# Solving the Group Cumulative Scheduling Problem with CPO and ACO

Lucas Groleaz, Samba Ndojh Ndiaye, Christine Solnon

# Solving the Group Cumulative Scheduling Problem with CPO and ACO

Lucas Groleaz[1,2], Samba N. Ndiaye[1], and Christine Solnon[3]

(1) LIRIS, CNRS UMR5205, INSA Lyon, F-69621 Villeurbanne - (2) Infologic
(3) CITI, INRIA, INSA Lyon, F-69621 Villeurbanne

**Abstract.** The Group Cumulative Scheduling Problem (GCSP) comes from a real application, i.e., order preparation in food industry. Each order is composed of jobs which must be scheduled on machines, and the goal is to minimize the sum of job tardiness. There is an additional constraint, called Group Cumulative (GC), which ensures that the number of active orders never exceeds a given limit, where an order is active if at least one of its jobs is started and at least one of its jobs is not finished. In this paper, we first describe a Constraint Programming (CP) model for the GCSP, where the GC constraint is decomposed using classical cumulative constraints. We experimentally evaluate IBM CP Optimizer (CPO) on a benchmark of real industrial instances, and we show that it is not able to solve efficiently many instances, especially when the GC constraint is tight. To explain why CPO struggles to solve the GCSP, we show that it is NP-Complete to decide whether there exist start times which satisfy the GC constraint given the sequence of jobs on each machine, even when there is no additional constraint. Finally, we introduce an hybrid framework where CPO cooperates with an Ant Colony Optimization (ACO) algorithm: ACO is used to learn good solutions which are given as starting points to CPO, and the solutions improved by CPO are given back to ACO. We experimentally evaluate this hybrid CPO-ACO framework and show that it strongly improves CPO performance.

## 1 Introduction

There exist numerous variants of scheduling problems [23]. In [9], a new scheduling problem is introduced, called the *Group Cumulative Scheduling Problem (GCSP)*. This problem comes from a real application: order preparation in food industry. Each order is composed of jobs which must be scheduled on machines, and the goal is to minimise the sum of job tardiness. There is an additional constraint, called the *Group Cumulative (GC)* constraint, which comes from the fact that a pallet is associated with each order: when starting the first job of an order is started, a pallet is set on the ground and this pallet is removed when the last job of the order is ended. As physical space is limited, the number of pallets on the ground must never exceed a given limit. In other words, jobs are grouped into orders, and GC ensures that the number of active groups never exceeds a limit, where a group is active if at least one of its jobs is started and at least one of its jobs is not finished.

Beyond the industrial application described in [9], the GC constraint may have other applications. In particular, it may be used each time a resource is required by a group of jobs, so that the resource is consumed when the first job of the group starts and it is released only when the last job of the group ends.

*Contributions.* In this paper, we describe a *Constraint Programming (CP)* model for the GCSP, and we show that GC may be decomposed using classical cumulative constraints. We experimentally evaluate *IBM CP Optimizer (CPO)*, a state-of-the-art solver for scheduling, on the industrial instances of [9], and we show that CPO struggle to solve many of them, especially when GC is tight. To provide insight into CPO performance, we study the complexity of GC: we show that it is NP-Complete to decide whether there exist start times which satisfy GC when the sequence of jobs on each machine is known, even if there is no additional constraint. Finally, we show how to hybridise CPO with the *Ant Colony Optimisation (ACO)* algorithm introduced in [9]: ACO is used to learn good solutions which are given as starting points to CPO, and solutions improved by CPO are given back to ACO. We experimentally evaluate this hybrid CPO-ACO framework, and show that it strongly improves CPO performance.

*Plan.* In Section 2, we describe the GCSP and we define GC. In Section 3, we introduce a CP model for the GCSP, and we report results obtained with CPO. In Section 4, we study the complexity of GC. In Section 5, we describe the ACO algorithm of [9]. In Section 6, we introduce and evaluate our hybrid CPO-ACO framework. Sections 2 and 5 are recalls from [9]. Sections 3, 4 and 6 contain new contributions with respect to [9].

*Notations.* We denote sets with calligraphic letters, constants with lowercase letters, and variables with uppercase letters. $\#\mathcal{A}$ denotes the cardinality of a set $\mathcal{A}$. $[l, u]$ denotes the set of all integers ranging from $l$ to $u$.

## 2 Description of the GCSP

The GCSP is a classical scheduling problem (referred to as the "basic" scheduling problem and described in Section 2.1) with an additional GC constraint (described in Section 2.2). In Section 2.3, we describe the benchmark of [9].

### 2.1 Basic scheduling problem

Given a set $\mathcal{M}$ of machines and a set $\mathcal{J}$ of jobs such that, for each job $j \in \mathcal{J}$, $r_j$ denotes its release date, $d_j$ its due date, and $p_j$ its processing time, the goal is to find a start time $B_j$, an end time $E_j$, and a machine $M_j$, for each job $j \in \mathcal{J}$. According to the notation introduced in [6], the basic scheduling problem underlying the GCSP is denoted $Rm, 1, 1; MPS|s_{ij}; r_j| \sum T_j$:

- $Rm, 1, 1$ means that $\mathcal{M}$ contains several machines working in parallel and each machine $m \in \mathcal{M}$ can process at most one job at a time;

- $MPS$ stands for *Multi-mode Project Scheduling* and means that every machine $m \in \mathcal{M}$ has its own speed denoted $sp^m$ (so that the duration of a job $j$ is $p_j * sp^{M_j}$);
- $s_{i,j}$ indicates that the setup time of a job $j \in \mathcal{J}$ depends on the job $i$ that precedes $j$ on the machine (*i.e.*, the time interval between the end time of $i$ and the start time of $j$ must be larger than or equal to this setup time);
- $r_j$ means that a job cannot start before its release date, *i.e.*, $\forall j \in \mathcal{J}, B_j \geq r_j$;
- $\sum T_j$ indicates that the goal is to minimize the sum of tardiness of every job, *i.e.*, $\sum_{j \in \mathcal{J}} \max(0, E_j - d_j)$.

### 2.2 GC Constraint

GC is a particular case of cumulative constraint [1, 2, 22, 21], and we show how to decompose GC using cumulative constraints in Section 3. Cumulative constraints are used to model the fact that jobs require resources (*e.g.*, human skills or tools) and that these resources have limited capacities, *i.e.*, the sum of resources required by all jobs started but not ended must never exceed resource capacities.

In the GCSP, the resource is not directly required by jobs, but by job groups. More precisely, jobs are partitioned into groups (corresponding to orders in the industrial application of [9]). The start (resp. end) time of a group is defined as the smallest start time (resp. largest end time) among all its jobs. A group is said to be *active* at a time $t$ if it is started and not ended at time $t$. The GC constraint ensures that the number of active groups never exceeds a given limit. More formally, we define the GC global constraint as follows.

**Definition 1.** *Given a set $\mathcal{J}$ of jobs, a partition $\mathcal{P}$ of $\mathcal{J}$ in $\#\mathcal{P}$ groups (such that each job $j \in \mathcal{J}$ belongs to exactly one group $\mathcal{G} \in \mathcal{P}$), an integer limit $l$ and, for each job $j \in \mathcal{J}$, an integer variable $B_j$ (resp. $E_j$) corresponding to the start time (resp. end time) of $j$, the constraint $GC_{\mathcal{J},\mathcal{P},l}(\{B_j : j \in \mathcal{J}\}, \{E_j : j \in \mathcal{J}\})$ is satisfied iff $\#\{\mathcal{G} \in \mathcal{P} : \min_{j \in \mathcal{G}} B_j \leq t < \max_{j \in \mathcal{G}} E_j\} \leq l$ for any time $t$.*

In Fig. 1, we display two examples of schedules: one that violates GC and one that satisfies it (we assume that setup times are null in this example).

### 2.3 Benchmark instances

A benchmark extracted from industrial data is introduced in [9]. It contains 548 instances such that the number of groups (resp. jobs and machines) ranges from 56 to 406 (resp. from 288 to 2909, and from 1 to 14). For each instance, an upper bound (denoted $x$) on the number of active groups is given. It is computed as follows: first, a greedy algorithm is used to compute a solution $s$ for the basic scheduling problem (without the GC constraint); then $x$ is assigned to the maximum number of active groups during the whole time horizon in $s$.

As our goal is to study the impact of GC on the solution process, we consider three classes of instances: in the first class, denoted *loose*, the limit $l$ is set to $l = 0.7 * x$, in the second class, denoted *medium*, $l$ is set to $0.5 * x$, and in the third class, denoted *tight*, $l$ is set to $0.3 * x$.

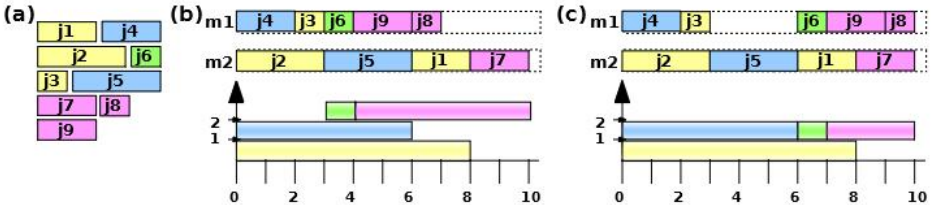


**Fig. 1.** Schedule examples. (a) A set $\mathcal{J}$ of 9 jobs and a partition $\mathcal{P}$ of $\mathcal{J}$ in 4 groups represented by colours. (b) Example of schedule on 2 machines which violates GC when $l = 2$ (there are 3 active groups from time 3 to time 6, as displayed on the bottom of (b)). (c) Example of schedule on 2 machines which satisfies GC when $l = 2$ (an iddle time is added between $j_3$ and $j_6$ to wait the end of the blue group).

$$\text{Minimize} \quad \sum_{j \in \mathcal{J}} max(0, endOf(A_j) - d_j)$$

$$\begin{aligned}
\text{subject to } & A_j^m = interval(d_j * sp^m) && \forall j \in \mathcal{J}, \forall m \in \mathcal{M} \quad (1)\\
& optional(A_j^m) && \forall j \in \mathcal{J}, \forall m \in \mathcal{M} \quad (2)\\
& A_j = alternative(\{A_j^m : m \in \mathcal{M}\}) && \forall j \in \mathcal{J} \quad (3)\\
& startMin(A_j^m, r_j) && \forall j \in \mathcal{J}, \forall m \in \mathcal{M} \quad (4)\\
& S^m = intervalSequence(\{A_j^m : j \in \mathcal{J}\}, jobTypes) && \forall m \in \mathcal{M} \quad (5)\\
& noOverlap(S^m, jobTypes, setupTimes) && \forall m \in \mathcal{M} \quad (6)
\end{aligned}$$

**Fig. 2.** CPO model for the basic scheduling problem described in Section 2.1. (*jobTypes* is an array which associates a type with every job and *setupTimes* is a transition matrix which defines the setup times between job types).

## 3 CPO Model

We describe a CPO model for the basic scheduling problem in Section 3.1, and a decomposition of GC in Section 3.2. We report results obtained with CPO in Section 3.3. We refer the reader to [16] for details on CPO.

### 3.1 Model of the basic scheduling problem

The CPO model associates an interval variable $A_j$ with every job $j \in \mathcal{J}$, *i.e.*, $A_j$ corresponds to the interval $[B_j, E_j]$. Also, an optional interval variable $A_j^m$ is associated with every job $j \in \mathcal{J}$ and every machine $m \in \mathcal{M}$: if job $j$ is executed on machine $m$, then $A_j^m = A_j$; otherwise $A_j^m$ is assigned to $\perp$ (*i.e.*, it is absent). Finally, an interval sequence variable $S^m$ is associated with every machine $m$ to represent the total ordering of the present interval variables in $\{A_j^m : j \in \mathcal{J}\}$.

The objective function and the constraints are described in Fig. 2. Constraint (1) defines the interval variable $A_j^m$ whose length is equal to the processing time of job $j$ multiplied by the speed of machine $m$; Constraints (2) and

$$span(F_{\mathcal{G}}, \{A_j^m : m \in \mathcal{M} \wedge j \in \mathcal{G}\}) \; \forall \mathcal{G} \in \mathcal{P} \quad (7)$$
$$Active = \sum_{\mathcal{G} \in \mathcal{P}} pulse(F_{\mathcal{G}}, 1) \quad (8)$$
$$lowerOrEqual(Active, l) \quad (9)$$

**Fig. 3.** CPO decomposition of GC.

(3) ensure that every job $j$ is scheduled on exactly one machine; Constraint (4) ensures that a job does not start before its release date; Constraint (5) defines the sequence of jobs on machine $m$; and Constraint (6) ensures that at most one job is executed at a time on machine $m$, and states that there are sequence-dependent setup times between jobs.

### 3.2 Decomposition of GC

We can easily decompose GC using a classical cumulative constraint. To this aim, we associate a new interval variable $F_{\mathcal{G}}$ with every group $\mathcal{G} \in \mathcal{P}$. This variable corresponds to a fictive job which starts with the earliest job of the group and ends with its latest job, and which consumes one unit of resource. A simple cumulative constraint on these fictive jobs ensures that the number of active groups never exceeds $l$.

More precisely, Fig. 3 describes a CPO model of this decomposition: Constraint (7) ensures that, for every group $\mathcal{G}$, the fictive job variable $F_{\mathcal{G}}$ spans over all jobs in the group; Constraint (8) defines the cumul function (denoted *Active*) corresponding to the case where each fictive job consumes one unit of the resource; and Constraint (9) ensures that *Active* never exceeds $l$, thus ensuring the cumulative constraint on fictive jobs.

### 3.3 Experimental evaluation of CPO

CPO is a state-of-the-art solver for scheduling problems, as demonstrated in [7] on the job shop, for example. In this section, we report CPO results with the model described in Sections 3.1 and 3.2 on instances described in Section 2.3. All experiments reported in this paper have been performed on a processor Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 7.2 GB RAM.

CPO provides different levels of filtering, and we have compared results obtained with two different levels: default filtering (based on *Timetable* [1]) and extended filtering (based on *energy reasoning* and *edge finding* [5, 15, 29, 22, 18]). For short time limits (less than 100s), CPO with default filtering usually finds better solutions than CPO with extended filtering. After one hour, for nearly half of the instances a better solution is found with the extended filtering, whereas for the other half a better solution is found with the default filtering, and this happens in all classes (loose, medium, and tight). In most cases, the difference between the two levels of filtering is rather small. Hence, we have chosen to report results obtained with the default level of filtering.

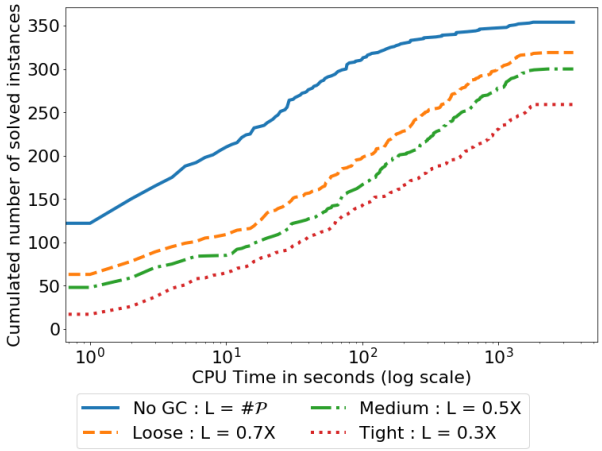


**Fig. 4.** Evolution of the cumulative number of solved instances with respect to time for the basic scheduling problem and for the GCSP when $l$ is loose, medium or tight.

In Fig. 4, we display the evolution of the cumulative number of solved instances with respect to time for the basic scheduling problem (we consider that an instance is solved when CPO has completed its run). In this case, CPO is able to solve 354 instances (among the 548 instances of the benchmark) within one hour. We also display results on the same set of instances when adding GC, for the three classes (which only differ on the value of $l$). Clearly, GC increases the hardness of the problem, and increasing the tightness of GC (by decreasing the limit $l$) also increases hardness: 319 (resp. 300 and 259) instances are solved within one hour for the loose (resp. medium and tight) class.

This may come from the fact that the decomposition of GC is not well propagated by CPO. A possible explanation is that interval variables $F_G$ associated with groups do not have known durations when starting the search. We can only compute bounds on group durations. For example, the duration of a group is lower bounded by the greatest duration of its jobs. However, these bounds are not very tight at the beginning of the search. In this case, energy-based propagation techniques are not efficient as the energy of a job is defined as its duration multiplied by its resource consumption.

In Fig. 5, we display the number of jobs and machines of solved and unsolved instances, for the basic scheduling problem and the tight GCSP. In both cases, some large instances (with more than 2500 jobs) are solved whereas some small instances (with less than 300 jobs) are not solved. Most instances with more than 6 machines are solved (only 8 are not solved for the basic scheduling problem) whereas many instances with one machine are not solved.

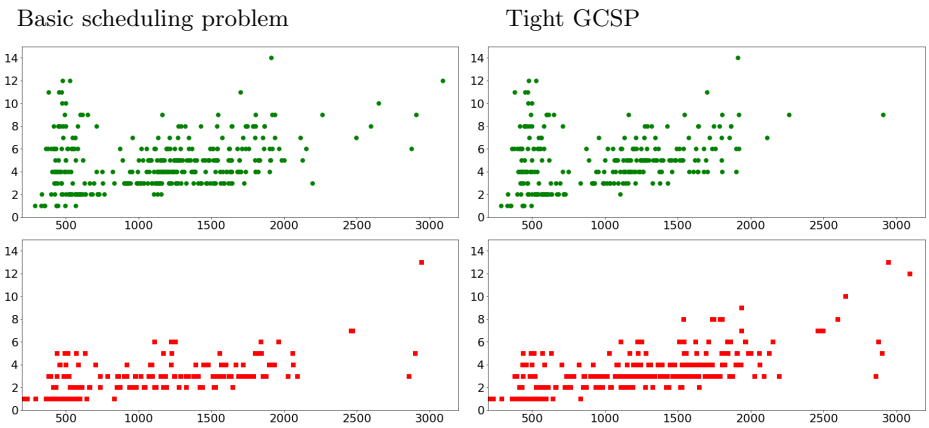


**Fig. 5.** Sizes of solved and unsolved instances for the basic scheduling problem (left) and the tight GCSP (right): each point $(x, y)$ corresponds to an instance with $x$ jobs and $y$ machines. Top (green points): instances solved in less than one hour. Bottom (red points): instances not solved within one hour.

## 4 Deciding of GC feasibility with list schedules

CPO exploits precedence relations to solve scheduling problems [16]: all temporal constraints are aggregated in a temporal network whose nodes represent interval start and end time-points and whose arcs represent precedence relations. Also, CPO integrates a *Large Neighborhood Search (LNS)* component which is based on the initial generation of a directed graph whose nodes are interval variables and edges are precedence relations between interval variables.

Reasoning on precedence relations often simplifies the solution process of scheduling problems. In particular, for the basic problem described in Section 2.1 (without GC), given a *list schedule* (*i.e.*, an ordered list of jobs for each machine), we can compute optimal start times in polynomial time: for each machine, we consider jobs according to the order defined by its associated list and schedule each of them as soon as possible [24, 10]. The basic scheduling problem is NP-hard because it is hard to find the list schedule which leads to the optimal solution. However, as optimal start times are easily derived from list schedules, search can focus on precedence relations.

If we add a classical cumulative constraint to the basic scheduling problem, the problem of computing optimal start times given a list schedule becomes NP-hard [21]. However, if we remove the objective function (*i.e.*, we simply search for a schedule which satisfies the cumulative constraint without having to minimize the tardiness sum), then we can easily compute start times that satisfy cumulative constraints given a list schedule: Again, this can be done greedily, by considering jobs in the order of the lists, and scheduling each job as soon as possible with respect to cumulative constraints.

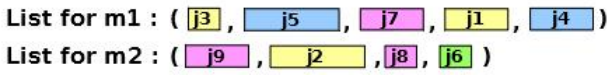


**Fig. 6.** Example of list schedule with 2 machines for the jobs of Fig. 1.

However, this is no longer true for GC. For example, let us consider the list schedule displayed in Fig. 6. We cannot find start times that satisfy GC for this list schedule when $l = 2$. Indeed, on machine $m_1$, the yellow job $j_1$ is between two blue jobs $j_5$ and $j_4$, and this implies that we must start the yellow group to be able to complete the blue group. Similarly, on machine $m_1$, the blue job $j_5$ is between two yellow jobs ($j_3$ and $j_1$) so that we must start the blue group to be able to complete the yellow group, and on machine $m_2$, the yellow job $j_2$ is between two pink jobs ($j_9$ and $j_8$) so that we must start the yellow group to be able to complete the pink group. This implies that both yellow, blue and pink groups must be active all together at some time.

More precisely, let us denote *LS-GC* the problem of deciding whether there exists a solution of GC which is consistent with a given list schedule, where a list schedule is consistent with a solution of GC iff, for every $j_1, j_2 \in \mathcal{J}$ such that $j_1$ occurs before $j_2$ in a same list, we have $E_{j_1} \leq B_{j_2}$.

**Theorem 1.** *LS-GC is $\mathcal{NP}$-complete.*

*Proof.* *LS-GC* clearly belongs to $\mathcal{NP}$ as we can check in polynomial time if a given assignment is a solution of GC which is consistent with a list schedule.

Now, let us show that *LS-GC* is $\mathcal{NP}$-complete by reducing the Pathwidth problem to it. Given a connected graph $G = (\mathcal{N}, \mathcal{E})$ (such that $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ a set of edges) and an integer $w$, Pathwidth aims at deciding whether there exists a sequence $(\mathcal{N}_1, ..., \mathcal{N}_n)$ of subsets of $\mathcal{N}$ such that (i) $\mathcal{N} = \bigcup_{i=1}^{n} \mathcal{N}_i$; (ii) $\forall \{u, v\} \in \mathcal{E}, \exists i \in [1, n], \{u, v\} \subseteq \mathcal{N}_i$; (iii) $\forall i, j, k \in [1, n], i \leq j \leq k \Rightarrow \mathcal{N}_i \cap \mathcal{N}_k \subseteq \mathcal{N}_j$; and (iv) $\forall i \in [1, n],\ \#\mathcal{N}_i \leq w$. Pathwidth is $\mathcal{NP}$-complete [11].

Let us first show how to construct an instance of *LS-GC* given an instance of Pathwidth defined by a graph $G = (\mathcal{N}, \mathcal{E})$ and an integer $w$. We assume that nodes of $\mathcal{N}$ are numbered from 1 to $\#\mathcal{N}$. For each edge $\{u, v\} \in \mathcal{E}$, we define three jobs denoted $j^1_{uv}$, $j^2_{uv}$, and $j^3_{uv}$ such that every job has a processing time equal to 1. The partition $\mathcal{P}$ associates one group $\mathcal{G}_u$ with every vertex $u$ such that $\mathcal{G}_u = \{j^1_{uv}, j^3_{uv} : \{u, v\} \in \mathcal{E} \wedge u < v\} \cup \{j^2_{uv} : \{u, v\} \in \mathcal{E} \wedge u > v\}$. In other words, for each edge $\{u, v\} \in \mathcal{E}$ such that $u < v$, $j^1_{uv}$ and $j^3_{uv}$ belong to group $\mathcal{G}_u$ whereas $j^2_{uv}$ belongs to group $\mathcal{G}_v$. There are $\#\mathcal{E}$ machines, and the list schedule associates the list $(j^1_{uv}, j^2_{uv}, j^3_{uv})$ with every edge $\{u, v\} \in \mathcal{E}$ such that $u < v$. Finally, we set the limit $l$ to $w$. Fig. 7 gives an example of this reduction.

Now, let us show that every solution $(\mathcal{N}_1, ..., \mathcal{N}_n)$ of an instance of Pathwidth corresponds to a solution of the corresponding instance of *LS-GC*. To this aim, we

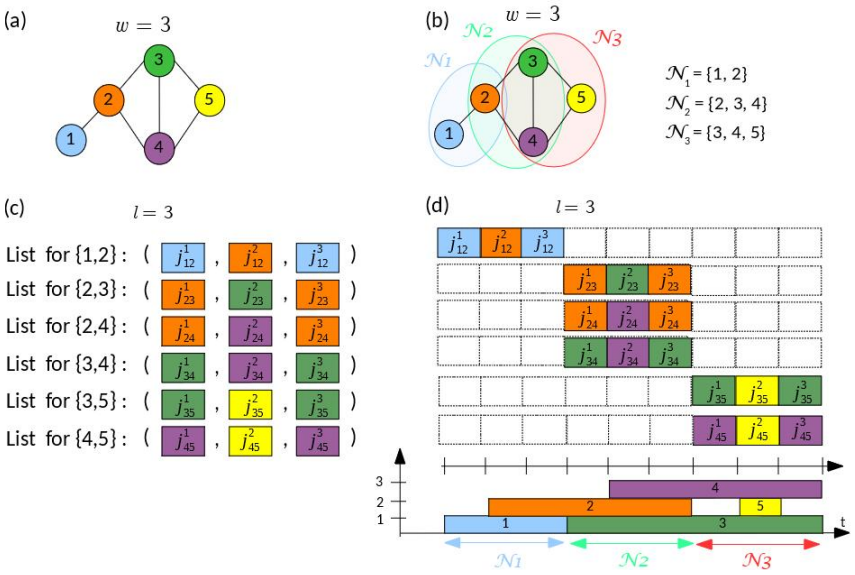


**Fig. 7.** Reduction from Pathwidth to *LS-GC*. (a): Example of instance of Pathwidth. (b): Example of solution of (a). (c): List schedule of the instance of *LS-GC* corresponding to (a). (d): Solution of (c) corresponding to (b).

show how to define the start time $B_{j^i_{uv}}$ of every job $j^i_{uv}$ associated with an edge $\{u, v\} \in \mathcal{E}$, with $i \in \{1, 2, 3\}$: let $a$ be the index of the first subset in $(\mathcal{N}_1, ..., \mathcal{N}_n)$ which contains both $u$ and $v$ (*i.e.*, $a = \min\{b \in [1, n] : \{u, v\} \subseteq \mathcal{N}_b\}$); we define $B_{j^1_{uv}} = 3*a-3$, $B_{j^2_{uv}} = 3*a-2$, and $B_{j^3_{uv}} = 3*a-1$; end times are computed by adding the processing time 1 to every start time. In Fig. 7(d), we display start and end times computed for a solution of the Pathwidth instance of Fig. 7(a). We can easily check that start and end times are consistent with the list schedule. To show that start and end times satisfy GC, we have to show that the number of active groups never exceeds $l$, and this is a consequence of the fact that the number of vertices in a set $\mathcal{N}_b$ never exceeds $w = l$. Indeed if we consider a time $t$ with $3 * a - 3 \leq t \leq 3 * a - 1$ ($a \in [1, n]$), then the only groups that can be active at time $t$ are those associated with nodes in $\mathcal{N}_a$ and $\#\mathcal{N}_a \leq w = l$.

Finally, let us show that every solution of the instance of *LS-GC* built from an instance of Pathwidth corresponds to a solution of this Pathwidth instance. A solution of an instance of *LS-GC* is an assignment of values to $B_j$ and $E_j$ for every job $j \in \mathcal{J}$ (defining start and end times of $j$). For each node $u \in \mathcal{N}$, we have a group of jobs $\mathcal{G}_u$, and the start time $B_u$ of this group is the smallest start time of its jobs (*i.e.*, $B_u = \min\{B_j : j \in \mathcal{G}_u\}$) whereas the end time $E_u$ of this group is the largest end time of its jobs (*i.e.*, $E_u = \max\{E_j : j \in \mathcal{G}_u\}$). Let $\mathcal{T} = \{B_u : u \in \mathcal{N}\}$ be the set of all group start times, and let $(t_1, \ldots, t_{\#\mathcal{T}})$ be the ordered sequence of values in $\mathcal{T}$. The solution of the Pathwidth instance is

$(\mathcal{N}_1, \ldots, \mathcal{N}_{\#\mathcal{T}})$ such that for each $i \in [1, \#\mathcal{T}]$, $\mathcal{N}_i = \{u \in \mathcal{N} : B_u \leq t_i < E_u\}$. We can check that $(\mathcal{N}_1, \ldots, \mathcal{N}_{\#\mathcal{T}})$ is a solution. Indeed, for each edge $\{u, v\} \in \mathcal{E}$ with $u < v$, the list $(j^1_{uv}, j^2_{uv}, j^3_{uv})$ ensures that when $j^2_{uv}$ starts both $\mathcal{G}_u$ and $\mathcal{G}_v$ are active groups. Hence, $\forall i \in [1, \#\mathcal{T}]$ such that $t_i = max(B_u, B_v)$, $\{u, v\} \in \mathcal{N}_i$. Now if we have $i, j, k \in [1, \#\mathcal{T}]$ with $i \leq j \leq k$, $u \in \mathcal{N}_i$ and $u \in \mathcal{N}_k$ then $B_u \leq t_i \leq t_j \leq t_k < E_u$. Hence $u \in \mathcal{N}_j$, and so $\mathcal{N}_i \cap \mathcal{N}_k \subseteq \mathcal{N}_j$. Finally, $\forall i \in [1, \#\mathcal{T}]$, all groups of $\mathcal{N}_i$ are active at time $t_i$. So $\#\mathcal{N}_i \leq l = w$

## 5 ACO algorithm for the GCSP

Many different ACO algorithms have been proposed for solving scheduling problems, and a review of 54 of these algorithms may be found in [27]. ACO algorithms use pheromone trails to learn promising solution components and progressively intensify the search around them. The two most widely considered definitions of pheromone trails for scheduling problems are: *Job trails*, where a trail $\tau(j, j')$ is associated with every couple of jobs $(j, j') \in \mathcal{J}^2$ to learn the desirability of scheduling $j'$ just after $j$ on a same machine; and *Position trails*, where a trail $\tau(j, m, n)$ is associated with each triple $(j, m, n) \in \mathcal{J} \times \mathcal{M} \times [1, \#\mathcal{J}]$ to learn the desirability of scheduling job $j$ at position $n$ on machine $m$.

Most ACO algorithms for scheduling problems follow the basic template displayed in Algo. 1. At each iteration of the loop lines 1-9, $n_{ants}$ solutions are constructed in a greedy randomised way, where $n_{ants}$ is a parameter which is used to control exploration (the larger $n_{ants}$, the stronger the exploration). At each iteration of the greedy construction (lines 4-8), a machine $m$ and a job $j$ are chosen, and $j$ is scheduled on $m$, until all jobs have been scheduled. The choice of $m$ is done according to some heuristics which depend on the scheduling problem. The choice of $j$ is done in a randomised way, according to a probability $p(j)$ which depends on two factors. The pheromone factor $f_\tau(j)$ represents the learned desirability of scheduling $j$ on $m$ and its definition depends on the pheromone trail definition: for *Job trails*, $f_\tau(j) = \tau(j', j)$ where $j'$ is the last job scheduled on $m$; for *Position trails*, $f_\tau(j) = \tau(j, m, k)$ where $k$ is the number of jobs scheduled on $m$. The heuristic factor $\eta(j)$ evaluates the interest of scheduling $j$ on $m$ and its exact definition depends on the scheduling problem. $\alpha$ and $\beta$ are two parameters which are used to balance these two factors.

At the end of each cycle (line 9), pheromone trails are updated in two steps. First, every pheromone trail is decreased by multiplying it with $1 - \rho$ where $\rho \in [0, 1]$ is a parameter which controls the speed of intensification: the larger $\rho$, the quicker search is intensified towards the best solutions found so far. In a second step, pheromone trails associated with the best solution among the $n_{ants}$ last computed solutions are increased in order to increase the probability of selecting the components of this solution in the next constructions.

In [9], Algo. 1 is adapted to solve GCSPs as follows. Line 5, $m$ is the machine which minimizes the end time of the last job assigned to it. Line 6, the heuristic factor $\eta(j)$ is set to 0 whenever the current number of active groups is equal to the limit $l$ and $j$ belongs to a group which is not yet active. In this case, the

**Algorithm 1:** ACO algorithm for scheduling problems

```
1  while time limit not reached do
2      for i in [1, n_ants] do
           /* Greedy randomised construction of one solution */
3          Cand ← J
4          while Cand ≠ ∅ do
5              choose a machine m ∈ M according to some heuristic
6              choose j ∈ Cand w.r.t. probability p(j) = [f_τ(j)]^α · [η(j)]^β / Σ_{j'∈Cand} [f_τ(j')]^α · [η(j')]^β
7              assign m to M_j, and assign values to B_j and E_j
8              remove j from Cand
9      update pheromone trails
10 return the best constructed solution
```

probability $p(j)$ of selecting $j$ is equal to 0, thus ensuring that solutions always satisfy GC. When $j$ can be scheduled without violating GC (*i.e.*, the number of active groups is smaller than $l$, or it is equal to $l$ and $j$ belongs to an active group), the heuristic factor $\eta(j)$ is defined as the *ATCS (Apparent Tardiness Cost with Setup-times)* score defined in [23]. Line 9, before updating pheromone trails, the best solution (among the last $n_{ants}$ constructed solutions) is improved by applying a local search step. Also, pheromone trails are updated according to the MMAS framework of [26], *i.e.*, every pheromone trail is bounded between two parameters $\tau_{min}$ and $\tau_{max}$. Also, every pheromone trail is initialised to $\tau_{max}$ at the beginning of the search process (before line 1).

This ACO algorithm has one hyper-parameter, which is used to choose the pheromone trail definition (*i.e.*, *Job trails*, *Position trails*, or a new definition introduced in [9] and called *Time trails*). The algorithm also has the following parameters: $n_{ants}$, $\alpha$, $\beta$, $\rho$, $\tau_{min}$ and $\tau_{max}$, plus two parameters for the local search step. In [9], a portfolio of nine complementary parameter configurations are identified, and the per-instance algorithm selector LLAMA [14] is used to select from this portfolio the configuration expected to perform best for every new instance to solve.

## 6 New Hybrid CPO-ACO approach

Many hybrid approaches combine exhaustive solvers (such as CP or Integer Linear Programming, for example) with meta-heuristics [4]. Some of these hybrid approaches are referred to as *matheuristics* [17]. A well known example of hybrid approach is LNS [25] which uses CP to explore the neighborhood of a local search.

Different hybrid CP-ACO approaches have been proposed such as, for example, [20, 19, 12, 13, 28, 8]. Some approaches use constraint propagation during the construction of solutions by ACO (lines 3-8 of Algo. 1), to filter the set of candidate components and remove those that do not satisfy constraints [19, 12].

Some other approaches use ACO to learn ordering heuristics which are used by CP [20, 13]. In [8], a bi-level hybrid process is introduced where ACO is used to assign a subset of variables, and the remaining variables are assigned by CP.

### 6.1 Description of the hybrid approach

In this section, we introduce a new hybrid CPO-ACO approach where ACO and CPO are alternatively executed and exchange solutions: solutions found by ACO are used as starting points for CPO, whereas solutions found by CPO are used to update pheromone trails. More precisely, we modify Algo. 1 as follows: every $k$ iterations of the loop lines 1-9, we call CPO. When calling CPO, we supply it with the best solution constructed during the $k$ last iterations of ACO, and this solution is used by CPO as a starting point. Each call to CPO is limited to a given number of backtracks. Once CPO has reached this limit, we get the best solution found by CPO and update pheromone trails according to this solution.

The limit on the number of backtracks follows a geometric progression, as often done in classical restart strategies: the first limit is equal to $b$, and after each call to CPO, this limit is multiplied by $g$. Hence, our hybrid CPO-ACO approach may be viewed as a particular case of restart where ACO is run before each restart in order to provide a new initial solution, and the best solution after each restart is given back to ACO to reinforce its pheromone trails.

Our hybrid CPO-ACO algorithm has three parameters: the number $k$ of ACO cycles which are executed before each call to CPO, and the values $b$ and $g$ which are used to fix the limit on the number of backtracks of CPO. In all our experiments, these parameters are set to $k = 5$, $b = 1000$, and $g = 20$. With these values, the number of calls to CPO (within a time limit of one hour) ranges from 3 for the smallest instances to 4 for the largest ones.

For the ACO algorithm used in CPO-ACO, we use a parameter setting which favors a quick convergence, as only $k = 5$ cycles of ACO are run before each CPO restart, and only 3 or 4 restarts are done: $\alpha = 5$, $\beta = 10$, $\rho = 0.2$, $n_{ants} = 40$, $\tau_{min} = 0.1$, $\tau_{max} = 4$, and the pheromone definition is *Position trails*.

In CPO-ACO, CPO is run with its default setting so that CPO performs restarts during each of its runs. We have made experiments with other settings (including the *DepthFirst* search mode of CPO, which performs a single search), and the best results were obtained with the default setting of CPO.

### 6.2 Experimental evaluation

*Compared approaches.* We compare CPO-ACO with CPO in its default setting (which is the best performing setting for the GCSP). We also report results obtained with the ACO algorithm introduced in [9] (we consider the ACO variant which uses Llama to select ACO parameters for each instance as this variant obtains the best results).

Finally, we report results obtained with *Solution-Guided Multi-Point Constructive Search (SGMPCS)* [3], which is a constructive search technique that

performs a series of resource-limited tree searches where each search begins either from an empty solution (as in randomized restart) or from an "elite" solution. SGMPCS has some similarities with our approach, as it provides initial elite solutions to the CP solver. Hence, the comparison with SGMPCS allows us to evaluate the interest of using pheromone to learn good solution components that are exploited to compute new starting points. For SGMPCS, we use CPO as CP solver, and we build elite solutions with the same greedy randomised algorithm as the one used in CPO-ACO, except that we ignore the pheromone factor $f_\tau$ when computing the probability of selecting a job. The parameters of SGMPCS are set as in [3], *i.e.* the probability of starting from an empty solution is 0.25 and the size of the elite list is 4.

We separate instances in two classes for analysing results: the *closed* class contains every instance for which the optimal solution is known (either because the objective function is equal to 0, or because an approach has been able to prove optimality); the *open* class contains all other instances. For the loose (resp. medium and tight) class, there are 361 (resp. 356 and 339) closed instances, and 187 (resp. 192 and 209) open instances.

*Results for closed instances.* On the left part of Fig. 8, we display the cumulative number of solved instances with respect to time, for closed instances. After one hour of CPU time, CPO-ACO clearly outperforms all other approaches for the three classes, and it has been able to solve nearly all closed instances. More precisely, the number of loose, medium and tight instances solved by CPO-ACO in one hour is equal to 356, 352, and 329, respectively, whereas it is equal to 344, 332, and 297 for SGMPCS, to 344, 318, and 242 for ACO, and to 324, 306, and 261 for CPO. We observe that the gap between CPO-ACO and other approaches increases when increasing the tightness of GC.

However, for short time limits (smaller than 10 seconds), conclusions are different. In particular, after 1 second, the best approach is SGMPCS: The number of loose, medium and tight instances solved by SGMPCS in 1 second is equal to 122, 100, and 48, respectively, whereas it is equal to 92, 78, and 35 for CPO-ACO. This shows that good starting points allow CPO to quickly find better solutions. However, after 10 seconds, ACO is able to build better starting points by exploiting good solutions previously constructed by ACO or CPO.

*Results for open instances.* We evaluate the quality of a solution of an open instance by computing its ratio to the best known solution: if this ratio is equal to 1, the solution is the best known solution; if it is equal to $r > 1$, the solution is $r$ times as large as the best known solution.

On the right part of Fig. 8, we display the evolution of the average ratio to best known solutions with respect to time. After one hour, CPO-ACO clearly outperforms all other approaches for the three classes. Its average ratio is rather close to 1, meaning that in many cases it has found the best known solution. More precisely, for loose, medium, and tight instances this ratio is equal to 1.37, 1.38, and 1.60, respectively, for CPO-ACO, whereas it is equal to 2.18, 2.51, and 3.24 for ACO, to 2.95, 3.06, and 5.50 for SGMPCS, and to 3.86, 5.95, and 13.44 for

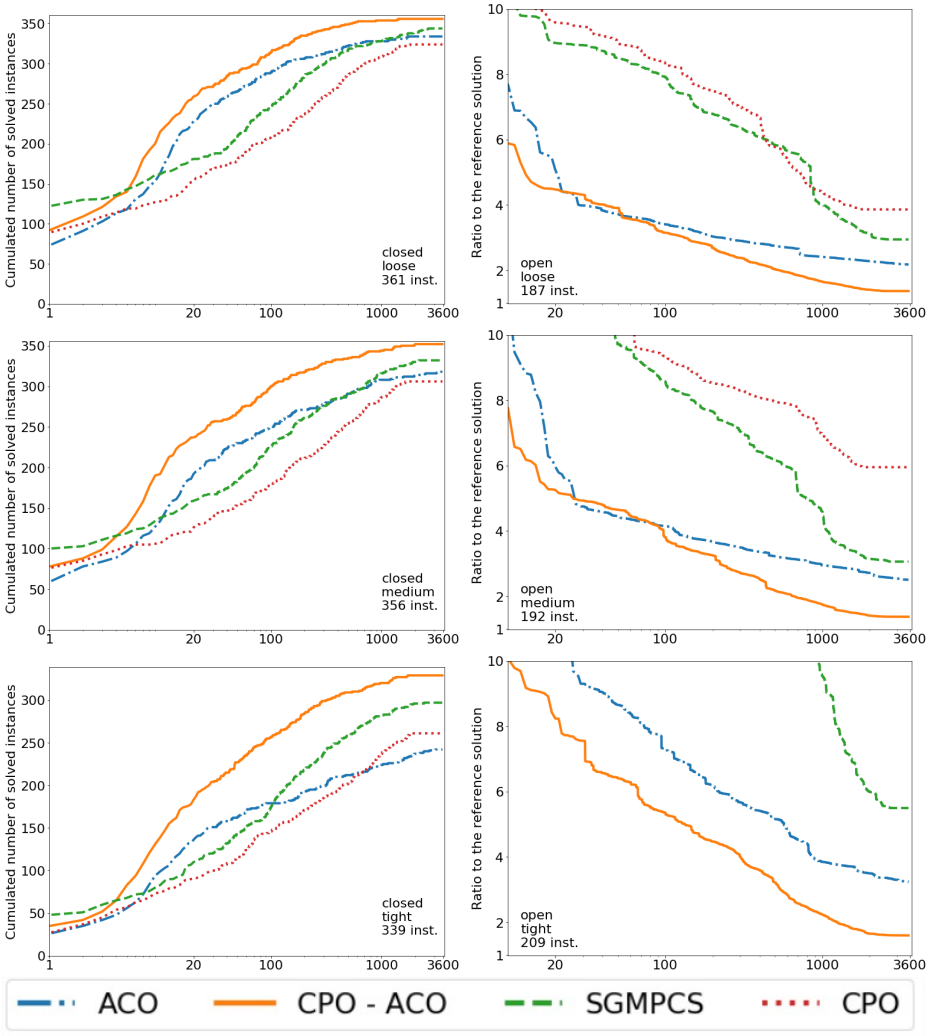


**Fig. 8.** Results of CPO-ACO, CPO, ACO, and SGMPCS. Left: Evolution of the cumulative number of solved instances for closed instances. Right: Evolution of the average ratio to the best known solution for open instances. Top: Loose instances. Middle: Medium instances. Bottom: Tight instances.

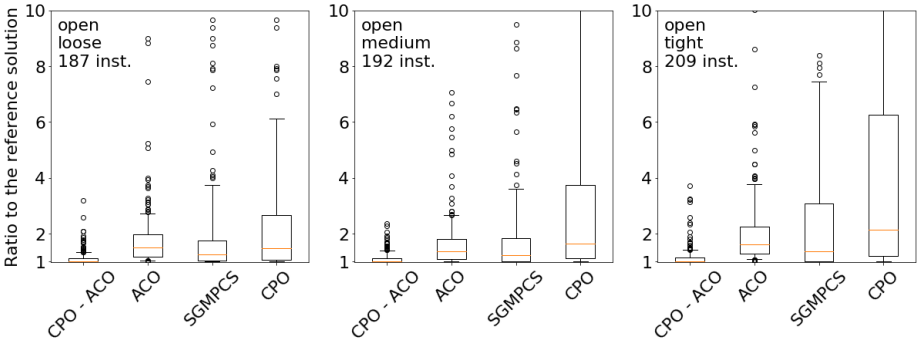


**Fig. 9.** Distribution of ratios to best known solutions after one hour on open instances.

CPO. Like for closed instances, the gap between CPO-ACO and other approaches increases when increasing the tightness of the constraint. This is particularly true for CPO which has very poor performance on tight instances. SGMPCS finds better solutions than CPO, and this shows us the interest of giving good starting points to CPO. However, like for open instances, we observe that the starting points learned by ACO allow CPO to find much better solutions.

In Fig. 9, we display ratio distributions. CPO-ACO has much smaller median ratios and inter-quartile ranges. For loose (resp. medium and tight) instances it finds the best known solution for 115 (resp.106 and 114) instances, whereas ACO finds it for 9 (resp. 21 and 14) instances, SGMPCS for 48 (resp. 44 and 34) instances and CPO for 27 (resp. 25 and 37) instances.

## 7 Conclusion

We have shown that GC is a challenging constraint for CP solvers such as CPO. In particular, reasoning on precedence relations (which is classical to solve scheduling problems) may be misleading as it is $\mathcal{NP}$-complete to find starting times that satisfy GC when a list schedule is provided. We have introduced an hybrid framework which drastically improves CPO performance by providing good starting points. These starting points are computed by an ACO algorithm which uses pheromone trails to learn good solution components.

This hybrid framework introduces new parameters: parameters to define the progression of the limit used to trigger restarts, and classical ACO parameters. In all experiments reported in this paper, we have used the same parameter setting. However, it could be interesting to use a per-instance algorithm selector such as Llama to dynamically choose the best parameter setting within a portfolio of representative and complementary settings. Other further work will concern the study of the GCSP: Fig. 5 shows us that some small instances are much harder than some large instances, and it would be interesting to identify instance parameters that characterize hardness.

## References


1. Aggoun, A., Beldiceanu, N.: Extending chip in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling **17**(7), 57–73 (Apr 1993). https://doi.org/10.1016/0895-7177(93)90068-A
2. Baptiste, P., Bonifas, N.: Redundant cumulative constraints to compute preemptive bounds. Discrete Applied Mathematics **234**, 168–177 (Jan 2018). https://doi.org/10.1016/j.dam.2017.05.001
3. Beck, J.C.: Solution-guided multi-point constructive search for job shop scheduling. J. Artif. Intell. Res. **29**, 49–77 (2007)
4. Blum, C., Puchinger, J., Raidl, G.R., Roli, A.: Hybrid metaheuristics in combinatorial optimization: A survey. Appl. Soft Comput. **11**(6), 4135–4151 (2011)
5. Bonifas, N.: A O(n 2 log(n)) propagation for the Energy Reasoning (Feb 2016)
6. Brucker, P., Drexl, A., Möhring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: Notation, classification, models, and methods. European Journal of Operational Research **112**(1), 3–41 (Jan 1999). https://doi.org/10.1016/S0377-2217(98)00204-5
7. Col, G.D., Teppan, E.C.: Industrial size job shop scheduling tackled by present day CP solvers. In: Principles and Practice of Constraint Programming - 25th International Conference, CP. Lecture Notes in Computer Science, vol. 11802, pp. 144–160. Springer (2019)
8. Gaspero, L.D., Rendl, A., Urli, T.: A hybrid ACO+CP for balancing bicycle sharing systems. In: Hybrid Metaheuristics. Lecture Notes in Computer Science, vol. 7919, pp. 198–212. Springer (2013)
9. Groleaz, L., Ndiaye, S.N., Solnon, C.: ACO with automatic parameter selection for a scheduling problem with a group cumulative constraint. In: GECCO 2020 - Genetic and Evolutionary Computation Conference. pp. 1–9. Cancun, Mexico (Jul 2020). https://doi.org/10.1145/3377930.3389818
10. Hartmann, S., Kolisch, R.: Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. European Journal of Operational Research **127**(2), 394–407 (Dec 2000). https://doi.org/10.1016/S0377-2217(99)00485-3
11. KASHIWABARA, T.: NP-completeness of the problem of finding a minimal-cliquenumber interval graph containing a given graph as a subgraph. Proc. 1979 Int. Symp. Circuit Syst pp. 657–660 (1979)
12. Khichane, M., Albert, P., Solnon, C.: Integration of ACO in a constraint programming language. In: Ant Colony Optimization and Swarm Intelligence, 6th International Conference, ANTS 2008. Lecture Notes in Computer Science, vol. 5217, pp. 84–95. Springer (2008)
13. Khichane, M., Albert, P., Solnon, C.: Strong Combination of Ant Colony Optimization with Constraint Programming Optimization. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, vol. 6140, pp. 232–245. Springer Berlin Heidelberg, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13520-0_26
14. Kotthoff, L.: LLAMA: Leveraging Learning to Automatically Manage Algorithms. arXiv:1306.1031 [cs] (Jun 2013)
15. Laborie, P.: Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. Artificial Intelligence **143**(2), 151–188 (Feb 2003). https://doi.org/10.1016/S0004-3702(02)00362-4

16. Laborie, P., Rogerie, J., Shaw, P., Vilím, P.: IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. Constraints **23**(2), 210–250 (Apr 2018). https://doi.org/10.1007/s10601-018-9281-x
17. Maniezzo, V., Stützle, T., Voß, S. (eds.): Matheuristics - Hybridizing Metaheuristics and Mathematical Programming, Annals of Information Systems, vol. 10. Springer (2010)
18. Mercier, L., Van Hentenryck, P.: Edge Finding for Cumulative Scheduling. INFORMS Journal on Computing **20**(1), 143–153 (Feb 2008). https://doi.org/10.1287/ijoc.1070.0226
19. Meyer, B.: Hybrids of Constructive Metaheuristics and Constraint Programming: A Case Study with ACO. In: Hybrid Metaheuristics, vol. 114, pp. 151–183. Springer Berlin Heidelberg, Berlin, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78295-7_6
20. Meyer, B., Ernst, A.T.: Integrating ACO and constraint propagation. In: Ant Colony Optimization and Swarm Intelligence, 4th International Workshop. Lecture Notes in Computer Science, vol. 3172, pp. 166–177. Springer (2004)
21. Neumann, K., Schwindt, C.: Project scheduling with inventory constraints. Mathematical Methods of Operations Research (ZOR) **56**(3), 513–533 (Jan 2003). https://doi.org/10.1007/s001860200251
22. Ouellet, P., Quimper, C.G.: Time-Table Extended-Edge-Finding for the Cumulative Constraint. In: Principles and Practice of Constraint Programming, vol. 8124, pp. 562–577. Springer Berlin Heidelberg, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40627-0_42
23. Pinedo, M.L.: Scheduling. Springer International Publishing, Cham (2016). https://doi.org/10.1007/978-3-319-26580-3
24. Schutten, J.: List scheduling revisited. Operations Research Letters **18**(4), 167–170 (Feb 1996). https://doi.org/10.1016/0167-6377(95)00057-7
25. Shaw, P.: Using constraint programming and local search methods to solve vehicle routing problems. In: Principles and Practice of Constraint Programming - CP98. Lecture Notes in Computer Science, vol. 1520, pp. 417–431. Springer (1998)
26. Stützle, T., Hoos, H.: Improvements on the Ant-System: Introducing the MAX-MIN Ant System, pp. 245–249. Springer Vienna, Vienna (1998). https://doi.org/10.1007/978-3-7091-6492-1_54
27. Tavares Neto, R., Godinho Filho, M.: Literature review regarding Ant Colony Optimization applied to scheduling problems: Guidelines for implementation and directions for future research. Engineering Applications of Artificial Intelligence **26**(1), 150–161 (Jan 2013). https://doi.org/10.1016/j.engappai.2012.03.011
28. C. Solnon: Constraint Programming with Ant Colony Optimization (232 pages). John Wiley and Sons (2010)
29. Vilím, P.: Timetable Edge Finding Filtering Algorithm for Discrete Cumulative Resources. In: Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, vol. 6697, pp. 230–245. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). https://doi.org/10.1007/978-3-642-21311-3_22