A Global Constraint for Total Weighted Completion Time for Unary Resources

András Kovács
Computer and Automation Research Institute
Hungarian Academy of Sciences
akovacs@sztaki.hu

J. Christopher Beck
Department of Mechanical and Industrial Engineering
University of Toronto, Canada
jcb@mie.utoronto.ca

November 30, 2009

Abstract

We introduce a novel global constraint for the total weighted completion time of activities on a single unary capacity resource. For propagating the constraint, we propose an $O(n^4)$ algorithm which makes use of the preemptive mean busy time relaxation of the scheduling problem. The solution to this problem is used to test if an activity can start at each start time in its domain in solutions that respect the upper bound on the cost of the schedule. Empirical results show that the proposed global constraint significantly improves the performance of constraint-based approaches to single-machine scheduling for minimizing the total weighted completion time. We then apply the constraint to the multi-machine job shop scheduling problem with total weighted completion time. Our experiments show an order of magnitude reduction in search effort over the standard weighted-sum constraint and demonstrate that the way in which the job weights are associated with activities is important for performance.

Keywords: Scheduling, constraint propagation, total weighted completion time

1 Introduction

Many successful applications of constraint programming (CP) to optimization problems exhibit a "maximum type" optimization criteria, characterized by minimizing the maximum value of a set of variables (e.g., makespan, maximum tardiness, or peak resource usage in scheduling). Such criteria exhibit strong back

propagation: placing an upper bound on the cost function results in the pruning of the domain (i.e., the reduction of the maximum value) of the constituent variables and the subsequent reduction in search space. Particularly in the scheduling domain, CP has not been as successful for other practically important optimization criteria such as "sum type" objective functions characterized by the minimization of the sum of a set of variables. Examples include total weighted completion time, weighted tardiness, weighted earliness and tardiness, and the number of late jobs. Even the recent and most efficient constraint-based approaches to scheduling with such criteria use only weak propagation, such as a weighted-sum constraint, although these are often coupled with strong lower bounding techniques to infer dead-ends [4]. Back propagation of the sum constraint is weak because it is often the case that the maximum value of each addend variable is supported by the minimum values of all the other variables. The significance of more efficient global constraints for back propagation has been emphasized by Focacci et al. [15, 16].

Our purpose is to develop algorithms for propagating "sum type" objective functions in constraint-based scheduling. In this paper, we address the total weighted completion time criterion on a single unary resource. This has equivalents in a wide range of applications. For example, in container loading problems it is typical to have constraints on the location of the centre of gravity (COG) of the cargo loaded into the container. The location of the COG in any selected dimension corresponds to the total weighted completion time in a schedule, in which activities stand for the boxes inside the container. Activity durations correspond to box lengths, resource requirements to the surfaces of the boxes, and activity weights to physical weight of the loaded boxes [23]. Another example is the capacitated lot-sizing problem and its discrete variant, where different items are produced on a resource with limited capacity, with specific deadlines [13]. The cost of a solution is composed of a holding cost and a setup or ordering cost where the former is computed as the total weighted difference of deadlines and actual production times. Apart from a constant factor, this is equivalent to the weighted distance of the activities from a remote point in time, which corresponds to the weighted completion time in a reversed schedule.

In all these applications, the total weighted completion time constraint appears as only one component of a complex satisfaction or optimization problem, in conjunction with various other constraints. Therefore, it appears appropriate to adopt a CP approach, where an inference algorithm is embedded inside a global constraint for total weighted completion time which is used to model and solve different optimization problems.

The remainder of this paper is organized as follows. In the next section, we introduce the notation used in the paper. In Section 3, we review the related literature. This is followed by the presentation and analysis of the proposed constraint propagation algorithm (Section 4). In Section 5, we turn to the empirical evaluation of the constraint on single-machine scheduling problems. Section 6 presents the modelling and solving of the multi-machine job shop scheduling problem using the constraint, including further experimental results. Section 7 discusses extensions and future work. Finally, in Section 8 we conclude.

2 Definitions and Notation

This paper introduces the COMPLETION global constraint for the total weighted completion time of activities on a unary resource. Formally, let there be given a set of n activities, A_i, to be executed without preemption on a single, unary capacity resource. Each activity is characterized by its processing time, p_i, and a non-negative weight, w_i. The start and end time variables of A_i will be denoted by S_i and C_i, respectively, and the constraint $C_i = S_i + p_i$ must hold. When appropriate, we call the current lower bound on a start time variable S_i the *release time* of the activity, and denote it by r_i. The total weighted completion time of the activities will be denoted by C. We assume that all data are integral. Thus, the constraint that enforces $C = \sum_i w_i(S_i + p_i)$ on activities takes the following form.

$$\text{COMPLETION}([S_1, ..., S_n], [p_1, ..., p_n], [w_1, ..., w_n], C)$$

Throughout this paper, we assume that p_i and w_i are constants. In applications where this assumption cannot be made, the lower bounds can be used during the propagation.

We will propose a propagation algorithm that filters the domains of the S_i variables (because in our case, domain filtering has the same complexity as adjusting only the bounds). It tightens only the lower bound on C, which is sufficient in applications where the cost is to be minimized.[1] The minimum and maximum values in the current domain of a variable X will be denoted by $\check{X}$ and $\hat{X}$, respectively.

Our algorithm will exploit preemptive relaxations. In a given preemptive schedule, an *activity fragment* is a maximal portion of an activity that is processed without interruption. We extend the above notation to activity fragments as well. Given a fragment α of activity A_i, $S(\alpha)$, $C(\alpha)$, and $p(\alpha)$ stand for the start time, end time, and the duration of α, respectively, with $S(\alpha) + p(\alpha) = C(\alpha)$ and $p(\alpha) \leq p_i$. Moreover, $r(\alpha) = r_i$ and $w(\alpha) = w_i \frac{p(\alpha)}{p_i}$.

We will make extensive use of the notion of the *mean busy time* of an activity, denoted by M_i. It stands for the average point in time at which the machine is busy processing A_i. In a fixed schedule, this is easily calculated by finding the mean of each time unit during which activity A_i executes. In *non-preemptive* scheduling problems $C_i = M_i + \frac{1}{2}p_i$ holds for each activity. In *preemptive* problems, only the inequality $C_i \geq M_i + \frac{1}{2}p_i$ holds. Note that M_i can be fractional even if all parameters in the problem are integer.

[1] In applications were it is important to tighten the upper bound of C, we propose to build a redundant model including a reversed schedule. The reversed schedule contains activities $A_i{}'$ with $S_i{}' = T - C_i$, where T is the length of the scheduling horizon. The reversed schedule is characterized by a total weighted completion time of $C' = \sum_i w_i C_i{}' = \sum_i w_i(S_i{}' + p_i) = \sum_i w_i(T - C_i + p_i) = nT + \sum_i w_i p_i - C$. Hence, posting a COMPLETION constraint on this reversed schedule tightens the upper bound of C via this channeling constraint.

3 Related Literature

The complexity, approximability, and algorithmic aspects of total weighted completion time scheduling problems have been studied extensively. Two of the problem variants most relevant for our contribution are the single and parallel machine versions with release dates. The classical scheduling notations for these problems are $1|r_i|\sum w_iC_i$ and $P|r_i|\sum w_iC_i$, respectively. Both variants are known to be NP-hard in the strong sense, even with uniform weights. Various polynomially solvable cases have been identified. For example, without release dates, ordering the activities according to the *Weighted Shortest Processing Time* rule, i.e., by non-decreasing p_i/w_i, yields an optimal solution. The preemptive version of the single machine problem with release dates and unit weights ($1|r_i, pmtn|\sum C_i$) is polynomially solvable using *Shortest Remaining Processing Time* rule, but adding non-uniform weights renders it NP-hard. A comprehensive overview of the complexity of related scheduling problems is presented in Chen et al. [9].

Linear programming (LP) and combinatorial lower bounds for the single machine problem have been studied and compared by Goemans et al. [18] and Dyer & Wolsey [14]. The *preemptive time-indexed formulation* corresponds to an assignment problem in which variables indicate whether activity A_i is processed at time t. In an alternative LP representation, the *non-preemptive time-indexed formulation*, variables express if activity A_i is completed at time t. Dyer & Wolsey [14] have shown that the latter relaxation is strictly tighter than the former. Since the number of variables in these formulations depend both on the number of activities and the number of time units, they can be solved in pseudo-polynomial time.

Another LP relaxation has been proposed by Schulz [33], using completion time variables. Subsequently, Goemans et al. [18] proved that this relaxation is equivalent to the preemptive time-indexed formulation, by showing that a preemptive schedule that minimizes the mean busy time yields the optimal solution for both relaxations. Moreover, this preemptive schedule can be found in $O(n \log n)$ time, where n is the number of activities. The authors also propose two randomized algorithms (and their de-randomized counterparts) to convert the preemptive schedule into a feasible solution of the original non-preemptive problem, $1|r_i|\sum w_iC_i$, and prove that these algorithms lead to 1.69 and 1.75-approximations, respectively. These results also imply a guarantee on the quality of the lower bound. Polynomial time approximation schemes for the single and parallel machines case, as well as for some other variants are presented in Afrati et al. [1]. The time complexity of the algorithm to achieve a $(1+\varepsilon)$-approximation for a fixed ε is $O(n \log n)$, but the complexity increases super-exponentially with ε.

Papers presenting complete solution methods for different versions of the total weighted completion time problem include earlier works by Belouadah et al. [7, 8], and more recent papers by Pan [28] for a single machine, Nessah et al. [26] for identical machines, and Della Croce et al. [11], as well as Akkan & Karabatı [2] for the two-machine flowshop problem. Most of these algorithms

make use of lower bounds similar to the ones discussed above, as well as a variety of dominance rules and customized branching strategies.

In constraint programming, the significance of cost-based global constraints for strong back propagation has been emphasized by Focacci et al. [16]. Cost-based constraints with effective propagation algorithms include the global cardinality constraint with costs [29], the minimum weight all-different constraint [34], the path cost constraint for traveling salesman problems [15], the cost-regular constraint [12], and the inequality-sum constraint [31]. The last work propagates objective functions of the form $\sum_i x_i$ where variables x_i are subject to binary inequality constraints of the form $x_j - x_i \leq c$. An efficient propagation algorithm based on Dijkstra's shortest path method is proposed. While the constraint can be applied to propagate the mean flow time or mean tardiness criterion in Simple Temporal Problems [10], it cannot exploit the information that activities must be performed on the same resource with limited capacity.

In the field of scheduling with "sum type" objective functions, Baptiste et al. [3] proposed a branch-and-bound method for minimizing the total tardiness on a single machine. While building the schedule chronologically, the algorithm makes use of constraint propagation to filter the set of possible next activities by examining how a given choice affects the value of the lower bound. Baptiste et al. [5] address the minimization of the number of late activities on a single resource, and generalize some well-known resource constraint propagation techniques for the case where there are some activities that complete after their due dates. The authors also propose propagation rules to infer if activities are on time or late, but the applicability of these inference techniques is restricted by the fact that they incorporate dominance rules that might be invalid in more general contexts. For propagating the weighted earliness/tardiness cost function in general resource constrained project scheduling problems, Kéri & Kis [21] define a simple method for tightening time windows of activities by eliminating values that would lead to solutions with a cost higher than the current upper bound.

4 Propagating Total Weighted Completion Time on a Unary Resource

Propagating the COMPLETION global constraint means removing the values from the domains of variables S_i $(i = 1, ..., n)$ and C that are inconsistent with the constraint. Deciding, in general, whether a given value is consistent with the COMPLETION constraint and the current domain bounds of the variables is equivalent to the decision version of the $1|r_i, d_i|\sum_i w_i C_i$ scheduling problem, which is known to be NP-complete [9]. This implies that, unless P=NP, there is no efficient algorithm that could find and remove *all* inconsistent domain values. The common approach in such cases is to consider a polynomially solvable relaxation of the property expressed by the constraint, and exploit the relaxation to prove the inconsistency of some of the domain values [30].

Our propagation algorithm for the COMPLETION constraint relies on a preemptive mean busy time scheduling problem investigated by Goemans et al. [18], which can be described as $1|r_i, pmtn|\sum_i w_i M_i$. This expression encodes a single-machine scheduling problem subject to release dates (r_i) with preemption allowed ($pmtn$) for minimizing the total weighted mean busy time, $\sum_i w_i M_i$. This relaxes the scheduling problem encoded in COMPLETION in three ways: (1) it disregards the deadlines of the activities; (2) it allows preemption; (3) instead of the activity completion times C_i, it considers the mean busy times M_i. All of these assumptions are necessary to achieve a relaxation for which a polynomial-time algorithm is known. Note that $\sum_i w_i C_i \geq \sum_i w_i M_i + \delta$ with $\delta = \frac{1}{2}\sum_i p_i$, hence, the optimal solution of the relaxed problem (plus the constant δ) indeed gives a lower bound on the original total weighted completion time problem.

In what follows, we first present an algorithm for solving the mean busy time problem, and then show how the solution of this relaxed problem can be exploited for propagating the COMPLETION constraint efficiently. Every algorithm will be illustrated on the sample problem in Figure 1.

	r_i	p_i	w_i	w_i/p_i
A_0	0	5	10	2
A_1	4	2	10	5
A_2	5	4	12	3
A_3	13	2	8	4

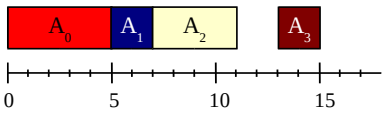

Figure 1: Left: The input data for the sample problem. Right: The optimal solution of the sample problem. The total weighted completion time is 372.

4.1 Computing a Lower Bound

The optimal solution of the preemptive mean busy time relaxation can be computed in $O(n \log n)$ time [18]. The algorithm maintains a priority queue of the activities sorted by non-increasing w_i/p_i. At each point in time, t, the queue contains the activities A_i with $r_i \leq t$ that have not yet been completely processed. Scheduling decisions must be made each time a new activity is released or an activity is completely processed. In either case, the queue is updated and a fragment of the first activity in the queue is inserted into the schedule. The fragment lasts until the next decision point. If the queue is empty, but there are activities not yet released, a gap is created. We represent gaps as *empty fragments*: fragments of an activity with a release date and a weight of 0 and with an infinite processing time. We also assume that the schedule ends with a sufficiently long empty fragment. Technically, the use of empty fragments is not an essential feature of our algorithm. However, they make our algorithm descriptions simpler, because they eliminate the need for differentiating the otherwise identical cases when an activity is followed by a gap or a fragment of another activity. Since there are at most $2n$ release time and activity completion events,

and updating the queue requires $O(\log n)$ time, the algorithm runs in $O(n \log n)$ time.

The optimal relaxed schedule for the sample problem is presented in Figure 2. The lower bound induced by this relaxed solution is 362. In the figure, fragments of activity A_i are denoted by $\alpha_i, \alpha_i', \alpha_i''$, etc. Empty fragments are named $\varepsilon, \varepsilon', \varepsilon''$, etc.

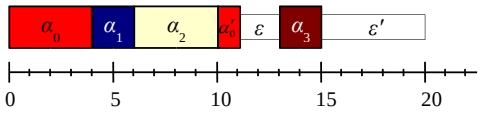

Figure 2: The solution of the relaxed problem, with objective value 295. As $\delta = 67$ for the problem, the lower bound derived from the relaxed solution is 362.

4.2 From a Lower Bound to Propagation – Direct Approach

The underlying idea of our constraint propagator is to compute the cost of the mean busy time relaxed problem *for each activity* A_i and *each possible start time* t of activity A_i, with the added constraint that activity A_i must start at time t. Such restricted problems will be denoted by $\Pi\langle S_i = t\rangle$. The solution cost of this relaxed problem (plus δ), denoted by $\check{C}\langle S_i = t\rangle = \sum_i w_i M_i + \delta$ is a valid lower bound on C in non-preemptive schedules where A_i starts at t. Therefore, we can exploit the following lemma to filter the domain of variable S_i.

Lemma 1 *If* $\hat{C} < \check{C}\langle S_i = t\rangle$*, then* t *can be removed from the domain of* S_i*.*

The lower bounding algorithm of Section 4.1 can easily be modified to compute $\check{C}\langle S_i = t\rangle$ by assigning $r_i = t$ and $w_i = \infty$. This gives activity A_i the largest w_i/p_i ratio among all the activities, ensuring that it starts at t and is not preempted. Relaxed solutions for various restrictions on the sample problems are presented in Figure 3. The last diagram in the figure displays $\check{C}\langle S_i = t\rangle$ as a function of t. As it will be shown later, this function is piecewise linear, and it can have an arbitrary number and order of increasing and decreasing sections.

4.3 From a Lower Bound to Propagation – Recomputation Approach

Obviously, it would be inefficient to re-solve the $\Pi\langle S_i = t\rangle$ problem separately for each possible value of t using the direct approach. Instead, we introduce

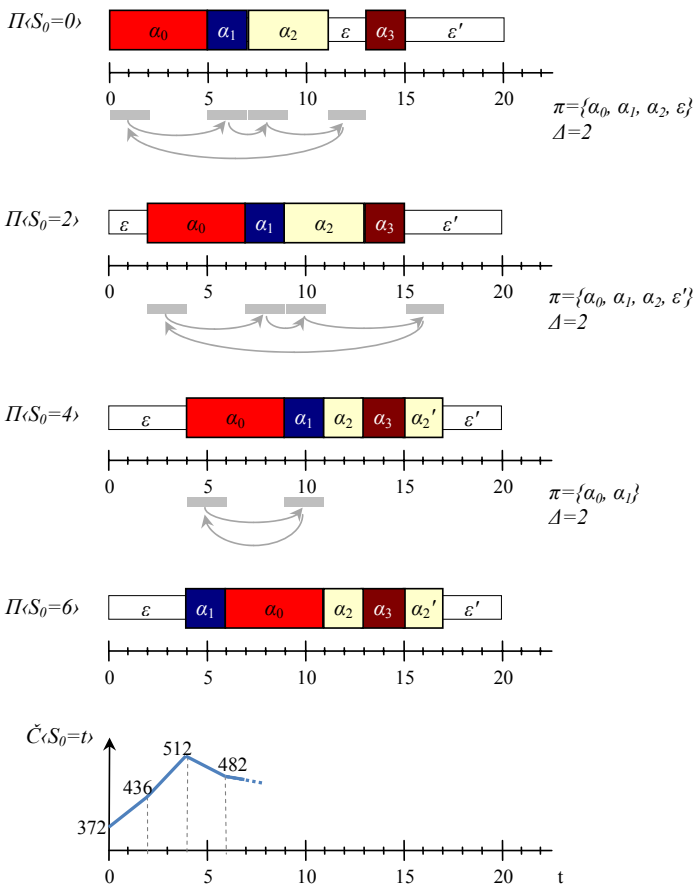

Figure 3: Relaxed solutions of various restricted problems, with corresponding restrictions displayed on the left. Between each pair of subsequent schedules, the transformation that maps one relaxed solution to the subsequent one are displayed. The parameters π and Δ of the transformation are shown on the right. The diagram at the bottom shows $\check{C}\langle S_i = t\rangle$ as a function of t.

a quick recomputation method for transforming an initial preemptive schedule into relaxed solutions for the relevant start time assignments. The relevant start times are described by an increasing series $(t_0, t_1, ..., t_L)$ with $t_0 = \check{S}_i$ and $t_{L-1} < \hat{S}_i \leq t_L$. The series will be chosen in such a way that $\check{C}\langle S_i = t\rangle$ changes linearly between the consecutive elements, i.e., for all $j \in [0, L-1]$ and $t \in [t_j, t_{j+1}]$

$$\check{C}\langle S_i = t\rangle \;=\; \frac{t_{j+1} - t}{t_{j+1} - t_j}\check{C}\langle S_i = t_j\rangle \;+\; \frac{t - t_j}{t_{j+1} - t_j}\check{C}\langle S_i = t_{j+1}\rangle.$$

The t_i values correspond to the break-points of the function $\check{C}\langle S_i = t\rangle$ over different values of t. The recomputation approach first builds a relaxed solution for the case that A_i starts at t_0 using the direct approach. Note that we are unable to determine all the values t_j at once, before the recomputations. Instead, we calculate them iteratively: t_{j+1} is computed based on the relaxed solution for t_j.

The underlying ideas of the recomputation approach are illustrated in Figure 3. The start time domain of activity A_0 (equivalent to fragment α_0) is being filtered, and the relevant time points t_j are 0, 2, 4, 6, etc. The figure displays the optimal solutions of the relaxed problems corresponding to these time points. Between each pair of subsequent schedules, the gray arrows show how (parts of) the fragments have to be moved to transform one relaxed solution to the subsequent one. For example, the transformation from $\Pi\langle S_0 = 0\rangle$ to $\Pi\langle S_0 = 2\rangle$ can be performed by moving the first 2 units of the fragments as follows: fragment α_0 is split, and its 2 first units are moved later in the schedule to start at time 5 (since the relocated fragment is continuous with the rest of α_0, they are merged back). The 2-unit long fragment α_1, which initially started at time 5, is moved later to start at time 7. In turn, α_2 is split, its first 2 units are moved later, and it is re-merged. Finally, the 2-unit long empty fragment ε is moved from time 11 to time 0, which was the initial start time of α_0. This move closes the cycle of movements. The second transformation, i.e., from $\Pi\langle S_0 = 2\rangle$ to $\Pi\langle S_0 = 4\rangle$, shows an example of splitting a fragment and not merging its parts back: this is the case for fragment α_2. In the sequel we give a formal characterization of these transformations.

Each recomputation step is performed by one call to a function called RecomputeSchedule. This function starts with determining the "structure" of the transformation it has to make on the current preemptive schedule σ. This structure is represented as a list $\pi = (\alpha_0 \equiv A_i, \alpha_1, ..., \alpha_K)$, where each α_k is an activity fragment in σ. The list π is constructed as follows:

1. Initialize with $\pi := (\alpha_0 = A_i)$ and $k := 0$;
2. Find the first fragment α to the right of α_k in the schedule such that $w(\alpha)/p(\alpha) < w(\alpha_k)/p(\alpha_k)$; append this α to the end of π, so in the continuation this fragment will be called α_{k+1}; $k := k + 1$;
3. If $r(\alpha_k) > t_j$ then goto step 2. Otherwise stop.

Note that the iteration stops when a fragment α_k is reached that can be moved earlier to the initial location of α_0. The iteration always terminates, because the empty fragment at the end of the schedule always satisfies the stopping condition. Also recall that $w(A_i) = \infty$ is assumed. Next, RecomputeSchedule sets $t_{j+1} = t_j + \Delta$ to be the maximum value such that a transformation with structure π leads to an optimal schedule for $\Pi\langle S_i = t_{j+1}\rangle$. For that purpose, Δ will be set to

$$\Delta \;=\; \min \begin{cases} \min_{k\in[1,K-1]}\{p(\alpha_k) \mid C(\alpha_k) \neq S(\alpha_{(k+1) \mod (K+1)})\} \\ \\ \min_{k\in[1,K-1]}\{r(\alpha_k) - S(\alpha_0)\} \end{cases}$$

Intuitively, the first line of the expression states that in general, Δ cannot be greater than the processing time of the fragments moved. Fragments α_k with $C(\alpha_k) = S(\alpha_{(k+1) \mod (K+1)})$ are an exception: we will show that we can manage fragments shorter than Δ in this case. The second line of the condition is necessary for the optimality of the transformed solution: if this condition on α_m bounds Δ (and this is the only bounding condition), then recomputing the optimal schedule for the next time point, i.e., from t_{j+1} to t_{j+2}, requires a substantially different transformation structure, namely $\pi' = (\alpha_0, \alpha_1, ..., \alpha_m)$. Accordingly, the stopping condition of the iteration to build π' will be hit at α_m, because this fragment is the first that can be moved in the place of α_0.

The values of π and Δ for several recomputation steps in the sample problem are displayed in the right hand side of Figure 3. Having parameters π and Δ computed, RecomputeSchedule sets $t_{j+1} := t_j + \Delta$ and transforms σ by performing the following steps on every fragment $\alpha_k \in \pi$:

1. If α_k is a *long* fragment, i.e., $p(\alpha_k) \geq \Delta$, then
   - If $p(\alpha_k) > \Delta$ then split α_k into two fragments. In the continuation, let α_k denote the first, Δ-long fragment;
   - Move α_k to start at $S'(\alpha_k) = S(\alpha_{k+1 \mod K+1})$;
2. Else α_k is a *short* fragment, i.e., $p(\alpha_k) < \Delta$;
   - Move α_k to start at $S'(\alpha_k) = S(\alpha_k) + \Delta$;
3. If after the move, α_k is placed next to another fragment of the same activity, then merge these fragments.

Lemma 2 *Given an optimal schedule σ for $\Pi\langle S_i = t_j\rangle$, the application of* RecomputeSchedule *converts it into an optimal schedule for $\Pi\langle S_i = t_{j+1}\rangle$*

Proof: First, we show that the transformed schedule is feasible. The selection of π and Δ ensures that π consist of sections of zero or more short fragments followed by one long fragment, such that $C(\alpha_k) = S(\alpha_{(k+1) \mod (K+1)})$ holds for the consecutive elements of the section, see Figure 4. Within each section, the transformation shifts the short fragments to the right by Δ units. It potentially splits the long fragment into two, and moves the created Δ-long fragment to the start of the next section. This guarantees that the moved sections will not overlap with each other or with fragments not moved. Also, release times will be respected.

To prove that the schedule is optimal, let us define a *bad pair* as a pair of activity fragments, α_{k1} and α_{k2}, such that $\frac{w(\alpha_{k1})}{p(\alpha_{k1})} < \frac{w(\alpha_{k2})}{p(\alpha_{k2})}$ and $r(\alpha_{k2}) <$

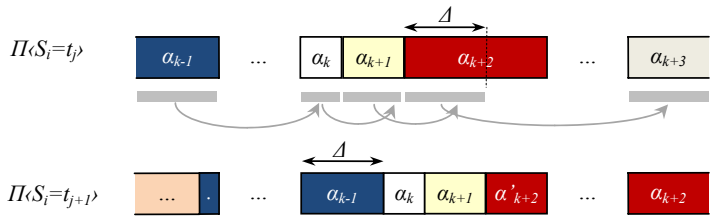

Figure 4: A section of the preemptive schedule moved in a transformation. Each section consists of zero or more *short* fragments (α_k and α_{k+1} in this example) and one *long* fragment (α_{k+1}). We exploit this structure to prove that the transformed schedule is feasible.

$C(\alpha_{k1}) \leq S(\alpha_{k2})$ hold. It is easy to see that a feasible solution of a relaxed problem is optimal if and only if it does not contain a bad pair, since the solution can be improved by (and only by) swapping a bad pair.

The initial schedule σ for the problem $\Pi\langle S_i = t_j\rangle$ does not contain a bad pair, because it is optimal for that problem. The transformed problem $\Pi\langle S_i = t_{j+1}\rangle$ differs from that initial problem only in having parameter $r_i = t_{j+1}$ instead of $r_i = t_j$. This means that the initial σ does not contain any bad pair for the transformed problem, either. Furthermore, RecomputeSchedule does not create bad pairs, because

- within the $\alpha_k \rightarrow \alpha_{k+1}$ $(k = 0, ..., K-1)$ move, α_k is swapped later across fragments that have a higher $\frac{w}{p}$ ratio than α_k.
- the $\alpha_K \rightarrow \alpha_0$ move swaps α_K earlier, and this fragment has the highest $\frac{w}{p}$ ratio among all the fragments that can be scheduled in the interval $[S(\alpha_0), S(\alpha_0) + \Delta]$.

□

Lemma 3 *The lower bound cost $\check{C}\langle S_i = t\rangle$ changes linearly for $t \in [t_j, t_{j+1}]$.*

Proof: Lemma 2 remains true if RecomputeSchedule uses a value of Δ lower than proposed above, which implies that a transformation with structure π leads to an optimal schedule for $\Pi\langle S_i = t\rangle$ with $t_j < t \leq t_{j+1}$. On the other hand, the application of RecomputeSchedule increases the lower bound cost by

$$\Delta \left(\sum_{k=0}^{K-1} (S(\alpha_{k+1}) - S(\alpha_k)) \frac{w(\alpha_k)}{p(\alpha_k)} \; - \; (S(\alpha_K) - S(\alpha_0)) \frac{w(\alpha_K)}{p(\alpha_K)} \right).$$

Since this expression is proportional to Δ, the lower bound cost changes linearly in $[t_j, t_{j+1}]$. □

Finally, we present how our propagation algorithm filters the variable domains. According to Lemma 1, if both $\check{C}\langle S_i = t_j\rangle > \hat{C}$ and $\check{C}\langle S_i = t_{j+1}\rangle > \hat{C}$ hold then the complete interval $[t_j, t_{j+1}]$ can be removed from the domain of S_i. If only the first condition holds, then the first, proportional part of the interval, i.e.,

$$\left[t_j,\ t_j + \left\lceil (\Delta \frac{\hat{C} - \check{C}\langle S_i = t_j\rangle}{\check{C}\langle S_i = t_{j+1}\rangle - \check{C}\langle S_i = t_j\rangle} \right\rceil - 1\right]$$

is removed. If only the second condition holds, then the last, proportional part of the interval, i.e.,

$$\left[t_j + \left\lfloor \Delta \frac{\hat{C} - \check{C}\langle S_i = t_j\rangle}{\check{C}\langle S_i = t_{j+1}\rangle - \check{C}\langle S_i = t_j\rangle} \right\rfloor + 1,\ t_{j+1}\right]$$

is removed from the domain of S_i. No removal is made otherwise. The new lower bound on the total weighted completion time is the best bound computed over the run of the propagator with different activities, i.e.,

$$\check{C}' \ = \ \max(\check{C},\ \max_i \min_t \check{C}\langle S_i = t\rangle).$$

4.4 Overall Algorithm and Its Complexity

The pseudo-code of the proposed constraint propagation algorithm is presented in Figure 5. The algorithm is implemented in two procedures: the main procedure PROPAGATE() calls procedure RECOMPUTESCHEDULE(σ, t, c) iteratively, whose parameters are a schedule σ, an integer t that represents the start time of the activity investigated, and a real number c, which equals the cost of σ.

Procedure PROPAGATE performs propagation using the recomputation approach for each activity A_i separately (lines 19-35). It initializes by determining the earliest start time t of A_i, building a schedule σ for $\Pi\langle S_i = t\rangle$, computing its cost $c = \check{C}\langle S_i = t\rangle$, and defining the variable $c_{\min}$ that will be used to tighten the lower bound of C. Note that the direct approach to build σ (line 21) is a standard preemptive scheduling algorithm, therefore it is not presented here in detail. Then, the algorithms iteratively recomputes schedule σ by calling RECOMPUTESCHEDULE (lines 24-34). Note that RECOMPUTESCHEDULE updates the values of σ, t, and c. At this point, variables t_{prev} and t store the values of t_j and t_{j+1}, respectively, while c_{prev} and c contain the corresponding lower bound costs. Finally, the induced domain filtering is executed for the cases that the complete interval $[t_{prev}, t]$, or its first or second part has to be filtered out (lines 28-33). Finally, the lower bound on C is tightened (line 35).

Procedure RECOMPUTESCHEDULE starts by building the list π that describes the structure of the transformation to be performed on σ (lines 3-7). Parameter Δ is computed in lines (8-9). The transformation of σ is carried out for each fragment in π in lines (10-15). The procedure ends with updating t and c (lines 16-17).

```
1 PROCEDURE RecomputeSchedule(σ, t, c)
2   LET A_i := the activity that starts at t
3   LET π := (A_i)
4   WHILE r(last(π)) > t OR size(π) = 1
5      LET α := the leftmost fragment in σ with
6         S(α) > S(last(π)) and w(α)/p(α) < w(last(π))/p(last(π))
7      Append α to π
8   LET Δ := min(min_{k∈[1,K]}{p(α_k) | S(α_{k+1 mod K+1}) ≠ S(α_k) + p(α_k)},
9                min_{k∈[1,K−1]}{r(α_k) − S(α_0)})
10  FORALL k IN [1, size(π)]
11     IF p(α_k) > Δ THEN
12        LET α_k be the first, Δ-long fraction of α_k
13     LET S(α_k) := max(S(α_k) + Δ, S(α_{k+1 mod K+1}))
14     IF α_k is preceded/succeeded by a fragment of the same activity THEN
15        Merge these fragments
16  LET t := t + Δ
17  LET c := c + Δ ( Σ_{k=0}^{K−1} (S(α_{k+1}) − S(α_k)) w(α_k)/p(α_k)  −  (S(α_K) − S(α_0)) w(α_K)/p(α_K) )


18 PROCEDURE Propagate()
19  FORALL activity A_i
20     LET t := Š_i
21     LET σ := schedule computed by the direct procedure for Π⟨S_i = t⟩
22     LET c := cost(σ)
23     LET c_min := c
24     WHILE t < Ŝ_i
25        LET t_prev := t
26        LET c_prev := c
27        RecomputeSchedule(σ, t, c)
28        IF c_prev > Ĉ and c > Ĉ THEN
29           Remove [t_prev, t] from domain(S_i)
30        ELSE IF c_prev > Ĉ THEN
31           Remove [t_prev, t_prev + ⌈(Δ (Ĉ − C_1)/(C_2 − C_1)⌉ − 1] from domain(S_i)
32        ELSE IF c > Ĉ THEN
33           Remove [t_prev + ⌊Δ (Ĉ − C_1)/(C_2 − C_1)⌋ + 1, t] from domain(S_i)
34        LET c_min := min(c_min, c)
35     LET Č := max(Č, c_min)
```

Figure 5: Pseudo-code of the constraint propagation algorithm.

To calculate the time complexity of the algorithm, we need an upper bound on the number of recomputation cycles required to perform the propagation on one activity (lines 24-34).

Lemma 4 *The number of recomputation steps required per activity is at most* $2n^2$.

Proof: Let us distinguish two types of recomputation steps based on how the size of time interval, $\Delta = (t_{j+1} - t_j)$, covered by one recomputation step is bound. If the condition in line 8 of Figure 5 bounds Δ, then we call it an A-type step. If the condition in line 9 bounds Δ, then we call it a B-type step. If lines 8 and 9 are equally bounding, then it is considered to be an A-type step.

Furthermore, let the number of inversions $I(\sigma)$ denote the number of fragment pairs $(\alpha_{k1}, \alpha_{k2})$ in the preemptive schedule σ such that $S(A_i) \leq S(\alpha_{k1}) < S(\alpha_{k2})$ and $\frac{w(\alpha_{k1})}{p(\alpha_{k1})} < \frac{w(\alpha_{k2})}{p(\alpha_{k2})}$. Since there are at most $2n$ fragments in σ, $I(\sigma)$ is at most $\sum_{j=1}^{2n-1} j = 2n^2 - n$. Observe that $I(\sigma)$ is strictly decreased by A-type recomputation steps, while it is not affected by B-type steps. Therefore, the number of A-type recomputation steps is at most $2n^2 - n$. On the other hand, the number of B-type steps is not greater than the number of different activity release times, which is at most n. □

Therefore, in procedure PROPAGATE, the outer loop (19-35) is repeated n times, while the inner loop (24-34) is repeated at most $n \cdot 2n^2 = 2n^3$ times. Assuming that a range removal from a variable domain can be performed in constant time, the complexity of the inner loop is determined by the procedure RECOMPUTESCHEDULE. Since there are at most $2n$ fragments in a preemptive schedule, the size of π is also at most $2n$, therefore one call to RECOMPUTESCHEDULE takes $O(n)$ time, which results in an overall complexity of $2n^3 \cdot O(n) = O(n^4)$. Hence, the complexity of the inner loop dominates the complexity of building σ in line (21) by the direct approach, which takes only $O(n \log n)$ time in each cycle, i.e., $O(n^2 \log n)$ in total. This means that the worst-case time complexity of one run of the proposed propagation algorithm is $O(n^4)$.

We note that in experiments, the practical behavior of the propagator differed from what is suggested by the above complexity considerations. Computing the initial preemptive schedules (line 21) took approximately 85% of the CPU time, while recomputations by RECOMPUTESCHEDULE took only 15%, even if the latter has the higher worst case complexity. This was due to the low number of recomputation steps required: on 40-activity instances we typically observed 4-6 steps (see Section 5 for details).

4.5 Discussion of the Implementation Details

While the pseudo-code depicted in Figure 5 captures the essence of the proposed propagation algorithm, its performance in applications depends also on various implementation details. In particular, we applied the following techniques to speed up the propagation algorithm:

- In constraint-based scheduling it is common to use so-called chronological solution strategies that build the schedule from its start towards its end. This implies that the propagator will often face situations where the start times of the activities scheduled first are bound. During propagation we ignore this bound beginning section of the schedule, and only consider its

known cost.

- Relaxed solutions are saved during each run of the propagator; filtering the domain of S_i is attempted again only after $\check{S}_j,\ j \neq i$ have increased, or $\hat{C}$ has decreased sufficiently to modify the relaxed solutions.

In contrast to the proposed propagation algorithm, most constraint propagators in scheduling apply bounds consistency instead of domain consistency. Despite this, our experiments showed that it is slightly faster to remove infeasible values from inside the domains for our COMPLETION propagator, since this results in a somewhat lower number of calls to this computationally expensive propagator. Nevertheless, the difference between the two domain reduction methods was minor, and a re-implementation in a different solver may bring different results.

5 Applying COMPLETION to Single-Machine Scheduling

In order to evaluate the proposed propagation algorithm in relative isolation (i.e., separated from more complex problems in which we are eventually interested in embedding it), we ran computational experiments to measure its performance on the single-machine total weighted completion time problem with release times, $1|r_i|\sum w_i C_i$.

The proposed algorithms have been implemented as a global constraint propagation algorithm in C++ and embedded into ILOG Solver and Scheduler versions 6.1. We set the resource capacity enforcement to use the following inference techniques provided by the ILOG libraries: the precedence graph, the disjunctive constraint, and edge finding [32]. We used an adapted version of the *SetTimes* branching heuristic [24, 32]: in each search node from the set of not yet scheduled (and not postponed) activities, the heuristic selects the activity that has the smallest earliest start time (EST) and then breaks ties by choosing the activity with the highest w_i/p_i ratio. Two branches are created according to whether the start time of this activity is bound to its EST or it is postponed. If any postponed activity can end before the EST of the selected activity, the search backtracks as no better schedule exists in the subtree. An activity is no longer postponed when constraint propagation increases its EST.

We compared the performance of three different models. The first used the standard weighted-sum (WS) constraint for propagating the optimization criterion. The second calculated the lower bound presented in Section 4.1 (WS+LB) at each node and used it for bounding. The third model made use of the proposed COMPLETION constraint.

These algorithms were tested on benchmark instances from the online repository [27], which were also used in Pan & Shi [28] and were generated in a similar fashion as in several earlier works [7, 20, 35]. The repository contains 10 single-machine problem instances for each combination of parameters n and R,

where n denotes the number of activities and takes values between 20 and 200 in increments of 10, while R is the relative range of the release time, chosen from $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.5, 1.75, 2.0, 3.0\}$. Activity durations are randomly chosen from $U[1, 100]$, weights from $U[1, 10]$, and release times from $U[0, 50.5nR]$, where $U[a, b]$ denotes the integer uniform distribution over interval $[a, b]$. Out of these 1900 instances in total, we ran experiments on the 300 instances with $n \leq 70$ and every second value of parameter R. The experiments were run on a 1.86 GHz Pentium M computer with 1 GB of RAM, with a time limit of 120 CPU seconds.

n	R	WS			WS+LB			COMPLETION		
		Solved	Nodes	Time	Solved	Nodes	Time	Solved	Nodes	Time
20	0.2	-	-	-	10	881	0.00	10	47	0.00
	0.6	2	1842024	62.00	10	2023	0.00	10	98	0.00
	1.0	10	114359	3.60	10	1788	0.10	10	109	0.00
	1.5	10	2518	0.00	10	224	0.00	10	67	0.00
	2.0	10	140	0.00	10	102	0.00	10	51	0.00
30	0.2	-	-	-	10	3078	0.10	10	116	0.00
	0.6	-	-	-	10	21455	3.20	10	424	0.00
	1.0	5	845422	58.60	9	115187	17.00	10	7127	2.90
	1.5	10	21555	1.30	10	1932	0.10	10	189	0.00
	2.0	10	2633	0.10	10	863	0.00	10	160	0.00
40	0.2	-	-	-	10	16235	2.50	10	263	0.10
	0.6	-	-	-	8	300649	58.25	9	4909	4.88
	1.0	2	164455	30.50	4	56536	11.75	10	27717	15.60
	1.5	10	40160	2.80	10	9111	1.30	10	602	0.20
	2.0	10	60602	3.70	10	3731	0.20	10	379	0.00
50	0.2	-	-	-	8	163551	37.62	10	1690	2.80
	0.6	-	-	-	-	-	-	8	32709	56.75
	1.0	-	-	-	-	-	-	2	27386	41.00
	1.5	3	92954	8.33	7	117120	27.00	10	12358	15.3
	2.0	8	36056	3.37	9	7050	1.22	10	1535	0.80
60	0.2	-	-	-	2	354131	97.00	10	17553	35.60
	0.6	-	-	-	-	-	-	-	-	-
	1.0	-	-	-	-	-	-	-	-	-
	1.5	-	-	-	4	199608	61.50	8	44098	33.00
	2.0	4	120345	12.75	9	49433	11.55	10	5433	4.40
70	0.2	-	-	-	-	-	-	6	3323	12.66
	0.6	-	-	-	-	-	-	-	-	-
	1.0	-	-	-	-	-	-	1	12899	48.00
	1.5	-	-	-	2	191104	65	3	9147	14.66
	2.0	3	134782	19.33	7	112472	35.71	9	14714	13.22

Table 1: Experimental results: number of instances solved (Solved), mean number of search nodes (Nodes) and mean search time (Time) for the different versions of the branch and bound. A '-' indicates that none of the instances were solved to optimality within the time limit. The means are computed only on the instances that the algorithm solved.

The experimental results are presented in Table 1, where each row contains combined results for the 10 instances with the corresponding number of activities, n, and release time range, R. For each of the three models, the table displays the number of the instances that could be solved to optimality (column *Solved*), the average number of search nodes (*Nodes*), and average search time in seconds (*Time*). The average is computed only on the instances that the algorithm solved. The results show that the classical WS model fails on some

of the 20-activity instances, whereas the COMPLETION constraint enabled us to solve–with one exception–all problems with at most 40 activities, and also performed well on the 50-activity instances. COMPLETION constraint adds significant pruning strength to the constraint-based approach: the COMPLETION model required up to 2 orders of magnitude less search node to find optimal solutions than WS. The pruning not only paid off in the means of the number of search nodes, but also decreased solution time on *every instance*, compared to both other models. Note that in some rows of Table 1, greater average computation times are displayed for COMPLETION than for WS+LB. This is purely because the average is computed over different sets of solved instances.

The results illustrate that instances with release time range $R \in \{0.6, 1.0\}$ are significantly harder for WS+LB and COMPLETION than other instances. This is explained by the fact that with $R \ll 1$, activities in the second half of the schedule can simply be ordered by non-increasing w_i/p_i. On this section of the schedule, the lower bound is exact and our propagator achieves completeness. On the other hand, $R \gg 1$ leads to problems where only a few activities can be chosen for scheduling at any point in time, which makes the instance easily solvable as well.

It is instructive to compare our results to state-of-the-art techniques for solving the single-machine problem. Our algorithms compare favorably to existing LP-based methods [35] that are able to solve instances with at most 30 to 35 activities, and earlier branch-and-bound methods [7], which solve problems with 40 to 50 activities. On the other hand, our approach is outperformed by two different, recent solution methods. One is a branch-and-bound algorithm combined with powerful dominance rules, constraint propagation, and no-good recording by Jouglet et al. [20], which has originally been developed for solving the more general total weighted tardiness problem. The other is a dynamic programming approach enhanced with dominance rules and constraint propagation by Pan & Shi [28]. These two approaches are able to solve instances with up to 100 and 200 activities, respectively. A part of the contributions of the previous work, especially the strong dominance rules [20, 28] are orthogonal and complementary to the COMPLETION constraint. We expect that combining such approaches with the COMPLETION constraint would lead to further performance improvements on the single-machine problem. However, as noted, our main aim is to address more complex problems where the sum type optimization criteria under capacity constraints appears as a sub-problem rather than to solve the single-machine problem itself. In the next section, we turn to one such problem, the job shop scheduling problem. In Section 7, we note preliminary results on other such problems.

6 Applying the COMPLETION Constraint to the Job Shop Scheduling Problem

An $n \times m$ job shop scheduling problem (JSP) has n jobs each composed of m completely ordered activities. Each activity requires exclusive use of one resource during its execution. The duration and the resource for each activity are given and may be different from that of other activities. Often, as in the problems studied here, a different resource is specified for each activity in a job. An activity cannot start until the activity immediately preceding it in the same job has finished. The standard JSP decision problem asks if, for a given makespan, D, all activities can finish by D. This is a well-known NP-complete problem [17]. It is not uncommon to solve the optimization version of the JSP with the goal of minimizing makespan, another metric such as sum of earliness and tardiness [6], or, in the present case, the sum of the weighted completion time of all jobs. More formally, given a set of jobs J and a weight, $w_j, j \in J$, our goal is to find a start time for each activity such that:

- no resource executes more than one activity at a time
- each activity starts after its preceding activity in the job-order ends
- $\sum_{j \in J} w_j C_{E_j}$ is minimized, where E_j is the last activity in job j.

We study square JSPs (i.e., $n = m$) where each job has exactly one activity on each resource.

6.1 From Job Weights to Activity Weights

Applying the COMPLETION constraint to the JSP is straightforward as the resources have unary capacity. The only complication is that the constraint uses a weight on each activity and the JSP has weights on each job. Therefore, we need to define a mapping from job weight to activity weight. We investigate three such mappings here.

1. *last*: The obvious approach is to assign the job weight to the last activity in each job and to assign all other activities a weight of zero. We then place a COMPLETION constraint on each resource that has a non-zero weight activity and the total weighted completion time is the sum of the C values of each COMPLETION constraint.

   This approach has two main drawbacks. First, a computationally expensive COMPLETION constraint is placed on each resource and the propagation algorithm is executed whenever there is a change in an activity time window. Second, the COMPLETION constraint makes inferences based on a relaxation that focuses on the interaction among activities on the same resource. Clearly, this interaction is not captured when the weighted activities are on different resources. In the extreme, the last activity in each job may be the only weighted activity on a resource. Under

such circumstances, the COMPLETION constraint is not able to make any inferences stronger than the simple weighted-sum constraint.

2. *busy*: To address the weaknesses of *last*, before solving we identify the most loaded resource, i.e., the "busy" resource, by summing the durations of the activities on each resource and selecting the resource with highest sum. The weight of each job is assigned to the activity of the job that is processed on the busy resource. All other activities have a weight of zero. A single COMPLETION constraint can then be posted on the busy resource. To calculate the total weighted completion time, we need to correct for the fact that the weighted activity is not necessarily the last activity in the job.

   Formally, as above, let E_j be the last activity in job j and let B_j be the single weighted activity in job j. Our optimization function is then: $C + \sum_{j \in J_i} w_j(C_{E_j} - C_{B_j})$ where C is the cost variable associated with the COMPLETION constraint.

3. *each*: The intuition for *busy* is that the important interactions among jobs are most likely to be observed on the busiest resource. However, this is clearly a heuristic as we do not necessarily identify the truly most constrained resource and it is likely that different resources will result in different and complementary inferences at different search states. Therefore, our final mapping places a COMPLETION constraint on each resource, correcting, as in *busy*, if an activity on a resource is not the last in the job.

   Formally, our optimization function is: $\max_i(C_i + \sum_{j \in J} w_j(C_{E_j} - C_{act_{ij}}))$ where i indexes the resources, C_i is the cost variable associated with the COMPLETION constraint on resource i, and act_{ij} is the last activity in job j that executes on resource i.

6.2 Experimental Details

To test the effectiveness of the COMPLETION constraint, we compare it against the standard weighted-sum, *WS*, form of the optimization function. For completeness, we also run WS with *last*, *busy*, and *each* weight allocations.

We experiment with two styles of search: chronological backtracking and randomized restart. For chronological backtracking (i.e., depth-first search) we use the same customized version of the SetTimes heuristic as applied for the single-machine problems (see Section 5).

For randomized restart, the limit on the number of backtracks before restarting evolves according to the universal limit developed by Luby et al. [25]. The heuristic is a randomized version of the customized SetTimes heuristic used above. Again, the set of non-postponed activities with minimum start time are selected. One activity from this set is randomly chosen by a biased roulette wheel weighted by the ratio of activity weight to duration. Higher weight, lower duration activities have a higher probability of being selected.

Two sets of 10×10 JSP problems are used. Initially ten makespan-minimization instances were generated with an existing generator [36]. The machine routings were randomly generated and the durations were randomly drawn from $U[1, 99]$. These instances were transformed into two sets of total weighted completion time problems with the only difference being the range of job weights: the first set has job weights randomly drawn from the interval $U[1, 9]$ and the second set has job weights randomly drawn from the interval $U[1, 99]$.

The models and algorithms were implemented in ILOG Scheduler 6.3. Experiments were run on 2GHz Dual Core AMD Opteron 270 with 2Gb RAM running Red Hat Enterprise Linux 4. We used an overall time-out of 1200 CPU seconds for each run. The randomized restart results are the mean over 10 independent runs.

6.3 Results

For this experiment, the dependent variable is the mean relative error (MRE) relative to the best solution known for the problem instance. The MRE is the arithmetic mean of the relative error over each run of each problem instance:

$$MRE(a, K, R) = \frac{1}{|R||K|} \sum_{r \in R} \sum_{k \in K} \frac{c(a, k, r) - c^*(k)}{c^*(k)} \tag{1}$$

where K is a set of problem instances, R is a set of independent runs with different random seeds, $c(a, k, r)$ is the lowest cost found by algorithm a on instance k in run r, and $c^*(k)$ is the lowest cost known for k. As these problem instances were generated for this experiment, the best-known solution was found either by the algorithms tested here or by variations used in preliminary experiments.

6.3.1 Chronological Backtracking

Figures 6 and 7 display the results for the two problem sets. The results fall into two groups: (1) the weighted-sum constraint with any weight mapping plus ⟨COMP, LAST⟩, and (2) ⟨COMP, BUSY⟩ and ⟨COMP, EACH⟩. The approaches in the latter group are clearly superior to those in the former. Overall, ⟨COMP, LAST⟩ appears to be, marginally, the worst performing approach. We interpret this to be an indication that the drawbacks of *last* noted above have a clear impact: on widely spread final activities, the COMPLETION propagation makes few inferences beyond what weighted-sum can do while incurring a much higher computational expense. The weighted-sum approaches are all close-to-identical. This is as we expected given that weighted-sum does not take into account any interactions between weighted activities.

For any CPU time cut-off less than 1200 seconds, ⟨COMP, EACH⟩ delivers the lowest MRE of any algorithm on both problem sets followed by ⟨COMP, BUSY⟩. The benefits appear modest given the formats of the graphs (i.e., a 0.01 decrease in MRE). However, another way to look at these results is based on the CPU time taken to achieve a particular level of MRE. In Figure 6 ⟨COMP, EACH⟩ takes less than 300 CPU seconds to achieve the same MRE as the

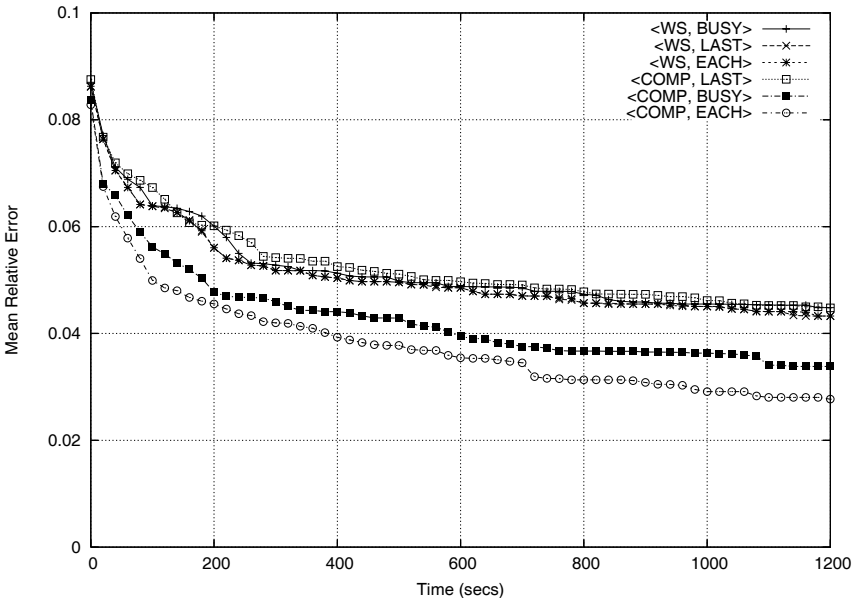

Figure 6: The mean relative error for different propagation techniques with chronological backtracking for the problems with job weight uniformly drawn from $U[1, 9]$. COMP: uses the COMPLETION constraint, WS: uses the weighted-sum constraint, LAST: places the job weight on the last activity in each job, BUSY: places the job weight on the activity of the job that is on the busiest resource, EACH: places the job weight on all activities of the job.

WS approaches achieve in 1200 CPU seconds: a four-fold improvement. In Figure 7 the results are a more than six times speed-up. It is also interesting to observe that ⟨COMP, EACH⟩ out-performs ⟨COMP, BUSY⟩ at each time point. Even though there are nine more COMPLETION constraints in the *each* condition, the substantial reduction in search nodes for a given solution quality is worthwhile in terms of CPU time. This can be clearly seen in Figure 8, where the MRE results from Figure 6 are plotted against the number of choice points. While ⟨COMP, EACH⟩ is only able to make a few more than 2,000,000 choice points, for the best achieved by ⟨COMP, BUSY⟩, ⟨COMP, EACH⟩ uses less than one-third of the choice points. To equal the best WS results, ⟨COMP, EACH⟩ uses one-tenth as many choice points.

6.3.2 Randomized Restart

As noted above, we also experimented with a randomized restart search. Figure 9 displays the results for the $U[1, 9]$. The $U[1, 99]$ results (not shown) are qualitatively similar. Comparing Figures 6 and 9, it is clear that all algorithms are substantially worse when using randomized restart than chronological back-

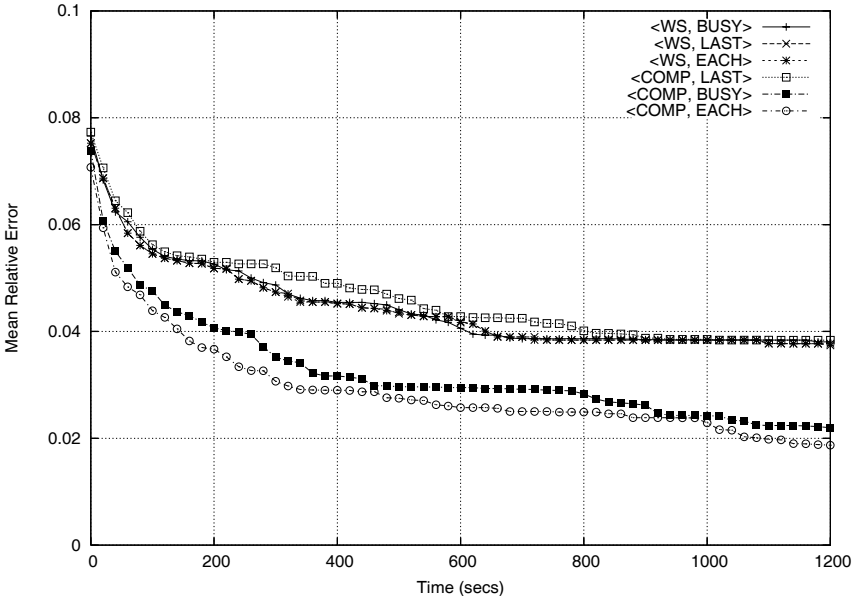

Figure 7: The mean relative error for different propagation techniques with chronological backtracking for the problems with job weight uniformly drawn from $U[1, 99]$. See the caption of Figure 6 for details.

tracking. In some cases, chronological backtracking with the weaker propagation substantially out-performs randomized restart with stronger propagation. Furthermore, the differences among the algorithms are narrowed.

Gomes et al. [19] demonstrated that randomized restart helps search when the run-times exhibit a heavy-tailed distribution. Such a distribution arises when the depths of search subtrees with no solution are exponentially distributed. Gomes et al. also observe that heavy-tailed behaviour is more likely with sophisticated algorithms that, for example, efficiently employ higher levels of propagation. The results here are, therefore, somewhat negative with respect to the goals of this work. While the COMPLETION constraint is able to improve performance over the weighted-sum constraint even with randomized restart, it does not appear strong enough to induce an exponential depth distribution of failed sub-trees. In other words, it does not provide enough propagation to induce "easy" sub-trees where the proof of infeasibility is quickly arrived at. Rather, all sub-trees must be exhaustively searched through.

6.3.3 Discussion

Our primary interest is the comparison between the performance of the approach employing the COMPLETION constraint and that of the search algorithms using the weighted-sum constraint. These experiments demonstrate that

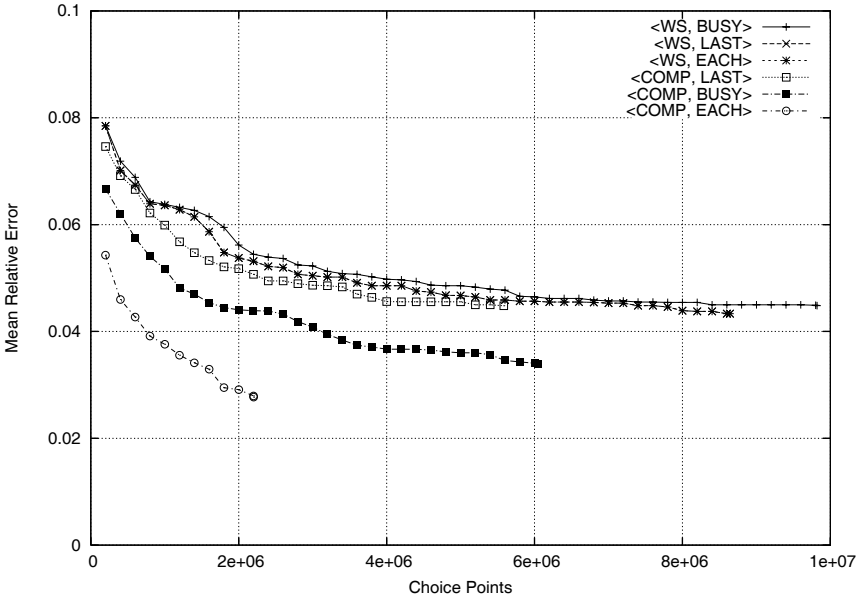

Figure 8: The mean relative error for different propagation techniques with chronological backtracking for the problems with job weight uniformly drawn from $U[1, 9]$ plotted versus the choice points in the search tree. Each instance was run with a 1200 CPU second time limit. See the caption of Figure 6 for details.

the COMPLETION constraint can be used to improve the problem solving capabilities of constraint-based scheduling algorithms beyond the single-machine case with no changes to the underlying algorithm, though with some experimentation to arrive at a good mapping between job weights and activity weights. The specialized single-machine approaches discussed at the end of Section 5 would require a significant re-engineering and re-implementation to be similarly adapted to the job shop scheduling problem.

7 Extensions and Future Work

An obvious research direction given the goals of this work is to use the COMPLETION constraint to model and solve a variety of optimization problems. We have been pursuing some problems that require relatively simple modeling extensions compared to the above work. For example, in Kovács & Beck [22], we introduced a constraint-based model of a single-machine scheduling problem with tool changes using the COMPLETION constraint. The experiments presented in that paper showed that our approach outperforms other MIP and customized branch-and-bound methods. Note that later we performed experiments on a different set of instances, where one of the MIP models was more

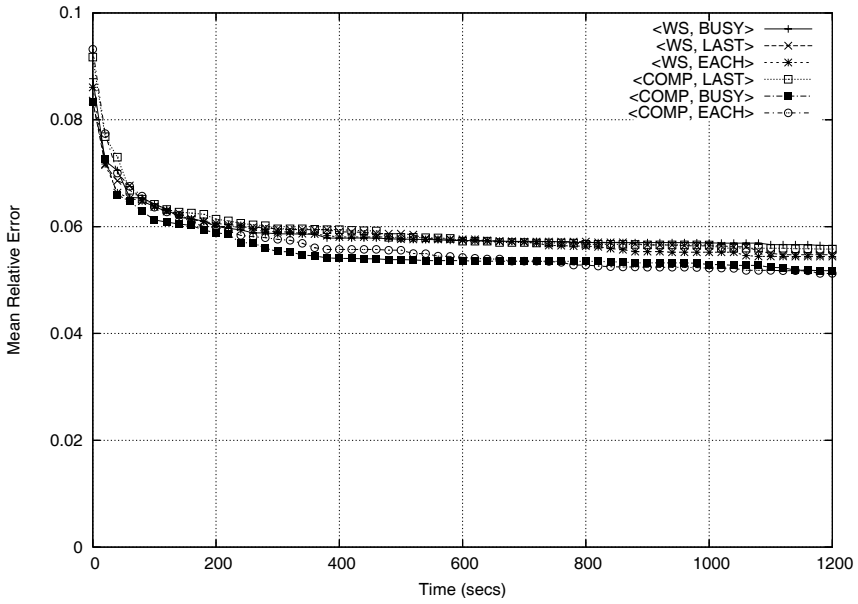

Figure 9: The mean relative error for different propagation techniques with randomized restart for the problems with job weight uniformly drawn from $U[1, 9]$. See the caption of Figure 6 for details.

efficient than our constraint-based approach.

The other research direction which we are pursuing is the generalization of the COMPLETION constraint from unary to cumulative capacity. We have defined a generalization as follows:

$$\text{COMPLETION}_m([S_1, ..., S_n], [p_1, ..., p_n], [\varrho_1, ..., \varrho_n], [w_1, ..., w_n], R, C)$$

As with the unary case, the scheduling problem involves n activities A_i to be executed without preemption on a single, cumulative resource. Activities are characterized by their processing times p_i, weights w_i, and resource requirements ϱ_i. R stands for the capacity of the resource. The total weighted completion time of the activities is denoted by C. Again, we assume that p_i, w_i, ϱ_i, and R are constants, though this limitation can be easily bypassed by reasoning with the lower and upper bounds of these parameters. The proposed propagation algorithm for COMPLETION_m is based on a novel variable-intensity relaxation of the cumulative resource scheduling problem. Similarly to the unary case, the propagator computes a series of relaxed solutions, each with an additional restriction stating that an activity A_i must start at time t. Then, if the cost of a relaxed solution is higher than the current upper bound, then t can be removed from the domain of the start time variable of the given activity A_i. The main difference between the unary and the cumulative cases is that for the latter, we

were unable to define efficient recomputation methods that convert a relaxed schedule to solutions for each possible value of t. Instead, we only estimate how the earliest or latest start times must be adjusted to achieve consistency. The results have been published in the paper [23].

In the same paper, we have applied the COMPLETION$_m$ constraint to model the container loading problem subject to constraints on the location of the COG, as noted in Section 1. Again, our initial empirical results appear promising.

8 Conclusions

In this paper, we propose an algorithm for propagating the COMPLETION constraint, which represents the sum of weighted completion times of activities on a single unary capacity resource. The propagation of the constraint exploits a lower bound arising from the optimal solution to the preemptive mean busy time scheduling problem. We introduce an algorithm that iteratively recomputes the lower bound cost for a carefully structured subset of the possible start time assignments, and filters start time variable domains by removing the values that would result in a cost higher than the current upper bound.

Empirical results on a set of single resource, minimum weighted completion time benchmarks show that the COMPLETION constraint significantly improves the performance of constraint-based approaches to this problem. The improvement occurs both compared to the classical weighted-sum representation of the objective function, and also compared to a model using the same relaxation only as a lower bound. We validate the applicability of the COMPLETION constraint in more complex problems by experiments on the multiple machine job shop scheduling problem with the criteria of minimizing total weighted completion time. Our results show a substantial improvement (i.e., a four to six times speed up in terms of run-time and an order of magnitude reduction in the number of choice points) over the standard weighted-sum constraint.

Acknowledgment

The authors acknowledge the support of the Canadian Natural Sciences and Engineering Research Council, ILOG, S.A., and the OTKA grant K 73376. A. Kovács acknowledges the support of the János Bolyai scholarship No. BO/00138/07.

References

[1] F. Afrati, E. Bampis, C. Chekuri, D. Karger, C. Kenyon, S. Khanna, I. Milis, M. Queyranne, M. Skutella, C. Stein, and M. Sviridenko. Approximation schemes for minimizing average weighted completion time with

release dates. In *Proc. of the 40th IEEE Symposium on Foundations of Computer Science*, pages 32–44, 1999.

[2] C. Akkan and S. Karabatı. The two-machine flowshop total completion time problem: Improved lower bounds and a branch-and-bound algorithm. *European Journal of Operational Research*, 159:420–429, 2004.

[3] Ph. Baptiste, J. Carlier, and A. Jouglet. A branch-and-bound procedure to minimize total tardiness on one machine with arbitrary release dates. *European Journal of Operational Research*, 158:595–608, 2004.

[4] Ph. Baptiste and C. Le Pape. Scheduling a single machine to minimize a regular objective function under setup constraints. *Discrete Optimization*, 2:83–99, 2005.

[5] Ph. Baptiste, L. Peridy, and E. Pinson. A branch and bound to minimize the number of late jobs on a single machine with release time constraints. *European Journal of Operational Research*, 144(1):1–11, 2003.

[6] J. C. Beck and P. Refalo. A hybrid approach to scheduling with earliness and tardiness costs. *Annals of Operations Research*, 118:49–71, 2003.

[7] H. Belouadah, M. E. Posner, and C. N. Potts. Scheduling with release dates on a single machine to minimize total weighted completion time. *Discrete Applied Mathematics*, 36:213–231, 1992.

[8] H. Belouadah and C. N. Potts. Scheduling identical parallel machines to minimize total weighted completion time. *Discrete Applied Mathematics*, 48:201–218, 1994.

[9] B. Chen, C. N. Potts, and G. J. Woeginger. *A Review of Machine Scheduling: Complexity, Algorithms and Approximation*, volume 3 of *Handbook of Combinatorial Optimization*. Kluwer, 1998.

[10] R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.

[11] F. Della Croce, M. Ghirardi, and R. Tadei. An improved branch-and-bound algorithm for the two machine total completion time flow shop problem. *European Journal of Operational Research*, 139:293–301, 2002.

[12] S. Demassey, G. Pesant, and L.-M. Rousseau. A cost-regular based hybrid column generation approach. *Constraints*, 11(4):315–333, 2006.

[13] A. Drexl and A. Kimms. Lot-sizing and scheduling – survey and extensions. *European Journal of Operational Research*, 99:221–235, 1997.

[14] M. Dyer and L. A. Wolsey. Formulating the single machine sequencing problem with release dates as mixed integer program. *Discrete Applied Mathematics*, 26:255–270, 1990.

[15] F. Focacci, A. Lodi, and M. Milano. Embedding relaxations in global constraints for solving TSP and TSPTW. *Annals of Mathematics and Artificial Intelligence*, 34(4):291–311, 2002.

[16] F. Focacci, A. Lodi, and M. Milano. Optimization-oriented global constraints. *Constraints*, 7(3–4):351–365, 2002.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[18] M. X. Goemans, M. Queyranne, A. S. Schulz, M. Skutella, and Y. Wang. Single machine scheduling with release dates. *SIAM Journal on Discrete Mathematics*, 15(2):165–192, 2002.

[19] Carla P. Gomes, Cèsar Fernández, Bart Selman, and Christian Bessière. Statistical regimes across constrainedness regions. *Constraints*, 10(4):317–337, 2005.

[20] A. Jouglet, P. Baptiste, and J. Carlier. Branch-and-bound algorithms for total weighted tardiness. In *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, chapter 13. Chapman & Hall / CRC, 2004.

[21] A. Kéri and T. Kis. Primal-dual combined with constraint propagation for solving rcpspwet. In *Proc. of the 2nd Multidisciplinary International Conference on Scheduling: Theory and Applications*, pages 748–751, 2005.

[22] A. Kovács and J. C. Beck. Single-machine scheduling with tool changes: A constraint-based approach. In *PlanSIG 2007, the 26th Workshop of the UK Planning and Scheduling Special Interest Group*, pages 71–78, 2007.

[23] A. Kovács and J. C. Beck. A global constraint for total weighted completion time for cumulative resources. *Engineering Applications of Artificial Intelligence*, 21(5):691–697, 2008.

[24] C. Le Pape, P. Couronné, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. In *Proceedings of the Thirteenth Workshop of the UK Planning Special Interest Group*, 1994.

[25] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[26] R. Nessah, F. Yalaoui, and C. Chu. A branch-and-bound algorithm to minimize total weighted completion time on identical parallel machines with job release dates. *Computers and Operations Research*, 35(4):1176–1190, 2009.

[27] Y. Pan. Test instances for the dynamic single-machine sequencing problem to minimize total weighted completion time, 2007. Available at www.cs.wisc.edu/˜yunpeng/test/sm/dwct/instances.htm.

[28] Y. Pan and L. Shi. New hybrid optimization algorithms for machine scheduling problems. *IEEE Transactions on Automation Science and Engineering*, 5(2):337–348, 2008.

[29] J.-C. Régin. Arc consistency for global cardinality constraints with costs. In *Proceedings of Principles and Practice of Constraint Programming (LNCS 1713)*, pages 390–404, 1999.

[30] J.-C. Régin. Global constraints and filtering algorithms. In M. Milano, editor, *Constraint and Integer Programming: Toward a Unified Methodology*, pages 89–135. Kluwer, 2003.

[31] J.-C. Régin and M. Rueher. Inequality-sum: a global constraint capturing the objective function. *RAIRO Operations Research*, 39:123–139, 2005.

[32] Scheduler. *ILOG Scheduler 6.1 Reference Manual*. ILOG, S.A., 2002.

[33] A. S. Schulz. Scheduling to minimize total weighted completion time: Performance guarantees of lp-based heuristics and lower bounds. In *Proc. of the 5th Int. Conf. on Integer Programming and Combinatorial Optimization*, pages 301–315, 1996.

[34] M. Sellmann. An arc consistency algorithm for the minimum weight all different constraint. In *Proceedings of Principles and Practice of Constraint Programming (LNCS 2470)*, pages 744–749, 2002.

[35] J.M. van den Akker, C.A.J Hurkens, and M.W.P. Savelsberg. Time-indexed formulations for machine scheduling problems: Column generation. *INFORMS Journal on Computing*, 12:111–124, 2000.

[36] J.P. Watson, L. Barbulescu, A.E. Howe, and L.D. Whitley. Algorithms performance and problem structure for flow-shop scheduling. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 688–695, 1999.