to work on a new cable. Some cables are too short for storing and thus require that the wiring of the cable must proceed without interruption. Note that direct successor constraints are specific to cavities, not cables. Depending on the position of the cavity on the palette, storage of one end might be possible, but storage of the other end might be not.

Definition 6 (Direct Successor Constraint, set $\mathscr{S}$) Let $\langle c_i, c_j \rangle$ be a job pair. A direct successor constraint $c_i \blacktriangleleft c_j$ (or $c_j \blacktriangleleft c_i$) formulates an atomic precedence constraint, which requires that the cable end j of a two-sided cable is immediately inserted after i is inserted or anytime before i (or i to follow immediately after or anytime before j respectively). A solution $\mathscr{P}$ satisfies a direct successor constraint $c_i \blacktriangleleft c_j$ for a job pair $\langle c_i, c_j \rangle$ if and only if $\mathscr{P} \vdash p(c_j) = p(c_i) + 1$ or $\mathscr{P} \vdash p(c_j) < p(c_i)$.

Direct successor constraints can also be formulated as atomic precedence constraints by adding the atomic constraint $c_i \triangleleft c_j$ to $\mathscr{A}$ and additional disjunctive constraints of the syntactic form $\mathscr{D}_2$ to specify that all other cavities must either come before or after the job pair $\langle c_i, c_j \rangle$, however, this formulation is less compact:

$$p(c_i) = p(c_j) + 1 \leftrightarrow c_i \triangleleft c_j \wedge \forall\, c_l (i \neq l \neq j)\ c_l \triangleleft c_i \vee c_j \triangleleft c_l \tag{2}$$

We now define a valid solution of a CTW instance as a solution, which satisfies all constraints defined for the instance.

Definition 7 (Valid Solution) Let I be a CTW instance with b two-sided and n one-sided cables, i.e., $k = 2b + n$ jobs, and sets of constraints $\mathscr{A}$, $\mathscr{A}_s$, $\mathscr{D}$, and $\mathscr{S}$. A permutation $\mathscr{P}$ is a valid solution for I if and only if $\mathscr{P}$ satisfies all constraints in $\mathscr{A} \cup \mathscr{D} \cup \mathscr{S}$. Note that if $k = 0$, all constraint sets are empty and any permutation of length 0 is a solution for the instance.

As an illustrating example consider a CTW instance with cables A, B and C, where A and B are two-sided cables defining the job pairs $\langle c_1, c_3 \rangle$ and $\langle c_2, c_4 \rangle$ respectively and C is a one-sided cable defining the job c_5. This instance has parameters $k = 5$ and $b = 2$ and is subject to the following constraints:

$$\begin{gathered} c_3 \triangleleft c_4 \\ c_4 \triangleleft c_1 \\ c_5 \triangleleft c_4 \\ c_2 \triangleleft c_5 \vee c_2 \triangleleft c_1 \\ c_4 \blacktriangleleft c_2 \end{gathered}$$

Of the $5! = 120$ possible job permutations, only 8 are valid solutions satisfying all the constraints. One example of such a valid solution is the permutation $(c_5, c_3, c_4, c_2, c_1)$.

3.2 Optimal solutions of CTW instances

In practice, one is not only interested in valid solutions, but in solutions of minimal cost. Four different cost functions are of interest. The first three cost functions aim to increase the robustness of the cable wiring actions by introducing penalties when a job pair is interrupted,

because interrupting the processing of a job pair and storing cables into storage can increase the risk of the robot arm in getting caught in stored cables. Furthermore, putting a cable end into storage and working on another cable, which is captured by the first cost function, can increase production time. The fourth function aims at keeping the violation of soft atomic constraints at a minimum. On the one hand, violating a soft atomic constraint allows the machine to make more flexible moves during wiring, but on the other hand, it can impact robustness negatively.

1. S = Number of interrupted job pairs, i.e., cable ends that are temporally added to the cable storage by the machine to pick up another cable for insertion.
2. M = Maximum number of cables that are contained in storage simultaneously.
3. L = Longest time a cable end resides in storage expressed in terms of number of jobs.
4. N = Number of violated soft atomic precedence constraints in $\mathscr{A}_s$.

The formalization of the four criteria S, M, L and N, makes repeated use of the indicator function $\mathbb{I}$. An indicator function $\mathbb{I}_A(x)$ for a set A returns 1 when x lies in the set A, and 0 otherwise. More formally, the indicator function $\mathbb{I}_A$ of a set A is defined as

$$\mathbb{I}_A(x) = \begin{cases} 1 & \text{if } x \in A \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

Criterion S counts how many of the job pairs were interrupted in a solution.

$$S = \begin{cases} 0 & \text{if } b = 0 \\ \sum_{i=1}^{b} \mathbb{I}_{\mathcal{S}}(i) & \text{otherwise} \end{cases} \tag{4}$$

where $\mathcal{S} = \{k \mid |p(c_k) - p(c_{k+b})| > 1\}$ and $\mathbb{I}_{\mathcal{S}}(x)$ is the indicator function of the set $\mathcal{S}$.

To determine the criterion M, i.e., the maximum number of cables that are stored simultaneously, we have to count for each job c_l in the solution, how many job pairs exist where one end is plugged before c_l, but the other end is plugged after c_l, i.e., how many job pairs are interrupted by a given job c_l.

$$M = \begin{cases} 0 & \text{if } b = 0 \\ \max\limits_{l \in \{1,\ldots,2b+n\}} \sum_{i=1}^{b} \mathbb{I}_{\mathcal{M}_i}(l) & \text{otherwise} \end{cases} \tag{5}$$

where $\mathcal{M}_j = \{l \mid \min\{p(c_j), p(c_{j+b})\} < p(l) < \max\{p(c_j), p(c_{j+b})\}\}$ and $\mathbb{I}_{\mathcal{M}_j}(l)$ is the indicator function of the set $\mathcal{M}_j$. Note that in this case, the set $\mathcal{M}_j$ depends on j.

The optimization criterion L is determined by measuring the number of jobs between job c_i and job c_{i+b} for each $i \in \{1, \ldots, b\}$, i.e., for each job pair, and takes the maximum value obtained for all job pairs. Note that if there are no jobs between two cable ends, the cable is plugged directly without accessing the storage.

$$L = \begin{cases} 0 & \text{if } b = 0 \\ \max\limits_{i \in \{1,\ldots,b\}} (|p(c_i) - p(c_{i+b})| - 1) & \text{otherwise} \end{cases} \tag{6}$$

For criterion N, we count the number of violated soft atomic constraints from the set $\mathscr{A}_s$, so

$$N = \sum_{k \lhd l \in \mathscr{A}_s} \mathbb{I}_{\mathcal{N}}(k, l) \tag{7}$$

where $\mathcal{N} = \{(x, y) \mid p(c_x) > p(c_y)\}$ and $\mathbb{I}_{\mathcal{N}}$ is the indicator function of the set $\mathcal{N}$.

Assuming $k \neq 0$, all four criteria are bounded by a lower value of 0 and an upper value depending on the size of the solution $k = 2b + n$ as follows:

$0 \leqslant S \leqslant b < k$

$0 \leqslant M \leqslant b < k$

$0 \leqslant L \leqslant k - 1 < k$

$0 \leqslant N \leqslant \frac{k \cdot (k-1)}{2} < k^2$ as there can be at most $\frac{k \cdot (k-1)}{2}$ soft atomic precedence constraints in a solution, otherwise the constraint graph contains a cycle and the instance is unsatisfiable.

As our objective function, we define the weighted sum of the four criteria S, M, L and N. By the weight assigned to each criterion, we want to ensure that their possible values fall into non-overlapping intervals, in order to ensure that solvers prioritize optimizing for the criteria based on a fixed ranking order: S before M before L before N. The value of N for the instances in the benchmark set that we consider is more tightly bound than the worst possible bound, i.e., in the benchmark set we always have $N < k$ instead of $N < k^2$. It is thus sufficient to weight the four objectives by powers of k to eliminate the influence of one to the others. For all experiments in this paper, we therefore use the weighted sum as defined in (8) below as optimization objective:[3]

$$k^3 \cdot S + k^2 \cdot M + k \cdot L + N \tag{8}$$

For the example considered above and the solution $(c_5, c_3, c_4, c_2, c_1)$, this formula returns costs 161 ($S = M = N = 1, L = 2$), which makes this solution one of the two optimal solutions for this instance.

3.3 NP-hardness of the CTW problem

We now prove that the CTW problem is NP-hard by a reduction from the Maximum Acyclic Subgraph problem. Note that for this reduction, we assume the presence of soft atomic constraints and one-sided cables.

Theorem 1 (NP-hardness of CTW) *The Cable Tree Wiring Problem (CTW) is NP-hard.*

Proof of Sketch We prove NP-hardness by a reduction from the Maximum Acyclic Subgraph problem (MAS). Let us assume an MAS instance where we are given a directed graph $G = (V, E)$, where $V = \{1, \ldots, n\}$ is the set of vertices of G and E is the set of directed edges of G. A solution to the MAS problem is a maximal set $E' \subseteq E$ of edges such that the resulting graph $G' = (V, E')$ is a directed acyclic graph. This problem is known to be NP-hard [33].

We now construct a CTW instance as follows. Let the set of vertices V correspond to jobs of one-sided cables $c_1, \ldots, c_n$. For each of the edges $e = (v, w) \in E$, where v corresponds to job c_j and w to job c_k, we introduce a soft precedence constraint $c_j \lhd c_k$. The graph G hence corresponds to the constraint graph of the CTW instance.

[3] Alternatively, one can use lexicographic optimization [2], but not all solvers considered in this paper support it.

The solution to the CTW problem is then a permutation $\mathscr{P}$ of the jobs with minimal cost, so with the fewest violated soft precedence constraints (since we have no pairs of jobs in this instance). Let E' be the set of all edges corresponding to soft atomic precedence constraints that are not violated by the permutation. By the optimality of the solution for the CTW instance, the set E' is the largest such set permitting a valid wiring sequence. Note further that a valid solution is possible if and only if the constraint graph has no cycles.

The set $E \setminus E'$ is hence the smallest set of edges that needs to be removed from G to obtain a directed acyclic graph, and so $G' = (V, E')$ is the maximum acyclic subgraph of G. The solution to the CTW hence gives a solution to the Maximum Acyclic Subgraph problem. Since the solution size is clearly polynomial in the size of the MAS instance, it is a polynomial many-one reduction, proving that CTW is NP-hard. □

Since our proof relies on the presence of soft atomic constraints, the complexity of the CTW problem without soft atomic precedence constraints is still open. Note further, that NP-completeness of the corresponding CTW decision problem follows from Theorem 1. For the special case that only (hard) atomic precedence constraints and no two-sided cables occur in an instance, the CTW problem can be solved in polynomial time.

Lemma 1 (CTW with only hard atomic constraints) *A CTW instance with only one-sided cables and $\mathscr{A}_s = \mathscr{D} = \mathscr{S} = \varnothing$ can be solved in polynomial time.*

Proof of Sketch First construct the constraint graph of the CTW instance by creating a vertex i for every job c_i and adding a directed edge from i to j to the graph if and only if the precedence constraint $c_i \triangleleft c_j$ exists in $\mathscr{A}$. By applying Kahn's algorithm [32] to the constraint graph, we obtain a topological ordering, which is necessarily a valid permutation satisfying all the precedence constraints. Note that Kahn's algorithm has time complexity $O(k + e)$, where k is the number of jobs and e is the number of constraints in the CTW instance. The CTW instance can thus be solved in polynomial time. □

Based on the relationship to the Traveling Salesperson problem that we discuss in the next subsection, we conjecture that the CTW problem is NP-hard once two-sided cables are added even if only the set $\mathscr{A}$ of atomic precedence constraints is non-empty and all other constraint sets are empty. We also conjecture that the CTW problem restricted to containing only disjunctive precedence constraints is NP-hard. Disjunctive precedence constraints can be compiled away by converting the constraint set of disjunctive precedence constraints into disjunctive normal form (DNF) where each disjunct only contains atomic constraints. The compilation can lead to an exponential "blow-up" in the number of disjuncts only containing atomic precedence constraints [40]. Solution length remains polynomially bounded in this potentially larger search space, i.e., membership in NP remains unchanged. Proving these conjectures is not straightforward as the CTW problem is a TSP variant with only tour-dependent edge costs, but no static edge costs, for which we could not find any formal complexity proofs.

If we, however, restrict a CTW instance to only contain direct successor constraints, it can be solved in linear time.

Lemma 2 (CTW with only direct successor constraints) *A CTW instance with $k = 2b + n$ jobs and b two-sided and n one-sided cables where $\mathscr{A} = \mathscr{A}_s = \mathscr{D} = \emptyset$ and $\mathscr{S} \neq \emptyset$ can be solved in linear time $O(k)$.*

Proof of Sketch As a direct successor constraint can only be defined for an end of a two-sided cable, an optimal solution permutation with cost 0 can be constructed by first wiring all job pairs of the two-sided cables without interruption and then adding the jobs for the one-sided cables:

$$\mathscr{P} = c_1, c_{1+b}, c_2, c_{2+b}, \ldots, c_b, c_{2b}, c_{2b+1}, \ldots, c_{2b+n}$$

This sequence ensures that if a direct successor constraint is defined for some job c_j, either the other cable end is directly following or preceding it. □

3.4 Relationship of CTW to TSP

The CTW problem can be considered as a variant of the traveling salesman problem (TSP) where cavities represent cities that need to be visited. TSP variants with precedence constraints have been studied in a number of papers with [35] being one of the first references, but see also [42, 52] for more recent overviews. The CTW problem can also be considered as a variant of the pickup and delivery TSP problem, where the pickup and delivery locations can be exchanged with each other, but the tour should visit them directly. Closely related variations of the pickup and delivery problem have been studied in [11, 60, 64], but none of them is identical to the CTW variant.

In the underlying graph, a city (job) is connected with any other city unless a precedence constraint $c_i \prec c_j$ is given, which removes the edge $c_j \mapsto c_i$ from the graph and also excludes all (sub)paths $c_j \cdots \mapsto \cdots c_i$ from the tour. Direct successor constraints remove even more edges from the graph, enforcing the edge $c_i \mapsto c_j$ as the only outgoing edge of c_i in the graph. In the resulting partially connected graph, we need to compute a hamiltonian path, but not a hamiltonian cycle, because we do not need to return to the starting job after having visited each job exactly once. Computing a hamiltonian path on general graphs is NP-complete. On complete graphs, i.e., in the unlikely case that no precedence constraints are given, the problem can be solved in linear time [56].

Note that the time to complete each wiring operation is independent of the position of a cavity in the permutation, because the robot arm of the machine begins each wiring operation from the storage location, where the cables are prepared, moves to the cavity, and back to the storage. Travel times can only be influenced by choosing a different layout, i.e., moving a harness on the palette closer to the cable storage, but once the layout is fixed, travel costs are identical for all permutations. Thus, travel times in a CTW instance are constant and do not need to be considered in the problem formalization. This makes our problem at first glance different from routing problems, which usually minimize travel time. However, we will show now how to encode our storage costs as edge costs.

Recall, that we have defined the underlying graph of this problem as follows: Each node represents a job and two nodes are connected unless a precedence constraint $c_i \prec c_j$ is given, which removes the edge $c_j \mapsto c_i$. A direct successor constraint between c_i and c_j further removes all outgoing edges from c_i except the edge $c_i \mapsto c_j$. Given a job pair of a two-sided cable $\langle c_i, c_j \rangle$ and a further job c_l where $l \neq i, j$, we define the edge costs as follows:

- Edges $c_i \mapsto c_j$ and $c_j \mapsto c_i$ have costs 0, because we plug the two ends of the two-sided cable immediately after one another.
- For an edge $c_i \mapsto c_l$ with $l \neq j$, costs are either 0 or 1 depending on whether the other end of the cable was visited on the tour to c_i or not.

- If c_j was visited on the tour to c_i, then edge $c_i \mapsto c_l$ has cost 0 because c_i is the "second" end of the cable that we take out of storage for plugging.
- If c_j was not visited, then edge $c_i \mapsto c_l$ has cost 1 because we put c_j into storage to work on the cable for c_l.

In the same way, costs are assigned to $c_j \mapsto c_l$. Intuitively, the edge cost is equal to 1 if a "remaining" cable end is put into storage and a new cable is picked for wiring, otherwise the edge cost is 0.

Let $1, \ldots, k$ be the cable ends in a CTW instance with corresponding jobs $c_1, \ldots, c_k$. Suppose further, as before, that b of the cables are two-sided. Let $q(x)$ denote the inverse of the function p introduced in Definition 3, i.e., for a given position x, the function q returns the job c_j at position x in the permutation sequence $\mathscr{P}$. We define a tour $\mathscr{P}$ in the graph to be a sequence of jobs (which correspond to nodes) $q(1), \ldots, q(k)$, where for each $x \in \{1, \ldots, k-1\}$ an edge between node $q(x)$ and $q(x+1)$ exists. A solution tour is Hamiltonian and visits every node in the graph exactly once.

Let $\mathfrak{c}\,(q(x) \mapsto q(x+1))$ be the cost of an edge $q(x) \mapsto q(x+1)$. S is the total number of interrupted job pairs, so the number of edges with cost 1 occurring on the path. Hence, S can be defined as

$$S = \sum_{x=1}^{2b+n-1} \mathfrak{c}\,(q(x) \mapsto q(x+1)) \tag{9}$$

We show that this definition of S in (9) is equivalent to our original definition of S in (4). Equation (9) can be rewritten as

$$= \sum_{x=1}^{2b+n-1} \mathbb{I}_K(x)$$

where $K = \{x \mid$ other end of the cable end corresponding to job $q(x)$ is put into storage$\}$. K is the set of all positions x in the tour such that the edge $q(x) \mapsto q(x+1)$ has cost 1. An edge $q(x) \mapsto q(x+1)$ has cost 1 if and only if the other end of the cable end plugged in job $q(x)$ has not been plugged before $q(x)$ and is not plugged right after $q(x)$, which is equal to

$$= \sum_{j=1}^{b} \sum_{x=1}^{2b+n-1} \mathbb{I}_{K_j^+}(x) + \mathbb{I}_{K_j^-}(x)$$

where $K_j^+ = \{x \mid q(x) = j \wedge x+1 < p(c_{j+b})\}$ and $K_j^- = \{x \mid q(x) = j+b \wedge x+1 < p(c_{j-b})\}$. K_j^+ is the set of positions in the permutation (tour) where a two-sided cable end j, where j is at most b, is plugged and such that the other end of j has not been plugged before $q(x)$ and is not plugged right after $q(x)$. The set K_j^- is defined similarly for j between $b+1$ and $2b$. This can then be further simplified to

$$= \sum_{j=1}^{b} \left(\sum_{x=1}^{2b+n-1} \mathbb{I}_{K_j^+}(x) \right) + \left(\sum_{x=1}^{2b+n-1} \mathbb{I}_{K_j^-}(x) \right)$$

which again can be simplified by joining the sets K_j^+ and K_j^- for all x.

$$= \sum_{j=1}^{b} \mathbb{I}_{S^+}(j) + \mathbb{I}_{S^-}(j)$$

where $S^+ = \{j \mid p(c_j) + 1 < p(c_{j+b})\}$ and $S^- = \{j \mid p(c_{j+b}) + 1 < p(c_j)\}$. With $S^+ \cup S^- = S$ and $S = \{i \mid |p(c_i) - p(c_{i+b})| > 1\}$ as defined in (4), we obtain

$$= \sum_{j=1}^{b} \mathbb{I}_S(j)$$

as required, which shows that the cable tree wiring problem can indeed be seen as a variant of the TSP problem combining precedence constraints and time-dependent edge costs.

4 The CTW benchmark set

The CTW benchmark set comprises 205 real-world and 73 artificial instances of cable tree wiring problems. Each instance is defined by constants k and b and its constraint sets. The original cable tree, i.e., the geometry of harnesses or cables is not contained in the instance description. Real-world examples originate from cable trees produced on the machines mostly for automotive applications. Artificial examples were constructed by industry specialists to highlight specific challenges when wiring cable trees during the development of the solution described in this paper. The benchmark set contains the following subsets:

- satisfiable: a set of 71 artificial and 185 real-world instances,
- unsatisfiable: a set of 2 artificial and 20 real-world instances where the layout generates contradicting constraints,
- challenge: a small subset of 5 artificial and 5 real-world instances from the satisfiable set where solvers have difficulties finding an optimal solution.

Table 1 shows the number of two-sided cables and the number of atomic, soft atomic, and disjunctive constraints for each instance in the challenge set. No instance of the challenge set contains direct successor constraints or one-sided cables, but they occur in 80 of the other benchmark instances. To give an idea on how many constraints apply when choosing a position value for a job in a permutation, we developed the notion of *constrainedness* (c-ness), which is inspired by the clause/variable ratio used to characterize the hardness of SAT instances [41]. For each job, we determined the number of atomic or disjunctive

Table 1 Parameters of the 10 challenge set instances

Instance	Two-sided Cables b	Atomic Constraints	Soft atomic Constraints	Disjunctive Constraints	Constraint Sum	Average c-ness	Max c-ness
A033	80	418	74	224	756	8.4	34.0
A060	100	2471	86	339	2,946	29.9	89.5
A066	170	7651	156	842	8,734	52.1	150.0
A069	186	9313	172	971	10,549	57.5	166.0
A073	198	9870	184	1211	11,364	56.8	168.0
R192	104	1270	74	197	1,593	14.7	64.5
R193	104	1056	86	99	1,293	11.9	43.5
R194	112	1471	81	204	1,812	15.5	68.5
R195	110	1525	77	243	1,900	16.7	70.5
R196	110	1416	79	201	1,751	15.3	67.5

constraints where the job occurs on the left-hand side of the constraint. Occurrence in an atomic constraint counts as 1, occurrence in one disjunct within a disjunctive constraint counts as 0.5. We calculated the maximum, and average constrainedness over all jobs of an instance. The parameter gives a rough indication of the difficulty of an instance when considered together with the permutation length, i.e., parameter $2b + n$.

A better way of predicting the difficulty is to just determine the *constraint sum* occurring in an instance, which we define as the sum of the number of two-sided cables, the number of atomic, soft atomic constraints, disjunctive constraints, and direct successor constraints. The number of two-sided cables is added to this sum as two-sided cables can also be viewed as soft direct successor constraints, because a penalty occurs if two ends of a two-sided cable are not plugged directly after one another. Using the constraint sum, a correlation with the solving state for each solver can be observed, see Section 7. The constraint sum in the CTW Benchmark set ranges from 0 to 11,766. It is an interesting question of how these preliminary measures can be improved to accurately predict the difficulty of a CTW instance. Note that the numbering of instances in the benchmark set does not reflect their difficulty in terms of the parameters discussed above.

The CTW benchmark set contains a variety of different instances. 20 instances require to compute permutations of length smaller than 10, 239 instances have a permutation length between 10 and 100, and 19 instances have a length of over 100 and up to 198 jobs. Real-world instances mostly range between 20 and 50 jobs with an average permutation length of 43. 40 instances contain one-sided cables ($k > 2b$) and all of them except one (A008) are real-world instances. The three largest satisfiable instances A071, A072, and A073 contain around 10,000 atomic, over 1,000 disjunctive, and nearly 200 soft atomic constraints, but no direct successor constraints. One of them, A073 is also part of the challenge set.

The 22 unsatisfiable instances contain 2 artificial and 20 real-world instances. The largest of them have a permutation length between 70 and 80, over 1,000 atomic and around 150 disjunctive constraints. Average permutation length is 51 and the smallest unsatisfiable instance has permutation length 6. All of them contain direct successor constraints. An overview of the average parameters and average solution time using CP solvers (across all models for these solvers) for the unsatisfiable instances is given in Table 2. Besides unsatisfiable instances, we show the same information for ten of the easiest to solve satisfiable instances. For this easy satisfiable set, we take the ten satisfiable instances with smallest average solving time across all models on all CP solvers considered in this paper. The largest of these instances has permutation length 10, 30 atomic and 2 disjunctive constraints. Further information on the unsatisfiable and easy satisfiable instances can be found in Appendix E. The benchmark set also contains a few pathological cases, such as for example instance R001 with no cables ($k = b = 0$) and instance R002 with just a single one-sided cable ($k = 1, b = 0$).

The complete benchmark set with all models and instance data can be downloaded from https://github.com/kw90/ctw_toolchain. Furthermore, this site also contains the code

Table 2 Average parameters of easy to solve satisfiable and all unsatisfiable instances

	Two-sided cables b	Atomic constraints	Soft atomic constraints	Disjunctive constraints	Constraint sum	Average CP solving time
Easy satisfiable	1.80	5.90	2.20	0.40	10.60	0.25s
Unsatisfiable	11.50	136.75	16.83	24.00	198.67	1.75s

and documentation of the tool chain that we developed for the conversion of models and data into the different formats required by the various solvers, see also Section 5. We also included an Excel file, which summarizes the parameters of all instances as well as all solver results that form the basis for Section 6. The CTW benchmark set was also included in the MiniZinc challenge 2020.

5 Modeling the CTW problem

Our models are original work by the authors and are based on a so-called quadratic permutation representation of the TSP as described in [27]. The models closely follow the formalization of the problem to provide a base line for an empirical study with various solvers. This also implies that we represent disjunctive precedence constraints and direct successor constraints directly without rewriting them as described earlier. Other modeling variants can be imagined, e.g., exploiting the relation to the TSP problem even more directly, modeling the problem as a scheduling problem, using a linear permutation representation, or a convex-hull encoding just to name a few. We focus on two major modeling approaches: a rather "natural" and "native" model **M** of the problem and an extension of this native model to a dual model **DM** exploiting a dual problem representation. For each solver, variants of the models **M** and **DM** are created as there is no single modeling language, which all solvers would support. We do not, however, try to achieve the best possible model for a specific solver, which would allow this solver to perform best on the problem. As such, our experiments provide a base line for the comparison of different solvers on the CTW problem, but they do not generalize to wider conclusions about the scalability or performance of a solver beyond the specific model or CTW problem.

In this section, we describe the constraint models **M** and **DM**. The model **DM** is used in the cable tree wiring solution with the IBM Cplex CP constraint solver and written in OPL, the proprietary language of Cplex. Using OPL had the advantage that the model could be easily reviewed and discussed with domain experts, because of the compact and natural representation of data structures and constraints provided by OPL. Based on the models **M** and **DM**, a number of further modeling variants in other languages were developed in order to compare this model and the results obtained with Cplex CP with those from other solvers. In the following, we summarize the derivation process of all model variants and give a short overview on their main characteristics. Appendix A provides further details.

5.1 The constraint models M and DM

The model **M** uses integer variables to represent cavities and cables. We assign the cable end numbers to the cavities as there is exactly one cable plugged into each cavity and we speak of cavities rather than cable ends in the model. For $k = 2b + n$ cable ends/cavities with b job pairs $\langle c_i, c_j \rangle$ numbered with $i = 1 \ldots b$ and $j = i + b$ we introduce the corresponding ranges in the model. The number of two-sided cables is captured by the range *CavityPairs*.

```
int k = ...; //number of cavities, permutation length
int b = ...;  //number of two-sided cables, job pairs
range Cavities = 1..k;
range Cablestarts = 1..b;
range Positions = 1..k;
range CavityPairs = 1..2*b ;
```

The three sets of atomic, soft atomic and disjunctive constraints are explicitly introduced into the model. Cables that are too short for storage and that are subject to direct successor constraints are represented in a list of integers of cable end numbers.

```
{Atomic} AtomicConstraints = ...;
{Atomic} SoftAtomicConstraints = ...;
{Disjun} DisjunctiveConstraints = ...;
{int} DirectSuccessors = ...;
```

Atomic and disjunctive constraints are represented as tuples:

```
tuple Atomic {int cbefore; int cafter;}
tuple Disjun {int c1before; int c1after; int c2before;
  int c2after}
```

An array decision variable *position_for_cavity (pfc)* of length $1, \ldots, k = 2b + n$ is introduced to represented the permutation sequence. This array uses the cavity number as index and stores the position as value.

```
dvar int pfc[Cavities] in Positions;
```

An *allDifferent* constraint is added for the *pfc* permutation sequence to implement the constraint implied by Definition 3.

```
allDifferent(pfc);
```

All hard and soft constraints as well as optimization criteria are formulated on the *pfc* permutation. Modeling the precedence and direct successor constraints is straightforward:

```
forall(c in AtomicConstraints)
pfc[c.cbefore] < pfc[c.cafter];

forall(c in DisjunctiveConstraints) {
pfc[c.c1before] < pfc[c.c1after] ||
pfc[c.c2before] < pfc[c.c2after];
if(c.c1before == c.c2before)
{maxl(pfc[c.c1after], pfc[c.c2after]) > pfc[c.c1before];}}

forall(i in DirectSuccessors: i<=b)
(pfc[i] < pfc[i+b]) => (pfc[i+b] - pfc[i] == 1);

forall(i in DirectSuccessors: i>b)
(pfc[i] < pfc[i-b]) => (pfc[i-b] - pfc[i] == 1);
```

The definitions of the optimization criteria S, M, L and N read as follows:

```
dexpr int S = (b == 0) ? 0 :
(sum(i in CableStarts) (abs(pfc[i] - pfc[i+b]) > 1));

dexpr int M = (b == 0) ? 0 :
(max(i in CavityPairs) (sum(j in CavityPairs: j<=b)
((pfc[j] < pfc[i] && pfc[i] < pfc[j+b]) ? 1 : 0)
```

```
+ sum(j in CavityPairs: j>b)
((pfc[j] < pfc[i] && pfc[i] < pfc[j-b]) ? 1 : 0)));

dexpr float L = (b == 0) ? 0 :
max(i in CableStarts) abs(pfc[i] - pfc[i+b]) - 1;

dexpr int N = sum(i in SoftAtomicConstraints)
(pfc[i.cbefore] > pfc[i.cafter]);
```

The objective function is stated as:

```
minimize S * pow(k, 3) + M * pow(k, 2) + L * pow(k, 1) + N;
```

Alternatively, Cplex and some other solvers offer the built-in *staticLex* function that defines a multi-criteria policy ordering the different criteria. However, in our experiments we found that Cplex performs slightly better when not using this function.

The constraint model **DM** follows the approach from [30] and uses two permutation sequences *cavity_for_position (cfp)* and *position_for_cavity (pfc)*. Whereas the *pfc* permutation uses the cavity number as index and stores the position as value, the *cfp* permutation assigns cavity identifiers to positions. An additional array decision variable *cfp* is thus added to the model.

```
dvar int cfp[Positions] in Cavities;
```

For example, the two permutations *cfp= 3,1,2* and *pfc=2,3,1* describe the same solution where cavity 3 is wired first, cavity 1 second, and cavity 2 last. The dual permutation representations are linked via a channeling constraint

$$\forall j \in \mathit{Cavities},\ p \in \mathit{Positions} :\ \mathit{pfc}[j] = p \Leftrightarrow \mathit{cfp}[p] = j$$

using the built-in *inverse* constraint in OPL. Furthermore, a (now) redundant *allDifferent* constraint is added for both permutation sequences. Experiments on the influence of the two redundant *allDifferent* constraints and the channeling constraint gave no clear picture. Different combinations of these three constraints increased or reduced solution time and costs on different instances for Cplex CP. In the end, we decided to keep the three global constraints in the dual model **DM**, which makes **DM** a strict extension of the model **M**.

```
allDifferent(pfc); allDifferent(cfp); inverse(cfp, pfc);
```

Using the dual model **DM** was key in scaling earlier versions of the Cplex CP solver to larger CTW instances, which it could not solve using only one of the permutation representations. All hard and soft constraints as well as optimization criteria remain formulated on the *pfc* permutation following the experimental results in [30].[4]

Benchmark instances are represented in the .DAT format used by Cplex. The .DAT files used in the experiments with Cplex CP and MIP are directly exported from cable tree data in the Zeta machines using an XML-based software interface and an exporter written in C#. Each file provides specific values for the integer parameters k (number of jobs, permutation length) and b (number of job pairs). Constraints are represented as sets of tuples of integer

[4]Note that in early experiments we had also worked with another non-dual model using only the *cfp* permutation and all constraints formulated over *cfp*. However, on this modeling variant Cplex CP performed much worse confirming the results in [30]. Thus, this non-dual modeling approach was quickly abandoned and not pursued further.

values enumerating the cavities. For example, instance R024 with 6 two-sided and 14 one-sided cables reads as follows (most constraints replaced by . . .)

```
k = 26;

b = 6;

AtomicConstraints = {<1,3>, <2,3>, <3,18>, <6,18>, <15,25>,
                     <17,21>, ...};

SoftAtomicConstraints = {<2,1>, <4,3>, <6,5>, <12,26>, ...};

DisjunctiveConstraints = {<8,15,8,16>, <16,12,6,16>,
                          <9,17,9,18>,...};

DirectSuccessors = {1,2,8,7,};
```

Note that direct successor constraints are represented in a list of integers of cable end numbers, because they are specific to two-sided cables and express that once a cable end is plugged into a cavity, the other end must be plugged immediately after or must have been plugged before. The label of the other cable end is obvious from our numbering scheme using the b parameter. Note that the integer i occurring in the DirectSuccessors list means that the direct successor constraint $c_i \blacktriangleleft c_j$, where j is the other end of the two-sided cable (so $j = i + b$ or $j = i - b$), exists in the problem instance. In the example above, we can see that both ends of the cables in job pairs $\langle c_1, c_7 \rangle$ and $\langle c_2, c_8 \rangle$ are too short for storage.

5.2 Overview on model variants, instance data formats, and the supporting tool chain

Based on the two models **M** and **DM**, we derived different implementations and model/data representations in order to perform a benchmarking using different solvers. This turned out to be a very time-consuming and mostly manual process, which also included the necessity to write software to achieve the desired conversions, because solvers use different modeling languages and data input formats. Figure 3 summarizes the derivation process for those solvers, which separate the model from instance data. It shows which model and data format is fed into which solver.

The model $\mathbf{M_Z}$ is an implementation of the non-dual model **M** in the MiniZinc language [44]. The model $\mathbf{DM'_Z}$ is an implementation of the dual model **DM** in the MiniZinc language [44] where we rewrote the definitions of the L and S criteria to eliminate the *absolute* functions. In contrast to the Cplex OPL language, the MiniZinc language does not provide the possibility to represent tuples. Therefore, OPL ranges are translated into sets of integers in MiniZinc and constraints are represented using arrays. Arguments of the arrays represent the cavities between which a constraint must hold. For disjunctive constraints, a 2-dimensional array is used. MiniZinc supports various global constraints, which are included into the model to express the *allDifferent* constraint over the elements of an array in a straightforward way.

The model $\mathbf{M_{GT}}$ is another non-dual model created by Guido Tack, Monash University. In this model, disjunctive constraints are rewritten using an array of booleans and an additional constraint over this array is added. The array captures the truth value of the two

disjuncts in a disjunctive constraint and the constraint states that at least one of the disjuncts must be true for the disjunctive constraint to be satisfied. Furthermore, an array of booleans is used to capture values of the optimization criteria, which are represented as sums over these boolean array values.

The models $\mathbf{M_Z}$, $\mathbf{DM'_Z}$, and $\mathbf{M_{GT}}$ are used in experiments with the Chuffed constraint solver and the Gurobi and Cplex MIP solvers. Whereas Chuffed can read a MiniZinc model directly, we used the MiniZinc command line tool and the provided wrappers for Gurobi and Cplex to feed the models into the MIP solvers. The MiniZinc models work with benchmark instances represented in the .DZN format. These data files are generated using a Python script, which replaces the delimiters used in the representation of the constraint sets in the .DAT files. This is the only required conversion as the .DAT and .DZN file formats are quite similar.

We also manually implemented variants of the **M** and **DM** models leading to models $\mathbf{M_{MIP}}$ and $\mathbf{DM_{MIP}}$ in the language OPL for Cplex MIP and models $\mathbf{M^{C\#}_{MIP}}$ and $\mathbf{DM^{C\#}_{MIP}}$ in the C# API of Gurobi. In these MIP model implementations, we implemented constraints as equations without any sophisticated rewriting such as for example described in [53]. Our goal was to keep the models close to the constraint models and to avoid unintuitive reformulations. The models represent constraints as tuples of integers in the same way as the **M** model. The *allDifferent* constraints are reformulated using pairwise inequalities. All other constraints rewrite the $<$ condition over permutation values as inequalities. In contrast to Cplex OPL, inequalities cannot be directly expressed in the Gurobi API, but must be rewritten as linear inequality expressions, which required us to introduce additional binary variables for each inequality constraint. Similarly, disjunctive constraints as well as the *allDifferent* constraint are rewritten using binary variables. In addition, we had to introduce additional variables for the optimization criteria, which capture whether a cable end comes before or after the other end of this cable in the permutation sequence. Cplex MIP uses the

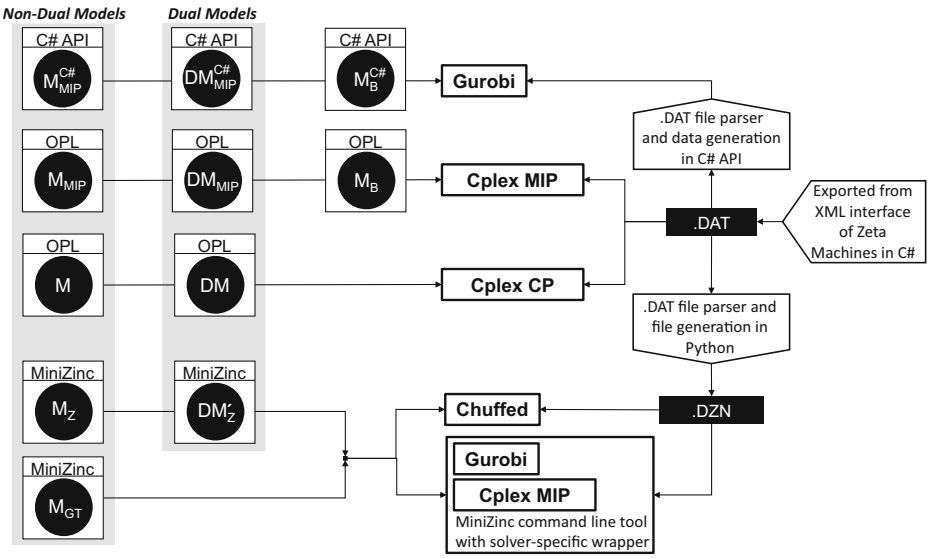

Fig. 3 Models and data formats for the Gurobi, Cplex CP, Cplex MIP, and Chuffed solvers that keep model and data in separate representations

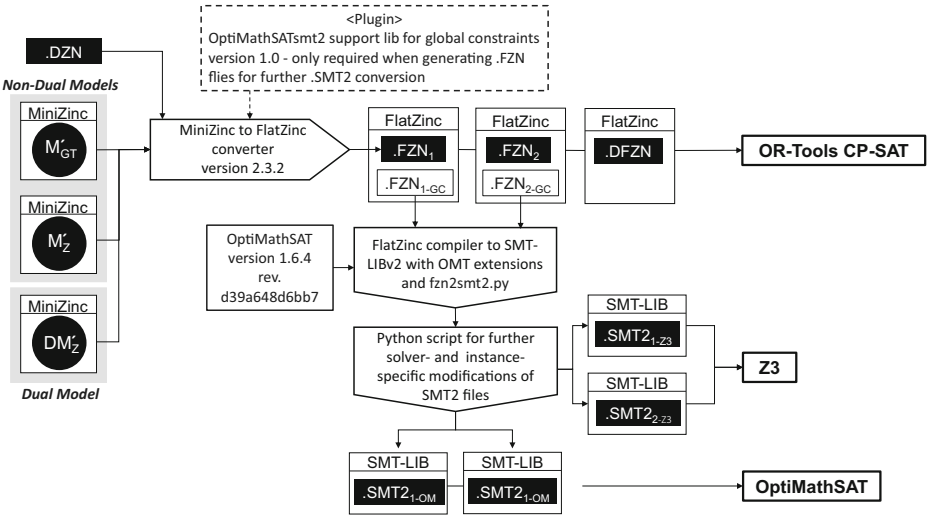

Fig. 4 Tool chain for the Google OR-Tools, Z3, and OptiMathSAT solvers that use a single integrated model-data file for each instance

same .DAT files as Cplex CP. To feed the instance data into Gurobi, a .DAT file parser was implemented in C# such that the data can be directly generated in the C# API of Gurobi.

The models $\mathbf{M_B}$ for Cplex MIP and $\mathbf{M_B^{C\#}}$ for Gurobi make use of a big-M reformulation for the disjunctive constraints [55] using additional decision variables and non-integer (floating point) optimization costs. An upper triangular matrix of booleans is used to represent that one cavity is plugged before another one. Constraints can be directly expressed in this matrix representation and the solution can be extracted from the matrix. Additional decision variables are used to capture the minimum and maximum positions of the cavities in the permutation. Their values are set by additional constraints.

Figure 4 summarizes the tool chain for the experiments with the Google OR-Tools CP-SAT solver and the two OMT solvers Z3 and OptiMathSAT, which use a single integrated model-data file for each instance. The non-dual MiniZinc models $\mathbf{M_Z}$ and $\mathbf{M_{GT}}$ were rewritten into MiniZinc models $\mathbf{M'_Z}$, $\mathbf{M'_{GT}}$, because the MiniZinc to Flatzinc converter does not support the *absolute* function. The model $\mathbf{DM'_Z}$ remained unchanged as the *absolute* function had already been removed from this model. Using one of these models and the data files in Chuffed's .DZN format, we generated two FlatZinc files $\mathbf{.FZN_1}$ and $\mathbf{.FZN_2}$ from the non-dual models $\mathbf{M'_Z}$ and $\mathbf{M'_{GT}}$, and one **.DFZN** file from the dual model $\mathbf{DM'_Z}$ in .FZN format, which provide the input into the Google OR-Tools CP-SAT solver.

In order to generate the .SMT2 files for the OMT solvers, this tool chain needs to be invoked again, but this time additionally using the OptiMathSATsmt2 support library marked with dashed lines in Fig. 4. This library generates another variant of the models and instance data named $.\mathrm{FZN}_{1-GC}$ and $.\mathrm{FZN}_{1-GC}$ supporting global constraints. We did not succeed in developing a tool chain for the dual model $\mathbf{DM'_Z}$ and therefore had to limit our experiments with the OMT solvers to the two non-dual model variants.

The $.\mathrm{FZN}_{1-GC}$ and $.\mathrm{FZN}_{1-GC}$ files can then be further processed with a FlatZinc to SMT2 compiler using OptiMathSAT, specific syntax support for Z3 and OptiMathSAT, and OMT extensions [17, 18]. Some information from our models is, however, not translated correctly to these files requiring postprocessing of the files with our own Python script. First,

the lower and upper bounds for the decision variable *pfc* are missing. Our scripts therefore need to add functions to the .SMT2 file setting the bounds larger than zero and lower or equal to the length of the permutation k. For example, for a permutation of length 20, these functions read as follows:

```
(define-fun lbound20 () Bool (> @pfc@20 0))
(define-fun ubound20 () Bool (<= @pfc@20 20))
```

The two functions have to be true in our model:

```
(assert lbound20)
(assert ubound20)
```

Second, the extraction of the *pfc* sequence from the solution model lacks a clear naming scheme. The sequence to be extracted corresponds to the first k variables starting with the name X_INTRODUCED. A workaround adds the relevant variables as comments to the output file. These comments are then used by our solution extractor to correctly extract the permutation values. For example if $k = 4$, the comments in the beginning of the file read:

```
;; k=4
;; Extract pfc from
;; X_INTRODUCED_0_
;; X_INTRODUCED_1_
;; X_INTRODUCED_2_
;; X_INTRODUCED_3_
```

For each model, we obtained two sets $\textbf{.SMT2}_{\textbf{Z3}}$ and $\textbf{.SMT2}_{\textbf{OM}}$ of files in .SMT2 format, which are specific to Z3 and OptiMathSAT. A project is available on github to access this elaborate tool chain for the generation of the .SMT2 formats for the two OMT solvers, see https://github.com/kw90/ctw_toolchain. The repository contains a Docker environment specified with scripts for installing all dependencies from the OptiMathSAT, Z3, libminizinc, and fzn2omt sources. A Jupyter notebooks automatically iterates over all files in the specified directory, and applies the translation and the necessary adjustments. For the experiments with OptiMathSAT, we forked a project on github https://github.com/kw90/omt_python_timeout_wrapper, which implements a Python wrapper to call the OptiMathSAT C library, such that we were able to run the solver with a given time limit and extract the best solution found. In contrast to the other solvers, which run directly under Windows 10, OptiMathSAT runs on a Linux Ubuntu 20.04 machine within the Windows Subsystem for Linux WSL.

6 Benchmarking models and solvers on the CTW problem

In the following, we summarize our findings from experiments with the various solvers and models. All experiments were run on a Windows 10 virtual machine with four 2.30 GHz processors and 8 GB memory. Solver/model combinations are called from

- our own C# code environment, which performs all necessary data and model conversions and result validation, or
- the MiniZinc command line tool, or
- the specific tool chain for Z3 and OptiMathSat.

We tested all solvers in their default configuration settings, which often means that a solver automatically performs strategy selection. For Chuffed and Google OR-Tools we used their freesearch option as default configuration. Furthermore, we tested some tuning options for the constraint solvers. For example, we investigated different settings of propagation levels for Cplex CP and ran some experiments using search annotations in MiniZinc for Chuffed and OR-Tools. MIP solvers were only tested in their default configuration. For the OMT solvers, specific strategies have to be selected. For Z3, we used its default solver for weighted MaxSAT problems called MaxRes [8, 43], but also tested the PD-MaxRes and WMax strategies of Z3. For OptiMathSat, we used its OMT-based encoding and engine setting as default strategy.

All solvers were tested on the entire benchmark set in a single run, except Gurobi, which we tested in chunks of 60 instances due to out-of-memory problems that we could not resolve otherwise. The numbers we report in the following are all from a single run of a solver for consistency reasons. We performed up to three runs for each solver and noticed minimal differences in the number of solved instances, solution costs or runtime, however felt that selecting numbers from a single (in our case the first) run gives a better impression than computing the average from 3 runs. For example, some solvers could solve up to two additional instances in different runs, but given the total subset of 256 instances, this difference is very small and does not change the overall picture.

The extraction of solution data is specific to each solver or modeling language. We used their generated logs and/or API to access the solver state and solution. Each solution was validated for correctness using our own software, which checks that a generated permutation sequence does satisfy all constraints. The software also recalculates the values of the optimization criteria and the overall objective. When showing solution costs, we used these recalculated values to make the solution costs comparable across different solvers as single cases of deviations occurred for some instances. We discuss these in more detail when we summarize our findings in Section 7.

For the experiments, we mapped the individual solver states to five possible outcomes of a solver on a benchmark instance:

- *unsatisfiable*: the solver has proven the instance to have no solution,
- *optimal*: the solver has found a solution and proven that no better solution with lower costs exists,
- *suboptimal*: the solver has found a solution, but was not able to prove it as optimal,
- *unsolved*: the solver was not able to find a solution or prove an instance as being unsatisfiable within a given time limit,
- *undefined*: any other state returned by a solver not mapped to one of the states above.

Details of the mapping of solver states for each of the tested solvers can be found in Appendix B. Appendix C summarizes our tests with selected solvers and models to find a time limit, which allows solvers to find good or even optimal solutions and still keeps the effort for a single run of a solver/model combination at a reasonable level. We decided for a 5-minute time limit, which satisfies these requirements and keeps a single run under 3 days.

6.1 Constraint solver performance on the CTW benchmark set

Table 3 summarizes the results for the constraint solvers on the non-dual model $\mathbf{M}$ for Cplex CP and its variants $\mathbf{M_Z}$ for Chuffed and $\mathbf{.FZN_1}$ for OR-Tools. Furthermore, the table shows how Chuffed performs on the model $\mathbf{M_{GT}}$ and OR-Tools on the corresponding variant $\mathbf{.FZN_2}$.

Table 3 Performance of constraint solvers in default configuration on entire benchmark set using non-dual models

Solver State	Cplex CP **M**	Chuffed $\mathbf{M_Z}$	OR-Tools $\mathbf{.FZN_1}$	Chuffed $\mathbf{M_{GT}}$	OR-Tools $\mathbf{.FZN_2}$
Optimal	135(+60)	144	224	135	228
Suboptimal	121	85	6	88	5
Unsolved	0	26	24	32	21
Unsatisfiable	22	22	22	22	22
Undefined	0	1	2	1	2
TOTAL Solved	256	229	230	223	233

Only Cplex CP finds solutions for all satisfiable instances, the other constraint solvers encounter between 21 and 32 unsolvable instances. Besides the 135 instances, for which Cplex can prove optimality of its solution, it finds 60 solutions of minimal cost, but cannot prove them as optimal.[5] All solvers identify the 22 unsatisfiable instances instantly. For the 6 instances, which OR-Tools using the $\mathbf{.FZN_1}$ model cannot solve optimally, no other solver finds an optimal solution. However, when using the $\mathbf{.FZN_2}$ model, OR-Tools solves 3 of these 6 instances optimally. For all of the 5 instances, which OR-Tools solves suboptimally using the $\mathbf{.FZN_2}$ model, Cplex CP finds suboptimal solutions of lower costs using the **M** model. On some instances, Chuffed and OR-Tools end up in an undefined state, which we discuss in more detail when we summarize our findings in Section 7.

Table 4 summarizes solution costs and runtimes for the constraint solvers using non-dual models. For a subset of 118 instances, all solvers can find optimal solutions. The largest instance R197 in this subset has permutation length 49 and constraint sum 432. There is only one instance R189, for which all solvers only find a suboptimal solution. This instance has permutation length $k = 100$ and its constraint sum parameter has value 1544 containing 1305 atomic constraints. Cplex CP finds the lowest cost solution for this instance. For OR-Tools, solution costs differ significantly between the $\mathbf{.FZN_1}$ and $\mathbf{.FZN_2}$ models.

There is a minimal deviation in the optimal solution costs for Cplex CP, which we discuss further in Section 7. It is worth looking at the time solvers need to find the 118 optimal solutions using their respective models. OR-Tools using the $\mathbf{.FZN_1}$ and $\mathbf{.FZN_2}$ models is the fastest solver and needs less than 5 minutes, followed by Cplex CP with a little bit more than half an hour. Chuffed needs nearly 45 minutes on both models to find optimal solutions and is thus almost 10 times slower than OR-Tools.

Table 5 summarizes the results for the constraint solvers using their respective dual model variants. Cplex CP and OR-Tools can find slightly more optimal solutions. For Chuffed, using the non-dual $\mathbf{M_Z}$ or the dual $\mathbf{DM_Z}$ models makes no difference at all – it solves exactly the same instances in the subsets of optimal or suboptimal solutions. Again, only Cplex CP finds solutions for all 256 solvable instances, and OR-Tools finds the most optimal solutions. As in the case when using the non-dual model **M**, Cplex CP finds cost-minimal solutions, but cannot prove their optimality. Compared to using the dual model, this number

[5] Google OR-Tools is the solver, which is able to prove optimality for most of its minimal cost solutions in our experiments. We therefore compare the costs of solutions, which other solvers marked as suboptimal, to the optimal solutions found by OR-Tools and then add these instances in braces to the optimal solution counts for the other solvers.

Table 4 Costs and runtimes for optimal and suboptimal solutions returned by constraint solvers in default configuration using non-dual models

Non-dual Models cost & runtimes	Cplex CP **M**	Chuffed $\mathbf{M_Z}$	OR-Tools $\mathbf{.FZN_1}$	Chuffed $\mathbf{M_{GT}}$	OR-Tools $\mathbf{.FZN_2}$
Solved Suboptimally (1)					
Total costs	20,158,417	44,327,625	42,227,828	44,287,834	23,178,029
Relative costs (in %)	100	220	209	220	115
Solved Optimally (118)					
Total costs	2,862,879	2,862,876	2,862,876	2,862,876	2,862,876
Total runtimes (in s)	2001	2517	267	2554	270
Relative runtimes (in %)	751	944	100	958	101

Percentages are rounded mathematically to the next integer value with setting the best-performing solver to 100%

drops from 60 down to 54, with 4 more solutions now proven as optimal. Two of these instances remain in the suboptimal subset, but their solution costs increase.

Table 6 summarizes solution costs and runtimes when using dual models. For a subset of 7 instances, all solvers only find suboptimal solutions. The largest instance in this set is again R189 with permutation length $k = 100$ and constraint sum 1544. Cplex returns the lowest sum of costs for these 7 instances. For 5 of these instances, it finds the lowest cost solutions. For 2 other instances, OR-Tools finds the lowest cost solutions.

For a subset of 127 instances, all solvers find optimal solutions. The two largest instances in this subset are A016 with $k = 80$ and constraint sum 751 and A017 with $k = 80$ and constraint sum 703. OR-Tools is also the fastest solver finding optimal solutions, being 2.5 times faster than Cplex CP and more than 5 times faster than Chuffed on this subset of instances.

In Table 7, we investigate how sensitive the constraint solvers are to changes from a non-dual model to a dual model and vice versa. There is some sensitivity in the number of optimally solved instances, but it is only small for Cplex CP and OR-Tools and not existent for Chuffed. For example, Cplex CP solves 135 instances optimally using the non-dual model **M**. In the dual model **DM**, 6 of these instances can only be solved suboptimally, but it solves an additional 10 instances optimally. Google OR-Tools solves 225 instances in the

Table 5 Performance of constraint solvers in default configuration on entire benchmark set using dual models

Solver State	Cplex CP **DM**	Chuffed $\mathbf{DM'_Z}$	OR-Tools **.DFZN**
Optimal	139 (+54)	144	225
Suboptimal	117	85	19
Unsolved	0	26	10
Unsatisfiable	22	22	22
Undefined	0	1	2
TOTAL Solved	256	225	244

Table 6 Costs and runtimes for optimal and suboptimal solutions returned by constraint solvers in default configuration using dual models

Dual Models	Cplex CP	Chuffed	OR-Tools
Cost & runtimes	**DM**	$\mathbf{DM'_Z}$	**.DFZN**
Solved Suboptimally (7)			
Total costs	60,285,838	172,558,537	82,715,780
Relative costs (in %)	100	286	137
Solved Optimally (127)			
Total costs	4,286,932	4,286,915	4,286,915
Total runtimes (in s)	1710	4006	739
Relative runtimes (in %)	231	542	100

Percentages are rounded mathematically to the next integer value with setting the best-performing solver to 100%

dual model **.DFZN**, but in the non-dual model $\mathbf{.FZN_1}$, 3 of these instances remain unsolved. The runtimes vary, however, quite significantly. Cplex CP is much faster when using a dual model, whereas Google and Chuffed slow down.

We ran a few experiments to investigate if the performance of constraint solvers using the dual model can be further improved by tuning search strategies, because using a dual model helped Cplex CP significantly and also allowed OR-Tools to find more solutions. For Cplex CP, we investigated the influence of extended *inference level setting*s, which allow the constraint solver to control the strength of domain reduction that it can achieve on the constraint variables by performing more or less constraint propagation. For the Chuffed solver, we experimented with different search annotations in MiniZinc and selected the best working one. Google OR-Tools only supports a small subset of search annotations. Therefore, we invoked it without search annotations, but using 8 workers, which are not active in the default configuration. For both solvers, we also disabled *freesearch*, i.e., the additional option to deviate from the annotated search strategy or the strategy that is used in their default configuration. Appendix D gives more details on the tested tuning configurations.

Table 8 summarizes the results for Cplex CP with extended inference level settings, Chuffed using search annotations and without *freesearch*, and Google OR-Tools using 8 workers and also no *freesearch*. Cplex CP now finds optimal solutions for 146 instances. This set contains 138 instances, which Cplex CP could solve optimally using the default

Table 7 Variance in the number of optimally solved instances for the Cplex CP and OR-Tools CP-SAT constraint solvers when using a dual or non-dual model

Dual vs. Non-Dual Models	Cplex CP		OR-Tools		Chuffed	
	M	**DM**	$\mathbf{FZN_1}$	**.DFZN**	$\mathbf{M_Z}$	$\mathbf{DM'_Z}$
Solved optimally in a model	135	139	224	225	144	144
Solved only suboptimally in other model	6	10	2	0	0	0
Unsolved in other model	0	0	0	3	0	0
Solved optimally in both models	129		222		144	
Runtimes per model on common subset (in s)	2,300	1,200	7,530	9,074	4,556	5,516

Table 8 Performance of constraint solvers with tuned search strategies using a dual model

Tuning	Cplex CP Dual Model **DM** + Extended Inference Level Settings	Chuffed Dual Model $\mathbf{DM}'_{\mathbf{Z}}$ + Search Annotation + No freesearch	OR-Tools Dual Model **.DFZN** + 8 Workers + No freesearch
Optimal	146 (+56)	116	236
Suboptimal	110	125	17
Unsolved	0	14	1
Unsatisfiable	22	22	22
Undefined	0	1	2
TOTAL Solved	256	241	253
Total costs (240 instances)	1,703,349,373	2,940,107,201	1,575,489,390
Relative costs (in %)	108	187	100
Total runtimes (in s)	31,226	40,998	7,948
Relative runtimes (in %)	393	516	100

Relative cost and runtime comparisons are calculated taking the values of OR-Tools as 100%

inference level settings, but not the instance R009, for which only a suboptimal solution is found. In addition, Cplex CP finds another 56 solutions of minimal costs, but cannot prove these solutions as optimal. Without extended inference level settings, Cplex CP found 54 cost-minimal solutions. It can now prove some of these solutions to be optimal. The number of cost-minimal solutions thus grows from 193 to 202. Total solution costs for the 256 solved instances is reduced by 3%. OR-Tools solves 236 instances optimally and finds suboptimal solutions for 17 instances. There is only a single instance, which it cannot solve when using 8 workers. This instance A072 has permutation length $k = 198$ and constraint sum 11766 and is the largest instance in the benchmark set. There is only one other instance A073 with the same permutation length $k = 198$, but with lower constraint sum 11346, for which OR-Tools finds a suboptimal solution.

All three solvers can solve the largest number of instances using the dual model and some tuning. Only for Chuffed, the number of optimally solved instances decreases from

Table 9 Performance of MIP solvers with different implementations of non-dual models on entire benchmark set

Solver State	Cplex MIP			Gurobi		
	$\mathbf{M_{MIP}}$	$\mathbf{M_Z}$	$\mathbf{M_{GT}}$	$\mathbf{M}^{\mathbf{C\#}}_{\mathbf{MIP}}$	$\mathbf{M_Z}$	$\mathbf{M_{GT}}$
Optimal	126(+10)	99(+9)	109(+12)	134(+8)	99(+8)	102(+8)
Suboptimal	77	92	79	17	45	42
Unsolved	46	65	68	104	112	112
Unsatisfiable	21	22	22	23	22	22
Undefined	8	0	0	0	0	0
TOTAL Solved	203	191	188	151	144	144

Table 10 Costs and runtimes for optimal and suboptimal solutions returned by MIP solvers with different implementations of non-dual models

Non-Dual Models	Cplex MIP			Gurobi		
Cost & runtimes	$\mathbf{M_{MIP}}$	$\mathbf{M_Z}$	$\mathbf{M_{GT}}$	$\mathbf{M_{MIP}^{C\#}}$	$\mathbf{M_Z}$	$\mathbf{M_{GT}}$
Solved Suboptimally (5)						
Total costs	2,328,362	2,801,932	1,922,460	2,142,257	4,318,355	2,890,439
Relative costs (in %)	121	146	100	111	225	150
Solved Optimally (87)						
Total costs	646,261	502,765	502,765	502,765	502,765	502,765
Total runtimes (in s)	279	1350	889	303	1681	1283
Relative runtimes (in %)	100	484	319	109	602	460

Percentages are rounded mathematically to the next integer value with setting the best-performing solver to 100%

144 using the $\mathbf{M_Z}$ and $\mathbf{DM'_Z}$ models to 116 using $\mathbf{DM'_Z}$ with tuning. However, its number of suboptimal solutions grows from 85 and 88 respectively, to 125. In total, it can now solve 241 instances compared to only 223, 225, or 229 instances in previous tests. Table 8 also compares the cost and runtimes on the subset of 240 instances, which all tuned solvers can now solve, i.e., they can find a suboptimal or optimal solution. Note that this subset is NOT a superset of the earlier subsets from Tables 4 and 6. OR-Tools shows the fastest performance, which is explained by the high number of optimal solutions that it finds for 236 instances and where it rarely exhausts the 5 minute time limit. Among all three solvers, the cost sum of the 240 instances for OR-Tools is also the lowest.

6.2 MIP solver performance on the CTW benchmark set

In the following, we summarize the results for tests with the Cplex MIP and Gurobi solvers. Overall, a significantly higher number of instances remains unsolved by the Cplex and Gurobi MIP solvers when compared to the constraint solvers.

Table 9 shows the results for Cplex MIP and Gurobi on the three variants of non-dual mixed-integer models. In all solver/model combinations, solutions of minimal costs are

Table 11 Performance of MIP solvers with different implementations of the dual and Big-M models on entire benchmark set

	Cplex MIP			Gurobi		
Solver State	$\mathbf{DM_{MIP}}$	$\mathbf{DM'_Z}$	$\mathbf{M_B}$	$\mathbf{DM_{MIP}^{C\#}}$	$\mathbf{DM'_Z}$	$\mathbf{M_B^{C\#}}$
Optimal	82	107(+10)	126(+14)	103(+3)	96(+13)	134(+5)
Suboptimal	56	86	24	25	49	19
Unsolved	118	63	106	127	111	102
Unsatisfiable	21	22	19	23	22	23
Undefined	1	0	3	0	0	0
TOTAL Solved	138	193	188	128	155	153

Table 12 Costs and runtimes for optimal and suboptimal solutions returned by MIP solvers with different implementations of the dual and big-M models

	Cplex MIP			Gurobi		
Dual/big-M Models	$\mathbf{DM_{MIP}}$	$\mathbf{DM'_Z}$	$\mathbf{M_B}$	$\mathbf{DM^{C\#}_{MIP}}$	$\mathbf{DM'_Z}$	$\mathbf{M^{C\#}_B}$
Solved Suboptimally (3)						
Total costs	3,682,181	2,136,996	1,015,795	1,865,705	2,069,799	2,486,636
Relative costs (in %)	362	210	100	184	204	245
Solved Optimally (81)						
Total costs	502,758	502,757	502,757	502,757	502,757	502,757
Total runtimes (in s)	2,543	341	230	1,087	1,045	135
Relative runtimes (in %)	1,880	252	170	804	773	100

Percentages are rounded mathematically to the next integer value with setting the best-performing solver to 100%

found, which cannot be proven as optimal. These numbers are shown in braces. Cplex MIP and Gurobi show a similar performance in the number of optimally solved instances (126 vs. 134) using the $\mathbf{M_{MIP}}$ and $\mathbf{M^{C\#}_{MIP}}$ model, resp. They differ significantly in the number of instances, for which they can find suboptimal solutions, where Cplex MIP solves at least two times more instances than Gurobi. These native implementations of the model $\mathbf{M}$ in Cplex OPL or the Gurobi API allow both solvers to solve more instances, whereas feeding the models $\mathbf{M_Z}$ and $\mathbf{M_{GT}}$ into the solvers with the help of the MiniZinc command line tool reduces the number of solved instances. For Cplex MIP, the model $\mathbf{M_{MIP}}$ leads to undefined solver states on some instances. Furthermore, Cplex MIP can only identify 21 out of the 22 unsatisfiable instances, whereas Gurobi identifies one solvable instance as unsatisfiable. Furthermore, Cplex MIP ends up in an *undefined* state on 8 instances. We discuss these issues in detail in Section 7.

Table 10 shows costs and runtimes when using non-dual models. For a subset of 5 instances, the two solvers only find suboptimal solutions. The largest instance in this set is R163 with permutation length $k = 56$ and constraint sum 642. For a subset of 87 instances, optimal solutions are found. The two largest instances in this set are R003 with $k = 36$ and constraint sum 207, and R205 with $k = 36$ and constraint sum 207. These instances are much smaller than the largest instances solved optimally by the constraint solvers. Cplex

Table 13 Performance of OMT solvers on entire benchmark set

	Z3 MaxRes		OptiMathSAT OMT	
Solver State	$\mathbf{SMT2_{1\text{-}Z3}}$	$\mathbf{SMT2_{2\text{-}Z3}}$	$\mathbf{SMT2_{1\text{-}OM}}$	$\mathbf{SMT2_{2\text{-}OM}}$
Optimal	94	96	26	26
Suboptimal	0	0	0	0
Unsolved	160	158	230	230
Unsatisfiable	22	22	22	22
Undefined	2	2	2	2
TOTAL Solved	94	96	26	26

MIP performs slightly better on the optimally solved subset, but the table also shows that feeding the MiniZinc models using the command line tool wrappers leads to a decline in performance, which is more dramatic for Gurobi than for Cplex. We discuss the deviation in the costs of optimal solutions for Cpex MIP in more detail in Section 7.

Table 11 summarizes the results for the MIP solvers using the dual model variants or the big-M model. Depending on the model, Cplex MIP identifies all or only 21 or 19 of the unsatisfiable 22 instances. Gurobi identifies all 22 unsatisfiable instances, but also returns *unsatisfiable* for one of the solvable instances. We discuss this issue in Section 7. The number of solved instances varies a lot for the different models and no clear favorite is visible. When using the directly implemented models $\mathbf{DM_{MIP}}$ and $\mathbf{DM^{C\#}_{MIP}}$, both solvers find significantly fewer solutions compared to when using the corresponding non-dual models. When using the MiniZinc command line tool, however, both solvers can solve a few more instances using a dual model as when using a non-dual model. Cplex MIP finds its largest number of optimal solutions using the big-M model $\mathbf{M_B}$. For Gurobi, the big-M model also plays off, but when using the dual model $\mathbf{DM'_Z}$ via the MiniZinc command line tool, the solver can solve slightly more instances.

Table 12 summarizes costs and runtimes on the subsets of models that both solvers can solve using either a dual model or the big-M model. It shows how much the subsets of suboptimally solved instances varies as there is only a small set of 3 instances solved by both solvers using any of the models. The largest instance in this subset is R049 with $k = 45$ and constraint sum 287. The costs of suboptimal solutions also vary significantly. For Cplex MIP, the cost sum is the lowest when using the model $\mathbf{M_B}$, whereas for Gurobi, it is the lowest when using the direct implementation of the dual model in the solver's API $\mathbf{DM^{C\#}_{MIP}}$. On the subset of 81 instances, which both solvers solve optimally using either a dual model or the big-M model, both solvers perform fastest when using the big-M model. The largest instance in the subset of optimally solved instances is R003 with $k = 36$ and constraint sum 149. Compared to the size of instances for which the constraint solvers can find optimal or suboptimal solutions, we can observe a significant decline.

6.3 OMT solver performance on the CTW benchmark set

Table 13 summarizes the results for the OMT solvers Z3 and OptiMathSAT using their variants of the two different SMT2 models derived from the models $\mathbf{M'_Z}$ and $\mathbf{M'_{GT}}$. Recall that no SMT2 model could be generated using the model $\mathbf{DM'_Z}$ and thus the OMT solvers are only tested on non-dual models. We show the results for OptiMathSAT using the OMT strategy and for Z3 using the MaxRes strategy.

Optimally solved instances are determined for Z3 by comparing its solution costs to the costs of known optimal solutions found by other solvers, see Appendix B for details. Z3 can solve 94 or 96 instances optimally. Interestingly, Z3 always either finds cost-optimal solutions or leaves an instance unsolved. From the solver state itself, all solutions found are marked as suboptimal by Z3. OptiMathSAT finds optimal solutions for 26 instances and cannot solve any other instances. Both solvers identify the 22 unsatisfiable instances easily, but end up in an *undefined* state for two more instances, which we discuss further in Section 7.

The permutation length of the largest instances R003 and R004, which Z3 can solve, is $k = 36$ with constraint sums 207 and 198, respectively. The average permutation length over all solved instances is only 17. On average, these instances contain 38 atomic constraints and 9 disjunctive constraints only. Only 1 instance solved by Z3 contains more than 222

Table 14 Comparison of different Z3 strategies on entire benchmark set

	Z3 MaxRes		Z3 PD-MaxRes		Z3 WMax	
Solver State	**SMT2$_{1\text{-}Z3}$**	**SMT2$_{2\text{-}Z3}$**	**SMT2$_{1\text{-}Z3}$**	**SMT2$_{2\text{-}Z3}$**	**SMT2$_{1\text{-}Z3}$**	**SMT2$_{2\text{-}Z3}$**
Optimal	94	96	94	97	91	97
Suboptimal	0	0	0	0	0	0
Unsolved	160	158	160	157	163	157
Unsatisfiable	22	22	22	22	22	22
Undefined	2	2	2	2	2	2
TOTAL Solved	94	96	94	97	91	97

atomic constraints, only 5 instances contain between 100 and 160 atomic constraints. For OptiMathSAT, the parameters are significantly lower. The average permutation length of the 26 solved instances is 7. The 5 largest solved instances have permutation length 12 and their constraint sum ranges between 29 and 56. On average, the instances solved by OptiMathSAT contain only 20 atomic, 8 soft atomic, and 7 disjunctive constraints, but no direct successor constraints.

Finally, we tested Z3 on the benchmark set with its other strategies WMax [6, 45] and PD-MaxRes [7] and compared them to the MaxRes strategy. All strategies compare slightly better on the **SMT2$_{1\text{-}Z3}$** model, but there are only small differences between the strategies as Table 14 shows. As with the MaxRes strategy, Z3 using Wmax or PD-MaxRes either returns an optimal solution or leaves a problem instance unsolved. We also tried to run OptiMathSAT with its MaxRes strategy, but returned an error message on all generated SMT2 files indicating that it had problems extracting the objective function when using this strategy.

7 Summary of findings and research challenges

Our empirical analysis illustrated a varying performance of the various solvers on the CTW benchmark set. In particular, modern constraint solvers show impressive results and notably the IBM Cplex CP and Google OR-Tools CP-SAT solvers excel in the tests. For some solvers, their performance can be further tuned by setting options in the search strategies, however, these tunings did not make a significant impact. We believe that the future will be in automatic, rather than human-provided search strategy selection. In addition, we are convinced that rewriting models has some more potential, in particular, for improving the performance of MIP solvers. In the following, we summarize our findings, derive research challenges for constraint solvers, and discuss some issues for further maturing solvers towards complex real-world applications.

Simplifying Benchmarking Experiments We invested about one person year into the empirical testing of the solvers, which turned out be much more complex than expected. In particular, the semi-automatic generation of SMT2 files, but also the manual rewriting of models for the MIP solvers were time-consuming and error-prone steps requiring to write substantial pieces of software. Having a software environment in place, which allowed us to integrate all solvers and in particular also to automatically write and analyze log files as

Table 15 Results of solvers that have problems on one or both of the "borderline" instances R001 and R002

Instance	k	b	Chuffed	OR-Tools	Z3/OptiMathSAT	Gurobi
R001	0	0	Error	Empty log	Empty log	Infeasible model
R002	1	0	Valid solution	Empty log	Empty log	Valid solution

well as to validate all solver solutions, was instrumental to obtain reproducible results. Our work also emphasizes the need for a unified modeling language as well as standardized data formats, which would ease the exchange of models and data between different algorithms. Furthermore, having a standardized output interface in place to extract results and optimization costs would lower the benchmarking burden. We believe that following a modeling approach, which keeps models and instance data separately, provides an easier-to-access interface to solvers.

Undefined Solver States At the beginning of the experimental series, we defined a mapping of individual solver states to a common set of states, see Table 17 in Appendix B. States, which a solver returned and which were not part of our mapping, are mapped to a value of *undefined*. Interestingly, we obtained more such states than expected. Several solvers have issues with instances R001 and R002, see the summary in Table 15. Instance R001 contains no cables and no constraints, for which the empty permutation is the solution. Instance R002 contains one one-sided cable, i.e., a single job, and no constraints, but it is solvable with the permutation containing this single job.

The Chuffed solver returns an error and no solution for instance R001 on all three of its models. The OR-Tools CP-SAT solver works on instances R001 and R002 for about 1.5 seconds and then returns empty logs without a solution. The Gurobi MIP solver reports an infeasible model for instance R001 with all models that are not fed into the solver with the MiniZinc command line tool and thus reports 23 unsatisfiable instances instead of 22 in these cases. Z3 and OptiMathSAT also fail on these 2 instances. For the instances R001 and R002, the SMT2 tool chain has problems in generating correct files causing an error returned by both solvers, which we count for the undefined state. However, the problem here is not with the solver, but with the instance file generation, which cannot deal with the absence of constraints. For example, the .FZN file for R001 contains an array with bounds set to 1..0. The generation of the SMT2 files for instances R001 and R002 generates an error message "error: failed to generate SMT-LIB formula" thrown by the OptiMathSAT binary.

The undefined state of 8 instances for the Cplex MIP solver using the $\mathbf{M_{MIP}}$ model as shown in Table 9 are caused by a state of *unbounded or infeasible* returned by the solver. One of these instances is from the set of 22 unsatisfiable instances, the remaining 7 are satisfiable. The result was surprising as none of these instances is infeasible and our models are not unbounded. Upon closer inspection of the behavior of the solver, we located the reasons for this state by the presolve strategy applied by Cplex MIP. Switching off presolving allows the solver to either find a suboptimal solution or run into the time-limit without finding a solution. This behavior is known for problems, which are "borderline infeasible".[6] Further investigating the borderline infeasibility of some our instances, which is likely caused by the high number of constraints, would definitely be an interesting avenue for future research.

[6]See also https://www.ibm.com/support/pages/turning-cplex-presolve-or- gives-inconsistent-results and https://www.ibm.com/support/knowledgecenter/SSSA5P_12.10.0/ilog.odms.cplex.help/CPLEX/Parameters/topics/PreInd.html.

With the $\mathbf{DM_{MIP}}$ and $\mathbf{M_B}$ models, 1 instance or 3 instances result in the state *unbounded or infeasible*, recall Table 11. These instances all belong to the set of 22 unsatisfiable instances.

Deviations in Solution Costs Our validation software ensures that all constraints are satisfied by a solution returned by a solver and it also recalculates the costs of the criteria S, M, L, and N as well as the value of the overall objective. All cost values shown in the figures are based on these recalculations. In case of the OMT solvers, we added the optimization criteria S, L, M and N as output variables in addition to the overall value of the optimization objective, but still had problems in accessing the value of the criteria S and L. However, we did not investigate this issue further as our recalculations were available.

The costs returned by the Cplex CP and Cplex MIP solvers for optimal solutions showed some deviations in Tables 6, 4, and 10. For example, in Table 6 the optimal solutions for four instances violate more soft atomic constraints (criterion N) than optimal solutions found by other solvers on the same instances: A001 (+2), R126 (+6), R127 (+3), and R128 (+6). For each instance, Cplex CP shows a best bound of lower value and these solutions are within the default optimality tolerance, which is 1.0 e^{-9}. Even with a very low relative tolerance, the absolute deviations can be significant as our objective functions yields high cost values. For example, for instance R128 an optimal solution of costs 129,729 is computed and the effective tolerance is 12.97. Revalidating solution costs is important when solutions from different algorithms are compared with each other. Furthermore, the specification of optimization objectives is a challenging task and it also heavily depends on how a problem is modeled. Having good support available in modeling languages to formulate and test optimization objectives is highly desirable in particular from an application-oriented perspective.

Algorithmic Insights For Improving Models We presented several modeling variants for the CTW problem, which were all created manually by applying different modeling approaches. The development process of a model proceeds over many iterations and is often an error-prone process. Quite often it can happen that incorrect formulations of constraints render an instance unsatisfiable. Although constraint solvers can quickly identify minimal conflict sets of constraints, finding and removing the root cause of an inconsistency caused by an incorrectly formulated constraint is not straightforward. Tool support to further analyze inconsistencies would be more than desirable and be another promising avenue for further research. A first attempt at tackling this is made in [37]. Furthermore, feedback from solvers, which helps in understanding what parts of a model make it difficult to solve, would be more than desirable. From a user's perspective, a model should be as compact and easy to understand as possible. From a solver's perspective, the model should, e.g., allow for maximum constraint propagation. Today's solvers are able to determine the best possible search strategy automatically, whereas algorithms for automatic problem reformulation are only in its beginnings, see for example [14, 29, 46, 54] and we argue that much more can be done here.

Recognizing Hard and Easy Instances The CTW benchmark set comprises instances of varying difficulty. The constraint sum measure introduced in Section 4 appears to be a good first indicator of the difficulty of each instance. In Table 16 we give an overview of the first and third quartile of the constraint sum for each solver and solver state. The entries in the table correspond to the results of the best model for each solver. Whereas a correlation can be observed between the constraint sum and the solver state, the constraint sum does not give any indication of which types of constraints contribute the most to the difficulty of the

Table 16 First (Q1) and third (Q3) quartile of the constraint sum for each solver

	Chuffed	Cplex CP Tuned	OR Tools 8 Workers	Cplex MIP	Gurobi	Z3 PDMaxRes	OptiMath-SAT
	$\mathbf{M_Z}$	**DM**	**.DFZN**	$\mathbf{M_{MIP}}$	$\mathbf{M_B^{C\#}}$	$\mathbf{SMT2_2}$	$\mathbf{SMT2_1}$
Opt. Q1	49.50	48.5	75.50	42.00	44.50	30.00	7.50
Opt. Q3	203.25	207.00	642.00	158.00	157.75	96.00	23.50
Subopt. Q1	410.00	415.25	5086.00	274.00	292.00		
Subopt. Q3	827.00	1533.00	9829.00	682.00	638.50		
Unsolv. Q1	1551.75		11766.00	755.25	576.25	320.00	118.75
Unsolv. Q3	9267.25		11766.00	7016.50	1551.50	902.00	751.75

No entry in a row means that no data was available for this solver state as the solver found for example only optimal, but no suboptimal solutions such as in the case of the OMT solvers

instance. From a theoretical point of view, better understanding the phase transitions [12] of this benchmark set is an interesting research problem. From a practical point of view, better understanding the hardness and possible solution quality is desirable. Cplex CP is the only solver which finds solutions for all instances in the benchmark set, but it reports for example a gap of over 98 % for the large instances A70 to A73 with permutation length between 190 and 198 and between 7,000 and 10,000 atomic constraints.

Comparison of Solutions found by Humans and Constraint Solvers The current software makes use of Cplex CP and Google OR-Tools. Solvers are allowed to work on instances for at most five minutes. Human technicians spend up to several days on working out good solutions for layout and insertion order by practically testing variants directly on the machine. The constraint solver-based tool is used to make suggestions for layout and insertion order, with an insertion order computed for a given layout. Technicians often modify a layout if they cannot find a working insertion order. The solutions computed by the constraint solvers differ quite significantly from solutions found by humans, notably for large and complex cable trees with many housings and cables. When working with the same layout as a human, the solvers find solutions of comparable costs for smaller instances and of significantly reduced costs for larger instances. Potential cost savings also depend on the experience of a human technician. Constraint solvers can sometimes reduce costs by up to 30% compared to solutions found by less experienced technicians. The constraints represent a very conservative digital twin model for the machine and exclude many insertion orders, which humans consider. Human solutions often violate various constraints, because the solutions work when tested on the machine. The constraint solvers use the much stricter model to ensure that their solutions work and that they do not make suggestions, which fail when tested on the machine.

8 Conclusion

We discuss the problem of cable tree wiring (CTW), which we position as a variant of a traveling sales person problem with atomic, soft atomic, and disjuntive precedence constraints, direct successor constraints as well as tour-dependent edge costs. The CTW problem can also be considered as the first known representative of the coupled task scheduling problem

with soft constraints and as a new variant of the pickup and delivery TSP. Using the relationships to these known problems, we prove the NP-hardness of various subclasses of the CTW problem and also show that certain restrictions of the various constraint sets can make the problem solvable in polynomial time. In addition, we identify interesting subclasses of the problem for which the complexity is open. We also discuss the constraint sum parameter as a promising predictor for the difficulty of solving an instance.

We present a benchmark set of 278 real-world and artificial instances, which was included in the MiniZinc challenge 2020, and test state-of-the-art constraint, mixed-integer, optimization modulo theory solvers on this set using also different variants of how the problem can be modeled. In particular, the IBM Cplex CP and the Google OR-Tools CP-SAT solver showed impressive results, with Cplex being the only tested solver to find solutions for each instance in the benchmark set and OR-Tools finding more optimal solutions than any of the other solvers we tested. Our results demonstrate the remarkable progress made over recent years, in particular in the field of constraint and CP-SAT solvers, and also raises several interesting questions for future research.

Appendix A: Details of Models $\mathbf{M_Z}$, $\mathbf{DM_Z}$, $\mathbf{M_{GT}}$, $\mathbf{M_{MIP}}$, $\mathbf{DM_{MIP}}$, $\mathbf{M_B}$

This appendix shows the details of the models $\mathbf{M_Z}$, $\mathbf{DM_Z}$, and $\mathbf{M_{GT}}$ written in the MiniZinc Language. We also provide more details on the models $\mathbf{M_{MIP}}$, $\mathbf{DM_{MIP}}$, and $\mathbf{M_B}$ written in the language OPL used with the Cplex MIP solver. All other models, for example those implemented in C# for Gurobi, are found on github https://github.com/kw90/ctw_toolchain.

A.1 The constraint models $\mathbf{M_Z}$ and $\mathbf{DM_Z}$ in MiniZinc language

In contrast to the Cplex OPL language, the MiniZinc language does not provide possibilities to represent tuples, so constraints are represented using arrays. We begin by declaring a number of integer parameters for our models. OPL ranges are translated into sets of integers in MiniZinc.

```
int: k;
set of int: Positions = 1..k;
set of int: Cavities = Positions;
int: b; set of int: CavityPairs = 1..2*b;
set of int: CableStarts = 1..b;
```

Constraints of the different types are represented using arrays. The arguments of the arrays are the cavities between which the constraint must hold. For disjunctive constraints, a 2-dimensional array is used.

```
array[int,int] of Cavities: AtomicConstraints;
array[int,int] of Cavities: DisjunctiveConstraints;
array[int] of Cavities: DirectSuccessors;
array[int,int] of Cavities: SoftAtomicConstraints;
```

In the model $\mathbf{M_Z}$, we only introduce the *position_for_cavity (pfc)* to represent the desired permutation sequence. MiniZinc supports various global constraints, which are included

into the model to express the *allDifferent* constraint over the elements of this array in a straightforward way.

```
array[Cavities] of var Positions: pfc;
include "globals.mzn";
constraint all_different(pfc);
```

Constraints are declared using the keyword *constraint* and make use of the *index_set* notation in MiniZinc. The formulation of the disjunctive constraint also makes use of the special case when the cavity on the left-hand side of the precedence relation is the same in both disjuncts similar to the OPL **M** and **DM** models.

```
constraint
forall (i in index_set_1of2(AtomicConstraints))
(pfc[AtomicConstraints[i,1]] < pfc[AtomicConstraints[i,2]]);

constraint
forall (i in index_set_1of2(DisjunctiveConstraints))
(( pfc[DisjunctiveConstraints[i,1]] <
pfc[DisjunctiveConstraints[i,2]]
\/
pfc[DisjunctiveConstraints[i,3]] <
pfc[DisjunctiveConstraints[i,4]])
/\
if DisjunctiveConstraints[i,1]==DisjunctiveConstraints[i,3] then
max(pfc[DisjunctiveConstraints[i,2]], pfc[DisjunctiveConstraints
[i,4]])> pfc[DisjunctiveConstraints[i,1]]
else true endif);

constraint
forall (i in index_set(DirectSuccessors))
(if DirectSuccessors[i]<= b
then pfc[DirectSuccessors[i]] < pfc[DirectSuccessors[i]+b]
-> pfc[DirectSuccessors[i]] +1 = pfc[DirectSuccessors[i]+b]
else pfc[DirectSuccessors[i]] < pfc[DirectSuccessors[i]-b]
-> pfc[DirectSuccessors[i]] +1 = pfc[DirectSuccessors[i]-b]
endif);
```

The various decision variables for the optimization criteria as well as the objective function *obj* are defined as follows using the *abs*-function:

```
var int: S =
if b=0 then 0
else sum(i in CableStarts) (abs(pfc[i]-pfc[i+b]) > 1)
endif;

var int: M =
if b=0 then 0
```

```
else (max(i in CavityPairs)
(sum(j in CavityPairs where j<=b)
(pfc[j] < pfc[i] /\ pfc[i] < pfc[j+b])
+sum(j in CavityPairs where j>b)
(pfc[j] < pfc[i] /\ pfc[i] < pfc[j-b])))
endif;

var int: L =
if b=0 then 0
else max(i in CableStarts) (abs(pfc[i]-pfc[i+b])-1)
endif;

var int: N =
sum(i in index_set_1of2(SoftAtomicConstraints))
(pfc[SoftAtomicConstraints[i,1]] >
 pfc[SoftAtomicConstraints[i,2]]);

var int: obj = S*pow(k,3)+M*pow(k,2)+L*pow(k,1)+N;
```

The dual model $\mathbf{DM_Z}$ adds the dual representation *cfp* and the *inverse* constraint as well as a redundant *allDifferent* constraints.

```
array[Positions] of var Cavities: cfp;
constraint all_different(cfp);
constraint inverse(pfc,cfp);
```

For the $\mathbf{M'_Z}$ and $\mathbf{DM'_Z}$ models, the *abs*-function in the criterion S needs to be rewritten to avoid the absolute value. To do so, an additional array variable is introduced:

```
array[int] of var bool:
vio_abs = [pfc[i] - pfc[i+b] > 1 \/ pfc[i+b] - pfc[i] >
          1 | i in CableStarts];

var int: S = if b=0 then 0 else sum(vio_abs) endif;
```

Similarly, the criterion L needs to be rewritten for these models to avoid the *abs*-function. To do so, two auxiliary variables are introduced:

```
var int:
diff1 = if b=0 then 0 else (max(i in CableStarts)
(pfc[i] - pfc[i+b])) endif;

var int:
diff2 = if b=0 then 0 else (max(i in CableStarts)
(pfc[i+b] - pfc[i])) endif;

var int: L = if b=0 then 0 else (max(diff1, diff2) - 1) endif;
```

A.2 The constraint model M_{GT} in MiniZinc language

A different modeling approach is adopted for the $\mathbf{M_{GT}}$ model, which was contributed by Guido Tack, Monash University. The disjunctive constraint can be rewritten using an array of booleans and an additional constraint over this array can be added. The array captures the truth value of the two disjuncts in the disjunctive constraint and the constraint states that at least one of the disjuncts must be true for the disjunctive constraint to be satisfied.

```
array[int] of var bool: disj =
[ pfc[DisjunctiveConstraints[i,2*j+1]]
 <pfc[DisjunctiveConstraints[i,2*j+2]]
| i in index_set_1of2(DisjunctiveConstraints), j in 0..1
];

constraint forall (i in 1..length(disj) div 2) (disj[(i-1)*2+1]
 \/ disj[(i-1)*2+2]);
```

An array of booleans is used to capture the values of the decision variables *S* and *N* by introducing two additional variables *vio_abs* and *vio* and then representing the variables as sums over these boolean array values. The *vio_abs* array captures whether a cable end must be stored or not, based on whether the "other end" of the cable is plugged before or after the "cablestart". The *vio* array captures if the positions of two cavities in the permutation sequence satisfy or violate a given soft atomic constraint over these two cavities.

```
array[int] of var bool: vio_abs =
[ pfc[i]-pfc[i+b] > 1 \/ pfc[i+b]-pfc[i] > 1 |
i in CableStarts];
var int: S = sum(vio_abs);
array[int] of var bool: vio =
[pfc[SoftAtomicConstraints[i,1]] >=
pfc[SoftAtomicConstraints[i,2]]
| i in index_set_1of2(SoftAtomicConstraints)];
var int: N = sum(vio);
```

A.3 The mixed-integer programming models M_{MIP} and DM_{MIP}

The mixed-integer programming model $\mathbf{M_{MIP}}$ was manually rewritten in OPL based on the model **M** without any sophisticated reformulation of the constraints as for example discussed in [53]. In order to keep the model close to the constraint model **M** it is based on the *pfc* permutation. The *allDifferent* constraint is replaced by pairwise inequalities over cavity numbers assigned to *pfc* positions. All constraints are represented using tuples. Precedence constraints are reformulated as inequalities and direct successors are formulated in equation form.

```
int k = ...; // number of cavities, permutation length
int b = ...; // number of wired cavity pairs

range Cavities = 1..k;
```

```
range Cablestarts = 1 ..b;

tuple Atomic{ int cbefore; int cafter;};

// disjunctive constraints
// tuple Disjun{ int c1before; int c1after; int c2before;
   int c2after; };

{Atomic} AtomicConstraints = ...;

{Atomic} SoftAtomicConstraints = ...;

{Disjun} DisjunctiveConstraints = ...;

{int} DirectSuccessors = ...;

range Positions = 1..k;

range CavityPairs = 1..2*b ;

dvar int pfc[Cavities] in Positions;

//alldifferent
forall(i, j in Cavities: i != j) pfc[i] != pfc[j];

//atomic constraints
forall(c in AtomicConstraints)
pfc[c.cbefore] - pfc[c.cafter] <= 0;

//disjunctive constraints
//forall(d in DisjunctiveConstraints)
(pfc[d.c1before] - pfc[d.c1after] <= 0 ||
pfc[d.c2before] - pfc[d.c2after] <= 0 );

// direct successor constraints
forall (i in DirectSuccessors: i <= b)
pfc[i] - pfc[i+b] <= 0 => pfc[i+b] == pfc[i] + 1;

forall (i in DirectSuccessors: i > b)
pfc[i] - pfc[i-b] <= 0 => pfc[i-b] == pfc[i] + 1;
```

The optimization criteria are rewritten similarly and the objective function remains unchanged. Note that the absolute-function cannot be used for the criterion L as it returns a float value, which would make the problem unbounded.

```
dexpr int S = ((b == 0) ? 0: sum(j in Cavities: 1<= j<= b)
(maxl(pfc[j], pfc[j+b])-minl(pfc[j], pfc[j+b]) >= 2));
```

```
dexpr int L = ((b == 0) ? 0 : max (j in Cablestarts)
(maxl(pfc[j] - pfc[j+b], pfc[j+b] - pfc[j]) - 1));

dexpr int M = ((b == 0) ? 0  : (max (i in CavityPairs)
(sum(j in CavityPairs: j<=b) (pfc[j] <= (pfc[i] - 1) &&
 pfc[i] <= (pfc[j+b] - 1))
+ sum(j in CavityPairs: j>b) (pfc[j] <= (pfc[i] - 1) &&
pfc[i] <= (pfc[j-b] - 1)))));

dexpr int N =  sum(s in SoftAtomicConstraints)
(pfc[s.cafter] - pfc[s.cbefore] <= 0);

minimize S * pow(k, 3) + M * pow(k, 2) + L * pow(k, 1) + N;
```

The dual model $\mathbf{DM_{MIP}}$ extends the model $\mathbf{M_{MIP}}$ with the dual *cfp* representation and the pairwise inequalities for *cfp*. The *inverse* constraint, which is not available in the MIP variant of OPL, is expressed by equalities.

```
// position for chamber dvar int pfc[Cavities] in Positions;
//chamber for position dvar int cfp[Positions] in Cavities;

//alldifferent
forall(i, j in Cavities: i != j) pfc[i] != pfc[j];
forall(i, j in Positions: i != j) cfp[i] != cfp[j];

//duality (channeling constraint)
forall(j in Cavities, p in Positions)
pfc[j] == p => cfp[p] == j;

forall(j in Cavities, p in Positions)
cfp[p] == j => pfc[j] == p;
```

A.4 The mixed-integer programming model $\mathbf{M_B}$

The model $\mathbf{M_B}$ makes use of big-M reformulations for disjunctive constraints [55] using additional decision variables and non-integer (floating point) optimization costs. The value of the big-M constant has to be chosen sufficiently big to prevent the problem from becoming unsolvable. As a rule of thumb, the big-M constant should be at least 100 times larger than the largest value of any of the variables. However, it should not be too large, because otherwise numerical instability and round-off errors can occur. Using an arbitrarily large M value also expands the feasible region of the LP relaxation unnecessarily and results in longer runtimes, see [10]. We decided to set `int bigM=k*100` as smaller or larger values led to incorrect values for some optimization criteria.

The following model $\mathbf{M_B}$ for the Cplex MIP solver uses an upper triangular matrix of booleans that indicates if a cavity is wired before another:

```
dvar boolean lt[Cavities][Cavities];
```

If $lt[i, j] = 1$, then *pfc[i]* < *pfc[j]*, otherwise *pfc[i]* > *pfc[j]*. Only the upper triangular matrix is defined, i.e., values are only defined for $i < j$. To find out if *pfc[i]* < *pfc[j]* when $i > j$, the value of $lt[j, i]$ is determined and inverted. The diagonal of the lt matrix is undefined, because a cavity cannot be placed before or after itself in the permutation. A solution can be directly extracted from the lt matrix as the subsequent example explains.

$$lt = \begin{bmatrix} 1 & 1 & 1 \\ & 0 & 1 \\ & & 1 \\ & & \end{bmatrix} \qquad \begin{array}{l} pfc = 1, 3, 2, 4 \text{ because} \\ pfc[1] < pfc[2], \quad pfc[1] < pfc[3], \\ pfc[1] < pfc[4], \quad pfc[2] > pfc[3], \\ pfc[2] < pfc[4], pfc[3] < pfc[4] \end{array}$$

Atomic constraints are expressed as entries in the lt matrix:

```
forall(a in AtomicConstraints)
if (a.cbefore < a.cafter) lt[a.cbefore,a.cafter] == 1;
else lt[a.cafter,a.cbefore] == 0;
```

The *allDifferent* constraint is expressed as inequalities over lt matrix values stating that either *pfc[i]* or *pfc[j]* has to be larger than the other.

```
forall(ordered i,j in Cavities)
pfc[i] - pfc[j] + 1 <= bigM * (1 - lt[i,j]);
forall(ordered i,j in Cavities)
pfc[j] - pfc[i] + 1 <= bigM * lt[i,j];
```

For example, if $lt[i, j]$ is equal to 0 and therefore *pfc[i]* > *pfc[j]*, the first constraint is always satisfied if the big-M constant is chosen big enough. The right term in the second constraint evaluates to 0 and the left term has to be smaller or equal to 0, which is only possible if the two values are different. Disjunctive constraints are formulated as inequality constraints. A satisfied disjunctive constraint is larger or equal to 1, because we sum up values for each atomic constraint in the disjunction from the lt matrix. As only the upper triangular matrix of the lt matrix is defined, case distinctions based on cavity indices have to be made and some values have to be inverted. With a fully populated matrix lt, each disjunctive constraint is expressed by the following conditional inequalities. Note that for each disjunctive constraint, only one of the if-clauses is satisfied.

```
forall(d in DisjunctiveConstraints){
if (d.c1before < d.c1after && d.c2before < d.c2after)
{lt[d.c1before,d.c1after] + lt[d.c2before,d.c2after]
>= 1;}
if (d.c1before < d.c1after && d.c2before > d.c2after)
{lt[d.c1before,d.c1after] + 1-lt[d.c2after,d.c2before]
>= 1;}
if (d.c1before > d.c1after && d.c2before < d.c2after)
{1-lt[d.c1after,d.c1before] + lt[d.c2before,d.c2after]
>= 1;}
if (d.c1before > d.c1after && d.c2before > d.c2after)
{1-lt[d.c1after,d.c1before] + 1-lt[d.c2after,d.c2before]
>= 1;}}
```

Direct successor constraints also make a case distinction based on cavity indices. If i is in *DirectSuccessors*, then $i + b$ must follow immediately in the permutation, which means that the left-hand side of the condition must be equal to 0 if and only if *pfc[i] < pfc[i+b]*.

```
forall(i in Cablestarts:
i in DirectSuccessors || (i+b) in DirectSuccessors){
if (i in DirectSuccessors)
pfc[i+b] - pfc[i] - 1 <= bigM * (1 - lt[i,i+b]);
if ((i+b) in DirectSuccessors)
pfc[i] - pfc[i+b] - 1 <= bigM * lt[i,i+b];}
```

Minimum and maximum positions of cavities in the permutation are stored in arrays *minimum* and *maximum*. Their values are set by constraints.

```
dvar float+ minimum[Cablestarts] in Positions;
dvar float+ maximum[Cablestarts] in Positions;

forall(i in Cablestarts) {
minimum[i] - pfc[i]  <= 0;                          (1)
minimum[i] - pfc[i+b] <= 0;                         (2)
pfc[i] - minimum[i]  <= bigM * (1-lt[i,i+b]);       (3)
pfc[i+b] - minimum[i] <= bigM * lt[i,i+b];          (4)
maximum[i] == pfc[i]+pfc[i+b] - minimum[i];}        (5)
```

Constraints (1) and (2) ensure that the value of *minimum[i]* is larger or equal to *min(pfc[i],pfc[i+b])*. Constraints (3) and (4) ensure that the value of *minimum[i]* is smaller or equal to *min(pfc[i],pfc[i+b])*, depending on which of the cavities in a job pair occurs first in the permutation. The value of *minimum[i]* is set correctly if and only if constraints (1) to (4) are satisfied. Calculating the value of *maximum[i]* is then trivial using constraint (5).

If a job pair is interrupted, its entry in the array *cableIsStored* is set to 1, which happens if the difference between the *minimum* and *maximum* position of cavities from a job pair in the permutation is larger than or equal to 2.

```
dvar boolean cableIsStored[Cablestarts];

forall(p in Cablestarts) {
2 - maximum[p] + minimum[p] <= bigM * (1-cableIsStored[p]);
maximum[p] - minimum[p] - 1 <= bigM * cableIsStored[p];}
```

The variable *cableIsStoredAtPosition[i,t]* is equal to 1 if one cavity from the job pair i occurs in the permutation before the position t, but the other does not. Otherwise *cableIsStoredAtPosition[i,t]* is equal to 0. The variable *cableIsPluggedBefore[i,t]* is equal to 1 if both cavities of a job pair i occur in the permutation before position t. The variable *cableIsPluggedAfter[i,t]* is equal to 1 if both cavities of a job pair i occur in the permutation after position t.

```
dvar boolean cableIsStoredAtPosition[Cablestarts,Positions];
dvar boolean cableIsPluggedBefore[Cablestarts,Positions];
dvar boolean cableIsPluggedAfter[Cablestarts,Positions];
```

Because a cable can either be completely inserted before one position in the permutation or only be inserted with one cable end or not be inserted at all, the following constraints hold:

```
forall(p in Cablestarts, t in Positions) {
cableIsStoredAtPosition[p,t] + cableIsPluggedBefore[p,t]
+ cableIsPluggedAfter[p,t] == 1;
minimum[p]-t+1 <= bigM*(1-cableIsStoredAtPosition[p,t]);
t-maximum[p]+1 <= bigM*(1-cableIsStoredAtPosition[p,t]);
t-minimum[p]  <= bigM*(1-cableIsPluggedBefore[p,t]);
maximum[p]-t  <= bigM*(1-cableIsPluggedAfter[p,t]);}
```

The maximum number of cables M that are simultaneously contained in storage is set implicitly by a constraint. Any value larger than M satisfies this constraint, however since M is part of the optimization objective, M is correctly set to be the smallest value satisfying this constraint. Similarly, the value L is also set implicitly by a constraint, whereas the calculation of N and S is straightforward:

```
forall(t in Positions) sum(p in Cablestarts)
cableIsStoredAtPosition[p,t] <= M;

forall(p in Cablestarts) maximum[p] - minimum[p] - 1 <= L;

N == sum(s in SoftAtomicConstraints)
(pfc[s.cafter] - pfc[s.cbefore] <= 0);

S == sum(p in Cablestarts) cableIsStored[p];
```

The objective function is identical to the one used in the $\mathbf{M_C}$ model.

```
minimize S * pow(k, 3) + M * pow(k, 2) + L * pow(k, 1) + N;
```

For the Gurobi solver, we manually implemented the models $\mathbf{M_I}$, $\mathbf{M_B}$, and $\mathbf{M_B}$ in the C# API of this solver.

Appendix B: Details on experimental setup: mapping solver states

Before beginning any experimentation, we needed to decide how to map the individual solver states to a common set of states. For the experiments, we mapped the individual solver states to five possible outcomes:

- *unsatisfiable*: the solver has proven the instance to have no solution,
- *optimal*: the solver has found a solution and proven that no better solution with lower costs exists,
- *suboptimal*: the solver has found a solution, but was not able to prove it as optimal,
- *unsolved*: the solver was not able to find a solution or prove an instance as being unsatisfiable within a given time limit,
- *undefined*: any other state returned by a solver not mapped to one of the states above.

Table 17 summarizes the mapping. For Chuffed and OR-Tools, the entries refer to a string syntax used by MiniZinc to represent the status of a solution. For Cplex CP, solver

Table 17 Mapping of solver-specific information about solver state and solution existence to four outcomes of a benchmark test, any other state returned by a solver is mapped to a state *undefined*

	Optimal	Suboptimal	Unsatisfiable	Unsolved
Chuffed	"=========="	"----------"	"UNSATISFIABLE"	"UNKNOWN"
OR-Tools	"=========="	"----------"	"UNSATISFIABLE"	"TIMEOUT"
Cplex CP	SearchCompleted and cp.Solve==true	SearchStopped and cp.Solve==true	SearchCompleted and cp.Solve==false	SearchStopped and cp.Solve==false
Cplex MIP	Optimal or OptimalTol	AbortTimeLim and cplex.Solve==true	Infeasible	AbortTimeLim and cplex.Solve==false
Gurobi	GRB.Status==2	GRB.Status==9 and SolCount>0	GRB.Status==3	GRB.Status==9 and SolCount==0
Z3	n/a (cost-based)	"sat"	"unsat"	"timeout"
OptiMathSat	"sat_optimal"	"sat"	"unsat"	"timeout"

states are defined by a parameter value IloCP::ParameterValues of the solver and there is a separate boolean parameter to indicate whether a solution was found or not. For Cplex MIP, we check the solver state in the Cplex.CplexStatus parameter and the existence of a solution. Gurobi returns a numerical solver status code in parameter GRB.Status and a solution count in parameter GRBModel.SolCount that we check in addition to the status code. Z3 distinguishes three different solver states in a status variable and adds the state in a string to the output file containing the solution. However, it has no explicit state for marking a solution as being optimal. Therefore, we checked its solution costs and if this is equal to the cost of a solution marked as optimal by another solver we also count it as an optimal solution found by Z3. OptiMathSAT outputs a solution status on the Linux console, to which we added a timeout output string via our Python wrapper in case the solver exhausted the time limit.

Appendix C: Details on experimental setup: setting an appropriate time limit

We also ran experiments to find an appropriate time limit for how long to invoke a solver on an instance. We are interested in setting a time limit, which allows solvers to find good solutions or even solve instances optimally. However, solvers can easily get stuck in large search spaces resulting from the very large instances in our benchmark set and investing more time will not allow them to significantly improve solution quality. Therefore, we ran a number of tests with the instances from the challenge set using selected combinations of the models **DM**, **M**, and $\mathbf{DM_{GT}}$ with some solvers under time limits of 2, 5, 10, and 20 minutes. For these tests, we selected all constraint solvers, the Cplex MIP solver, and Z3. We compared the costs of solutions found by these solvers under different time limits. Table 18 summarizes the results for the Cplex CP solver using the **DM** model, which solves all challenge set instances under all time limits, but cannot prove any of its solutions as optimal.

Table 18 Results for Cplex CP on the challenge set under different time limits, entries show solution costs, none of the solutions found were marked as optimal

	Cplex CP **DM**			
Instance	2 minutes	5 minutes	10 minutes	20 minutes
A033	7,742,021	7,742,021	7,741,861	7,741,781
A060	43,318,041	43,308,025	42,338,828	41,317,903
A066	374,829,658	364,859,160	359,918,105	335,380,634
A069	529,386,562	503,541,657	484,268,711	464,931,037
A073	630,266,347	536,520,241	420,367,496	288,011,609
R192	26,007,513	22,587,155	20,337,530	20,337,530
R193	15,827,902	10,191,828	10,191,619	10,181,118
R194	36,724,491	28,217,186	19,748,757	19,748,757
R195	34,748,796	28,069,153	28,069,153	28,069,150
R196	34,782,897	32,105,826	30,774,826	18,685,725

Cplex CP can improve solution costs on all instances when given more time. However, as Fig. 5 illustrates, the improvement is much less significant from 10 to 20 minutes when compared with the decrease in costs made when going from 2 to 5. Investing 10 minutes yields relevant cost reductions for only three instances.

Table 19 summarizes the results for all other solvers, which only solve a few instances from the challenge set. Only OR-Tools using a non-dual model is able to solve one instance optimally in 5 minutes and 2 more instances optimally in 20 minutes. Cplex MIP was only able to solve one instance, whereas Z3 was not able to solve any instance even when given a 20 minutes time limit.

A 5 minute time limit allows solvers to find solutions as the experiments show. For instances, for which only suboptimal solutions are found, running the solvers for 10 or 20

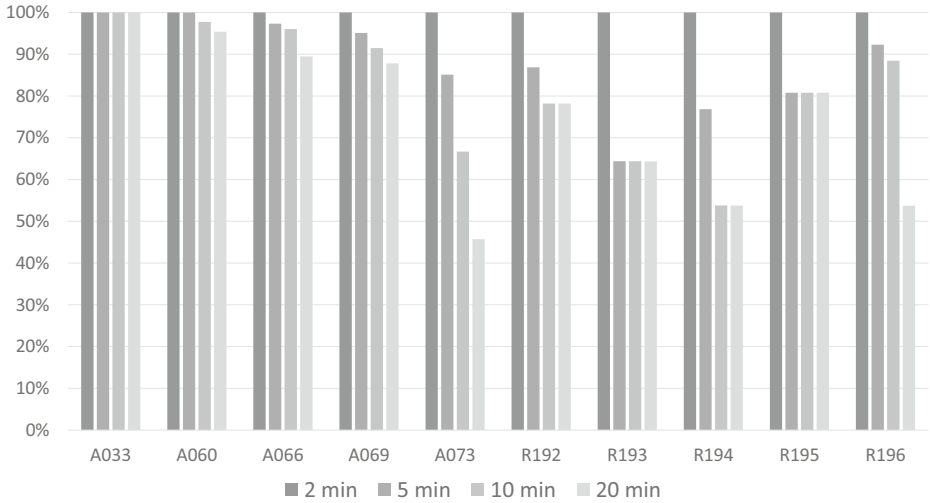

Fig. 5 Relative decrease in solution costs for Cplex CP on the challenge set over all tested time limits, costs of the solution found within 2 minutes is set as 100%

Table 19 Summary of results obtained by Chuffed, Cplex MIP, and OR-Tools on the challenge set under different time limits

Instance	2 minutes	5 minutes	10 minutes	20 minutes	Solver
A033		20,671,476	19,634,521	18,610,991	Chuffed $\mathbf{M_Z}$
	19,660,117	20,099,646	19,615,230	18,563,257	Chuffed $\mathbf{M_{GT}}$
			18,026,120	16,465,075	Cplex MIP $\mathbf{M_{MIP}}$
		5,153,238	5,153,238	5,153,238	OR-Tools $\mathbf{.FZN_1}$
	11,391,543	**5,153,238**	5,153,238	5,153,238	OR-Tools $\mathbf{.FZN_2}$
A060				46,359,032	Cplex MIP $\mathbf{M_{MIP}}$
R192			58,965,217	58,965,217	Chuffed $\mathbf{M_Z}$
			28,202,636	**12,408,258**	OR-Tools $\mathbf{.FZN_1}$
			36,118,707	**12,408,258**	OR-Tools $\mathbf{.FZN_2}$
R193				54,215,758	Chuffed $\mathbf{M_{GT}}$
				11,284,655	OR-Tools $\mathbf{.FZN_1}$
				19,234,102	OR-Tools $\mathbf{.FZN_2}$
R194				78,936,063	OR-Tools $\mathbf{.FZN_1}$
			49,328,866	25,369,932	OR-Tools $\mathbf{.FZN_2}$
R195				73,481,904	Chuffed $\mathbf{M_{GT}}$
				28,107,669	OR-Tools $\mathbf{.FZN_1}$
				26,750,039	OR-Tools $\mathbf{.FZN_2}$
R196				24,021,937	OR-Tools $\mathbf{.FZN_1}$
		61,487,932	40,095,576	**14,679,739**	OR-Tools $\mathbf{.FZN_2}$

Only those instances are shown where at least one solution was found by a solver. Instances from the challenge set not shown in this table where not solved by any of the selected solvers under the given time limit(s). Entries show solution costs. The few optimal solutions found are marked in bold. Cells with no entries mean that no solution was found by any of the solvers

minutes yields improvements, but they rarely allow solvers to find optimal solutions. We thus set the time limit for all experiments to 5 minutes.

Appendix D: Details on tuning constraint solver performance

The following tuning options were investigated for the three constraint solvers: For Cplex CP, we investigated the influence of extended *inference level setting*s, which allow the constraint solver to control the strength of domain reduction that it can achieve on the constraint variables by performing more or less constraint propagation. We set three inference levels to the value *extended*: *default inference level*, *precedence inference level*, and *allDifferent inference level*.

For the Chuffed solver, we experimented with different search annotations in MiniZinc and Chuffed using non-dual models. We first experimented with an annotation on the *allDifferent* constraint in the $\mathbf{M_Z}$ and $\mathbf{DM'_Z}$ models to use *bounds* or *domain* propagation:

```
constraint all_different(pfc)::bounds;
constraint all_different(pfc)::domain;
```

Table 20 Impact of different combinations of search annotations on solution costs in Chuffed compared to Chuffed's performance without search annotations taken as 100% baseline, best values are marked in bold

		Variable Choice				
	Variable Value Choice	dom_w_deg	first_fail	impact	most_constrained	occurence
$\mathbf{M_Z}$	indomain_min	99.91	99.87	99.66	99.99	100.00
$\mathbf{M_Z}$	indomain_split_random	97.71	**97.69**	97.73	98.28	97.84
$\mathbf{M_{GT}}$	indomain_min	99.90	99.83	100.00	99.94	99.80
$\mathbf{M_{GT}}$	indomain_split_random	97.66	97.65	97.85	**97.50**	97.55

However, these annotations had no significant impact on the number of instances that Chuffed can solve. Chuffed can find none more suboptimal solution when using the $\mathbf{M_{GT}}$ model with annotated domain propagation on the *allDifferent* constraint. Solution costs even increased slightly. We therefore abandoned this annotation.

We then experimented with various combinations of search annotations in order to control how the Chuffed solver conducts its variable choices and how it selects the domain values for a variable, which seemed appropriate for the CTW domain. None of the combinations had a relevant impact on the number of instances solved by Chuffed, but sometimes solutions of lower cost are found. Table 20 summarizes the relative changes in costs on the same subset of 208 (43 artificial and 165 real-world) instances where Chuffed finds an optimal or suboptimal solution in its default configuration (costs are set to 100%) or when using any of the different combinations of search annotations. For the choice of how to constrain a variable, *indomain_split_random*, which assigns a random value from the variable's domain, works best, whereas the variable choice settings lead to no clear picture. For the $\mathbf{M_Z}$ model, the *first_fail* strategy (choose the variable with the smallest domain size) works best and for the $\mathbf{M_{GT}}$ model, the *most_constrained* strategy (choose the variable with the smallest domain, breaking ties using the number of constraints) works best.

As a result, we decided to tune Chuffed with the best working search annotation and applied this annotation also to the dual model $\mathbf{DM'_Z}$:

```
::int_search(pfc, first_fail,indomain_split_random)
```

Table 21 Parameters of the ten instances with smallest average CP solving time

Instance	Two-sided Cables b	Atomic Constraints	Soft atomic Constraints	Disjunctive Constraints	Constraint Sum	Average CP Solving time
A005	2	3	2	0	7	0.2420s
A007	1	0	0	0	2	0.2529s
A008	3	9	2	1	15	0.2502s
A012	2	0	2	0	6	0.2512s
A013	2	5	2	0	9	0.2506s
A054	1	0	0	0	1	0.2251s
A055	3	10	3	1	23	0.2491s
R026	0	30	7	2	39	0.2391s
R034	3	9	3	0	15	0.2546s
R170	1	1	1	0	3	0.2370s

Table 22 Parameters of the 22 unsatisfiable instances in the benchmark set

Instance	Two-sided Cables b	Atomic Constraints	Soft atomic Constraints	Disjunctive Constraints	Constraint Sum	Average CP Solving time
A049	35	1394	57	115	1671	2.5021s
A050	35	1392	57	134	1637	2.4412s
R005	18	137	18	15	212	0.7713s
R007	23	322	16	85	461	2.0450s
R024	6	89	19	11	129	0.3476s
R027	6	96	16	10	132	0.3309s
R028	6	102	16	11	139	0.3318s
R029	6	91	16	9	126	0.3241s
R030	6	105	17	12	144	0.3180s
R071	3	10	3	1	23	0.2491s
R072	6	4	10	7	39	0.3049s
R074	10	12	17	9	64	0.6241s
R100	23	196	26	47	310	1.5554s
R153	37	1060	47	137	1299	2.7011s
R154	38	1060	48	142	1306	2.8043s
R155	37	1066	47	138	1306	2.7057s
R156	39	1100	48	152	1358	3.2733s
R157	39	1106	48	156	1368	3.2711s
R158	40	1113	56	119	1346	3.2018s
R159	40	1055	56	120	1289	3.1522s
R161	40	723	54	204	1037	3.8566s
R198	25	477	28	71	605	1.3635s

Appendix E: Easily solvable and unsatisfiable instances

In Table 21, we give an overview of the parameters and average solving time of CP solvers for the ten instances with smallest average CP solving time. These instances may be considered the easiest instances in the benchmark set. Note that we consider for the average CP solving time the solving times of the Cplex CP solver on the **M** and **DM** models, the Chuffed solver on the $\mathbf{M_Z}$, $\mathbf{M_{GT}}$, and $\mathbf{DM_Z}$ models, and the Google OR-Tools solver on the $\mathbf{.FZN_1}$, $\mathbf{.FZN_2}$, and **.DFZN** models. In Table 22 we give a similar overview for all unsatisfiable instances in the benchmark set.

Acknowledgements This work was partially supported by the Swiss Innovation Agency innosuisse and the RICAIP project that has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 857306. We thank Stefan Bucheli, Beat Estermann, Roland Liem, Georg Moravitz, Kurt Ulrich, and Zeta technicians from Komax AG for their support, fruitful cooperation, and access to this interesting data set. Deep thanks goes to Bernhard Nebel, Albert-Ludwigs-University Freiburg, for a fruitful discussion on the complexity of the CTW problem and to Guido Tack, Monash University, for feedback on an earlier version of this paper and for contributing one of the models. Deep thanks also goes to the reviewers for their helpful comments.

Funding Open Access funding enabled and organized by Projekt DEAL.

Declarations

Conflict of Interests The authors declare that they have no conflict of interest.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Abbou, R., Barman, J., Martinez, C., Verma, S. (2017). Dynamic route planning and scheduling in flexible manufacturing systems with heterogeneous resources, a max-plus approach. In *Control & Automation (ICCA), 2017 13th IEEE international conference on* (pp. 723–728): IEEE.
2. Arora, J.S. (2017). *Multi-objective optimum design concepts and methods, Chap. 18*, (pp. 771–794). Cambridge: Academic Press.
3. Austrin, P., Manokaran, R., Wenner, C. (2013). On the NP-hardness of approximating ordering constraint satisfaction problems. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and techniques, LNCS*, (Vol. 8096 pp. 26–41): Springer.
4. Baptiste, P. (1996). Disjunctive constraints for manufacturing scheduling: Principles and extensions. *International Journal of Computer Integrated Manufacturing*, *9*(4), 306–310.
5. Benoist, T. (2008). Soft car sequencing with colors: Lower bounds and optimality proofs. *European Journal of Operational Research*, *191*(3), 957–971.
6. Bjørner, N. (2011). Engineering theories with Z3. In *Asian Symposium on Programming Languages and Systems* (pp. 4–16): Springer.
7. Bjørner, N., Phan, A.D., Fleckenstein, L. (2015). *νz*-An Optimizing SMT Solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 194–199): Springer.
8. Bonet, M.L., Levy, J., Manya, F. (2007). Resolution for Max-SAT. *Artificial Intelligence*, *171*(8), 606–618.
9. Bredström, D., & Rönnqvist, M. (2008). Combined vehicle routing and scheduling with temporal precedence and synchronization constraints. *European Journal of Operational Research*, *191*(1), 19–31.
10. Camm, J.D., Raturi, A.S., Tsubakitani, S. (1990). Cutting Big M down to size. *Interfaces*, *20*(5), 61–66.
11. Chalasani, P., & Motwani, R. (1999). Approximating capacitated routing and delivery problems. *SIAM Journal on Computing*, *28*(6), 2133–2149.
12. Cheeseman, P., Kanefsky, B., Taylor, W.M. (1991). Where the really hard problems are. In *12Th International Joint Conference on Artificial Intelligence, IJCAI'91* (pp. 331–337).
13. Chen, C.P. (1990). AND/OR precedence constraint traveling salesman problem and its application to assembly schedule generation. In *Systems, Man and Cybernetics, 1990. Conference proceedings., IEEE international conference on* (pp. 560–562): IEEE.
14. Chu, G., & Stuckey, P.J. (2015). Dominance breaking constraints. *Constraints*, *20*(2), 155–182.
15. Chu, G., Stuckey, P.J., Schutt, A., Ehlers, T., Gange, G., Francis, K. Chuffed: A lazy clause solver. https://github.com/chuffed/chuffed.
16. Condotta, A., & Shakhlevich, N.V. (2012). Scheduling coupled-operation jobs with exact time-lags. *Discrete Applied Mathematics*, *160*(16-17), 2370–2388.
17. Contaldo, F., Trentin, P., Sebastiani, R. An enhanced mzn2fzn compiler for OptiMathSAT. https://github.com/PatrickTrentin88/emzn2fzn.
18. Contaldo, F., Trentin, P., Sebastiani, R. (2020). From MINIZINC to optimization modulo theories, and back. In *Proceedings of the 17th International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research, LNCS*: Springer.

19. Da Col, G., & Teppan, E. (2019). Google vs IBM: A constraint solving challenge on the job-shop scheduling problem. arXiv:1909.08247.
20. De Moura, L., & Bjørner, N. (2008). Z3: an efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337–340): Springer.
21. Donati, A.V., Montemanni, R., Casagrande, N., Rizzoli, A.E., Gambardella, L.M. (2008). Time dependent vehicle routing problem with a multi ant colony system. *European Journal of Operational Research*, *185*(3), 1174–1191.
22. Fagerholt, K., & Christiansen, M. (2000). A travelling salesman problem with allocation, time window and precedence constraints—an application to ship scheduling. *International Transactions in Operational Research*, *7*(3), 231–244.
23. Gao, T., & Liu, C. (1996). Minimum crosstalk channel routing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *15*(5), 465–474.
24. Gottlieb, J., Puchta, M., Solnon, C. (2003). A study of greedy, local search, and ant colony optimization approaches for car sequencing problems. In *EvoWorkshops*, (Vol. 2611 pp. 246–257): Springer.
25. Grabowski, J., & Wodecki, M. (2004). A very fast tabu search algorithm for the permutation flow shop problem with makespan criterion. *Computers & Operations Research*, *31*(11), 1891–1909.
26. Gurobi. http://www.gurobi.com/.
27. Gutin, G., & Punnen, A.P. (Eds.) (2007). *The Traveling Salesman Problem and its Variations*. Berlin: Springer.
28. Haghani, A., & Jung, S. (2005). A dynamic vehicle routing problem with time-dependent travel times. *Computers & Operations Research*, *32*(11), 2959–2986.
29. Heinz, S., Schulz, J., Beck, J.C. (2013). Using dual presolving reductions to reformulate cumulative constraints. *Constraints*, *18*(2), 166–201.
30. Hnich, B., Smith, B.M., Walsh, T. (2004). Dual modelling of permutation and injection problems. *JAIR*, *21*, 357–391.
31. IBM: Cplex. https://www.ibm.com/products/ilog-cplex-optimization-studio/.
32. Kahn, A.B. (1962). Topological sorting of large networks. *Communications of the ACM*, *5*(11), 558–562.
33. Karp, R.M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations* (pp. 85–103): Springer.
34. Ku, W.Y., & Beck, J.C. (2016). Mixed integer programming models for job shop scheduling: A computational analysis. *Computers & Operations Research*, *73*, 165–173.
35. Kubo, M., & Kasugai, H. (1991). The precedence constrained traveling salesman problem. *Journal of the Operations Research Society of Japan*, *34*(2), 152–172.
36. Lageweg, B.J., Lenstra, J.K., Kan, A.H.G.R. (1978). A general bounding scheme for the permutation flow-shop problem. *Operations Research*, *26*(1), 53–67.
37. Leo, K., & Tack, G. (2017). Debugging unsatisfiable constraint models. In *International conference on AI and OR techniques in constraint programming for combinatorial optimization problems* (pp. 77–93): Springer.
38. Manlove, D.F., & McBride, I. (2017). Trimble, J.: "Almost-stable" matchings in the hospitals/residents problem with couples. *Constraints*, *22*(1), 50–72.
39. Mapa, S.M.S., & Urrutia, S. (2015). On the maximum acyclic subgraph problem under disjunctive constraints. *Information Processing Letters*, *115*(2), 119–124.
40. Miltersen, P.B., Radhakrishnan, J., Wegener, I. (2005). On converting CNF to DNF. *Theoretical Computer Science*, *347*(1-2), 325–335.
41. Mitchell, D., Selman, B., Levesque, H. (1992). Hard and easy distributions of SAT problems. In *Proceedings of the 10th national conference on AI (AAAI)* (pp. 459–465).
42. Moon, C., Kim, J., Choi, G., Seo, Y. (2002). An efficient genetic algorithm for the traveling salesman problem with precedence constraints. *European Journal of Operational Research*, *140*(3), 606–617.
43. Narodytska, N., & Bacchus, F. (2014). Maximum satisfiability using core-guided MaxSat resolution. In *Proceedings of the 28th AAAI conference on artificial intelligence, AAAI'14*: AAAI press.
44. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G. (2007). MiniZinc: Towards a standard CP modelling language. In *International Conference on Principles and Practice of Constraint Programming* (pp. 529–543): Springer.
45. Nieuwenhuis, R., & Oliveras, A. (2006). On SAT modulo theories and optimization problems. In *International conference on theory and applications of satisfiability testing* (pp. 156–169): Springer.
46. Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P. (2017). Automatically improving constraint models in Savile Row. *Artificial Intelligence*, *251*, 35–61.
47. Orman, A.J., & Potts, C.N. (1997). On the complexity of coupled-task scheduling. *Discrete Applied Mathematics*, *72*(1-2), 141–154.
48. Google OR-tools. https://developers.google.com/optimization/.

49. Osman, I.H., & Potts, C. (1989). Simulated annealing for permutation flow-shop scheduling. *Omega. Int. Journal of Management Science*, *17*(6), 551–557.
50. Pferschy, U., & Schauer, J. (2013). The maximum flow problem with disjunctive constraints. *Journal of Combinatorial Optimization*, *26*(1), 109–119.
51. Picard, J.C., & Queyranne, M. (1978). The time-dependent traveling salesman problem and its application to the tardiness problem in one-machine scheduling. *Operations Research*, *26*(1), 86–110.
52. Rashid, M.F.F.A., Jusop, M., Mohamed, N.M.Z. (2018). R.romlay, F.: Optimization of travelling salesman problem with precedence constraint using modified GA encoding. *Advanced Science Letters*, *24*(2), 1484–1487.
53. Refalo, P. (2000). Linear formulation of constraint programming models and hybrid solvers. In *Intertanional Conference on Principles and Practice of Constraint Programming* (pp. 369–383): Springer.
54. Rendl, A. (2010). Effective compilation of constraint models. Ph.D. thesis, University of St Andrews.
55. Ruiz, J.P., & Grossmann, I.E. (2017). Global optimization of non-convex generalized disjunctive programs: a review on reformulations and relaxation techniques. *Journal of Global Optimization*, *67*(1), 43–58.
56. Rytter, W., & Szreder, B. (2012). Computing maximum hamiltonian paths in complete graphs with tree metric. In *International Conference on Fun with Algorithms* (pp. 346–356): Springer.
57. Sawada, H., Mukai, R., Araki, S., Makino, S. (2004). A robust and precise method for solving the permutation problem of frequency-domain blind source separation. *IEEE Transactions on Speech and Audio Processing*, *12*(5), 530–538.
58. Sebastiani, R., & Trentin, P. (2015). OptiMathSAT: a tool for optimization modulo theories. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I* (pp. 447–454).
59. Solnon, C. (2000). Solving permutation constraint satisfaction problems with artificial ants. In *Proceedings of the 14th European Conference on Artificial Intelligence* (pp. 118–122): IOS press.
60. Stein, D.M. (1978). An asymptotic, probabilistic analysis of a routing problem. *Mathematics of Operations Research*, *3*, 89–101.
61. Tasgetiren, M.F., Liang, Y.C., Sevkli, M., Gencyilmaz, G. (2007). A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *European Journal of Operational Research*, *177*(3), 1930–1947.
62. Vajda, S. (1961). Mathematical programming, Addison-Wesley, Boston.
63. Vander Wiel, R.J., & Sahinidis, N.V. (1996). An exact solution approach for the time-dependent traveling-salesman problem. *Naval Research Logistics (NRL)*, *43*(6), 797–820.
64. Veenstra, M., Roodbergen, K.J., Vis, I.F.A., Coelho, L.C. (2017). The pickup and delivery traveling salesman problem with handling costs. *European Journal of Operational Research*, *257*(1), 118–132.
65. Walsh, T. (2001). Permutation problems and channelling constraints. In *Logic for Programming, Artificial Intelligence, and Reasoning* (pp. 377–391): Springer.
66. Wang, J.B., & Wang, J.J. (2013). Single-machine scheduling with precedence constraints and position-dependent processing times. *Applied Mathematical Modelling*, *37*(3), 649–658.

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