Using Approximation within Constraint Programming to Solve the Parallel Machine Scheduling Problem with Additional Unit Resources

Arthur Godet,[1] **Xavier Lorca,**[2] **Emmanuel Hebrard,**[3] **Gilles Simonin**[1]
[1]IMT Atlantique, LS2N - UMR CNRS 6004, 44307, Nantes, France
[2]Centre Génie Industriel, IMT Mines Albi, France
[3]CNRS, LAAS-CNRS, Université de Toulouse
{arthur.godet, gilles.simonin}@imt-atlantique.fr, xavier.lorca@mines-albi.fr, hebrard@laas.fr

Abstract

In this paper, we consider the Parallel Machine Scheduling Problem with Additional Unit Resources, which consists in scheduling a set of n jobs on m parallel unrelated machines and subject to exactly one of r unit resources. This problem arises from the download of acquisitions from satellites to ground stations. We first introduce two baseline constraint models for this problem. Then, we build on an approximation algorithm for this problem, and we discuss about the efficiency of designing an improved constraint model based on these approximation results. In particular, we introduce new constraints that restrict search to executions of the approximation algorithm. Finally, we report experimental data demonstrating that this model significantly outperforms the two reference models.

Introduction

Scheduling problems have long been studied for their importance and relevance to real-life applications, especially in but not limited to manufacturing. In the Parallel Machines Scheduling problem, as many jobs as machines can be processed simultaneously. There are several variants of this problem (Allahverdi 2015). Among them, those involving additional resources are of particular interest and their complexity and approximability was the focus of significant research (Blazewicz, Lenstra, and Kan 1983).

The Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR) is a scheduling problem with n jobs to be processed on m parallel machines and each job is subject to exactly one of r disjunctive resources. This problem is a generalisation of $P||C_{max}$ which is strongly NP-hard (Blazewicz, Lenstra, and Kan 1983; Garey and Johnson 1978). A survey by Edis, Oguz, and Ozkarahan reports known approximation bounds for several variants of parallel machines scheduling problem, including a generalization of PMSPAUR (2013). However, the best approximation algorithm for this problem was proposed more recently in (Hebrard et al. 2016). This problem has several real-life applications. One of them is the problem of planning the download of acquisitions made by agile observation satellites (Pralet et al. 2014; Hebrard et al. 2016). These satellites rotate around the Earth and make acquisitions of data that are saved in memory banks inside the satellites. A memory bank can be seen as a hard drive, and so only one picture can be read at a time from a memory bank. When making an acquisition, the satellite uses r different optic frequencies, and so generates r different files. There are exactly r memory banks, storing the different files. The pictures can be transmitted to ground stations only at certain periods of time, called *download windows*, because of the rotation around the Earth. Moreover, when the satellite is well positioned and so can transfer data to ground stations, m communication channels are open, but only files from distinct memory banks can be transferred in parallel. The objective is to minimise the total completion time, called *makespan*. Indeed, the objective is to know whether it is possible to download all the files during a given download window or not. This problem is generally part of a bigger one, as described by Pralet et al. (2014).

The main contribution of this paper is to show how insights from approximation algorithms for PMSPAUR (Hebrard et al. 2016) can be leveraged to design an exact constraint programming (CP) model for this problem. In particular, decision variables stand for potential executions of a greedy approximation algorithm, which are in turn channeled to a more conventional constraint model for scheduling problems. Moreover, although the goal is to emulate the approximation algorithm, this is achieved declaratively, by adding dedicated constraints and dominance relations to the model. Experimental results clearly show that our method outperforms other state-of-the-art CP approaches.

The rest of the paper is organised as follows. The next section precisely describes the studied problem and recalls the necessary background. The following section gives a baseline CP model of PMSPAUR. Then, we show how to use approximation results into the CP model, and we demonstrate that finding an optimal schedule can be achieved by finding the right sequence in which processing the jobs through a specific algorithm. The following section details the CP model built on these results. And finally, the last section, we discuss the results of the three models on our instances.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Background and Notation

In the remainder of the paper, we write $[i, j]$ for the set of integers $\{i, i+1, \ldots, j\}$ and $[i]$ as a shortcut for $[1, i]$.

Approximation

Considering a $\mathcal{NP}$-hard minimisation problem $\mathcal{P}$, an *approximation algorithm* is a polytime/space algorithm that offers a guarantee on the retrieved solution quality. Basically, it ensures that the computed result is never larger than a certain value times a trivial lower bound, defining $\rho > 1$ the *ratio* of the approximation algorithm. Note the same algorithm can have several distinct ratios on different parameters. An example of this is given by Lemmas 2 and 3.

The Parallel Machines Scheduling Problem with Additional Unit Resources

We consider a set $\mathcal{T} = \{1, \ldots, n\}$ of jobs of fixed *duration* p_i ($i \in \mathcal{T}$) to be processed on m *parallel machines*. Moreover, every *job* requires exactly one among r *resources* $R_1, \ldots, R_r$. We denote $res(i)$ the resource of the job i. To simplify the notations we write R_j for the set of jobs requiring the resource R_j, *i.e.*, $R_j \equiv \{i \mid res(i) = R_j\}$.

A *schedule* is a mapping $\sigma : \mathcal{T} \mapsto \mathbb{N}$ from jobs to starting times. Let s_i be the starting time of job i, obviously we have $s_i = \sigma(i)$. We denote $e_i = s_i + p_i$ the completion time of job i, and σ is said feasible if and only if at any time $t \in \mathbb{N}$:

$$|\{i \mid s_i \leq t \leq e_i\}| \leq m \tag{1}$$

$$|\{i \mid i \in R_j, s_i \leq t \leq e_i\}| \leq 1 \quad \forall j \in [r] \tag{2}$$

Equation (1) ensures that no more than m jobs can be simultaneously processed and equation (2) that no two different jobs requiring the same resource can be simultaneously processed. We consider the problem of finding a feasible schedule σ with minimum *makespan* $C_{max} = \max\{e_i \mid i \in \mathcal{T}\} - \min\{s_i \mid i \in \mathcal{T}\}$. Given a set of jobs $\mathcal{T}$ and a schedule σ, we denote[1] $e^{\mathcal{T}}_{min}$ the earliest idle time of any parallel machine, $e^{\mathcal{T}}_{max}$ the earliest time at which all parallel machines are idle, and $e^{\mathcal{T}}_{R_j}$ the latest usage time of resource R_j:

$$\begin{aligned} e^{\mathcal{T}}_{min} &= \min\{t \mid t \in \mathbb{N} \wedge |\{i \mid s_i \leq t \leq e_i\}| < m\} \\ e^{\mathcal{T}}_{max} &= \min\{t \mid t \in \mathbb{N} \wedge |\{i \mid s_i \leq t \leq e_i\}| = 0\} \\ e^{\mathcal{T}}_{R_j} &= \max\{e_i \mid i \in R_j\} \end{aligned}$$

Finally, we denote $L_{\mathcal{T}} = \sum_{i \in \mathcal{T}} p_i$ the sum of the processing times of all the jobs, and $L_{\mathcal{T}}(R_j)$ the sum of the processing times of the jobs in $\mathcal{T}$ requiring resource j:

$$L_{\mathcal{T}}(R_j) = \sum_{i \in R_j} p_i$$

Existing Constraint programming approach

To solve scheduling problems in Constraint Programming, it is common to use *task variables* to model the problem. A *task variable* generally has an integer variable to represent the starting time of the task, as well as an integer variable for the ending time. The processing time of the task is either an integer variable or an integer constant. A task variable also has a height, which can either be an integer variable or an integer constant, and corresponding to the consumption of resources of the task. These task variables are used within the CUMULATIVE (Aggoun and Beldiceanu 1992) constraints. For DISJUNCTIVE constraints, the task variables have necessarily a height of 1. In the following, we use DISJUNCTIVE constraints as described in (Fahimi, Ouellet, and Quimper 2018). For CUMULATIVE constraints, we use the *edge-finding* algorithm from (Vilím 2009), the *time-tabling* algorithm from (Ouellet and Quimper 2013) and the *not-first/not-last* and *overload-checking* algorithms from (Fahimi, Ouellet, and Quimper 2018).

For all $i \in [n]$, let T_i be a task variable, which starts at s_i lasts p_i and ends at e_i. The variables $\{T_i, i \in [n]\}$ are all of height 1. For all $i \in [n]$, the domain of s_i is $[0, L_{\mathcal{T}} - p_i]$, and since the duration p_i is constant, e_i is functionally dependent on s_i. Moreover, we have an integer variable C_{max} with domain $\left[\max\left(\frac{L_{\mathcal{T}}}{m}, \max_j L_{\mathcal{T}}(R_j)\right), L_{\mathcal{T}}\right]$ standing for the time needed to execute a schedule, to be minimised. Then, the constraints are:

$$\text{DISJUNCTIVE}(\{T_i, i \in R_j\}), \quad \forall j \in [r] \tag{3}$$

$$\text{CUMULATIVE}(\{T_i, i \in [n]\}, m) \tag{4}$$

$$C_{max} = \max_{i \in [n]}(e_i) \tag{5}$$

The DISJUNCTIVE constraint (3) ensures that no more than one job from a resource can be processed at any time. The CUMULATIVE constraint (4) ensures that no more than m jobs can be processed concurrently, that is that we do not exceed the machines capacity. Finally, the MAX constraint (5) is here to link the makespan C_{max} with the starting variables of the jobs. Notice that although this model does not specify on which machine each job is processed, given a solution, it is possible to compute a valid assignment of jobs to machines in polynomial time.

To explore the search space, a search heuristic is applied to select variables and values on which branching along a depth-first exploration strategy of a binary tree. In CP, there exist generic search heuristics as domOverWDeg (Boussemart et al. 2004). In scheduling problems, it is natural to try to schedule tasks as early as possible. The search called *smallest* represents this idea. It selects the variable s_i with the smallest lower bound among the uninstantiated ones and branches by instantiating this variable to its lower bound. On the other hand, Godard, Laborie, and Nuijten present in 2005 a search heuristic called *SetTimes* (Godard, Laborie, and Nuijten 2005) which consists in selecting, as for *smallest*, the variable s_i with the smallest lower bound among the uninstantiated ones, but, when backtracking, *SetTimes* would not try to branch on this variable again until the other constraints, that are CUMULATIVE and DISJUNCTIVE constraints here, increase the variable's lower bound.

[1] In the following, the schedule σ is always clear from the context, so we do not include it in the subscript.

Using approximation results

In this section, we recall some results from (Hebrard et al. 2016) and extend them in order to be used in a CP solver. Algorithm 1 shows the basic procedure `EnQueue`. It simply

Algorithm 1: `EnQueue`

Data: `schedule` : σ, `job` : i
Result: Schedule σ with job i enqueued
$\sigma(i) \leftarrow \max(e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{res(i)})$
return σ

inserts the job i given as argument at the "back" of the schedule, at the earliest possible time. More precisely, it schedules it at time t equals to the maximum between the earliest idle time of any machine $e^{\mathcal{T}}_{min}$ and the maximum usage time $e^{\mathcal{T}}_{res(i)}$ of $res(i)$. Applying the procedure on a sequence of the jobs gives a feasible schedule, as stated by Lemma 2 in (Hebrard et al. 2016). Moreover, we get from Corollary 1 of the same paper that calling `EnQueue` on any sequence of jobs is a $(2 - \frac{1}{m})$-approximation algorithm, for m parallel machines.

In the remainder of this section, we prove that there exists a sequence of operations $\sigma \leftarrow \texttt{EnQueue}(\sigma, i)$ that yields an optimal schedule, and hence can be the basis for a complete algorithm. Moreover, we show how to cut branches in the corresponding search tree.

Getting an optimal schedule via a sequence of `EnQueue` operations

Definition 1. *A schedule is* left-shifted *if no job can possibly be processed earlier without violating a resource constraint.*

Definition 2. *A schedule is* persistent *if every pair of jobs sharing a resource and immediately consecutive are processed on the same machine.*

Definition 3. *A schedule is* dense *if there is no $t_1 < t_2$ such that a machine is idle during $[t_1, t_2[$ and in process at t_2.*

Lemma 1. *There exists a left-shifted persistent dense optimal schedule.*

Proof. The fact that there exists an optimal left-shifted schedule is trivial.

Consider a left-shifted optimal schedule with two jobs i_1 and i_2 requiring the same resource such that $e_{i_1} = s_{i_2}$ but processed on two distinct machines, M_x and M_y, respectively. Then we can reassign i_2 and all the trailing jobs on M_y to M_x, and all the jobs subsequent to i_1 on M_x to M_y. Clearly, this operation changes no start time and cannot violate a resource constraint if it did not before the operation. The solution obtained is therefore equivalent and the operation can be repeated until there is no such occurrence. Therefore, there exists a left-shifted persistent optimal schedule.

Now, suppose that, in such a left-shifted persistent schedule, a machine M is idle in an interval $[t_1, t_2[$ and in process at time t_2. Since the job starting at time t_2 on this machine is left-shifted, then there must exist a job ending at time t_2 on some other machine. However, this would contradict the fact that this schedule is persistent. □

Theorem 1. *There exists a sequence of operations $\sigma \leftarrow \texttt{EnQueue}(\sigma, i)$ that lead to an optimal schedule.*

Proof. Let σ be a left-shifted persistent dense optimal schedule, and order the jobs by their start times in σ in increasing order, and let rename them accordingly.

We show by induction on the rank of the jobs in the ordering, that the corresponding sequence of calls to `EnQueue` produces σ (up to machine symmetries). This is trivially true for a single job. Suppose this is true until job i, and call σ_i the schedule restricted to jobs 1 to i.

If the start time of $i + 1$ is the earliest idle time t of any machine in σ_i, then there is no job requiring $res(i + 1)$ in process after t. If there was such a job, since its start time is less than or equal to t, $i + 1$ could not start at t in σ. Therefore, `EnQueue` inserts that job on a machine with earliest idle time in σ_i yielding the same start time as in σ.

If the start time of $i+1$ is not the earliest idle time t of any machine in σ_i, then, since σ is dense, no job $i' > i$ may be processed on the first-ending machine in σ. Therefore, since the schedule is left-shifted, all jobs subsequent to i must require one the resources used at time t (of which there are at most $m - 1$). Since σ is persistent it has only one possible configuration: all the jobs of each resource are processed by a single machine. Since `EnQueue` does insert job $i + 1$ following the previous job requiring the same resource, job $i + 1$ will have the same starting time as in σ.

Therefore, there exists a sequence of calls to `EnQueue` that produces an optimal schedule. □

The search tree that explores the permutations of jobs to `EnQueue` is thus guaranteed to contain an optimal solution.

Remark 1. *Moreover, from the proof of Theorem 1, we can observe that it is sufficient to explore only the orderings consistent with the chronological order in the resulting schedule. Notice that this might not be the case when the resource required by the next job i is in use at time t and there is another job i' with $i' > i$ that might be inserted to start at time t. In that case, inserting first i and then i' or the reverse yields the same schedule.*

Algorithm 2 explores only *chronological-compatible* orderings of `EnQueue` operations. At each choice point, it computes a set $\mathcal{I}$ of jobs that can possibly start at $e^{\mathcal{T}}_{min}$. If this set is not empty, it is sufficient to branch on the possible permutations of `EnQueue` operations restricted to these jobs, by Remark 1. Otherwise, by the same argument, it is sufficient to branch on the jobs whose resource finishes first.

Additional rules to cut branches in the search tree

In the subsequent paragraphs, we propose rules to reduce the size of the search space by cutting branches that are dominated. We assume that jobs are indexed by their order on the branch we consider. Therefore, when inserting i, previous jobs are already allocated to machines and have a fixed start time, whereas following jobs are not scheduled yet.

Let recall another algorithm presented in (Hebrard et al. 2016) that will be of use here: `MaxLoad` (see Algorithm 3).

Algorithm 2: `Search`

Data: $\texttt{jobs} : \mathcal{T}, \texttt{schedule} : \sigma, \texttt{int} : ub$
Result: Minimum makespan of a completion of σ to $\mathcal{T}$
$C_{max} \leftarrow \max\{e^{\mathcal{T}}_{R_j} \mid 1 \leq j \leq r\}$
if $ub \leq C_{max}$ **then return** ub
if $\mathcal{T} = \emptyset$ **then return** C_{max}
$\mathcal{I} \leftarrow \{i \in \mathcal{T}, e^{\mathcal{T}}_{res(i)} \leq e^{\mathcal{T}}_{min}\}$
if $\mathcal{I} \neq \emptyset$ **then**
  return $\min_{i \in \mathcal{I}}\{$`Search` $(\mathcal{T} \setminus \{i\},$ `EnQueue` $(\sigma, i)\ )\}$
else
  $\mathcal{J} \leftarrow (\arg\min_j e^{\mathcal{T}}_{R_j} \cap \mathcal{T})$
  return $\min_{i \in \mathcal{J}}\{$`Search` $(\mathcal{T} \setminus \{i\},$ `EnQueue` $(\sigma, i)\ )\}$

Algorithm 3: `MaxLoad`

Data: $(\texttt{jobs} : \mathcal{T})$
Result: A schedule with an approximation guarantee
$\sigma \leftarrow \emptyset$
while $\mathcal{T} \neq \emptyset$ **do**
  let R_j be a resource with maximum load $L_{\mathcal{T}}(R_j)$
  pick any job $i \in \mathcal{T}$ such that $i \in R_j$
  $\mathcal{T} \leftarrow \mathcal{T} \setminus \{i\}$
  $\sigma \leftarrow \texttt{EnQueue}(\sigma, i)$
return σ

Theorem 2. `MaxLoad` *finds the optimal completion of any sequence of jobs if there are no more than* m *resources required by the remaining jobs.*

Proof. First, assume that the set of resources in use in the interval $[e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{max}[$ is disjoint from the resources required by the remaining jobs. In this case we can view the problem as a generalisation where each parallel machine M_z has a release date e_{M_z}.

We prove that MAXLOAD is optimal in this case by induction on the number of machines m. For $m = 1$ this is trivial. Now suppose that this is true for m machines and consider the case for $m + 1$ machines. MAXLOAD picks the resource R_j that maximises $L_{\mathcal{T}}(R_j)$ and will eventually process every job requiring R_j on the machine M_x with earliest release time. If one of these jobs is the last completed job, then the solution is optimal, so we assume that the last completed job is on another machine M_y and requires $R_{j'}$. The other choices made by the algorithm are completely independent of those on machine M_x and therefore we know that the other tasks are optimally scheduled on the m other machines. In particular, there is no resource except for R_j whose jobs we can swap for jobs requiring $R_{j'}$ in order to improve the schedule. However swapping the jobs of R_j with those of $R_{j'}$ does not help either since $L_{\mathcal{T}}(R_j) \geq L_{\mathcal{T}}(R_{j'})$ and the release date of M_y is larger than or equal to that of M_x.

Now, suppose that there exist resources in use during $[e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{max}[$ required by at least one remaining job. We can transform the instance as follows: for every machine M_z which is idle at time e_{M_z}, there is a unique resource in use in the interval $[e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{max}[$ (Lemma 2 in (Hebrard et al. 2016)). If this resource is required by some remaining job, we create a job of length $e_{M_z} - e^{\mathcal{T}}_{min}$ requiring that resource and we set the release time of this machine to $e^{\mathcal{T}}_{min}$. Otherwise, we simply set the release time of this machine to e_{M_z}. This new instance corresponds to the previous case and is thus optimally solved by MAXLOAD. Moreover, observe that the completion is such that every job requiring a given resource is processed by the same machine, and therefore the permutation of jobs on each machine does not matter. Therefore, this solution is also optimal for the original case. □

Let $L_{\mathcal{T}}(\overline{R_j})$ denote the sum of the processing times of the jobs in $\mathcal{T}$ that do not require resource R_j.

Lemma 2. *Let* j *be such that* $L_{\mathcal{T}}(R_j)$ *is maximum. If* $L_{\mathcal{T}}(R_j) \geq 2L_{\mathcal{T}}(\overline{R_j})/m$ *then the minimum makespan of any schedule of* $\mathcal{T}$ *is* $L_{\mathcal{T}}(R_j)$.

Proof. Trivially, $L_{\mathcal{T}}(R_j)$ is a lower bound, therefore we only need to prove that the condition of the lemma entails that there is a solution with that makespan.

By Theorem 3 in (Hebrard et al. 2016) MAXLOAD is a $(2 - \frac{2}{m+1})$-approximation algorithm. Moreover, the proof of this theorem shows that for any instance with set of jobs $\mathcal{T}$, either:

1. one resource is in use at all times in the optimal schedule;
2. or the makespan σ found by MAXLOAD is at most $\left(2 - \frac{2}{m+1}\right) \frac{\sum_{i \in \mathcal{T}} p_i}{m}$.

Now suppose that the condition of the lemma holds and consider the schedule where every job requiring resource R_j is processed on the same machine, and all the remaining jobs on the $m - 1$ other machines. The makespan of this solution is thus the maximum between $L_{\mathcal{T}}(R_j)$ and the makespan C_{max} of the optimal solution of the sub-instance containing every job in $\mathcal{T}$ that do not require R_j and only $m - 1$ parallel machines. By Theorem 3 in (Hebrard et al. 2016), the optimal makespan C_{max} for this instance is either $L_{\mathcal{T}}(R_{j'})$ for some j', or at most $(2 - \frac{2}{m}) \frac{L_{\mathcal{T}}(\overline{R_j})}{m-1} = 2L_{\mathcal{T}}(\overline{R_j})/m$. Since $L_{\mathcal{T}}(R_j) \geq L_{\mathcal{T}}(R_{j'})$, we have in both cases $\max(L_{\mathcal{T}}(R_j), C_{max}) = L_{\mathcal{T}}(R_j)$. □

Lemma 3. *Let* j *be such that* $L_{\mathcal{T}}(R_j) \geq ((m-2)p_{max} + L_{\mathcal{T}}(\overline{R_j}))/(m-1)$ *then the minimum makespan is* $L_{\mathcal{T}}(R_j)$*, with* p_{max} *the duration of the longest job not requiring* R_j.

Proof. The proof is exactly the same as that of Lemma 2, except that it uses Theorem 2 instead of Theorem 3 in (Hebrard et al. 2016), where case 2 is: $C_{max} \leq \frac{\sum_{i \in \mathcal{T}} p_i}{m} + \left(1 - \frac{1}{m}\right) p_{max}$ □

Model using approximation results

The model we present in this section leverages the previous results in order to improve on the baseline constraint programming model. In particular, we add variables standing for the ordering of operations `EnQueue`, and constraints emulating their behavior and channelling with the original variables. Backtrack search will then find a sequence of operations `EnQueue` (σ, i) that yields an optimal schedule, the existence of which is assured by Theorem 1. To that extent, we use integer variables standing for a permutation:

$\forall k \in [n]$, o_k is an integer variable of domain $[n]$

Variable o_k taking value i indicates that job i is enqueued k^{th}. We also add to the model the following constraints:

(i) ALLDIFFERENT($\{o_k, k \in [n]\}$)

(ii) ENQUEUECSTR($\{o_k, k \in [n]\}, \{s_i, i \in [n]\}$)

(iii) APPROXCSTR($\{o_k, k \in [n]\}, \{s_i, i \in [n]\}$)

The ALLDIFFERENT constraint (Régin 1994) assures that $\{o_k, k \in [n]\}$ is a permutation of $[n]$. The ENQUEUECSTR constraint ensures the channeling between the ordering variables and the variables standing for start time. The dominance relation it enforces is correct by Theorem 1. The constraint APPROXCSTR enforce the stopping conditions corresponding to Lemmas 2 and 3.

ENQUEUECSTR

Definition 4. *The constraint* ENQUEUECSTR *ensures that start times are consistent with a sequence of* `EnQueue` *operations given by the ordering variables.*

$$\text{ENQUEUECSTR}(\{o_k, k \in [n]\}, \{s_i, i \in [n]\})$$
$$\iff \forall k \in [n], s_{o_k} = \max\left(e_{\min}^{\{o_l | l<k\}}, e_{res(o_k)}^{\{o_l | l<k\}}\right)$$

The propagation of this constraint is decomposed in two separate part: the *forward* channeling from the ordering to the start time, and the *backward* channeling from start time to the ordering. We do not try to enforce generalized arc-consistency on this constraint. Instead, we focus on the variable o_k such that for all $l < k$, o_l is instantiated. This is sufficient to emulate Algorithm 2, while being efficient.

Notice that ENQUEUECSTR is related to the *SetTimes* heuristic. However, whereas the latter prevents assignments that do not yield different orderings, the former also forbids assignments that do not yield dense schedules.

Forward Channeling

The forward channeling consists in iterating over the variables $\{o_k, k \in [n]\}$ in lexicographic order while they are instantiated, and setting the start time s_{o_k} according to Definition 4. Moreover, the following proposition makes it possible to implement forward channeling very efficiently, *if we make the three following assumptions*:

1. The model has no further constraints.
2. We branch on the variables $\{o_k \mid k \in [n]\}$ only.
3. The propagator for forward channeling has a strictly lower priority than constraints (3) and (4)

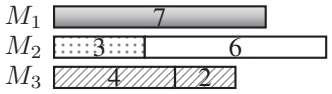

Figure 1: `EnQueue` procedure result

Proposition 1. *If every variable in $\{o_k \mid k \leq f\}$ is instantiated, and the solver has reached a fix point with respect to constraints (3) and (4), then the smallest value in the domain of s_{o_f} is* $\max\left(e_{\min}^{\{o_k|k<f\}}, e_{res(o_f)}^{\{o_k|k<f\}}\right)$.

Proof. Suppose that the variables $\{o_k \mid k \leq f\}$ are all instantiated. Observe that the variables $\{s_{o_k} \mid k < f\}$ are all instantiated as well. As a result, very weak hypothesis on the propagators for constraint (4) are sufficient to ensure that the minimum value in the domain of s_{o_f} is larger than or equal to $t_1 = e_{\min}^{\{o_k|k<f\}}$, since the resource is fully used until this time. By Lemma (2) in (Hebrard et al. 2016), if a resource R_j is used later than t_1, then it is in use in the whole interval $[t_1, t_2]$ with $t_2 = e_{R_j}^{\{o_k|k<f\}}$. Therefore, again weak hypothesis on the propagators for constraint (3) are sufficient to ensure that no value smaller than t_2 are in the domain of s_{o_f}.

On the other hand, there exists at least one branch leading to an optimal schedule where the value $\max(t_1, t_2)$ is in the domain of s_{o_f}, by Theorem 1 □

Therefore, in that situation we can simply set the value of the variable s_{o_f} to its minimum value.

Proposition 2. *Forward channeling can be implemented to run in $O(n)$ time over a branch.*

Proof. We can keep a reversible integer f for the smallest value for which the variables $\{s_{o_l} \mid l \leq f\}$ are all instantiated. When the variable o_{f+1} is instantiated, we increment f until the invariant is satisfied, and we set the domain of every variable along the way to its minimum value. So each variable is explored at most once along a branch. □

Example 1. *Let consider the example depicted by the following resources: R_1 with Jobs' processing time: $7, 2, 1, 4$; R_2 with Jobs' processing time: 3; R_3 with Jobs' processing time: $4, 2$; R_4 with Jobs' processing time: $6, 3, 2$; where the jobs being ordered as they come. That is, for example job 2 with a processing time of 2 requires R_1, and job 6 with a processing time of 4 requires R_3. If we have $o_1 = 1$, $o_2 = 5$, $o_3 = 6$, $o_4 = 7$ and $o_5 = 8$, which corresponds to the partial schedule represented on Figure 1, the filtering algorithm of* ENQUEUECSTR *will instantiate s_1, s_5, s_6 to 0, s_7 to 4 and finally s_8 to 3.*

Backward Channeling

As shown in the proof of Proposition 1, the minimum in the domain of a variable s_i is correctly maintained to the maximum of $e_{\min}^{\{o_k|k<g\}}$ and $e_{res(i)}^{\{o_k|k<g\}}$ via constraint propagation. Therefore, any assignment of the variable o_g is consistent for

ENQUEUECSTR. However, from Remark 1, we see that we can restrict search to sequences of `EnQueue` operations that are chronologically compatible. Therefore, if t is the minimum value in the domain of any job in $[n] \setminus \{o_k \mid k < g\}$, then we can prune the value i from the domain of o_g if the domain of s_i does not contain t. Indeed, the job whose start time can be t cannot require $res(i)$ and therefore, enqueueing this job or job i in any order yields the same schedule.

Proposition 3. *The filtering algorithm for backward channeling runs in $O(n)$ time.*

Proof. Computing t is done in $O(n)$. The first encountered job i whose start time variable's domain contains t can be stored, so this part is done in the size of the domain of the current o variable. □

Remark 2. *Note that this dominance rule, and the associated filtering, must be called **after** that all other constraints have been propagated. This can be achieved by setting a low priority of the propagator.*

Example 2. *Let consider the same example as Example 1. The next variable $order$ to be instantiated is o_6. In this example, t is equal to 7. Despite M_3 being idle at time 6, since there are jobs only from resources R_1 and R_4 that are not yet placed, the earliest time at which a job can be placed is 7 for jobs of resource R_1. Since t equals 7,* ENQUEUECSTR*'s filtering algorithm will remove values 9 and 10 from $dom(o_6)$ because none of the corresponding start variables have 7 in their domains. Therefore, after the call to* ENQUEUECSTR*'s filtering algorithm and to* ALLDIFFERENT*'s propagator, $dom(o_6) = \{2, 3, 4\}$.*

APPROXCSTR

The constraint APPROXCSTR detects some situations where the approximation algorithm is exact and hence the current branch can be pruned.

Proposition 4. *Let j be such that $L_{\mathcal{T}}(R_j)$ is maximum, $L_{\mathcal{T}}(\overline{R_j})$ denote the sum of the processing times of the jobs that do not require resource R_j and finally p_{max} the duration of the longest job not requiring R_j.*

If $L_{\mathcal{T}}(R_j) \geq \frac{2L_{\mathcal{T}}(\overline{R_j})}{m}$ or if $L_{\mathcal{T}}(R_j) \geq \frac{(m-2)p_{max}+L_{\mathcal{T}}(\overline{R_j})}{m-1}$ or if there are no more than m resources required by the remaining jobs, then the current schedule can be completed by algorithm MAXLOAD *to the optimal schedule that we can get from the current state.*

Proof. This proposition is the direct application of Theorem 2, Lemma 2 and Lemma 3. □

Proposition 5. APPROXCSTR*($\{o_k, k \in [n]\}, \{s_i, i \in [n]\}$) runs in $O(n \times (r+m))$, r being the number of unit resources.*

Proof. Computing $L_{\mathcal{T}}(R_j)$, $L_{\mathcal{T}}(\overline{R_j})$ and p_{max} is done in $O(n)$. Looking if there are no more than m resources required by the remaining jobs is done in $O(r \times n)$. Finally, completing the current schedule into an optimal one given the current state by algorithm MAXLOAD is done in $O(n \times (r+m))$: APPROXCSTR runs in $O(n \times (r+m))$. □

Experimentation

This section is decomposed into three parts. First we introduce the benchmark configuration. Then, we present the six approaches that were evaluated. Finally, we report and discuss the results obtained by these approaches. The instances of the benchmark and the code developed for these research can be found on GitHub (Godet 2019).

Benchmark presentation

Our benchmark is composed of 234 instances of the Parallel Machine Scheduling Problem with Additional Unit Resources, all of which have been randomly generated. The randomness of the instance is on the processed time of the jobs, but is also on the number of jobs in each resources.

# parallel machines	# Resources
2	3, 4
3	4, 5, 6
5	6, 7, 8, 10
10	12, 15, 17, 20

Table 1: Configurations #ParallelMachines - #Resources

Before generating the jobs and their processing times, we fixed several configurations for the number of machines and the number of resources, all of whom are summarized in Table 1. The basic idea was to generate, for each number of machines, instances such that the number of resources was 1.25, 1.5, 1.75 and 2.0 times higher than the number of machines. For each of this 13 configurations, we generated two types of instances: the first one has the same number of jobs for each resource (which is the case in our application, *i.e.* planning the download of acquisitions made by agile observation satellites); the second type has a random, positive, number of jobs for each resource. We had two other ways to configure instances generation: the maximal number of jobs requiring a resource, and the maximal processing time of any job. The maximal number of jobs requiring a unit resource could be either 5, 10 or 20. For the first type of instances, each unit resource is required by exactly these numbers of jobs. The maximal processing time of a job could be either 10, 100 or 1000. The generated instances have a total number of jobs between 6 and 400 and have very different shapes (size, number of machines and resources, short or large in time, etc.).

Solvers configurations

The experiments were done on an Intel core i7-8650U (up to 4.2 GHz) processor. A time limit of 30 minutes was given for each model for each instance. The source code, the MiniZinc models and the data files at .dzn format are all available on the project repository (Godet 2019). Our experiments compare four different models: a baseline *Integer* model and the three models using approximation results, respectively named *Order*, *Order+A* (using APPROXCSTR) and *Order+AM* (using `MaxLoad` as value heuristic). All of them were encoded and tested using *Choco Solver* (Prud'homme, Fages, and Lorca 2017). The *Integer* model was also implemented on the *Chuffed lazy-clause generation solver*

	domOverWDeg	smallest	SetTimes	Chuffed	Order	Order+A	Order+AM
#Proofs / #Solutions	86 / **234**	148 / **234**	127 / **234**	77 / 115	172 / **234**	174 / **234**	**213** / **234**
TimeToProof (ms)	48757	1933	1916	20329	1875	342	**137**
Objective	1.8780	1.0304	1.0617	1.5653	1.0201	1.0180	**1.0000**

Table 2: Results of the benchmark on the 234 generated instances

	domOverWDeg	smallest	SetTimes	Chuffed	Order	Order+A	Order+AM
#Proofs / #Solutions	56 / **86**	75 / **86**	81 / **86**	50 / **86**	81 / **86**	81 / **86**	**83** / **86**
TimeToProof (ms)	48910	765	828	19111	737	283	120
Objective	1.01638	1.00006	**1.00000**	1.01258	**1.00000**	**1.00000**	**1.00000**

Table 3: Results of the benchmark - small instances (86 instances of size < 40)

	domOverWDeg	smallest	SetTimes	Chuffed	Order	Order+A	Order+AM
#Proofs / #Solutions	25 / **97**	58 / **97**	39 / **97**	16 / 35	76 / **97**	76 / **97**	**87** / **97**
TimeToProof (ms)	45765	3926	3911	30463	3791	641	204
Objective	1.3960	1.0212	1.0624	1.4768	1.0016	1.0014	**1.0000**

Table 4: Results of the benchmark - medium instances (97 instances of size $40 \geq$ and < 120)

	domOverWDeg	smallest	SetTimes	Chuffed	Order	Order+A	Order+AM
#Proofs / #Solutions	5 / **51**	15 / **51**	7 / **51**	11 / 13	15 / **51**	17 / **51**	**43** / **51**
TimeToProof (ms)	59004	17490	15780	294	17166	231	199
Objective	4.2295	1.0975	1.1638	4.6521	1.0893	1.0801	**1.0000**

Table 5: Results of the benchmark - large instances (51 instances of size $120 \geq$ and ≤ 400)

(Chuffed) with its default search heuristic (Chu et al. 2019). Three search heuristics are used in the experiments: *domOverWDeg* (Boussemart et al. 2004), *smallest* and *SetTimes* as described in the section presenting the existing CP model. As *SetTimes* is close to the one we use in our approaches, we expect to find similar results in terms of efficiency. The main difference between the two approaches is that *SetTimes* will eventually branch on jobs that do not have minimum earliest start time (est). Say that you have two mutually exclusive jobs a and b, with est t and $t+1$ respectively. *SetTimes* will schedule a first, and if failing, will not try to schedule a first again, but it will try to schedule b first and a second. In our model, a is the single candidate for the next insertion, and hence if that fails we will backtrack and the ordering $b \rightarrow a$ will not be tried. The Order and Order+A models are using the *inputOrderLB* search heuristic, which consists in selecting variables by order and selecting the lower bound of the selected variable as instantiation value. As part of our search approach, the *inputOrderLB* search heuristics acts of course on variables $\{o_k, k \in [n]\}$. The Order+AM model uses a custom search heuristic based on a Algorithm 3 that act on variables $\{o_k, k \in [n]\}$.

Results

Table 2 reports an overall of the results obtained for the 234 generated instances. Our approach obtains very good results at doing the optimality proof for most of the instances. Our approaches find at least one solution, whatever its quality, for every instance, while Chuffed does not find any solution for 119 instances. We can observe that our approaches are generally faster to prove optimality than any other approach. Moreover, in general, our approaches get better solutions, as shown by the line *Objective*. The values of the line *Objective* are computed as the mean of the ratio between the result of the approach (the makespan of the returned solution) and the best solution obtained among the seven approaches. We observe a clear advantage of our approaches (Order, Order+A and Order+AM), also with a clear improvement from Order to Order+A and from Order+A to Order+AM. It is interesting to note that the cutting rule allows to prove optimality at root node for 60 instances for Order+A and Order+AM models. Analysing the results by considering the size of the problems in terms of jobs confirms the previous observations. Consequently, for respectively small instances (Table 3), medium instances (Table 4) and large instances (Table 5) observations remain the same.

Conclusion

In this paper, we presented an improved way to solve the Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR), using results from Approximation theory. After recalling definitions and classical models, we have proved that finding an optimal schedule for PMSPAUR

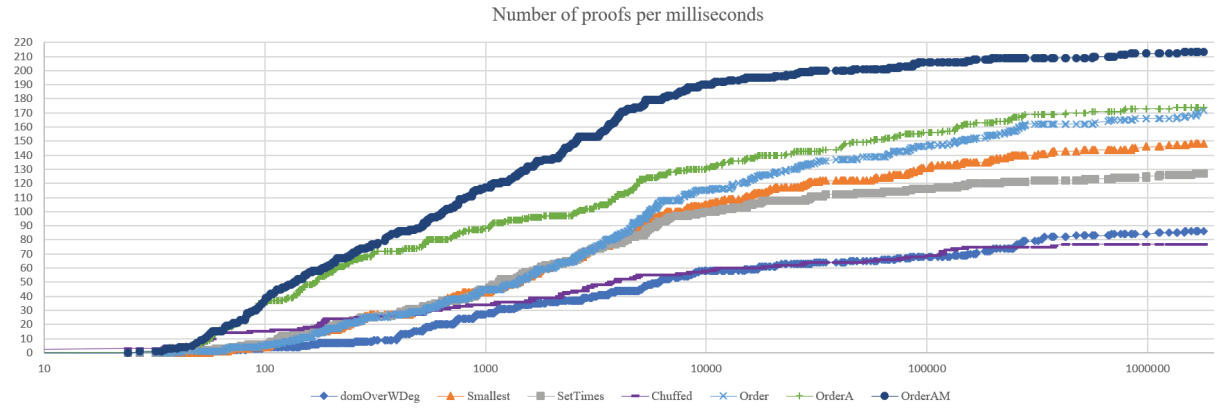

Figure 2: Number of proofs per milliseconds

can be done by searching a sequence of the jobs to apply the `EnQueue` procedure on. We also proved additional rules that allow to cut branches during the search. Finally, we presented an advanced model based on finding the sequence of tasks on which applying `EnQueue` procedure and based on these new cutting rules. We experimentally showed that our Order approach outperforms the classical ones and is really effective to find an optimal schedule for most of the instances of the benchmark, even the large ones, especially when we add the cutting rule and a specific search heuristic (respectively Order+A and Order+AM). Former research would consist in implementing this approach on other kind of scheduling problems, and find potential other cutting rules in the process. Former research would also consist in adapting this approach to different kind of problems, like ones based on *bin-packing* constraints.

References

Aggoun, A., and Beldiceanu, N. 1992. Extending CHIP in order to solve complex scheduling and placement problems. In *Actes de la 1ères Journées Francophones de Programmation Logique (JFPL)*, 51.

Allahverdi, A. 2015. The third comprehensive survey on scheduling problems with setup times/costs. *European Journal of Operational Research* 246(2):345–378.

Blazewicz, J.; Lenstra, J. K.; and Kan, A. H. G. R. 1983. Scheduling subject to resource constraints: classification and complexity. *Discrete Applied Mathematics* 5(1):11–24.

Boussemart, F.; Hemery, F.; Lecoutre, C.; and Sais, L. 2004. Boosting systematic search by weighting constraints. In *Proceedings of the 16th Eureopean Conference on Artificial Intelligence (ECAI)*, 146–150.

Chu, G.; Stuckey, P. J.; Schutt, A.; Ehlers, T.; Gange, G.; and Francis, K. 2019. *Chuffed, a lazy clause generation solver*. Department of Computing and Information Systems University of Melbourne, Australia.

Edis, E. B.; Oguz, C.; and Ozkarahan, I. 2013. Parallel machine scheduling with additional resources: Notation, classification, models and solution methods. *European Journal of Operational Research* 230(3):449–463.

Fahimi, H.; Ouellet, Y.; and Quimper, C. 2018. 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.

Garey, M. R., and Johnson, D. S. 1978. "strong" np-completeness results: Motivation, examples, and implications. *J. ACM* 25(3):499–508.

Godard, D.; Laborie, P.; and Nuijten, W. 2005. Randomized large neighborhood search for cumulative scheduling. In *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS)*, 81–89.

Godet, A. 2019. Github repository of the project. https://github.com/ArthurGodet/PMSPAUR-public.

Hebrard, E.; Huguet, M.; Jozefowiez, N.; Maillard, A.; Pralet, C.; and Verfaillie, G. 2016. Approximation of the parallel machine scheduling problem with additional unit resources. *Discrete Applied Mathematics* 215:126–135.

Ouellet, P., and Quimper, C. 2013. Time-table extended-edge-finding for the cumulative constraint. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP)*, 562–577.

Pralet, C.; Verfaillie, G.; Maillard, A.; Hebrard, E.; Jozefowiez, N.; Huguet, M.; Desmousceaux, T.; Blanc-Paques, P.; and Jaubert, J. 2014. Satellite data download management with uncertainty about the generated volumes. In *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS)*.

Prud'homme, C.; Fages, J.-G.; and Lorca, X. 2017. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S.

Régin, J.-C. 1994. A filtering algorithm for constraints of difference in csps. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*, 362–367.

Vilím, P. 2009. Edge finding filtering algorithm for discrete cumulative resources in $\mathcal{O}(kn\log(n))$. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP)*, 802–816.