Toward a Global Constraint for Minimizing the Flowtime

Camille Bonnin, Arnaud Malapert, Margaux Nattaf, Marie-Laure Espinouse

Toward a Global Constraint for Minimizing the Flowtime

Camille Bonnin[1], Arnaud Malapert[2][a], Margaux Nattaf[1] and Marie-Laure Espinouse[1][b]
[1]*Univ. Grenoble Alpes, CNRS, Grenoble INP[c], G-SCOP, 38000 Grenoble, France*
[2]*Université Côte d'Azur, CNRS, I3S, France*
{*camille.bonnin, margaux.nattaf, marie-laure.espinouse*}*@grenoble-inp.fr, arnaud.malapert@univ-cotedazur.fr*

Keywords: Constraint Programming, Global Constraint, Operation Research, Scheduling, Flowtime

Abstract: This work is a study toward a global constraint minimizing the flowtime of a single machine scheduling problem. Classical methods for filtering algorithms use a lower bound coming from the solution of a relaxation. Notably, there are several polynomial relaxations to minimize the flowtime on a single machine. A general scheme for the global constraint is proposed that allows the use of a subset of polynomial relaxations that lays the ground for more complex filtering algorithms. The filtering algorithm has a complexity of $O(n \cdot M \cdot R)$, where n is the number of tasks, M is an upper bound on the time windows of these tasks, and R is the complexity of the algorithm used for solving the relaxation. The constraint has been tested on both single machine and flowshop problems. Experimental results show that the performance improvement depends on the type of problem. The number of branches reduction is promising for designing new filtering rules.

1 INTRODUCTION

In today's society, scheduling problems are encountered everywhere with broad constraints and goals variety. The most famous and most studied objective function is the minimization of the makespan (*i.e.* the end time of the schedule) (Brucker, 2004; Strusevich, 2022; Stewart et al., 2023). However, other objectives are also often used in real life. Such is the case of the minimization of the flowtime (*i.e.* the average completion time of the tasks) which can be used to model works in progress. This objective has many applications that goes from the industrial field, such as foundry (Bewoor et al., 2018) to the healthcare service sector (Cho et al., 2023), going through multiprocessor scheduling (Awerbuch et al., 2002). There are many methods to solve those problems: heuristics, linear programming, and constraint programming are among them. Each of them has pros and cons. Some solve the problem exactly, while others give an approximation. Some methods outperform others, depending on the problem, and on the instance.

Constraint programming (CP) is a well-established method for solving scheduling problems. In fact, today, some of the largest companies (Google, IBM, Oracle) implement their own solver. However, today, CP is more focused on satisfaction problems than optimization ones. Indeed, the classical approach for optimization is to solve a sequence of satisfaction problems where additional constraints bound the domain of the objective variable. So, the flowtime minimization can be done without other additional mechanisms by independently propagating the resource constraints and the sum objective constraint (Baptiste et al., 2001a). Nevertheless, there are more and more studies on the integration of optimization in scheduling constraint problems. However, most of those works are focused on makespan minimization. As far as we know, this is not yet the case for other objectives, such as sum objectives, for which techniques propagating simultaneously the resource and optimization constraints do not exist much. Still, there are some works about the integration of such objectives in CP (Baptiste et al., 2006). In particular, (Kovács and Beck, 2011) presented a global constraint for minimizing the weighted flowtime for a single machine problem.

This work aims to contribute to creating a framework dedicated to the integration of sum objectives in scheduling constraint problems. This is strongly related to the `completion` constraint (Kovács and Beck, 2011) for the weighted flowtime. Here, we define a new global constraint, the `flowtime` constraint, that is more specialized, as it assumes that all tasks have the same weight, but has more can-

[a] https://orcid.org/0000-0003-0099-479X
[b] https://orcid.org/0000-0003-0120-661X
[c] Institute of Engineering Univ. Grenoble Alpes

didate relaxations for designing efficient filtering algorithms. Thanks to the declarative aspect of CP, a global constraint can be used to model different problems without having to modify the algorithms behind it. The relaxations of the `flowtime` and `completion` constraints are identified and classified according to their complexity. Then, a general but simple schema for filtering is designed that allows to use and compare multiple relaxations. The interest of such global constraint is validated experimentally, and the relaxations are compared within the constraint which opens prospects for improving and refining filtering rules and algorithms to minimize the flowtime.

The outline of the paper is the following. First, Section 2 introduces some notations and defines the global `flowtime` constraint. Secondly, Section 3 gives a short overview of the literature on global scheduling constraints as well as the classification of the relaxations for the `flowtime` and `completion` constraints. Then, Section 4 presents the global schema of the `flowtime` constraint including its filtering rules and algorithms. Finally, Section 5 gives some experimental results on a single machine problem and on the permutation flowshop problem.

2 NOTATIONS, DEFINITIONS

In order to give the state-of-the art and describe our constraint, some notations and definitions must be provided. This section introduces some classical notations and definitions in scheduling, and formally defines the `flowtime` constraint proposed in this paper.

Although the proposed constraint has a general schema that allows changing the relaxation internally used to compute the lower bound on the flowtime, the scheduling problem on which this constraint is based is always the same. This problem is written $1|r_j;d_j|\sum C_j$ in the Graham notation (Graham et al., 1979). The machine field 1 indicates that tasks are scheduled on a single unary capacity resource. The constraint field $r_j;d_j$ indicates that a schedule of the tasks must satisfy their release dates r_j, and deadlines d_j. The objective field $\sum C_j$ states that the optimization criterion is the total completion time (*i.e.* the sum of the completion times C_j) minimization. This objective is also often called the flowtime through misuse of language in scheduling theory (Pinedo, 2012) where the flowtime is defined formally as $\sum F_j$ with $F_j = C_j - r_j$. Since they both are equivalent to within a constant, let us call the total completion time flowtime in the following and let $F = \sum C_j$ denote the variable representing the flowtime throughout this paper.

In the following, let us assume that there are n tasks in the scheduling problem, written $T_1, \ldots, T_n$ or simply $1,\ldots,n$ when there is no ambiguity. In addition to a release date, r_j, and a deadline, d_j, each task j has a fixed duration, p_j. Let us assume that all data are positive integers and that the durations are non-zeros. When scheduled, the time at which task j begins its execution is called the starting time of j and written S_j. $C_j = S_j + p_j$ represents the time at which task j finishes its execution and is called the completion time of j.

$$\begin{array}{c}\texttt{flowtime}\,(([S_1,\ldots,S_n],[p_1,\ldots,p_n]),F)\\ \stackrel{\text{def}}{\Longleftrightarrow}((\forall_{1\leq i<j\leq n},(S_i+p_i\leq S_j \vee S_j+p_j\leq S_i))\wedge\\ \sum_i(S_i+p_i)=F)\end{array}$$

Here, the durations are assumed constant, but they could be adapted easily for variable durations by taking their lower bounds. The proposed filtering rules update the lower bound of F and the lower and upper bounds of the S_j variables. As it is assumed that the solver enforces $C_j = S_j + p_j$, it also implicitly updates the C_j variables. Given a variable X, let $\underline{X}$ denote its lower bound, and $\overline{X}$ its upper bound.

3 RELATED WORK

This section gives the state-of-the-art related to the `flowtime` constraint. First, Section 3.1 describes the most famous global constraints in scheduling. Then, Section 3.2 lists the relaxations of $1|r_j;d_j;prec|\sum C_j$, the scheduling problem used for defining the `flowtime` constraint with the additions of precedence constraints (*prec*). The reasons for this addition is explained in Section 3.2. Finally, Section 3.3 briefly describes the algorithms for solving the identified polynomial relaxations.

3.1 Scheduling and Global Constraints

The idea of using global constraints to improve the performances of constraint programming in solving scheduling problems is not recent and has shown significant results. Indeed, the disjunctive constraint (Carlier, 1982; Carlier and Pinson, 1990; Baptiste et al., 2001b; Vilím, 2004; Fahimi and Quimper, 2014) and the cumulative constraint (Aggoun and Beldiceanu, 1993; Letort et al., 2012; Gay et al., 2015) are well known and efficient constraints for modeling scheduling problems.

The disjunctive constraint is one of the first global constraints created for scheduling problems, and its filtering algorithm has many versions (Baptiste et al., 2001b). Some of those algorithms are based on the edge-finding rule, a filtering technique that takes a set

of tasks $\mathcal{T}$ and tests for each task of $\mathcal{T}$ if it must, can, or cannot be executed first (or last) in $\mathcal{T}$. For example, (Carlier and Pinson, 1990) used edge-finding for the disjunctive constraint, and (Vilím, 2004) improved it by using a specific structure called θ-λ tree.

The disjunctive constraint is based on a single machine problem that can be used as a block to model more complex scheduling problems. The `completion` constraint proposed in (Kovács and Beck, 2011) for minimizing the weighted flowtime uses the same approach. It is based on the single machine problem, tests the tasks one by one, and makes deductions according to the results of those tests. These tests allow filtering combinations of values that cannot lead to a better-cost solution than the one found so far, making the filtering algorithm of `completion` part of the cost-based domain filtering approach defined in (Focacci et al., 1999). In order to do so, those tests use a classical technique that consists of computing a lower bound on the objective by solving a relaxation of their problem (*i.e.*, a version of the problem with weakened constraints). By doing so, the efficiency of the filtering rule and algorithm is linked directly to the quality and the time needed to solve the relaxation even when incrementality is involved. This means that the choice of relaxation is very important in this approach.

3.2 Flowtime Relaxations

To design the filtering algorithm of the `flowtime` constraint, it is necessary to identify and classifies the relaxations of the single machine problem that minimizes the flowtime with respect to the release dates, the deadlines, and the precedence constraints, $1|r_j;d_j;prec|\sum C_j$. The addition of the precedence constraints is crucial here in order to have a more complete view of what is possible or not in polynomial time. Another important reason for this addition is that some precedences can be deduced from the time windows of the tasks, and so relaxations with precedences can still be used as relaxations of $1|r_j;d_j|\sum C_j$. However, note that here and in the following, when the constraint *prec* is used, it means precedences inherent to the problem and not induced by the data. Figure 1 is a diagram that gives the relations between these relaxations and classify them according to their complexity. $\mathcal{NP}$-hard problems are in purple, and polynomial problems are in blue. Problems surrounded by a dashed box allow preemption, which means that the execution of a task can be stopped for executing another one and then resumed later. $A \dashrightarrow B$ means that A is a relaxation of B and $A \rightarrow B$ that A is both a relaxation and a reduction of B. Both relations are transitive and therefore, all listed problems are relaxations of $1|r_j;d_j;prec|\sum C_j$.

The general problem considered in this section, $1|r_j;d_j;prec|\sum C_j$ is $\mathcal{NP}$-hard because it has three $\mathcal{NP}$-hard relaxations that are also reductions: $1|r_j|\sum C_j$, the problem with release dates (Lenstra et al., 1977); $1|r_j;d_j|\sum C_j$, the problem with release dates and deadlines whose complexity is the result of being a reduction of the previous one which is the general problem of the `flowtime` constraint; and $1|prec|\sum C_j$, the problem with precedence constraints (Lenstra and Rinnooy Kan, 1978).
It also has three other $\mathcal{NP}$-hard relaxations that are not reductions because of the preemption: $1|r_j;d_j;pmtn|\sum C_j$, the preemptive problem with release dates and deadlines (Du and Leung, 1993); $1|chains;r_j;pmtn|\sum C_j$, the preemptive problem with release dates and precedence constraints in the form of chains graph (Lenstra, 2023); and $1|prec;pmtn|\sum C_j$, the preemptive problem with precedence constraints which has the same complexity as $1|prec|\sum C_j$ as proven in (Brucker, 2004) because there are no release dates.

In order to have a filtering algorithm with a polynomial complexity, those problems will not be used in the `flowtime` constraint. However, they can give some indications of what is impossible to do in polynomial time. For example, $1|chains;r_j;pmtn|\sum C_j$ indicates that it is impossible to minimize the flowtime in polynomial time if there are both release dates and precedence constraints even if preemption is allowed. Indeed, chain graphs are the root of all precedence graphs, so what is impossible with them is impossible with all other forms of precedence graphs. The complexity of $1|prec;pmtn|\sum C_j$ shows that it is impossible to minimize the flowtime in polynomial time if there are precedence constraints, even if preemption is allowed. This justifies the choice made not to consider precedence constraints for the general problem of the `flowtime` constraint. Finally, in order to have a polynomial relaxation of $1|r_j;d_j;prec|\sum C_j$, at least two of those three constraints must be dropped.

$1|r_j;d_j;prec|\sum C_j$ has five polynomial relaxations: $1||\sum C_j$, the problem without any particular constraint except the fact that tasks cannot be preempted (Brucker, 2004); $1|d_j|\sum C_j$, the problem with deadlines, (Chen et al., 1998); $1|sp\text{-}graph|\sum w_jC_j$, the one where precedence constraints form a series-parallel graph and with the objective of minimizing the weighted flowtime (Lawler, 1978); $1|r_j;pmtn|\sum C_j$, the preemptive problem with release dates (Brucker, 2004); $1|r_j;pmtn|\sum w_jM_j$, the preemptive problem with release dates and whose objective is to minimize the weighted mean busy time (*i.e.*

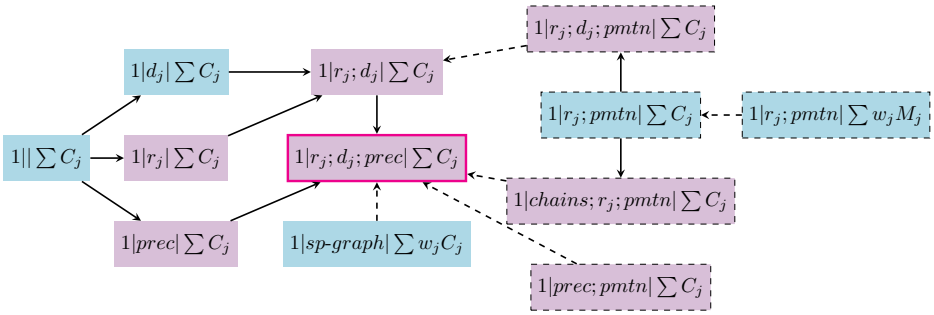


Figure 1: Reductions and relaxations of $1|r_j;d_j;prec|\sum C_j$.

the weighted sum of the average execution times of the tasks) (Kovács and Beck, 2011).

In fact, $1|r_j;pmtn|\sum w_jM_j$ is not exactly a relaxation of $1|r_j;d_j;prec|\sum C_j$ as the objective is not the same, but as explained in (Kovács and Beck, 2011), a lower bond of $\sum w_jC_j$ is given by $\sum w_jM_j$, where $M_j = (\sum t_{i,j})/p_j$, with $t_{i,j} = i$ if j is executed at time i and 0 otherwise. Note also that $1|r_j;pmtn|\sum w_jC_j$ is $\mathcal{NP}$-hard (Labetoulle et al., 1984). For simplification purposes, $1|r_j;pmtn|\sum w_jM_j$ will still be referred to as a relaxation of $1|r_j;d_j;prec|\sum C_j$ in this article.

$1||\sum C_j$ can be solved in polynomial time by the algorithm described in (Brucker, 2004). However, as its value is constant no matter the domains of the S_j variables, it is not interesting to compute a lower bound. Still, it is difficult to anticipate which of the four other polynomial relaxations will give the best results for filtering the `flowtime` constraint. So, in order to compare those relaxations, the `flowtime` constraint uses a global scheme allowing changing the relaxation. Let us now present the algorithms solving the polynomial relaxations for a better understanding.

3.3 Solving the Polynomial Relaxations

Out of the five polynomial relaxations of $1|r_j;d_j|\sum C_j$ found in the previous subsection, four can be solved by a (priority) list algorithm. A list algorithm schedules the tasks in a given order, as soon as possible, and without preemption. For example, Smith's algorithm (or SPT algorithm) for $1||\sum C_j$ orders the tasks by the shortest processing time and then schedules them as soon as possible, a new task begins when the previous one ends (Brucker, 2004). The complexity of a list algorithm is $O(n \cdot \log(n))$. A priority list algorithm schedules at each time point the task with the highest priority. In most cases, the complexity of a priority list algorithm is also $O(n \cdot \log(n))$.

The preemptive problem with release dates $1|r_j;pmtn|\sum C_j$ is solved by a priority list algorithm (Brucker, 2004) that uses modified Smith's rule. This rule consists of scheduling the available unfinished task with the smallest remaining processing time at each release date or completion date. Therefore, a task can only be preempted at the release date of another task. The only possible idle times are when no unfinished task is available because of the release dates. The preemptive problem with release dates of minimizing the weighted mean busy time $1|r_j;pmtn|\sum w_jM_j$ is also solved by a priority list algorithm (Kovács and Beck, 2011). This is similar to $1|r_j;pmtn|\sum C_j$, except that the priority is given to the available unfinished task with the largest ratio w_j/p_j.

The mandatory part is an important notion for time-tabling methods for the disjunctive (Fahimi and Quimper, 2014) and cumulative (Aggoun and Beldiceanu, 1993) constraints. The mandatory part of a task (Lahrichi, 1982) is a time interval in which the task is scheduled in any feasible solution. If the mandatory part is not empty, it is computed by $[d_j - p_j; p_j + r_j]$. Enforcing the mandatory parts in $1|r_j;pmtn|\sum C_j$ does not change the complexity of the solving algorithm (Bonnin et al., 2022). Indeed, the algorithm fixes the mandatory part of the tasks and schedules the remaining parts around them. The reasoning is similar for $1|r_j;pmtn|\sum w_jM_j$. This new constraint on the tasks will be noted *mand* so that the problems with mandatory parts, $1|r_j;pmtn;mand|\sum C_j$ and $1|r_j;pmtn;mand|\sum w_jM_j$, are also polynomial relaxations. The problems without mandatory parts are relaxations of the ones with mandatory parts.

The problem with deadlines $1|d_j|\sum C_j$ is solved by a priority list algorithm (Chen et al., 1998). This is a backward scheduling algorithm in which the tasks are scheduled from the last to the first in the schedule. The priority is given to the available unfinished task (whose deadline is bigger or equal to the current time) with the largest processing time. The schedule ends at time $\sum p_j$ which is then the first time point. Let us remark that the schedule has no idle time.

The last identified polynomial problem where the

precedence constraints form a series-parallel graph $1|sp\text{-}graph|\sum w_jC_j$ is not solved by a priority list algorithm. It is based on the traversal of the tree decomposition of the series-parallel graph that involves more complex operations for labeling the nodes (Lawler, 1978). However, this algorithm still has a complexity of $O(n \cdot \log(n))$.

To conclude, the state-of-the-art shows that global constraints improve the performance in constraint-based scheduling and that some efficient ones are based on single machine problems which are used as building blocks for modeling complex problems. The polynomial relaxations of global constraints play a role in designing filtering rules and algorithms. Such relaxations have been identified and described for $1|r_j;d_j|\sum C_j$ which defines the `flowtime` constraint. It is now possible to explain the filtering rules and algorithm of the `flowtime` constraint.

4 THE FLOWTIME CONSTRAINT

This section describes the filtering algorithm of the `flowtime` constraint. Section 4.1 gives the global schema of the filtering algorithm which is composed of two rules described in Sections 4.2 and 4.3.

For now, the `flowtime` constraint does not use the series-parallel graph relaxation, $1|sp\text{-}graph|\sum w_jC_j$. Indeed, it is more complex as to our knowledge there does not exist any non-trivial algorithm that transforms efficiently an instance of $1|r_j;d_j|\sum C_j$ into an instance of $1|sp\text{-}graph|\sum w_jC_j$.

4.1 Global Schema

Section 3 showed that the choice of the relaxation is important to develop a global constraint for scheduling problems using lower bounds. It is then interesting to compare the performances of multiple relaxations of the $1|r_j;d_j|\sum C_j$ problem for the flowtime minimization. To achieve this goal with the least bias possible, the schema of the `flowtime` constraint is the same for all relaxations. In fact, from one relaxation to another, the only change in the filtering algorithm is the computation of the lower bound which is done by solving the relaxation. For simplification purposes, from now on, the solution of the relaxation will be considered as a black box. The algorithms for solving the different relaxations are described in Section 3.3.

The filtering of the `flowtime` constraint is composed of two rules which are described in Sections 4.2 and 4.3. The first one is the update of the lower bound of F, $\underline{F}$. The second one is the filtering of the bounds of the domains of the $S_1, \ldots, S_n$ variables, the variables representing the starting times of the tasks. To illustrate the filtering algorithm of the `flowtime` constraint, the following running example is used.

Example 1 (An optimal solution of $1|r_j;d_j|\sum C_j$)**.** The instance of $1|r_j;d_j|\sum C_j$ used for this example is described in Table 1. T_j represents the task j, p_j its duration, r_j its release date and d_j its deadline.

Table 1: Instance of $1|r_j;d_j|\sum C_j$ with six tasks.

T_j	1	2	3	4	5	6
p_j	14	5	2	3	6	3
r_j	0	0	1	12	16	17
d_j	24	∞	10	∞	26	∞

An optimal solution is given in Figure 2 where the flowtime $\sum C_j$ is 129. The tasks are represented by coloured rectangles. The time is represented on the lower axis. On this axis, only the times corresponding to the release dates, deadlines or completion times are indicated. The upper axis represents the release dates (positive indices) and deadlines (negative indices) of the tasks. For instance, the release date of the task 3 is indicated by a 3 on the upper axis at time 1 on the lower axis and its deadline is indicated by a -3 on the upper axis at time 10 on the lower axis. The coloured intervals under the tasks are the mandatory parts, the number of the corresponding task is written in the middle. For instance, the dark blue interval $[d_1 - p_1; p_1 + r_1] = [24 - 14; 14 + 0] = [10; 14]$ is the mandatory part of task 1 drawn above the lower axis.

4.2 Update the Lower Bound

The first filtering rule updates the lower bound of the flowtime objective variable. First, a lower bound of F, $\underline{F'}$ is computed by solving the selected relaxation. Then, the current lower bound of F, $\underline{F}$ is updated with the value of $\underline{F'}$. Constraint programming solvers automatically check that there is no contradiction with $\overline{F}$ when updating $\underline{F}$. It means that if $\underline{F'} > \overline{F}$, then the value of $\underline{F}$ does not change and the constraint fails.

Example 2 (flowtime lower bound update)**.** Here and in Example 4, the problem selected as the relaxation is $1|r_j;pmtn|\sum C_j$. Let us also assume that previous choices and deductions have reduced the domains of the S_j variables as in Table 2. Those values of S_j can be used to create the tasks (T'_j) with the release dates r'_j and deadlines d'_j indicated in Table 2. Those T'_j tasks correspond to the T_j tasks of Table 1 with their release dates advanced and/or their deadline reduced to be able to begin only at a time present in the domain of the corresponding S_j variables. The durations are the same for T'_j and T_j. The values of r'_j and d'_j are

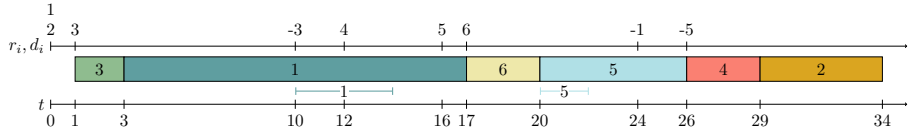


Figure 2: An optimal schedule of the instance of Table 1 for the $1|r_j;d_j|\sum C_j$ problem, where $\sum C_j = 129$.

Table 2: Domains of the S_j variables and the corresponding r'_j and d'_j values.

T'_j	1	2	3	4	5	6
S_j	[0, 10]	[0, 46]	[1, 8]	[12, 49]	[16, 20]	[17, 17]
r'_j	0	0	1	12	16	17
d'_j	24	51	10	51	26	20

obtained by those formulas: $r'_j = \underline{S_j}$ and $d'_j = \overline{S_j} + p_j$. It can be noted that T'_6 is fixed. Let us also assume that the domain of F has been reduced to $[100, 130]$.

A solution of this relaxation is described in Figure 3. The value of the lower bound, $\underline{F'}$ found by the relaxation is 104, it is more than 100, the current value of $\underline{F}$, so the value of $\underline{F}$ is updated to 104. It can be noticed that since the $1|r_j;pmtn|\sum C_j$ relaxation does not enforce the respect of the deadlines, task 1 is late. In the same way, since the respect of the mandatory parts of the tasks is not a constraint of $1|r_j;pmtn|\sum C_j$, some tasks are not executed on their mandatory part even if they should. This observation is also true for fixed tasks. This is the reason why the integration of mandatory parts will also be evaluated.

4.3 Filtering the Bounds of the Domains of the Start Time Variables

The second filtering rule of `flowtime` updates the bounds of the domain of the start time variables, S_j. For sake of simplification, only the filtering of the lower bounds is presented. The algorithm for filtering the upper bounds is symmetric to Algorithm 1.

For each T'_j, the algorithm schedules T'_j without preemption from its earliest starting time (*i.e.* $\underline{S_j}$). This time is noted t in the following. It then tries to schedule the other tasks around T'_j following the solving algorithm of the relaxation. The optimality proof of schedule for the relaxation with T'_j fixed is similar to those for the mandatory parts (Bonnin et al., 2022). If the objective value found by the relaxation is strictly bigger that $\overline{F}$, then there is no feasible solution for which T'_j begins at time t and t is filtered from the domain of S_j. Indeed, as the objective value found by the relaxation is a lower bound of F if T'_j begins at time t, then a similar reasoning than what is done in Section 4.2 can be made.
It can also happen that the relaxation does not have a solution. For instance, if T'_j is scheduled on the mandatory part of another task and the relaxation ensure the respect of mandatory parts. In this case, there is no feasible solution for which T'_j begins at time t, t is then filtered from the domain of S_j.
In the other case, if there is a solution and the objective value computed by the relaxation is no bigger than $\overline{F}$, a support can be found in which T'_j begins at time t, so no deduction is made. It is then not possible to filter more the lower bound of S_j, so the algorithm stops for this task and start looking at the next task.

Algorithm 1: Filtering algorithm of $\underline{S_1}, \ldots, \underline{S_n}$

Data: $\{S_1, \ldots, S_n\}, \{p_1, \ldots, p_n\}, \overline{F}$
Result: Updated $\{\underline{S_1}, \ldots, \underline{S_n}\}$
$\{T'_1, \ldots, T'_n\} \leftarrow$ the tasks created from $\{S_1, \ldots, S_n\}$ and $\{p_1, \ldots, p_n\}$;
for $j \leftarrow 1$ ***to*** n **do**
  $t \leftarrow \underline{S_j}$;
  while $t \leq \overline{S_j}$ **do**
    $relax_{j,t} \leftarrow$ value of the objective found by solving the relaxation while setting T'_j to be executed in $[t, t + p_j]$;
    if $relax_{j,t} = \varnothing$ ***or*** $relax_{j,t} > \overline{F}$ **then**
      $t \leftarrow t + 1$;
    else break;
  $\underline{S_j} \leftarrow t$

Proposition 3. *The complexity of Algorithm 1 is $O(n \cdot M \cdot R)$ where $M = \max_j(|S_j|)$ and R is the complexity of the relaxation algorithm.*

Proof. Creating one task T'_j is an $O(1)$ operation. It is repeated for each of the n tasks to be created. So the first line has an $O(n)$ complexity.

For the loop, the action is repeated for all n tasks. The first and last line of the **for** loop are assignations that take $O(1)$, so the complexity is bounded by the complexity of the **while** loop. The **while** loop is repeated at most for all possible values of the domain of S_j (from $\underline{S_j}$ to $\overline{S_j}$), and so at a maximum of $M = \max_j(|S_j|)$ times. Setting the task to be executed in $[t, t + p_j]$ and computing the relaxation can be done with the same complexity R as simply computing the relaxation by the same argument as what

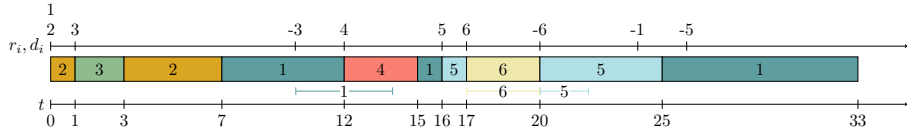


Figure 3: An optimal solution of the $1|r_j;pmtn|\sum C_j$ relaxation for the T'_j tasks, where $\sum C_j = 104$.

is done in (Bonnin et al., 2022). The **if** and **else** parts are only simple tests and assignations and so have an $O(1)$ complexity. The complexity of the **for** loop is then bound by $O(n \cdot M \cdot R)$.

The complexity of Algorithm 1 is then bound by $O(n \cdot M \cdot R)$ where $M = \max_j(|S_j|)$ and R is the complexity of the relaxation algorithm. □

Example 4 (Execution of Algorithm 1)**.** This example uses the same values as Example 2 and only the filtering of $\underline{S_1}$ is given. Figure 4 shows the results of the relaxation at each step of the algorithm. At steps 1, 2, and 3, where $t = 0$, 1, and 2 respectively, the objective value of the relaxation is strictly greater than those of $\overline{F}$, 130. So the value of $\underline{S_1}$ is updated to 3. But at step 4, where $t = 3$, the objective value of the relaxation is no greater than 130, it is not possible to filter more the value of $\underline{S_1}$ and the algorithm stops looking at T'_1 and begins examining the next task.

A choice is made in Algorithm 1 to only filter the bounds of the S_j variables as a classical representation of a task in constraint programming is an interval variable for which most of the solvers only allow bounds filtering. Algorithm 1 is a naive version of a filtering rule for `flowtime` as it tests each possible value for $\underline{S_j}$. Once the promising relaxations are identified, specific rules or algorithms should be designed for those relaxations. To obtain the complete filtering algorithm of the `flowtime` constraint, the bound update of Section 4.2 must be executed first. Then, if there is no contradiction with $\overline{F}$, the lower bounds of the S_j variables must be updated with Algorithm 1 and the last step is to update the upper bounds of the S_j variables with a symmetric algorithm to Algorithm 1.

5 EXPERIMENTAL RESULTS

This section aims to evaluate the relaxations and filtering rules available for the global constraint `flowtime`. The experiment framework is defined so the following questions are addressed: **Q1.** Which relaxation is the best proving the optimality or finding good upper bounds? **Q2.** What are the performance trade-offs between the propagation of the lower bound and the filtering of the starting times? In terms of solving time? Number of branches? **Q3.** What is the efficiency of the constraint depending on the problem?

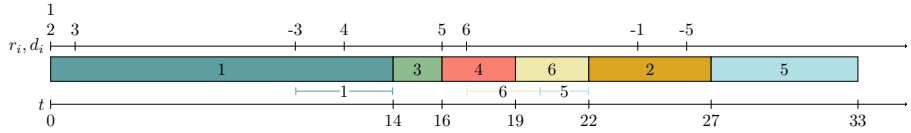
Section 5.1 describes the experimental protocol with the alternatives for the `flowtime` constraint. The constraint is evaluated on two flowtime minimization problems: a single machine problem in Section 5.2; and a permutation flowshop problem in Section 5.3.

5.1 Experimental Protocol

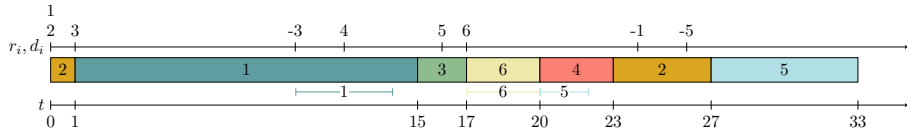
The following alternatives are considered for the global constraint `flowtime`: `sum` uses a standard sum constraint for propagating the flowtime; `pmtnFlow` propagates the lower bound as presented in Section 4.2 using $1|r_j;pmtn|\sum C_j$; `pmtnBusy` propagates the lower bound using $1|r_j;pmtn|\sum M_j$; `mandFlow` propagates the lower bound using $1|r_j;pmtn;mand|\sum C_j$; `mandBusy` propagates the lower bound using $1|r_j;pmtn;mand|\sum M_j$; `norelFlow` propagates the lower bound using $1|d_j|\sum C_j$; `filtFlow` does as `pmtnFlow` but also filters the starting times as presented in Section 4.3 using $1|r_j;pmtn|\sum C_j$; `filtBusy` does as `pmtnBusy` and filters the starting times using $1|r_j;pmtn|\sum M_j$; `complBusy` adapts the `Completion` constraint (Kovács and Beck, 2011), which uses $1|r_j;pmtn|\sum M_j$, as described in the following.

The proposed alternatives have been implemented as a global constraint in C++ and embedded into IBM ILOG CPLEX Optimization Studio 22.1 (IBM, 2023). All models state the standard sum constraint and a `noOverlap` constraint. The `noOverlap` constraint sets the resource capacity enforcement and uses the default inference techniques.

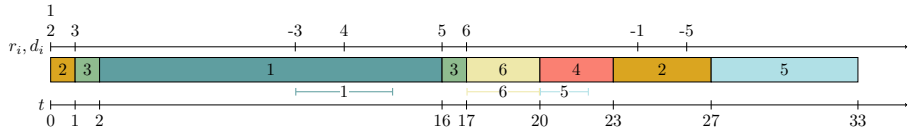
The `Completion` constraint proposed in (Kovács and Beck, 2011, `complBusy`) has been made compatible with the latest version of CP Optimizer with the minimum possible changes on the code. Indeed, the IlcActivity type is not available anymore because it has been replaced by IloIntervalVar. IlcActivity allowed to remove values from an enumerated domain, creating holes, but IloIntervalVar only accepts changes on the bounds of its interval domain. The `Completion` constraint has then been adapted to use IloIntervalVar variables instead of IlcActivity ones. Therefore, keep in mind that the original constraint has been weakened as it only updates the bounds of the domains. For the



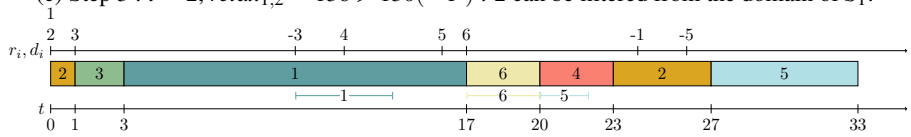
(a) Step 1 : $t = 0$, $relax_{1,0} = 131 > 130 (= \overline{F})$: 0 can be filtered from the domain of S_1.



(b) Step 2 : $t = 1$, $relax_{1,1} = 135 > 130 (= \overline{F})$: 1 can be filtered from the domain of S_1.



(c) Step 3 : $t = 2$, $relax_{1,2} = 136 > 130 (= \overline{F})$: 2 can be filtered from the domain of S_1.



(d) Step 4 : $t = 3$, $relax_{1,3} = 123 \leq 130 (= \overline{F})$: STOP, a support of $\underline{S_1}$ has been found.

Figure 4: Example of the filtering rule of the lower bound of S_1 variable (second rule) with the $1|r_j;pmtn|\sum C_j$ relaxation. The example uses the instance of Table 2 and $\underline{S_1}$, whose previous value was 0, is updated to 3.

same reason, the version tested is not optimized, so the time results for `complBusy` should be read as upper bounds and not precise measures.

Only results obtained using a CP model are presented as results obtained by implementing a classical MIP time-indexed model (Keha et al., 2009) with the same configuration are always at least 2 times slower than those obtained using the `sum` alternative.

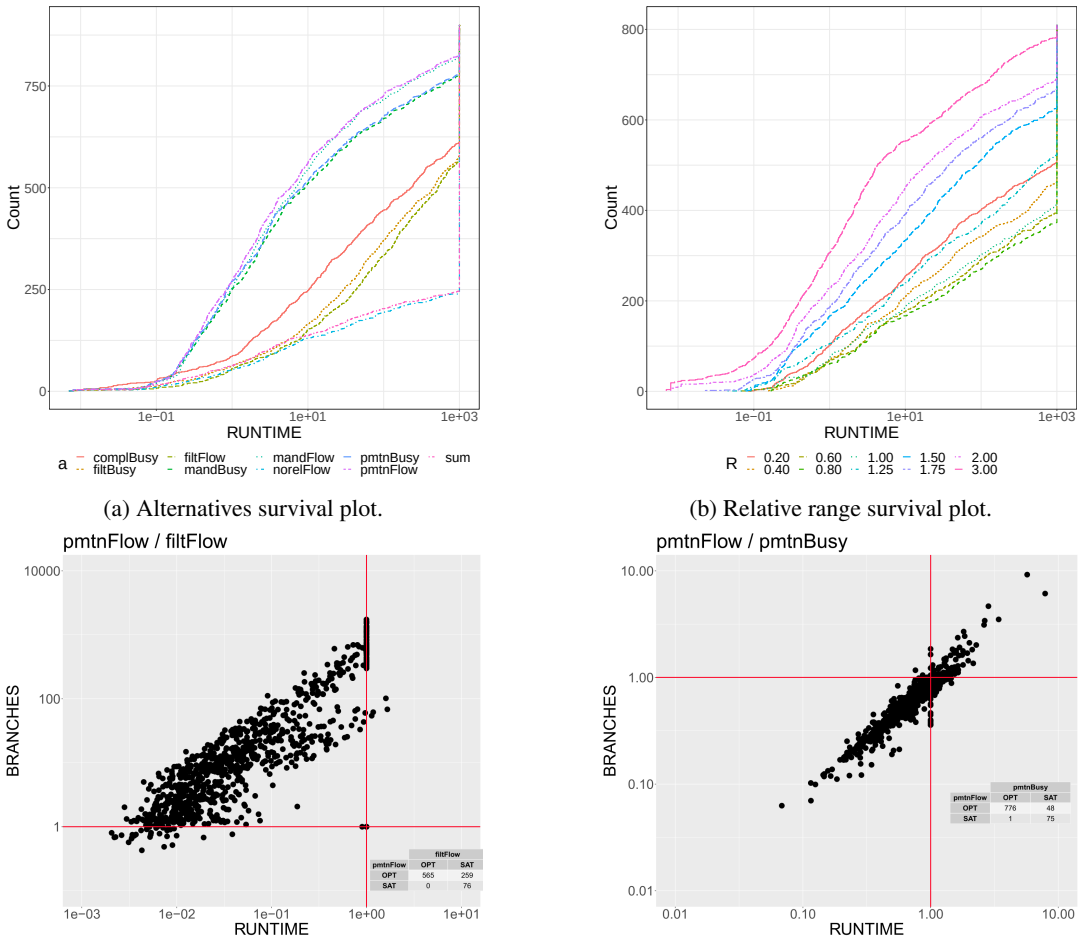
All experiments were run on a Dell computer with 256 GB of RAM and 4 Intel E7-4870 2.40 GHz processors running on CentOS Linux release 7.9 (each processor has 10 cores). The parallelism is disabled in order to ease the comparative analysis. Last, the time limit is 1000 seconds for each run.

5.2 Single Machine Problem

The problem instances for the single machine problem have been proposed in (Pan and Shi, 2008) for $1|r_j|\sum w_j C_j$ and also used in (Kovács and Beck, 2011). The repository contains 10 problem instances for each combination of parameters n and R, where n is the number of tasks, while R is the relative range of the release date. We have selected all 900 instances with fewer than 100 tasks for every relative range.

Figure 5a gives the number of instances optimally solved depending on the time grouped by alternative. All instances that have reached the time limit are in the vertical line at time $1e+03$. A faster alternative is above and on the left of a slower alternative. Clearly, the alternatives `sum` and `norelFlow` are the worst alternatives because they only optimally solve around 250 instances over 900 and are an order of magnitude slower than the other alternatives. So, the alternative `norelFlow` can be discarded as inefficient in future work. The alternatives that filter the starting times improve the efficiency by optimally solving more than 500 instances. The alternative `filtBusy` is slightly faster than `filtFlow`, and they are both slightly less efficient and slower than `complBusy`. It shows an interest in the incrementality of the filtering compared to our more naive, but simpler, approach. Currently, the alternatives that only propagate the lower bound are the most efficient and the fastest. Surprisingly, the integration of the mandatory part does not give any significant improvement. To conclude, `pmtnFlow` is the most efficient closely followed by `pmtnBusy`. Filtering the starting times takes too much time and leads to fewer optimality proofs even when the filtering is incremental as in `complBusy`.

Figure 5b gives the number of instances optimally solved depending on the time it took grouped by the relative range R of the release dates. The relative range is relevant with respect to the problem hard-



(a) Alternatives survival plot.

(b) Relative range survival plot.

(c) `pmtnFlow` versus `filtFlow`

(d) `pmtnFlow` versus `pmtnBusy`

Figure 5: Experimental results for the single machine problem.

ness because the curves can be easily distinguished. The easiest instances have the greatest ranges (3.00, 2.00, 1.75, 1.50, 1.25) because, for a given time window, only a few tasks overlap, reducing the degree of freedom for scheduling. Then, instances that have the lowest ranges (0.20, 0.40) are also pretty easy, because the problem becomes close to $1||\sum C_j$ and the relaxation is likely to give good lower bounds. Finally, the middle ranges (0.60, 0.80, 1.00) lead to the hardest instances because the relaxation is weaker and the degree of freedom for scheduling remains high.

In Figures 5c and 5d, each point represents one instance and its x coordinate is the ratio of the solving time of the first alternative over the solving time of the second alternative, whereas its y coordinate is the ratio of the branches. Notice also that both scales are logarithmic. All points located above and on the right of the point (1,1) are instances improved by the second alternative. On the contrary, all points located below and on the left of the point (1,1) are instances improved by the first alternative (bottom-left quadrant).

As expected, all points are around the diagonal as the number of nodes is roughly proportional to the time (top-left and bottom-right quadrants are empty). Figure 5c shows that filtering the starting times (`filtFlow`) reduces by an order of magnitude on the number of branches compared to the propagation of the lower bound (`pmtnFlow`), but it is so slow that the solving time remains larger by an order of magnitude. Therefore, filtering the starting times is promising, but this requires further improvements beyond the incrementality.Figure 5d shows that the relaxation with the flowtime gives more optimality proofs, explores fewer branches and is faster than the relaxation with the mean busy time.

Table 3 summarizes the experimental results as in (Kovács and Beck, 2011). Each row contains combined results for the 10 instances with the same number of activities, n, and release time range, R. For each of the 4 alternatives, the table displays the num-

Table 3: Experimental results for four alternatives on single machine problems.

Problem		sum			pmtnFlow			filtFlow			complBusy		
n	R	o	b	t	o	b	t	o	b	t	o	b	t
20	0.2	–	–	–	10	14.7K	0.3	10	0.8K	1.8	10	0.9K	0.1
	0.6	3	10.0M	292.5	10	23.8K	0.5	10	2.8K	5.2	10	8.7K	3.2
	1.0	10	1.9M	57.1	10	14.6K	0.3	10	6.1K	8.7	10	9.0K	2.7
	1.5	10	81.4K	2.0	10	7.5K	0.2	10	2.2K	3.4	10	4.2K	0.8
	2.0	10	9.2K	0.2	10	5.6K	0.1	10	1.9K	1.4	10	2.8K	0.5
30	0.2	–	–	–	10	30.1K	0.9	10	1.5K	10.3	10	3.5K	1.7
	0.6	–	–	–	10	43.0K	1.2	10	3.9K	26.1	10	16.5K	19.3
	1.0	2	2.5M	101.4	10	45.6K	1.6	10	16.2K	69.2	10	25.9K	37.8
	1.5	9	494.7K	17.7	10	11.2K	0.3	10	5.4K	15.6	10	9.4K	2.2
	2.0	10	81.3K	2.7	10	10.2K	0.3	10	5.3K	14.0	10	7.8K	1.1
40	0.2	–	–	–	10	34.0K	1.4	10	3.0K	44.1	10	13.5K	12.5
	0.6	–	–	–	10	94.4K	4.1	10	8.2K	116.3	10	82.4K	90.5
	1.0	2	7.5M	368.8	10	43.7K	2.0	10	12.9K	82.0	10	44.0K	19.9
	1.5	10	1.4M	64.8	10	13.8K	0.5	10	7.5K	38.7	10	23.8K	6.4
	2.0	9	191.4K	7.8	10	13.1K	0.5	10	5.8K	27.8	10	10.4K	8.8
50	0.2		–	–	10	52.4K	3.7	10	6.3K	171.8	10	48.6K	39.5
	0.6	–	–	–	10	213.8K	12.3	8	14.5K	383.1	10	282.5K	291.1
	1.0	–	–	–	10	2.2M	143.8	5	31.9K	423.1	5	199.7K	331.7
	1.5	4	2.5M	118.6	10	70.2K	4.4	10	18.8K	168.7	10	75.3K	158.0
	2.0	8	630.0K	31.3	10	18.6K	0.9	10	8.1K	74.0	10	18.0K	10.49
60	0.2	–	–	–	10	109.1K	9.1	6	10.8K	517.0	8	150.8K	239.2
	0.6	–	–	–	10	1.3M	107.1	2	12.9K	662.9	–	–	–
	1.0	–	–	–	9	342.1K	26.3	6	21.9K	528.2	3	150.7K	784.3
	1.5	–	–	–	10	32.6K	2.2	10	16.3K	199.4	10	36.8K	32.5
	2.0	6	3.7M	187.6	10	24.0K	1.2	10	6.8K	88.9	10	41.4K	14.7
70	0.2	–	–	–	10	430.1K	40.0	3	10.1K	674.8	3	69.7K	632.1
	0.6	–	–	–	8	2.7M	267.1	–	–	–	–	–	–
	1.0	–	–	–	10	2.1M	192.0	2	38.1K	831.4	2	197.3K	332.3
	1.5	–	–	–	10	119.1K	10.1	8	26.0K	488.0	9	174.7K	215.1
	2.0	3	5.2M	312.2	10	41.0K	2.7	10	20.3K	291.1	10	52.8K	55.0

ber of instances that were solved to optimality (column o), the average number of branches (b), and the average search time in seconds (t). The averages are computed only on the instances that the algorithm optimally solved. Table 3 confirms that the 3 alternatives for the `flowtime` constraint are more efficient and reduce the number of branches compared to the `sum` constraint. More and larger instances are optimally solved in Table 3 than in (Kovács and Beck, 2011) where only a few instances with 50 tasks or more are optimally solved. The comparison between the propagation of the lower bound (`pmtnFlow`) and the filtering of the starting times (`filtFlow` and `complBusy`) also differs. In (Kovács and Beck, 2011), the filtering of the starting times outperforms the propagation of the lower bound in terms of the number of optimality proofs, solving times, and number of branches. The reason is not obvious, but could include the change of objective function, or our adaptation of the `completion` constraint.

The detailed results are not given here, but the situation is similar when comparing the upper bounds on instances that are not optimally solved.

To conclude on the single machine, $1|d_j|\sum C_j$ is not efficient, but other preemptive relaxations are more efficient in terms of solving time and reducing the number of branches, with a slight advantage for the flowtime over the mean busy time. Mandatory parts do not have much practical importance for filtering the starting times. Still, the results obtained using $1|d_j|\sum C_j$ and, to some extent, the mandatory part might be biased due to instances lacking deadlines. Last, filtering the starting times is not yet efficient enough to balance the additional time required for it compared to the lower bound propagation. Yet, it drastically reduces the number of branches which is promising for designing more efficient algorithms through incrementality and triggering.

5.3 Flowshop Problem

The flowshop problem consists of determining a processing sequence of n tasks in a set of m machines that are arranged in series. All tasks must be processed sequentially in all machines. Each task needs a given processing time at each machine. A flowshop is a common production setting in factories where products start processing at machine or stage 1 and continue processing until they are finished in the last machine. The permutation flowshop problem also assumes that the production sequence of tasks for the first machine is kept unaltered for all other machines. The 300 instances have between 5 and 20 machines: 60 instances of (Taillard, 1993) have between 20 and

50 jobs; and 240 instances of (Vallada et al., 2015) have between 10 and 60 jobs. Table 4 and Table 5 present the experimental results for the flowshop for five relevant alternatives identified in Section 5.2.

Table 4 gives for each alternative the number of instances solved optimally, the average search time (in seconds), the average number of branches, and their standard deviations. Those last four values are computed over the optimally solved instances only, otherwise, the time limit is reached and these are not significant. It shows that the improvement provided by the global constraint is not as significant as for the single machine problem. In fact, `pmtnFlow` is the quickest to prove the optimality, followed by `complBusy` and `pmtnBusy`, then `sum` and `filtFlow` is the one that takes the longest by far. However, `pmtnFlow` is doing one fewer proof than `complBusy`, `pmtnBusy`, and `sum`, and `filtFlow` is doing very fewer proofs than the others. In terms of the number of branches, `filtFlow` is doing smaller proofs, but as it does less than $3/5$ of the proofs done by the other alternatives, the comparison is hard to do. As expected, for the other alternatives, `complBusy` is the one with the fewer branches, followed by `pmtnFlow`, then `pmtnBusy`, and by far `sum`. Note that at most 36 instances over 300 are optimally solved as even the smallest flowshop instance has more operations (100) than the bigger instances tested for the single machine problem.

Table 4: Time and number of branches per alternative.

Alternative	Opt	Time (s)		Branches	
a	*o*	*avg*	*std*	*avg*	*std*
`sum`	36	317	269	2.26M	1.60M
`pmtnBusy`	36	257	244	1.86M	1.43M
`pmtnFlow`	35	237	228	1.67M	1.25M
`complBusy`	36	257	231	1.28M	1.03M
`filtFlow`	21	412	336	0.54M	0.46M

Table 5 summarizes the relative error of each alternative over all instances. The relative error for an instance is the ratio of the absolute error to the best known objective value for this instance. For each alternative (column a), the table displays the number of the instances optimally solved (column o), the average relative error (*avg*), and its standard deviation (*std*) in per-thousand. The averages are computed over all 300 instances. For the relative error, the alternative `sum` gives the best results closely followed by `pmtnBusy`, and less closely by `pmtnFlow`. The filtering of the starting times gives significantly worse upper bounds (`complBusy` and mostly `filtFlow`).

A possible explanation is that the instances for the flowshop problem are larger and harder than those for the single machine so the search speed becomes more important than the propagation strength for finding good upper bounds. Furthermore, many scheduling decisions, even for the permutation flowshop, have less influence over the relaxation because they concern earlier machines. Indeed, the schedule of the last machine entirely determines the flowtime and it depends on the schedule of all previous machines, giving less visibility to the bound and the filtering. It shows that even the propagation of the lower bound should be more carefully triggered to save time and that filtering the relative order of tasks could help as in most global constraints for scheduling.

Table 5: Mean relative error (over all instances).

a	*o*	*avg* (‰)	*std* (‰)
`sum`	36	0.48	0.63
`pmtnBusy`	36	0.50	0.66
`pmtnFlow`	35	0.55	1.19
`complBusy`	36	1.35	2.18
`filtFlow`	21	5.01	4.76

To conclude this section, the `flowtime` constraint helps to improve the performances of the solver on single machine problems. However those results do not carry to more complex scheduling problems like the flowshop problem. The main part to improve is the filtering part as it considerably reduces the number of branches, but currently takes too much time.

6 CONCLUSIONS

This article proposes a global constraint minimizing the flowtime that allows using multiple polynomial relaxations of $1|r_j;d_j|\sum C_j$. For now, the `flowtime` constraint is more efficient than a standard sum constraint on a single machine problem but only has a moderate effect on the flowshop problem. The experimental results have identified the most efficient relaxations and shown that filtering the starting times takes too long to improve the solving times despite reducing the number of branches.

This work opens new perspectives for designing more efficient filtering rules and algorithms for the `flowtime` constraint. First, incremental filtering of the starting times would improve the performances but is not enough for compensating the additional time. Second, the events that trigger the filtering are key to efficiency because most of the filtering stages do not reduce the domains. Third, the detection of precedences between tasks may strengthen the filtering while reducing the number of steps as in the disjunctive constraint. Finally, the last avenue is to integrate the last untested relaxation $1|sp\text{-}graph|\sum w_j C_j$ that is not solved by a priority list algorithm.

ACKNOWLEDGEMENTS

We thank András Kovács and J. Christopher Beck for sharing their code, and the latter for his advice.

REFERENCES

Aggoun, A. and Beldiceanu, N. (1993). Extending chip in order to solve complex scheduling and placement problems. *Math Comput Model*.

Awerbuch, B., Azar, Y., Leonardi, S., and Regev, O. (2002). Minimizing the flow time without migration. *SIAM J Comput*.

Baptiste, P., Laborie, P., Pape, C. L., and Nuijten, W. (2006). Constraint-based scheduling and planning. In Rossi, F., van Beek, P., and Walsh, T., editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence. Elsevier.

Baptiste, P., Le Pape, C., and Nuijten, W. (2001a). *Constraint-based scheduling: applying constraint programming to scheduling problems*. Springer Science & Business Media.

Baptiste, P., Le Pape, C., and Nuijten, W. (2001b). *Constraint-based scheduling: applying constraint programming to scheduling problems*. Int Ser Oper Res Man. Springer.

Bewoor, L. A., Prakash, V. C., and Sapkal, S. U. (2018). Production scheduling optimization in foundry using hybrid particle swarm optimization algorithm. *Procedia Manuf.*

Bonnin, C., Nattaf, M., Arnaud, A. M., and Espinouse, M.-L. (2022). Extending smith's rule with task mandatory parts and release dates. In *PMS 2022*.

Brucker, P. (2004). *Single Machine Scheduling Problems*. Springer Berlin, Heidelberg.

Carlier, J. (1982). The one-machine sequencing problem. *Eur J Oper Res*.

Carlier, J. and Pinson, É. (1990). A practical use of jackson's preemptive schedule for solving the job shop problem. *Ann Oper Res*.

Chen, B., Potts, C. N., and Woeginger, G. J. (1998). *A Review of Machine Scheduling: Complexity, Algorithms and Approximability*. Springer US.

Cho, B.-H., Athar, H. M., Bates, L. G., Yarnoff, B. O., Harris, L. Q., Washington, M. L., Jones-Jack, N. H., and Pike, J. J. (2023). Patient flow time data of covid-19 vaccination clinics in 23 sites, united states, april and may 2021. *Vaccine*.

Du, J. and Leung, J. Y. (1993). Minimizing mean flow time with release time and deadline constraints. *J Algorithm*.

Fahimi, H. and Quimper, C.-G. (2014). Linear-time filtering algorithms for the disjunctive constraint. *Proceedings of the AAAI Conference on Artificial Intelligence*.

Focacci, F., Lodi, A., and Milano, M. (1999). Cost-based domain filtering. In Jaffar, J., editor, *Lect Notes Comput SC*. Springer Berlin Heidelberg.

Gay, S., Hartert, R., and Schaus, P. (2015). Simple and scalable time-table filtering for the cumulative constraint. In Pesant, G., editor, *Lect Notes Comput SC*. Springer International Publishing.

Graham, R., Lawler, E., Lenstra, J., and Kan, A. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. In Hammer, P., Johnson, E., and Korte, B., editors, *Discrete Optim II*, Annals of Discrete Mathematics. Elsevier.

IBM (2023). Cplex optimisation studio 22.1. https://www.ibm.com/products/ilog-cplex-optimization-studio.

Keha, A., Khowala, K., and Fowler, J. (2009). Mixed integer programming formulations for single machine scheduling problems. *Comput Ind Eng*.

Kovács, A. and Beck, J. C. (2011). A global constraint for total weighted completion time for unary resources. *Constraints*.

Labetoulle, J., Lawler, E., Lenstra, J., and Rinnooy Kan, A. (1984). Preemptive scheduling of uniform machines subject to release dates. In *Progress in combinatorial optimization*. Academic Press.

Lahrichi, A. (1982). Ordonnancements. la notion de parties obligatoires et son application aux problèmes cumulatifs. *RAIRO-Oper Res*.

Lawler, E. (1978). Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In Alspach, B., Hell, P., and Miller, D., editors, *Algorithmic Aspects of Combinatorics*, Ann. of Discrete Math. Elsevier.

Lenstra, J., Rinnooy Kan, A., and Brucker, P. (1977). Complexity of machine scheduling problems. *Ann. of Discrete Math.*

Lenstra, J. K. (accessed in 2023). Unpublished. http://schedulingzoo.lip6.fr/ref2bibtex.php?ID=Lenstra::.

Lenstra, J. K. and Rinnooy Kan, A. (1978). Complexity of scheduling under precedence constraints. *Oper Res*.

Letort, A., Beldiceanu, N., and Carlsson, M. (2012). A scalable sweep algorithm for the cumulative constraint. In Milano, M., editor, *Lect Notes Comput SC*. Springer Berlin Heidelberg.

Pan, Y. and Shi, L. (2008). New hybrid optimization algorithms for machine scheduling problems. *IEEE T Autom Sci Eng*.

Pinedo, M. L. (2012). *Scheduling: theory, algorithms, and systems*. Springer New York, NY.

Stewart, R., Raith, A., and Sinnen, O. (2023). Optimising makespan and energy consumption in task scheduling for parallel systems. *Comput Oper Res*.

Strusevich, V. A. (2022). Complexity and approximation of open shop scheduling to minimize the makespan: A review of models and approaches. *Comput Oper Res*.

Taillard, É. (1993). Benchmarks for basic scheduling problems. *Eur J Oper Res*.

Vallada, E., Ruiz, R., and Framinan, J. M. (2015). New hard benchmark for flowshop scheduling problems minimising makespan. *Eur J Oper Res*.

Vilím, P. (2004). O(nlogn) filtering algorithms for unary resource constraint. In Régin, J.-C. and Rueher, M., editors, *Lect Notes Comput SC*. Springer Berlin Heidelberg.