A Constraint Programming Approach to Ship Refit Project Scheduling

Raphaël Boudreault
Thales Digital Solutions, Québec, Canada

Vanessa Simard
NQB.ai, Québec, Canada

Daniel Lafond
Thales Digital Solutions, Québec, Canada

Claude-Guy Quimper
Université Laval, Québec, Canada

Abstract

Ship refit projects require ongoing plan management to adapt to arising work and disruptions. Planners must sequence work activities in the best order possible to complete the project in the shortest time or within a defined period while minimizing overtime costs. Activity scheduling must consider milestones, resource availability constraints, and precedence relations. We propose a constraint programming approach for detailed ship refit planning at two granularity levels, daily and hourly schedule. The problem was modeled using the CUMULATIVE global constraint, and the *Solution-Based Phase Saving* heuristic was used to speedup the search, thus achieving the industrialization goals. Based on the evaluation of seven realistic instances over three objectives, the heuristic strategy proved to be significantly faster to find better solutions than using a baseline search strategy. The method was integrated into *Refit Optimizer*, a cloud-based prototype solution that can import projects from Primavera P6 and optimize plans.

2012 ACM Subject Classification Computing methodologies → Planning and scheduling; Theory of computation → Constraint and logic programming

Keywords and phrases Ship refit, planning, project management, constraint programming, scheduling, optimization, resource-constrained project scheduling problem

Digital Object Identifier 10.4230/LIPIcs.CP.2022.10

Supplementary Material *Software (Source Code)*: `https://github.com/raphaelboudreault/chuffed/releases/tag/SBPS`; archived at `swh:1:dir:3eb166e6188719513b62f762559a2420c13c9997`

Funding This project has received financial support from the Scale AI Canadian Innovation Supercluster and from the Mitacs Accelerate program.

Acknowledgements Thanks are due to the many members of the Refit Optimizer project team and our collaborators at Dalhousie University, Polytechnique Montréal, Sōdan, Simwell and Genoa Design International. We are very grateful to the many domain experts consulted and to Seaspan Victoria Shipyards for their invaluable feedback.

1 Introduction

Ship refit planning is a complex and tedious endeavor that requires scheduling several hundred (or thousand) tasks across a time horizon that may span several weeks, months or even over a year [2]. Planners must ensure that precedence relations between tasks are respected, that the required human and material resources are available, and that the scheduled work is completed within the maximum allocated project duration. For instance, dry-dock work periods need to be fixed years in advance, thus leaving no flexibility for increasing project time span. Potential goals of planners in this context are to create schedules minimizing the project total duration or, in case the planning horizon prevents the work to be accomplished in time, minimizing the overtime labor costs. Some planners focus on other needs, such as creating robust schedules leaving flexibility to adjust to unforeseen delays. Indeed, while initial plans must be free of conflicts, unplanned events, delays and their arising work require ongoing re-planning efforts throughout the project. While initial planning may take several weeks for large projects, leaving time for planners to manually attempt optimizing the task scheduling, replanning leaves very little time for planners to consider their options and thus is mainly an opportunistic and reactive process.

There exist multiple enterprise resource planning tools such as Microsoft Project [25], Primavera P6 [33], and IBM Maximo [18], which are typical software solutions to support ships refit planning. Yet beyond the core project management functionalities, the support for optimizing schedules with computational methods from operations research remains limited to resources leveling, i.e. spreading the workload more evenly across the project duration. Some optimization solutions have been previously created for custom projects, yet lack reusability. To our knowledge, the only generic and reusable schedule optimization capability currently available is the Aurora (Stottler Henke) intelligent scheduling solution [35]. While the available information about Aurora's proprietary optimization algorithms is limited, these are described as being based on heuristics, as opposed to exact methods, derived from domain experts. While this satisfying approach is highly relevant and effective for human problem solving given the human brain's bounded computational capabilities, we posit that optimization algorithms can be developed to do better than human-derived heuristics. The current work aims to push further the state-of-the-art in this area by producing a general purpose exact method enhanced with metaheuristics.

Thales Canada has set out to create *Refit Optimizer*, a prototype solution for multi-objective optimization in the ship refit domain, while also designing it to be reusable across a wide variety of other scheduling contexts [22]. The key motivation for this work comes from challenges and innovation opportunities identified in the context of the Arctic/Offshore Patrol Ships and Joint Support Ships in-service support (AJISS) program with the Royal Canadian Navy. Herein, the focus is on detailed planning, using either days or hours as the basic time unit. The Refit Optimizer prototype is currently operational, deployed on a secure cloud platform, and combines several complementary services for importing/exporting project data (from/to Primavera P6), visualizing the schedule using an interactive Gantt chart and tasks list, editing the schedule, freezing scheduled tasks, and optimizing the schedule according to one of three different objectives (*makespan*, *overtime costs* or *robustness*). Additional capabilities include comparing options, analyzing and visualizing geospatial conflicts, forecasting progress, modeling uncertainties and assessing risks using discrete-event simulations (see [22]).

We propose a constraint programming approach for detailed ship refit planning that is currently fully integrated into Refit Optimizer. The problem considered is closely related to the *Resource-Constrained Project Scheduling Problem* (RCPSP) [16, 17, 34] and is modeled using the efficient CUMULATIVE global constraint [1, 38]. We use the *Solution-Based Phase Saving* (SBPS) value selection heuristic [13, 42] to speedup the search and obtain better solutions in a reasonable time. We evaluate our approach on a benchmark formed of seven instances supplied by our industrial partners, namely Sōdan, the AJISS team and Seaspan Victoria Shipyards, which are compared to each other using RCPSP complexity metrics from the literature.

The paper is organized as follows. Section 2 describes the ship refit planning problem. Section 3 presents background notions on scheduling, constraint programming, SBPS and RCPSP complexity measures. The CP model is presented in Section 4, as well as its extensions and the search heuristics developed in our context. Our approach is evaluated on the benchmark instances in Section 5. Finally, Section 6 addresses the applicability of the solution to an industrial setting, followed by a conclusion suggesting directions for future work.

2 Problem Description

Ship refitting is an important shipyard event during which all ship's activities are suspended for improvements. The objective of a refit is to restore, customize, modify, or modernize part of a ship. Most of the time, however, stopping all activities can become costly, which makes efficient ships refit planning important. The time window, or *horizon*, during which a refit takes place can be decided years in advance. When the horizon is exceeded, the dock is no longer available and the ship has to leave. Thus, in order to estimate the required duration, planners have to consider a large number of daily or even hourly tasks depending on multiple capacity-limited resources, both human and material. Furthermore, due to physical limitations, a maximum number of workers must be simultaneously allowed in some work areas. Precedence relationships must be considered between many of the tasks, while some date constraints, such as milestones, must be achieved. Finally, specific tasks must be idle over the weekends. In practice, the initial planned time is often insufficient, in which case overtime for some tasks can be scheduled to fit in the restrictive horizon.

Three main objectives are targeted in this project according to the challenges faced by shipyards. First, to support planners in their horizon estimation, the prototype solution has to offer to minimize the refit total duration, also known in scheduling as **makespan**. This helps the planner at the tactical level by identifying the minimum time needed for a certain ship refit, according to the constraints on tasks. Then, it has to allow the user to produce an operational plan over a fixed horizon that minimizes the **overtime** costs. Since a lot of unplanned delays happen during the actual refit execution, this option helps a shipyard respect their obligation while minimizing costs associated with overtime labor. Finally, the solution has to propose an operational plan taking the **robustness** into account when planning overtime. The idea is to minimize the risk of exceeding the refit deadline by planning the overtime work, as much as possible, at the beginning of the horizon. Thus, if unforeseen events during the execution increase the need for overtime before the end of the refit, it is still possible to proceed. With the option of these three objectives, a user can efficiently estimate a ship refit duration, while being supported during its execution.

Our research is based on the practical needs of our industrial partners which supplied us with realistic use cases. Table 1 presents seven instances of different sizes that were made available for our tests. Each instance is defined by a horizon, given in days or in hours depending on the planning granularity, which represents the available time to complete all tasks. The proposed horizon comes from the initial planners overestimate and can be seen as a baseline for the makespan minimization. The number of tasks, precedence relations, and resources, as well as the task duration range (without overtime), help to better evaluate and compare the size of the instances. The number of tasks that can be performed in overtime (**#O**) is given, where a "*" indicates that those tasks must be idle during weekends. Some tasks do not follow any work hours requirement and thus cannot be shortened nor suspended. A common example of this in a ship refit is paint drying tasks. The number of work areas (**#WA**) is also presented and included as a specific type of resource.

Table 1 Size comparison of the seven instances supplied by our industrial partners.

Instance	Horizon	#Tasks (#O)	Task duration	#Precedence relations	#Resources (#WA)
`day-yacht21`	29 days	21 (20)	1-3 days	32	9 (2)
`hour-yacht21`	704 hours	21 (20)	1-8 hours	32	9 (2)
`generic136`	178 days	136 (136*)	1-20 days	99	9 (4)
`software138`	183 days	138 (138*)	1-10 days	341	8 (0)
`navy253`	728 hours	253 (253*)	1-8 hours	246	92 (87)
`cruise510`	268 days	510 (464*)	1-15 days	550	32 (24)
`navy830`	6200 hours	830 (830*)	1-200 hours	816	146 (128)

The first four instances, `day-yacht21`, `hour-yacht21`, `generic136`, and `software138`, were artificially created by the team for testing purposes. They are used to test the limits of the optimization algorithms, since realistic instances are somewhat simpler because they are manually created by human experts. The instances `day-yacht21` and `hour-yacht21` are two versions of the same ship with different task durations and planning granularities, respectively days and hours. This simple problem has been used in user workshops to compare the result of a manual optimization to the result of our approach. Instance `generic136` does not describe a particular ship refit and was created to test simultaneously various precedence and date constraint types. Instance `software138` describes the management of a software development project, which is used to show the genericity of our approach to scheduling problems with resources. Other instances are anonymized versions of real, or closely inspired by real, refit use cases from recent years. Instances `navy253` and `navy830` are two versions of a real use case provided by the AJISS team, while `cruise510` is inspired by a sample problem provided by Seaspan Victoria Shipyards.

3 Background

3.1 Scheduling

The problem we consider is part of the great family of *scheduling problems*. In the operations research and optimization literature, scheduling problems are many and varied. Given a set of tasks $\mathcal{I}$, these problems require finding when to execute each task over a definite timeline $\mathcal{T} := \{0, 1, \ldots, t_m\}$, where each $t \in \mathcal{T}$ is a discrete time point, so that an objective function is optimized while different constraints are satisfied. Specifically, our scheduling use case is highly related to the *Resource-Constrained Project Scheduling Problem* (RCPSP). Introduced in 1969 by Pritsker et al. [34], its standard definition (see e.g. [16, 17]) supposes first that *preemption* is forbidden, i.e. that each task cannot be interrupted once started. Then, a finite set of resources $\mathcal{R}$ is considered, where each task $i \in \mathcal{I}$ requires an amount $h_{i,r} \in \mathbb{Z}^{\geq 0}$ of resource $r \in \mathcal{R}$ used for its whole duration. Each resource $r \in \mathcal{R}$ has a constant usage capacity $c_r \in \mathbb{Z}^{>0}$ and is fully available at any time (*renewable*). Also, the resources are *cumulative*, i.e. more than one task can use a resource at a time. Thus, the RCPSP assumes at each time point $t \in \mathcal{T}$ that each resource's total usage by tasks does not exceed its capacity. Finally, precedence relationships between some tasks are considered. The objective is to find a schedule with the earliest project ending date or, in other words, the minimal makespan.

Blazewicz et al. [7] have shown that the RCPSP belongs to the strongly NP-hard problems. Thus, its computation complexity and industrial application interest has led to a plethora of techniques, both exact and heuristic, in various research domains. These approaches include notably specialized *branch-and-bound methods* [12, 21, 40], *mixed-integer programming* [11, 20] and, as presented in Section 3.2, *constraint programming*. We refer the reader to Pellerin et al. [30] for a recent survey of current heuristic approaches.

Among the various RCPSP benchmark instance sets in the literature, three are usual: PSPLIB [19], BL [5] and PACK [8]. While these contain from 17 to 120 tasks, our instances listed in Table 1 are significantly larger. Furthermore, the number of resources in these benchmarks is at most 5, while ours can go up to 146. Planning horizons, as well as `navy830` maximal task duration, are also greater than the ones in these benchmarks that go up to 139 and 19 time points respectively, but are comparable to the ones in the more realistic instances of Koné et al. [20].

3.2 Constraint Programming

Constraint Programming (CP) is a powerful programming paradigm to solve combinatorial problems. In particular, it can be used to optimally solve many large-scale optimization problems under constraints [36]. A CP model is formed of *decision variables*, each provided with a finite set of possible values called *domain* and denoted $\text{dom}(X)$ for a variable X. The relationships between the variables are defined by *constraints*, each provided with a specialized inference algorithm. Optimization problems also have an *objective function* to minimize. CP solvers generally perform a tree search to find feasible solutions, where each node of the tree corresponds to a partial solution, and each *branching* is a node created from its parent with an additional assignment of an unfixed variable to a value. The branching selection rules for variables and values are defined using *heuristics*.

Significant efforts have been made in the constraint programming community to efficiently solve scheduling problems involving resource constraints. Introduced by Aggoun and Beldiceanu [1], the CUMULATIVE global constraint enforces the usage of a resource by tasks to be at most its capacity for each time point in the optimization timeline. Formally, for a resource $r \in \mathcal{R}$, given variables S_i and D_i are respectively the starting time and duration of task $i \in \mathcal{I}$, $\textsc{Cumulative}([S_i \mid i \in \mathcal{I}], [D_i \mid i \in \mathcal{I}], [h_{i,r} \mid i \in \mathcal{I}], c_r)$ is logically equivalent to

$$\sum_{\substack{i \in \mathcal{I}: \\ S_i \leq t < S_i + D_i}} h_{i,r} \leq c_r \qquad \forall t \in \mathcal{T}.$$

Over the years, many efficient rules for the CUMULATIVE constraint have been developed to detect failures and filter variables' domains (see e.g. [4, 6, 14, 29, 41]). Furthermore, important progress towards solving large-scale RCPSP instances with CP has been made by Schutt et al. [37, 38] by combining some of these rules with *lazy clause generation*. Introduced by Ohrimenko et al. [28], this technique is a hybrid between CP and *boolean satisfiability* (SAT) solvers. During the search, each filtered value is now recorded with an explanation as a SAT clause. When a failure is detected, the solver uses its explanations to learn a *nogood*, a core reason for what led to this conflict. This nogood is then added as a new constraint to the solver's underlying SAT mechanism. As a result, this process allows avoiding reproducing the same choices later during the search. Furthermore, it enables SAT-based branching heuristics depending on variables' activity in conflicts, notably *Variable State Independent Decaying Sum* (VSIDS) [26]. Several modern and efficient CP solvers, such as Chuffed [9] and OR-Tools [31], are based on the lazy clause generation technique.

3.3 Solution-Based Phase Saving

Large Neighborhood Search (LNS) [32, 39] is a metaheuristic that has been successfully used in many contexts for scaling exact solving methods to large optimization problems. Given an initial solution, the technique iteratively improves the best known solution according to the considered objective. At each iteration, a *neighborhood* is chosen such as a part of the variables are fixed to their value in the current solution, whereas the others are relaxed, which generates a smaller subproblem. Solutions of the latter can then be quickly found by any chosen method.

One of the major drawbacks of relying on LNS to find better solutions for an optimization problem is the loss of exactness from the initial solving method. Thus, a relatively simple, efficient and closely related to LNS value selection heuristic for CP solvers has been introduced by Vion and Piechowiak [42] as *Best-Solution*. Demirović et al. [13] introduced the same heuristic soon after under the name *Solution-Based Phase Saving* (SBPS). We refer in the following to this heuristic as the latter terminology.

Given an optimization problem with variables $X_1, \ldots, X_n$, a variable selection heuristic H_{var} and a value selection heuristic H_{val}, let $b = (b_1, \ldots, b_n)$ denote the current best solution, if one exists, where b_i corresponds to the value of X_i in this solution. If X_k is the variable chosen by H_{var}, the SBPS branching strategy does the following:

a) If b exists and $b_k \in \text{dom}(X_k)$, then choose the value b_k for X_k;

b) Else, choose a value for X_k following H_{val}.

Thus, the strategy focuses the search around the current best solution as much as possible. Combined with a restart strategy and a dynamic variable selection heuristic such as VSIDS, SBPS effectively mimics LNS [13]. Indeed, starting from the root node, the search fixes almost all variables to their current best solution value, and then searches around this solution for a subset of unassigned variables with backtracking, thus implicitly building a neighborhood. The size of the latter is then limited by the restart strategy. Also, the dynamic aspect of the search produces a built-in diversification of neighborhoods, besides that VSIDS tends to select closely related variables. The resulting strategy produced interesting results on a variety of instances [13, 42].

3.4 RCPSP Complexity

In order to properly assess the performance of our approach on the instances in Table 1, it is important to compare them on a similar scale. To do so, Artigues et al. [3] listed a selection of state-of-the-art indicators that characterize the complexity of RCPSP instances. These indicators are typically used to generate instances of a targeted complexity level, but are also relevant to evaluate existing instances. They can be classified in four categories: precedence-oriented, time-oriented, resource-oriented, and hybrid.

The *Order Strength* (OS) is a precedence-oriented indicator showing how much the instance's precedence constraints induce an ordering of the tasks [3, 24]. If P denotes the set of task pairs $\{i, j\}$ $(i, j \in \mathcal{I},\ i \neq j)$ which cannot be executed in parallel due to a chain of precedence constraints between them, OS is defined as the ratio of $|P|$ over the total number of task pairs:

$$\text{OS} := \frac{|P|}{|\mathcal{I}|(|\mathcal{I}| - 1)/2}.$$

We have $\text{OS} \in [0, 1]$. It has been observed that the closer the value is to 1, the more ordered the tasks and the lower the complexity.

The *Resource Factor* (RF) is a resource-oriented indicator which evaluates the resource usage by tasks [3]. It is defined as the ratio of the average number of required resources by task over the number of resources:

$$\mathrm{RF} := \frac{\sum_{i \in \mathcal{I}, r \in \mathcal{R}} u_{i,r}}{|\mathcal{I}||\mathcal{R}|}.$$

where $u_{i,r}$ equals 1 if task $i \in \mathcal{I}$ requires resource $r \in \mathcal{R}$ ($h_{i,r} > 0$), and 0 otherwise. We have $\mathrm{RF} \in [0,1]$. It has been shown that as the RF value increases, the complexity also does.

The *Resource Strength* (RS) indicator combines a time-oriented view with the resource complexity [3, 10]. For each resource $r \in \mathcal{R}$, it considers its maximal usage $c_r^{\max}$ when tasks are scheduled at their earliest while satisfying precedence constraints,

$$c_r^{\max} := \max_{t \in \mathcal{T}} \sum_{i \in \mathcal{I}_t^{\mathrm{ES}}} h_{i,r},$$

where $\mathcal{I}_t^{\mathrm{ES}} \subseteq \mathcal{I}$ is the subset of tasks executed at time point t in this schedule. The resource's strength RS_r is then defined by the ratio of its overall availability over its availability in the earliest schedule:

$$RS_r := \frac{c_r - c_r^{\min}}{c_r^{\max} - c_r^{\min}},$$

where $c_r^{\min} := \max_{i \in \mathcal{I}} h_{i,r}$. In the case where $c_r^{\max} \leq c_r$, RS_r is instead fixed to 1. Thus, every resource $r \in \mathcal{R}$ with $RS_r = 1$ is always sufficiently available and is not a constraint. The RS value is obtained by averaging RS_r over all resources. We have $\mathrm{RS} \in [0,1]$, and the complexity generally increases as RS decreases.

The *Disjunctive Ratio* (DR) indicator is a hybrid between the precedence and resource complexities [3, 5]. The set of task pairs P which cannot be executed in parallel from OS is extended with a set D of pairs that would violate a resource constraint if both tasks overlapped in time, i.e. $D := \{\{i,j\} : \exists r \in \mathcal{R}, h_{i,r} + h_{j,r} > c_r\}$. DR is then defined as the ratio of the number of elements in this new set over the total number of task pairs:

$$\mathrm{DR} := \frac{|P \cup D|}{|\mathcal{I}|(|\mathcal{I}| - 1)/2}.$$

We also have $\mathrm{DR} \in [0,1]$. It has been established that the higher DR is, the more disjunctive the instance is.

4 Methodology

4.1 Main model

In the following, we present the main CP model developed for the ship refit planning problem, as described in Section 2. It is inspired from the classical RCPSP model for CP [38]. Note that in this model, we assume a planning granularity in days. Extensions to support hours and other specific constraints are discussed in Section 4.2.

Reusing the scheduling notation introduced in Section 3.1, we define the input parameters. The horizon $t_m \in \mathbb{Z}^{>0}$ determines the scheduling timeline as the set $\mathcal{T} = \{0, 1, \ldots, t_m\}$. If $\mathcal{I}$ is the given set of tasks to schedule, each task $i \in \mathcal{I}$ is associated with s_i^L, $s_i^U \in \mathcal{T}$ and e_i^L, $e_i^U \in \mathcal{T}$ which are lower (L) and upper (U) bounds on the task starting (s) and ending (e) times as implied by the date constraints. Each task $i \in \mathcal{I}$ also has a processing time

$p_i \in \mathcal{T}$ which corresponds to the task duration in days without overtime. The precedence requirements form the set $\mathcal{P}$ and are encoded as triples (i, j, l), asking for task $i \in \mathcal{I}$ to be completed $l \in \mathcal{T}$ days before task $j \in \mathcal{I} \setminus \{i\}$ starts. Each resource r in the given set of resources $\mathcal{R}$ has a constant capacity $c_r \in \mathbb{Z}^{>0}$, a daily standard usage cost $w_r^S \in \mathbb{R}^{\geq 0}$, and a daily overtime usage cost $w_r^O \in \mathbb{R}^{\geq 0}$, with $w_r^S \leq w_r^O$. The amount of resource $r \in \mathcal{R}$ required by task $i \in \mathcal{I}$ is given by $h_{i,r} \in \mathbb{Z}^{\geq 0}$. A working day is defined by three parameters, $d^S, d^O, d^E \in \{0, 1, \ldots, 23\}$, with $d^S < d^O \leq d^E$, where $[d^S, d^O - 1]$ are standard hours and $[d^O, d^E]$ are overtime hours. Finally, tasks that can be performed in overtime are contained in the set $\mathcal{I}^* \subseteq \mathcal{I}$.

The integer decision variables of our model are as follows. For each task $i \in \mathcal{I}$, S_i is the task starting time, while E_i is the task total elapsed time between its start and its completion. We define $\text{dom}(S_i) = [s_i^L, s_i^U]$ and $\text{dom}(E_i) = \mathcal{T}$, for each $i \in \mathcal{I}$.

The constraints of the model are presented below.

$$\textsc{Cumulative}([S_i \mid i \in \mathcal{I}], [E_i \mid i \in \mathcal{I}], [h_{i,r} \mid i \in \mathcal{I}], c_r) \qquad \forall r \in \mathcal{R} \qquad (1)$$

$$e_i^L \leq S_i + E_i \leq e_i^U \qquad \forall i \in \mathcal{I} \qquad (2)$$

$$S_i + E_i + l \leq S_j \qquad \forall (i, j, l) \in \mathcal{P} \qquad (3)$$

$$\left\lceil \frac{\left(d^O - d^S\right) p_i}{d^E - d^S} \right\rceil \leq E_i \leq p_i \qquad \forall i \in \mathcal{I}^* \qquad (4)$$

$$E_i = p_i \qquad \forall i \in \mathcal{I} \setminus \mathcal{I}^* \qquad (5)$$

The Cumulative constraints (1) ensure that the cumulative usage of each resource by tasks does not overload its capacity at any given point in the timeline. Since $S_i + E_i$ represents the ending time of task $i \in \mathcal{I}$, constraints (2) force the ending time of each task to respect its upper and lower bounds, as implied by the problem date constraints. Constraints (3) impose the precedence requirements from set $\mathcal{P}$. Constraints (4) define the possible values for the elapsed time E_i of task $i \in \mathcal{I}^*$ that can include some overtime work hours. First, the value must be at most p_i since it corresponds to the task duration without overtime. Second, note that performing overtime hours reduces the overall elapsed time in days of a task. The number of standard hours required by task $i \in \mathcal{I}^*$ is given by $\left(d^O - d^S\right) p_i$, which is redistributed over longer days of $d^E - d^S$ hours, fully using the overtime hours each day, thus a lower bound on E_i. For example, if $(d^S, d^O, d^E) = (8, 16, 20)$, a typical day is formed of 8 standard hours and 4 overtime hours. A task $i \in \mathcal{I}^*$ with $p_i = 3$ requires $3 \times 8 = 24$ standard hours, but can be completed in $\lceil \frac{24}{12} \rceil = 2$ days when working 12-hour shifts (8 hours is in overtime). Finally, constraints (5) force tasks that cannot be executed in overtime $(\mathcal{I} \setminus \mathcal{I}^*)$ to have their standard duration.

We considered three different objective functions, where the model can be used with either of them. First, the **makespan** objective, which is the minimization of the schedule duration, is modeled as follows:

$$\min \max_{i \in \mathcal{I}} (S_i + E_i) .$$

This objective is considered without allowing overtime, which is done by assuming $\mathcal{I}^* := \emptyset$.

Then, the **overtime** objective is to minimize the costs associated with overtime work. For a task $i \in \mathcal{I}^*$, the number of standard working days transformed into overtime is given by $p_i - E_i$. Since each overtime day costs $w_r^O - w_r^S$ per resource $r \in \mathcal{R}$ used, the total cost is given by $\sum_{i \in \mathcal{I}^*} \sum_{r \in \mathcal{R}} h_{i,r}(w_r^O - w_r^S)(p_i - E_i)$. Equivalently, we minimize the following linear expression where the inner summation is pre-computed for each $i \in \mathcal{I}^*$:

$$\min \sum_{i \in \mathcal{I}^*} (p_i - E_i) \left(\sum_{r \in \mathcal{R}} h_{i,r}(w_r^O - w_r^S) \right) .$$

Finally, the **robustness** objective is to minimize the risk of exceeding the deadline of a schedule by planning the overtime early in the project. To evaluate this criterion for a task $i \in \mathcal{I}^*$, we multiply its amount of used overtime $p_i - E_i$ by its starting time S_i, leading to the following non-linear function:

$$\min \sum_{i \in \mathcal{I}^*} (p_i - E_i) S_i.$$

4.2 Extensions

The model presented in Section 4.1 has been extended in several ways to better suit the ship refit planning reality. First of all, more types of precedence constraints were considered other than the ones in (3). Indeed, our current model supports any precedence of the form $X_i + l \leq Y_j$, where $X, Y \in \{S, S+E\}$ for $l \in \mathcal{T}$ and $i, j \in \mathcal{I}$ $(i \neq j)$. It was also asked that our model consider that some tasks, but not all, should be suspended on weekends. To this end, additional variables N_i representing the non-working (idle) time points of task $i \in \mathcal{I}$ were considered. In the model, the non-working time is included in the elapsed time of the task. Thus, constraints (4) and (5) are instead applied on the working time $E_i - N_i$. Additional constraints are considered to enforce a value for N_i when the task overlaps at least one weekend. Finally, the model has been extended to support a scheduling granularity in hours. In this case, the overtime constraints (4) and (5) are replaced by additional variables O_i encoding the number of hours in overtime for task $i \in \mathcal{I}^*$. Each of these variables is closely related via special constraints to its associated N_i which also includes the non-working hours during the nights. Thus, in hours, the elapsed times are simply equal to $p_i + N_i$.

4.3 Search Heuristics

Two branching heuristics for the CP model are considered herein as a basis. For the **makespan** objective, we select the starting time variable S_i, $i \in \mathcal{I}$, with the smallest value in its domain, and we assign it to this value. This way, the search focuses as much as possible around the schedule where each task begins at its earliest starting time. For the **overtime** and **robustness** objectives, the branching heuristic selects the task $i \in \mathcal{I}$ that has a starting time variable S_i with the smallest value in its domain. Then, it assigns, in order, the smallest value in $\text{dom}(S_i)$ to S_i and the greatest value in $\text{dom}(E_i)$ to E_i. In the hour granularity case, this last branching is replaced by assigning the smallest value in $\text{dom}(O_i)$ to O_i, and then the smallest value in $\text{dom}(E_i)$ to E_i. In both cases, the intuition is to place the tasks as early as possible, while simultaneously choosing a task duration with as few overtime time points as possible. The resulting heuristic can be formulated with the *priority search* annotation from the MiniZinc modeling language [15, 27], and is supported by Chuffed CP solver [9].

5 Experimentation

The benchmark we considered for our experiments is formed of the seven instances presented in Section 2. In order to compare them on the same basis and assess their complexity, we computed for each instance the indicators OS, RF, RS and DR introduced in Section 3.4. The resulting values are presented in Table 2. In the case of RS, since many resources gave a value $RS_r = 1$ which makes the comparison difficult, we decided to compute RS by averaging RS_r over all resources $r \in \mathcal{R}$ that are in fact restrictive $(RS_r < 1)$. Values in bold font highlight the most complex instances according to each indicator.

Table 2 Complexity indicators of the seven instances.

Instance	OS	RF	RS	DR
`day-yacht21`	0.72	**0.28**	0.40	**0.82**
`hour-yacht21`	0.72	**0.28**	0.40	**0.82**
`generic136`	**0.02**	0.23	0.24	0.06
`software138`	0.27	0.12	**0.01**	0.32
`navy253`	0.19	0.02	0.71	0.18
`cruise510`	0.07	0.06	0.27	0.07
`navy830`	**0.02**	0.01	0.66	0.02

Each instance has its strengths and weaknesses. Looking first at OS, instance `generic136` is one with the least ordered structure. It can be explained by the fact that the instance was created with arbitrary precedence relations as a means of testing. It seems nonetheless that our three realistic use cases, `navy253`, `cruise510` and `navy830`, are also relatively complex according to this indicator. In terms of resource usage (RF) and disjunctive structure (DR), the two smallest instances, `day-yacht21` and `hour-yacht21` are the most complex ones. A big part of that comes from the fact these instances contain a lot less resources, but are more often used at full capacity. Finally, considering the time-oriented view (RS), the typical RCPSP instance `software138` is the most constrained in terms of resources. Although the realistic instances seem to be less affected by their resource constraints, their complexity lies in the large number of tasks to schedule and the lack of artificially induced ordering.

The CP model presented in Section 4 was modeled in the MiniZinc 2.5.5 language [27]. We implemented[1] the SBPS scheme as described in Section 3.3 into the solver Chuffed [9], which we used to solve the instances. The CUMULATIVE constraint was set to apply the optional Time-Table-Edge-Finding (TTEF) checking and filtering rules [37, 41]. The experiments were performed on a MSI GP63 Leopard 8RE machine with an Intel i7-8750H CPU at 2.2 GHz, 6 cores and 16 GB of RAM. Each optimization execution was given a timeout of 4 hours, and a constant restart strategy of 100 failures.

Each instance was solved for each objective, **makespan**, **overtime** and **robustness**. For the last two objectives, we have empirically chosen a “restricted” horizon for each instance corresponding to a reduction between 2% and 30% of the best known makespan. However, due to its specific constrained nature preventing overtime to be performed, `generic136` could not be considered for these objectives.

For each optimization, two search methods were compared. The *BASELINE* strategy consists simply of using the search heuristics defined in Section 4.3. The *SBPS* strategy, on the other hand, also uses these heuristics until a first solution is found. When it is the case, the SBPS branching procedure activates. Furthermore, the variable selection scheme of choosing S_i, $i \in \mathcal{I}$, with the smallest value in $\text{dom}(S_i)$ is replaced by selecting S_i with the greatest conflict activity, as provided by the VSIDS score of Chuffed [9, 26]. The latter modification allows the resulting procedure to effectively reproduce an LNS [13]. We did not directly use the *free search* (`-f`) option of Chuffed, which is alternating between the user-defined heuristic and the VSIDS strategy (on all the variables), since we observed the solving process was generally slowed down by its usage. However, since no solution was found before the timeout without it, we added the free search for instance `software138` when optimizing the **overtime** and the **robustness**.

[1] The code is available at `https://github.com/raphaelboudreault/chuffed/releases/tag/SBPS`. We thank Emir Demirović for providing his original implementation [13].

Table 3 Results on the benchmark instances when considering the **makespan** objective.

Instance	*Baseline*		*SBPS*		Time (s)
	Objective	Time (s)	Objective	Time (s)	improv.
`day-yacht21`	**28 days**	0.2*	**28 days**	0.2*	0.2
`hour-yacht21`	**78 hours**	0.4*	**78 hours**	0.4*	0.4
`generic136`	**178 days**	0.7*	**178 days**	0.7*	0.7
`software138`	144 days	1.4	**119 days**	41.6	1.1
`navy253`	**389 hours**	4.2	**389 hours**	3.7	3.7
`cruise510`	228 days	14.7	**227 days**	785.7	229.3
`navy830`	5216 hours	18.7	**5144 hours**	199.7	18.2

Table 4 Results on the benchmark instances when considering the **overtime** objective.

Instance	*Baseline*		*SBPS*		Time (s)
	Objective	Time (s)	Objective	Time (s)	improv.
`day-yacht21`	**1560**	0.3*	**1560**	0.3*	0.3
`hour-yacht21`	**485**	0.4*	**485**	0.4*	0.4
`software138`	5600	14 359.6	**2600**	153.4	34.3
`navy253`	70	4.2	**66**	5.0	4.0
`cruise510`	26 000	11.7	**15 760**	7555.3	5.8
`navy830`	227	25.2	**36**	276.5	26.6

Table 5 Results on the benchmark instances when considering the **robustness** objective.

Instance	*Baseline*		*SBPS*		Time (s)
	Objective	Time (s)	Objective	Time (s)	improv.
`day-yacht21`	**47**	0.3*	**47**	0.3*	0.3
`hour-yacht21`	**192**	0.4*	**192**	0.4*	0.4
`software138`	900	13 571.8	**258**	320.2	15.3
`navy253`	10 686	5057.6	**3480**	1411.9	6.7
`cruise510`	4870	13 022.5	**842**	1321.7	14.2
`navy830`	146 794	11 208.9	**9076**	13 863.4	41.1

Tables 3, 4, and 5 present respectively the results obtained when considering the **makespan**, **overtime** and **robustness** objectives. In each table, we report the best objective value found (**Objective**) as well as the solving time (**Time**) in seconds. When the timeout was reached, we instead show the required time to find the best solution. A "*" next to a solving time value indicates that the instance was optimally solved. For comparison purposes, we also report the time in seconds taken with *SBPS* to find a solution with an objective value less than or equal to the best one found with *Baseline* (**Time improv.**).

For the **makespan** objective (Table 3), instances `day-yacht21`, `hour-yacht21`, and `generic136` are quickly solved optimally using both strategies. For the other instances, the *Baseline* method finds its best solution in the first 20 seconds of the search, without being able to improve it after. In comparison, the *SBPS* strategy improves the minimal makespan for `software138` by 25 days, `cruise510` by 1 day and `navy830` by 9 days, while instance `navy253` gave the same solution. The use of *SBPS* thus reduced the best makespan by 5% on average. Furthermore, the improved solutions are found in a similar time than *Baseline*, except for `cruise510` where the solution of 227 days is found 15.6 times slower.

For the **overtime** objective (Table 4), the objective value corresponds to the costs induced by overtime work. Since our benchmark is formed of abstract and anonymized realistic instances, the obtained costs are of different sizes and units, thus incomparable in-between instances. Note that instances `day-yacht21` and `hour-yacht21` are still trivially solved optimally with both strategies. Bigger instances see their best objective value from *Baseline* considerably improved with *SBPS*. The best cost is reduced by 48% on average using *SBPS*, while the best solution of *Baseline* is found 94 times faster for `software138`, and 2 times faster for `cruise510`.

For the **robustness** objective (Table 5), the unconventional way of representing it is a challenge for the *Baseline* method. While `day-yacht21` and `hour-yacht21` are still optimally solved, the larger instances need a lot of computation time to settle on a good solution. In fact, for `software138` and `cruise510`, the time to find the best solution is close to the timeout (14 400 seconds). In comparison, the *SBPS* strategy finds a solution with the same robustness value much faster, in no more than 42 seconds. For `cruise510`, the solution is found 917 times faster. Furthermore, the best objective found by *Baseline* is reduced on average by 79%.

6 Discussion

The main goal of this research was to create a prototype solution for multi-objective optimization in the ship refit domain. By successfully proposing plans to the targeted instances supplied by our industrial partners within a reasonable time limit, we have demonstrated the applicability of our constraint programming model. It was important for practical use to obtain a good solution under predetermined time limits: 15 minutes for instances under 100 tasks, one hour for instances between 100 and 500 tasks, and four hours for instances over 500 tasks. In comparison, an expert manually planning smaller instances like `day-yacht21` could take up to four hours. While it was possible to consider the *Baseline* strategy as an attempt to solve the biggest instances, the computation time and the solution's quality were sometimes less than satisfactory for industrial purposes. The use of *SBPS* proved to be a fairly good strategy, leading to improved objective values in a significantly shorter amount of time for the **makespan**, **overtime** and **robustness** objectives. These experiments also demonstrated the Refit Optimizer prototype's relevance for real-world project planning and ongoing project management with re-optimization. Qualitative user feedback from usability tests with domain experts also supports this assessment.

A lot of effort was put in designing Refit Optimizer to be reusable across a wide variety of other scheduling contexts. The terminology and architecture of the product database were made consistent with the terms and structures from the project management field. Plus, the genericity of the constraints formulation allows to consider more than ten different types of precedence and time constraints on tasks. Resources can include workers as well as locations and equipment, which opens to future improvements with geospatial constraints [22]. Preliminary tests on real projects in the naval, avionics, and ground transportation domains also show that the solution has a strong cross-domain potential.

The main reason why we chose Chuffed [9] over other CP solvers was its proven efficiency on large-scale RCPSP instances by combining state-of-the-art CUMULATIVE filtering algorithms with lazy clause generation [37, 38]. Its built-in VSIDS branching heuristic allowed us to easily reproduce the gains obtained in recent SBPS-related work [13, 42]. Furthermore, Chuffed could directly support the *priority search* MiniZinc annotation [15, 27] used to formulate our baseline search heuristics. We did try the OR-Tools CP-SAT solver [31] via its FlatZinc implementation, but preliminary results showed greater computation times to find similar or worse solutions. We did also try a standard LNS procedure prior implementing SBPS. However, we rapidly found that the technique was rather inefficient for the **overtime** and **robustness** objectives, while it was difficult to find a suitable solution deconstruction rule.

There were many challenges in working in an industrial setting. First, the importance of anonymity for the industrial partners made it difficult to analyze some results. For many instances, the estimated workforce costs were changed to abstract values, which produced unrealistic execution costs. Having access to real data would have allowed us to produce more complex realistic instances to challenge our prototype. Explainability of results was also an important challenge. Since the tool needs to be used in an industrial setting by many different people, it is important to document and explain each potential source of incoherent results encountered. Thus, a lot of effort was put on explaining the input data format and importance of each parameter to untrained users in order to avoid as much illogical data as possible. It was also important to focus on results interpretation and solution selection. Furthermore, one recurring issue was that, in a lot of situations, real projects could not be optimized because of an unsatisfiability proof by the algorithm. Without any feedback to the users as to why it is the case, users were at a loss for identifying which constraints to relax or remove. An automated method for identifying causes and potential solutions to help overcome over-constrained problems appears to be an essential requirement for the successful use of the prototype in the field. We did use FindMUS [23] from the MiniZinc tool suite to help us identify the data inconsistencies. However, its usage required complete knowledge of the optimization model, thus was not a viable option for end users.

A comparison of the complexity of our seven instances to the complexity of PSPLIB [19], BL [5] and PACK [8] benchmarks, as evaluated by Artigues et al. [3], shows the difference between real-life and theoretical applications. Our set of realistic instances is more complex on average when looking at OS (precedence) and RS (resources over time), although the difference is small. This can be explained by the significant greater size of our instances. However, the literature benchmarks are widely more complex in terms of RF (resource usage) and DR (resources and precedence). That can be explained by the changes made by experts so the instances could be manually planned. Manually defined resources are also less restrictive than in computer generated instances.

It would have been interesting to test our CP approach on the three literature benchmarks, as well as the extended ones from Koné et al. [20]. We established that our set of seven instances, although closer to the ship refit reality, were less complex than the computer generated instances, thus more extensive tests would be necessary to complete the constraint programming prototype. Although our industrial partners may not be subject to high levels of precedence-resource complexity, it is important to be aware of the prototype's limits in the hopes of further improving the solution. Being able to consider more complex problems could also become a strategic advantage for shipyards, allowing them to include more constraints normally not considered with manual plans.

7 Conclusion

In this paper, we introduced a CP approach to the ship refit planning problem. Our prototype solution was successfully tested on seven realistic instances supplied by our industrial partners with varying levels of complexity, which demonstrated the CP applicability for this problem. The proposed CP model is highly related to the classical RCPSP model, while multiple extensions are considered to address problem-specific constraints and objectives. As a means to speedup the search of better solutions, we proposed to use the SBPS value selection heuristic. Its usage improved on average the objective value by 5%, 48% and 79% when minimizing respectively the makespan, the overtime costs and the robustness, as better solutions are found significantly faster than with our baseline heuristics.

Directions for future work include the integration of an optimization algorithm based on mixed-integer programming, and extending algorithms by considering task priority levels for scope optimization, i.e. when work requirements surpass the capacity. We also aim to further explore the use of probabilistic discrete-event simulations for robustness assessment, and the use of geospatial modeling and visualization to improve planning as well as users understanding.

References

1 Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and Computer Modelling*, 17(7):57–73, April 1993. doi:10.1016/0895-7177(93)90068-A.

2 Rashpal Ahluwalia and Denis Pinha. Decision support system for production planning in the ship repair industry. *Industrial and Systems Engineering Review*, 2(1):52–61, July 2014.

3 Christian Artigues, Oumar Koné, Pierre Lopez, Marcel Mongeau, Emmanuel Néron, and David Rivreau. Benchmark instance indicators and computational comparison of methods. In *Resource-Constrained Project Scheduling*, pages 107–135. John Wiley & Sons, Ltd, 2008.

4 Philippe Baptiste, Claude Le Pape, and Wim Nuitjen. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. International Series in Operations Research & Management Science. Springer, Boston, MA, first edition, 2001.

5 Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1):119–139, January 2000. doi:10.1023/A:1009822502231.

6 Nicolas Beldiceanu and Mats Carlsson. A new multi-resource cumulatives constraint with negative heights. In Pascal Van Hentenryck, editor, *Principles and Practice of Constraint Programming - CP 2002*, Lecture Notes in Computer Science, pages 63–79, Berlin, Heidelberg, 2002. Springer. doi:10.1007/3-540-46135-3_5.

7 J. Blazewicz, J. K. Lenstra, and A. H. G. Rinnooy Kan. Scheduling subject to resource constraints: Classification and complexity. *Discrete Applied Mathematics*, 5(1):11–24, January 1983. doi:10.1016/0166-218X(83)90012-4.

8 Jacques Carlier and Emmanuel Néron. On linear lower bounds for the resource constrained project scheduling problem. *European Journal of Operational Research*, 149:314–324, September 2003. doi:10.1016/S0377-2217(02)00763-4.

9 Geoffrey G. Chu. *Improving Combinatorial Optimization*. PhD thesis, The University of Melbourne, 2011. GitHub: https://github.com/chuffed/chuffed.

10 Bert De Reyck and Willy Herroelen. On the use of the complexity index as a measure of complexity in activity networks. *European Journal of Operational Research*, 91(2):347–366, June 1996. doi:10.1016/0377-2217(94)00344-0.

11 Sophie Demassey. Mathematical programming formulations and lower bounds. In *Resource-Constrained Project Scheduling: Models, Algorithms, Extensions and Applications*, pages 49–62. John Wiley & Sons, Ltd, 2008.

12 Erik L. Demeulemeester and Willy S. Herroelen. New benchmark results for the resource-constrained project scheduling problem. *Management Science*, 43(11):1485–1492, November 1997. `doi:10.1287/mnsc.43.11.1485`.

13 Emir Demirović, Geoffrey Chu, and Peter J. Stuckey. Solution-Based Phase Saving for CP: A Value-Selection Heuristic to Simulate Local Search Behavior in Complete Solvers. In John Hooker, editor, *Principles and Practice of Constraint Programming*, Lecture Notes in Computer Science, pages 99–108, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-319-98334-9_7`.

14 Hamed Fahimi, Yanick Ouellet, and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint and a quadratic filtering algorithm for the cumulative not-first not-last. *Constraints*, 23(3):272–293, July 2018. `doi:10.1007/s10601-018-9282-9`.

15 Thibaut Feydy, Adrian Goldwaser, Andreas Schutt, Peter J Stuckey, and Kenneth D Young. Priority Search with MiniZinc. In *ModRef 2017: The Sixteenth International Workshop on Constraint Modelling and Reformulation*, 2017.

16 Sönke Hartmann and Dirk Briskorn. An updated survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 297(1):1–14, February 2022. `doi:10.1016/j.ejor.2021.05.004`.

17 Willy Herroelen, Bert De Reyck, and Erik Demeulemeester. Resource-constrained project scheduling: A survey of recent developments. *Computers & Operations Research*, 25(4):279–302, April 1998. `doi:10.1016/S0305-0548(97)00055-5`.

18 IBM Maximo Application Suite. IBM, 2021. Website: `https://www.ibm.com/ca-en/products/maximo`.

19 Rainer Kolisch and Arno Sprecher. PSPLIB - A project scheduling problem library. *European Journal of Operational Research*, 96(1):205–216, January 1997. `doi:10.1016/S0377-2217(96)00170-1`.

20 Oumar Koné, Christian Artigues, Pierre Lopez, and Marcel Mongeau. Event-based MILP models for resource-constrained project scheduling problems. *Computers & Operations Research*, 38(1):3–13, January 2011. `doi:10.1016/j.cor.2009.12.011`.

21 Philippe Laborie. Complete MCS-based search: Application to resource constrained project scheduling. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 181–186, San Francisco, CA, USA, July 2005. Morgan Kaufmann Publishers Inc.

22 Daniel Lafond, Dave Couture, Justin Delaney, Jessica Cahill, Colin Corbett, and Gaston Lamontagne. Multi-objective schedule optimization for ship refit projects: Toward geospatial constraints management. In Tareq Ahram, Redha Taiar, and Fabienne Groff, editors, *Human Interaction, Emerging Technologies and Future Applications IV*, Advances in Intelligent Systems and Computing, pages 662–669, Cham, 2021. Springer International Publishing. `doi:10.1007/978-3-030-74009-2_84`.

23 Kevin Leo and Guido Tack. Debugging unsatisfiable constraint models. In Domenico Salvagnin and Michele Lombardi, editors, *Integration of AI and OR Techniques in Constraint Programming*, Lecture Notes in Computer Science, pages 77–93, Cham, 2017. Springer International Publishing. GitLab: `https://gitlab.com/minizinc/FindMUS`. `doi:10.1007/978-3-319-59776-8_7`.

24 Anthony A. Mastor. An experimental investigation and comparative evaluation of production line balancing techniques. *Management Science*, 16(11):728–746, July 1970. `doi:10.1287/mnsc.16.11.728`.

25 Microsoft Project. Microsoft, 2019. Website: `https://www.microsoft.com/en-ca/microsoft-365/project/project-management-software`.

26 M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference*, pages 530–535, June 2001. `doi:10.1145/378239.379017`.

27 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. MiniZinc: Towards a standard CP modelling language. In Christian Bessière, editor, *Principles and Practice of Constraint Programming – CP 2007*, Lecture Notes in Computer Science, pages 529–543, Berlin, Heidelberg, 2007. Springer. Website: `https://www.minizinc.org/`. `doi:10.1007/978-3-540-74970-7_38`.

28 Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, September 2009. `doi:10.1007/s10601-008-9064-x`.

29 Yanick Ouellet and Claude-Guy Quimper. A $O(n \log^2 n)$ checker and $O(n^2 \log n)$ filtering algorithm for the energetic reasoning. In Willem-Jan van Hoeve, editor, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Lecture Notes in Computer Science, pages 477–494, Cham, 2018. Springer International Publishing. `doi:10.1007/978-3-319-93031-2_34`.

30 Robert Pellerin, Nathalie Perrier, and François Berthaut. A survey of hybrid metaheuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research*, 280(2):395–416, January 2020. `doi:10.1016/j.ejor.2019.01.063`.

31 Laurent Perron and Vincent Furnon. OR-Tools. Google, 2022. Website: `https://developers.google.com/optimization/`.

32 David Pisinger and Stefan Ropke. Large Neighborhood Search. In Michel Gendreau and Jean-Yves Potvin, editors, *Handbook of Metaheuristics*, International Series in Operations Research & Management Science, pages 399–419. Springer US, Boston, MA, 2010. `doi:10.1007/978-1-4419-1665-5_13`.

33 Primavera P6 Enterprise Project Portfolio Management (P6 EPPM). Oracle, 2022. Website: `https://docs.oracle.com/en/industries/construction-engineering/primavera-p6-project/index.html`.

34 A. Alan B. Pritsker, Lawrence J. Waiters, and Philip M. Wolfe. Multiproject scheduling with limited resources: A zero-one programming approach. *Management Science*, September 1969. `doi:10.1287/mnsc.16.1.93`.

35 Robert Richards and Richard Stottler. Complex project scheduling lessons learned from NASA, boeing, general dynamics and others. In *2019 IEEE Aerospace Conference*, pages 1–9, March 2019. `doi:10.1109/AERO.2019.8741996`.

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

37 Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In Carla Gomes and Meinolf Sellmann, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 234–250, Berlin, Heidelberg, 2013. Springer. `doi:10.1007/978-3-642-38171-3_16`.

38 Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. Explaining the cumulative propagator. *Constraints*, 16(3):250–282, July 2011. `doi:10.1007/s10601-010-9103-2`.

39 Paul Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. In Michael Maher and Jean-Francois Puget, editors, *Principles and Practice of Constraint Programming — CP98*, Lecture Notes in Computer Science, pages 417–431, Berlin, Heidelberg, 1998. Springer. `doi:10.1007/3-540-49481-2_30`.

40 Arno Sprecher. Scheduling resource-constrained projects competitively at modest memory requirements. *Management Science*, 46(5):710–723, 2000.

41 Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In Tobias Achterberg and J. Christopher Beck, editors, *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Lecture Notes in Computer Science, pages 230–245, Berlin, Heidelberg, 2011. Springer. `doi:10.1007/978-3-642-21311-3_22`.

42 Julien Vion and Sylvain Piechowiak. Une simple heuristique pour rapprocher DFS et LNS pour les COP. In *Actes des 13e Journées Francophones de la Programmation par Contraintes, JFPC 2017*, pages 38–45, Montreuil sur Mer, France, June 2017.