ACO with automatic parameter selection for a scheduling problem with a group cumulative constraint

Lucas Groleaz, Samba Ndojh Ndiaye, Christine Solnon

ACO with automatic parameter selection for a scheduling problem with a group cumulative constraint

Lucas Groleaz
Infologic, INSA Lyon, LIRIS UMR5201, CNRS, F-69621 Villeurbanne

Samba N. Ndiaye
CITI, INRIA, Univ. Lyon 1, LIRIS UMR5201, CNRS, F-69621 Villeurbanne

Christine Solnon
INSA Lyon, CITI, INRIA CHROMA, F-69621 Villeurbanne

ABSTRACT

We consider a RCPSP (resource constrained project scheduling problem), the goal of which is to schedule jobs on machines in order to minimise job tardiness. This problem comes from a real industrial application, and it requires an additional constraint which is a generalisation of the classical cumulative constraint: jobs are partitioned into groups, and the number of active groups must never exceeds a given capacity (where a group is active when some of its jobs have started while some others are not yet completed). We first study the complexity of this new constraint. Then, we describe an Ant Colony Optimisation algorithm to solve our problem, and we compare three different pheromone structures for it. We study the influence of parameters on the solving process, and show that it varies from an instance to another. Hence, we identify a subset of parameter settings with complementary strengths and weaknesses, and we use a per-instance algorithm selector in order to select the best setting for each new instance to solve. We experimentally compare our approach with a tabu search approach and an exact approach on a data set coming from our industrial application.

CCS CONCEPTS

• **Applied computing → Industry and manufacturing**; • **Theory of computation → Theory of randomized search heuristics**; *Problems, reductions and completeness.*

KEYWORDS

Ant Colony Optimization (ACO), Scheduling, Cumulative Constraint, Algorithm selection

ACM Reference Format:
Lucas Groleaz, Samba N. Ndiaye, and Christine Solnon. 2020. ACO with automatic parameter selection for a scheduling problem with a group cumulative constraint. In *Genetic and Evolutionary Computation Conference (GECCO '20), July 8–12, 2020, Cancún, Mexico.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3377930.3389818

1 INTRODUCTION

In this paper we deal with a scheduling problem coming from a real application which aims at preparing orders in food industry. More precisely, each order is composed of a set of jobs which must be scheduled on machines. The goal is to assign a machine and a start time to each job so that the sum of tardiness of all jobs is minimised. This schedule must also satisfy an additional constraint which comes from the fact that we must load a palet for each order: this palet is installed when starting the first job of the order, and it is removed when the last job of the order is completed. As the physical space is limited, the number of active palets (which are installed but not yet completed) must never exceed a given limit.

A first contribution of our paper is to study this new constraint, called *Group Cumulative (GC)* constraint. In particular, we show that we can easily model GC constraints with classical cumulative constraints by adding fictive jobs. However, as the duration of these fictive jobs is not known *a priori*, the resulting model hardly scales. We also show that the GC constraint is more difficult to tackle than the classical cumulative constraint because we cannot decide in polynomial time if it is possible to satisfy it even when we know the order of jobs on every machine.

A second contribution is an Ant Colony Optimization (ACO) algorithm for our problem. We consider and compare three pheromone structures: two of them are classical structures which have been already used to solve scheduling problems, whereas the third one is new.

Finally, we study the influence of parameters on the solution process. This sensitivity analysis highlights the fact that several parameter configurations have complementary performances, and the best configuration varies from an instance to another. We show that better results can be obtained by using a per-instance algorithm selector in order to select the best parameter configuration according to instance features.

Organisation of the paper: We describe the scheduling problem and the GC constraint in Section 2, and the ACO algorithm in Section 3. We study the influence of parameters on the solution process in Section 4, and we show how to use a per-instance algorithm selector to improve results in Section 5.

2 DESCRIPTION OF THE PROBLEM

We first describe the basic problem without the GC constraint, in Section 2.1, and then introduce the GC constraint in Section 2.2. In Section 2.3, we study the complexity of the GC constraint, and in Section 2.4 we describe our benchmark instances.

2.1 Basic scheduling problem

Scheduling problems [24] basically involve scheduling a set J of jobs on a set M of machines. More precisely, for each job $j \in J$, r_j denotes its release date, d_j its due date, and p_j its processing time. The goal is to find, for each job $j \in J$, a start time b_j, an end time e_j and a machine m_j. Different constraints and objective functions may be considered, and [4] introduces a notation to formally specify them. According to this notation, our problem is denoted $Rm, 1, 1; MPS; b_{brkdwn}|s_{ij}; r_j| \sum T_j$:

- $Rm, 1, 1$ means that M contains several machines working in parallel and each machine $m \in M$ can process at most one job at a time;
- MPS stands for multi-mode project scheduling and means that every machine $m \in M$ has its own speed denoted sp_m (so that the duration of a job j is $p_j * sp_{m_j}$);
- b_{brkdwn} indicates that each machine has its own breaks during which it cannot process any job, *i.e.*, the end time of a job j is equal to $e_j = b_j + p_j * sp_{m_j} + \Delta_{break}$ where Δ_{break} is equal to the break duration if there is a break which starts between b_j and $b_j + p_j * sp_{m_j}$ on machine m_j, and to 0 otherwise;
- $s_{i,j}$ indicates that the setup-time of a job $j \in J$ 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); a specificity of our problem is that setups cannot be done during breaks (because, in our application, a setup involves modifying the machine configuration by the machine operator);
- r_j means that each job cannot start before its release date, *i.e.*, $\forall j \in J, b_j \geq r_j$;
- $\sum T_j$ indicates that the goal is to minimize the sum of tardiness of every job, *i.e.*, $\sum_{j \in J} \max(0, e_j - d_j)$.

We have modelled this scheduling problem with IBM CPOptimizer (CPO) [15]: each job is associated with an interval variable, and we use *noOverlap* constraints to ensure that jobs scheduled on a same machine do not overlap and are separated by setup times, and *intensity* constraints to ensure that jobs are not scheduled during breaks (see [15] for details on these constraints). The choice of CPO is motivated by the fact that it has state-of-the-art results on a large range of scheduling problems. We also designed an Integer Linear Programming (ILP) model (using CPLEX), and noticed that CPO strongly outperforms CPLEX on our data-set.

2.2 Group Cumulative Constraint

In Resource Constrained Project Scheduling Problems (RCPSPs), jobs need resources (such as electricity or human skills, for example), and these resources are limited by capacities. RCPSPs are often modelled by using *cumulative* constraints [1, 2, 21, 23]: each job is associated with a height corresponding to the amount of resource needed by the job, and the cumulative constraint ensures that, at any time t, the total height of all jobs that are started but not yet finished at time t does not exceed the capacity.

In our application, we also have limited resources but these resources are needed by job groups. More precisely, our application aims at preparing a set O of orders such that each order is a set of jobs. Hence, each job $j \in J$ is associated with exactly one order denoted o_j. We define the start (resp. end) time of an order as the smallest start time (resp. largest end time) among all its jobs, and we say that an order is active at a time t when it is started and not yet ended at time t. We denote *active*(t) the set of orders which are active at time t, *i.e.*,

$$active(t) = \{o \in O : \exists j, j' \in J, o_j = o_{j'} = o \wedge b_j \leq t \leq e_{j'}\}.$$

Each active order occupies some physical space (corresponding to a pallet). As the physical space is limited, the number of active orders must be smaller than or equal to a given limit L, at any time of the schedule. Hence, our RCPSP is obtained by adding to the scheduling problem defined in Section 2.1 a *Group Cumulative (GC)* constraint which ensures that, for each time t, $\#active(t) \leq L$.

We can extend the CPO model described in Section 2.1 in order to model this constraint in a rather straightforward way:

- for each order $o \in O$, we define a fictive job f_o with an undefined duration and a height of 1;
- for each job $j \in J$, we add the constraints $b_{f_o} \leq b_j$ and $e_{f_o} \geq e_j$, where f_o is the fictive job associated with o_j order;
- we add a classical cumulative constraint on all fictive jobs in order to ensure that the number of fictive jobs started and not ended at any time t does not exceed L.

However, CPO does not scale very well on this model. This comes from the fact that durations of fictive jobs are not known (until all jobs of the corresponding order have been scheduled) and constraint propagation algorithms used by CPO (which are based on energy Reasoning [3]) do not reduce the search space in this case.

2.3 Complexity

The scheduling problem described in Section 2.1 (without resource constraints) is NP-hard [24]. However, if we know the ordered list of jobs that must be scheduled on every machine, then we can compute the start times that minimize the tardiness sum in polynomial time [9, 26]. More precisely, a *list schedule* is a set of $\#M$ ordered lists $l_1, \dots l_{\#M}$ such that each job of J occurs in exactly one list. Given a list schedule, we compute optimal start times in a greedy way: for each machine m, we consider jobs according to the order defined by l_m and schedule each of these jobs as soon as possible. Therefore, solving the scheduling problem of Section 2.1 amounts to finding the best list schedule (and start times are derived in polynomial time from these lists).

Let us now consider the cases where we add a classical cumulative constraint (Section 2.3.1), or a GC constraint (Section 2.3.2).

2.3.1 *Classical cumulative constraints.*

If we add a classical cumulative constraint to the scheduling problem of Section 2.1, the problem of computing the best 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 the problem of finding start times that satisfy cumulative constraints given a list schedule is polynomial: Again, this can be done greedily, by considering jobs in the order of the list l_m for each machine $m \in M$, and scheduling each job as soon as possible with respect to cumulative constraints.

For example, let us consider the list schedule displayed on top of Fig. 1, and let us assume that blue and yellow jobs require one unit of resource (whereas green and pink do not require any resource), and the capacity of this resource is 2. In this case, the greedy approach computes start times displayed on bottom of Fig. 1.

2.3.2 *GC constraint.*

However, this is no longer true for the GC constraint, *i.e.*, deciding if there exist start times that satisfy the GC

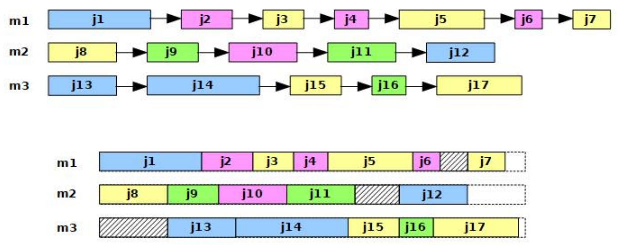

Figure 1: Computation of start times given list schedules. Top: a list schedule with 3 machines and 17 jobs. Bottom: start times computed in a greedy way in the case of a classical cumulative constraint when blue and yellow jobs require one unit of a resource which is limited to 2.

constraint given a list schedule is an NP-complete problem (even when there is no objective function to minimize)[1].

For example, let us consider the list schedule displayed on top of Fig. 1, and let us assume that there are 4 orders: the blue (resp. yellow, pink, and green) order contains jobs $\{j_1, j_{12}, j_{13}, j_{14}\}$ (resp. $\{j_3, j_5, j_7, j_8, j_{15}, j_{17}\}$, $\{j_2, j_4, j_6, j_{10}\}$, and $\{j_9, j_{11}, j_{16}\}$). Finally, let us assume that we have a GC constraint which ensures that the number of active orders never exceeds 2. In this case, it is not possible to find start times that satisfy the GC constraint for this list schedule. Indeed, on machine m_1, the yellow job j_3 is between two pink jobs j_2 and j_4, and this implies that we must start the yellow order to be able to complete the pink order. Similarly, on machine m_2, there is a pink job between two green jobs so that we must start the pink order to be able to complete the green order, and on machine m_3, there is a green job between two yellow jobs so that we must start the green order to be able to complete the yellow order. This implies that both yellow, green and pink orders must be active all together at some time and, therefore, there do not exist start times that satisfy the GC constraint for this list schedule.

2.4 Data set

We use a benchmark of instances coming from our industrial partner Infologic. Our instances have been extracted from 1182 work days in a warehouse. In these instances, the number of orders (resp. jobs and machines) ranges from 55 to 478 (resp. from 207 to 3460, and from 1 to 14). As our goal is to study the impact of the GC constraint on the solution process, we have generated two classes of instances: the first class, denoted *loose*, contains instances with a rather large capacity $L = 0.8 * X$ whereas the second class, denoted *tight*, contains instances with a smaller capacity $L = 0.5 * X$, where X is computed as follows: we use a heuristic algorithm[2] to solve the instance without the GC constraint (*i.e.*, without limiting the number of active orders), and define $X = \max_t \#active(t)$ for this solution. Finally, we have removed from our data-set every instance for which a simple greedy construction finds an optimal solution without any tardiness. We obtain 1459 instances (728 *loose* instances and 731 *tight* instances)[3].

Among these 1459 instances, 674 instances have been solved to optimality (either because CPO has been able to prove optimality, or because one of our approaches has been able to find a solution with an objective function cost equal to 0). These instances are said to be *closed*. The 785 remaining instances are said to be *open*: for these instances, we do not know the optimal solution. We evaluate the quality of a solution for an *open* instance by computing its ratio to a reference solution which is the best solution found by all approaches introduced in this paper.

3 ANT COLONY OPTIMIZATION

Ant Colony Otimization (ACO) [5, 6] is a meta-heuristic which has been used to solve various optimization problems including scheduling problems such as, for example, the resource-constrained scheduling problem [20]. A survey on solving scheduling problems with ACO is provided in [30].

ACO algorithms use pheromone trails to learn promising solution components and progressively intensify the search around them. In this paper, we consider three different kinds of pheromone trails, which are defined in Section 3.1.

Our ACO algorithm iterates over three steps: in a first step, N_{ants} solutions are constructed in a greedy randomised way, as described in Section 3.2; in a second step, the best solution of the cycle is improved with local search, as described in Section 3.3; in a third step, pheromone trails are updated, as described in Section 3.4.

3.1 Pheromone trails

We consider three different kinds of pheromone trails. The first two ones (denoted *Jobs* and *Position*, respectively) have been widely used to solve scheduling problems (according to [30], *Jobs* is used in 38 papers, and *Position* in 17 papers). The third one (denoted *Time*) is a new pheromone factor which has never been used to solve a scheduling problem (as far as we know).

Jobs pheromone trails. For each couple of jobs $(j, j') \in J^2$, we define a pheromone trail $\tau(j, j')$ which represents the learned desirability of scheduling job j' just after job j on a same machine. We add a fictive job to J (used to represent the job that precedes the first job on a machine).

Position pheromone trails. For each triple $(j, m, n) \in J \times M \times [1, \#J]$, we define a pheromone trail $\tau(j, m, n)$ which represents the learned desirability of scheduling job j on machine m at position n.

Time pheromone trails. Let H be the time horizon, discretised according to a given step size *stepSize* (*i.e.*, H is a finite set of contiguous time intervals such that the first interval starts at the beginning of the day, the last interval ends at the end of the day, and the size of each interval is equal to *stepSize*). For each couple $(j, h) \in J \times H$, we define a pheromone trail $\tau(j, h)$ which represents the learned desirability of scheduling job j at time step h.

[1]The NP-completeness proof has been done by reducing the decision version of the PATHWIDTH problem, but we do not detail it as it is out of the scope of this paper.
[2]This heuristic algorithm corresponds to a single solution construction as described in Algo. 1 when pheromone is ignored (*i.e.*, $\alpha = 0$), and line 7 is replaced with a greedy choice (*i.e.*, we choose $j \in Cand$ which maximizes $p(j)$).

Algorithm 1: Construction of a solution

```
1  $S \leftarrow \emptyset$
2  while $J \neq \emptyset$ do
3      $m_{next} \leftarrow \text{argmin}_{m \in M}\ endTime(m)$
4      $O_{open} = \{o : \exists (j, j') \in J \times S, o_j = o_{j'} = o\}$
5      if $\#O_{open} < L$ then $Cand \leftarrow J$;
6      else $Cand \leftarrow \{j \in J : o_j \in O_{open}\}$ ;
7      choose $j \in Cand$ with probability $p(j)$
8      $m_j \leftarrow m_{next}$
9      compute the start time $b_j$ and the end time $e_j$ of $j$
10     remove $j$ from $J$ and add it to $S$
```

3.2 Construction of a solution

The greedy randomised procedure used to build a solution is described in Algo. 1. It starts from an empty schedule $S = \emptyset$, and iteratively adds jobs to S (lines 2-10) until all jobs are scheduled.

At each iteration of lines 2-10, we first select the machine m_{next} which has the smallest end time, where the end time of a machine m (denoted $endTime(m)$) is equal to the end time of the last job assigned to it, *i.e.*, $endTime(m) = \max_{j \in S, m_j = m} e_j$. In lines 4-6, we build the set *Cand* of jobs that are not yet scheduled and that can be scheduled on m_{next} without violating the GC constraint. To this aim, we first compute the set O_{open} of orders o such that at least one job of o has been scheduled in S and at least one job of o has not yet been scheduled. If $\#O_{open} < L$, then we can select any job of J without violating the GC constraint (line 5). Otherwise, $\#O_{open}$ is equal to L, and in this case we restrict *Cand* to the jobs that are not yet scheduled and that belong to an order of O_{open} (line 6). Note that this filtering procedure may remove from *Cand* some jobs that could lead to better schedules. However, as it is NP-complete to decide if a job can be scheduled without violating the GC constraint, we use this simple filtering procedure to ensure feasibility.

Then, we randomly choose a job $j \in Cand$ according to the probability:

$$p(j) = \frac{[f_\tau(j, m_{next}, S)]^\alpha [\eta(j, m_{next}, S)]^\beta}{\sum\limits_{j' \in Cand} [f_\tau(j', m_{next}, S)]^\alpha [\eta(j', m_{next}, S)]^\beta}$$

where $f_\tau(j, m_{next}, S)$ is the pheromone factor, $\eta(j, m_{next}, S)$ is the heuristic factor, and α and β are parameters that are used to balance pheromone and heuristic factors.

The exact definition of the pheromone factor depends on the pheromone trails:

- for *Jobs*, we define

$$f_\tau(j, m_{next}, S) = \tau(prev_j, j)$$

  where $prev_j$ is equal to the fictive job if there is no job scheduled on m_{next}, and to the last job scheduled on machine m_{next} (*i.e.*, $prev_j = \text{argmax}_{j' \in S, m_{j'} = m_{next}} s_{j'}$) otherwise;
- for *Position*, we define

$$f_\tau(j, m_{next}, S) = \tau(j, m_{next}, n)$$

  where $n = \#\{j \in S : m_j = m_{next}\}$ is the number of jobs already scheduled on machine m_{next} in S;
- for *Time*, we define

$$f_\tau(j, m_{next}, S) = \tau(j, h)$$

  where h is the time interval associated with $endTime(m_{next})$.

The heuristic factor is the ATCS (Apparent Tardiness Cost with Setup-times) score introduced in [24]. It is a compromise between the duration of the job, the remaining time before its due date and the setup-time incurred by doing this job just after the preceding one :

$$\eta(j, m_{next}, S) = \frac{1}{p_j} e^{-\frac{max(d_j - p_j * sp_{m_{next}} - t, 0)}{\overline{p}}} e^{-\frac{s_{prev_j, j}}{\overline{s}}}$$

where $\overline{p}$ is the mean processing time of the remaining jobs, and $\overline{s}$ is the mean setup-time between all the remaining jobs.

Finally, we assign j to the machine m_{next} (line 8), and we compute the start time b_j for j (line 9): this start time is equal to the end time $endTime(m_j)$ plus the setup time between j and the last job scheduled on m_j (including breaks).

3.3 Local search

Once N_{ants} solutions have been constructed, we select the best solution among them, and improve it by local search. However, as this local search step is rather time consuming, we do not perform it at each cycle, and introduce a parameter q_{LS} to control the frequency of this local search step: q_{LS} is the probability of applying local search to the best solution of the cycle.

We consider a classical neighborhood for scheduling problems (used, for example, in [16]): we select the job with the largest tardiness, remove it from its machine, and explore all neighbor solutions obtained by inserting this job elsewhere. We consider a first improvement policy, *i.e.*, we stop exploring the neighborhood when finding a neighbor which improves the objective function and satisfies the GC constraint. If there is no improving neighbor, we select the best neighbor. We use a tabu list to prevent the search from cycling by forbidding to move a job which has been recently moved [7].

This local search process is stopped when the number of non-improving moves is equal to $d_{LS} * \frac{\#J}{100}$, where d_{LS} is a parameter.

3.4 Pheromone updating step

We consider the Max-Min Ant System (MMAS) framework [29][4]. Hence, we introduce two parameters τ_{min} and τ_{max}, and every pheromone trail is bounded between τ_{min} and τ_{max}. Also, we initialize every pheromone trail to τ_{max} at the beginning of the search process.

At the end of each cycle (once N_{ants} solutions have been constructed, and the best of these solutions has been improved by local search), pheromone trails are updated in two steps. In a first step, pheromone evaporation is simulated by multiplying every pheromone trail with $1 - \rho$ where $\rho \in [0, 1]$ is the pheromone evaporation rate.

[3]These instances are available at `perso.citi-lab.fr/csolnon/gc-sched.html`.

[4]We also made experiments with P-ACO [8], and obtained rather similar performance than with MMAS.

In a second step, the best solution of the cycle (denoted s_{cycle}) is rewarded. The quantity of pheromone added is defined by

$$\Delta = 1 - \frac{f(s_{cycle}) - f(s_{run})}{f(s_{run})}$$

where s_{run} is the best solution found since the beginning of the run, and $f(s)$ is the objective function value of a solution s. Note that when $f(s) = 0$, then we have found an optimal solution (such that every job ends before its due date) and we stop the run.

This quantity Δ of pheromone is added on pheromone trails associated with s_{cycle}:

- for *Jobs* pheromone trails, it is added on every trail $\tau(j, j')$ such that either job j immediately precedes job j' on a machine, or j is the fictive job and j' is the first job on a machine;
- for *Position*, it is added on every trail $\tau(j, m, n)$ such that j is the n^{th} job scheduled on machine m;
- for *Time*, it is added on every trail $\tau(j, h)$ such that job j is scheduled at time step h. We also reward pheromone trails $\tau(j, h-k)$ and $\tau(j, h+k)$. Indeed, if it is good to schedule j at time step h, then it should be good too to schedule j at time steps close to h. We consider a Gaussian reward, as introduced in [27] for continuous problems. More precisely, we introduce a parameter std_{dev} and, for every integer value $k \geq 1$, we define $\Delta_k = \Delta \cdot e^{-\frac{1}{2}(\frac{k}{std_{dev}})^2}$. For every job j which is scheduled at time step h, and for every integer value $k \geq 1$ such that $\Delta_k > 0.01$, we add Δ_k to $\tau(j, h-k)$ and $\tau(j, h+k)$.

4 INFLUENCE OF PARAMETERS ON THE SOLUTION PROCESS

Our algorithm has classical ACO parameters: the number of solutions constructed at each cycle N_{ants}, the pheromone factor weight α, the heuristic factor weight β, the pheromone evaporation rate ρ, and the pheromone bounds τ_{min} and τ_{max}.

There is also one hyper-parameter which is used to select the kind of pheromone trails, as described in Section 3.1. This hyper-parameter is denoted $\tau_{struture}$ and its possible values are *Jobs*, *Position*, and *Time*. When $\tau_{struture} = Time$, there are two additional parameters: the size of time intervals *StepSize* (in seconds), and the standard deviation considered in the Gaussian reward std_{dev}.

Finally, there are two parameters which are used to configure the local search step: the probability of applying local search to the best solution of the cycle q_{LS}, and a parameter d_{LS} which controls the number of non improving moves before stopping local search.

When α is set to 0, pheromone trails are not used and we skip the pheromone updating step (and parameters ρ, τ_{min}, τ_{max}, and $\tau_{structure}$ are ignored). In this case, our algorithm may be viewed as a kind of Greedy Randomized Adaptive Search Procedure (GRASP), as described in [25], for example.

4.1 Automatic parameter configuration

Automatic configuration tools are now widely used to tune parameters of ACO algorithms (see [19], [18] or [28], for example).

We have used ParamILS [10] to search for a good setting of our parameters. This search has been performed on a subset of 10 representative instances: all these instances are non trivial ones (*i.e.*, CPO is not able to solve them within one hour), and have various sizes; half of them are *loose* instances, and the other half are *tight* instances. Each run has been limited to one hour of CPU time, and the total running time of ParamILS has been limited to one week. All experiments reported in this paper have been performed on a processor Intel(R) Xeon(R) CPU E5-2650 v4 @ 2.20GHz with 4 GB RAM.

Table 1: Automatic parameter configuration: each column corresponds to a parameter and contains the values initially provided to paramILS for this parameter. We highlight in grey the parameter setting chosen by paramILS.

α	β	N_{ants}	ρ	τ_{min}	τ_{max}
0	1	10	0.02	0.01	1
1	5	30	0.1	0.1	3
3	10	50	0.25	1	5
5	15		0.5		
10			0.9		

$\tau_{structure}$	StepSize	std_{dev}
Job	10	1
Position	60	5
Time	300	10

q_{LS}	d_{LS}
0	0.1
0.1	1
0.5	10
1	100

Table 1 gives, for each parameter, the set of values which have been considered by paramILS and highlights in grey the parameter chosen by ParamILS. Our algorithm with this parameter configuration is denoted *ACO paramILS*. In this configuration, $\tau_{structure}$ is set to Time, *i.e.*, pheromone is used to learn the desirability of scheduling jobs at time steps, with time steps of 60 seconds and a standard deviation std_{dev} equal to 1. Also, q_{LS} is set to 0.5, meaning that local search is applied (on average) every 2 cycles.

4.2 Sensitivity analysis

To evaluate the influence of parameters on the solution process, we fix some parameters to the setting found by paramILS, and change the values of the other parameters. For each configuration, we plot the evolution of the ratio $\frac{f}{f^*}$ (on the y-axis) with respect to time t (on the x-axis, with a log-scale), where f is the total tardiness of the best solution found after t seconds of CPU time, and f^* is the total tardiness of a reference solution (which is the best solution found by all configurations within a time limit of one hour). Hence, when $\frac{f}{f^*} = 1$, the configuration has found the reference solution.

Results are presented separately for four different instances which have been randomly chosen, and we plot average results on 5 runs (with different random seeds) for each instance. In all figures, the curve in orange corresponds to the configuration computed by paramILS.

Figure 2 displays results with different configurations for α and ρ, ranging from $\alpha = 0$ (where pheromone is not considered at all) to $\alpha = 10$; $\rho = 0.25$ (where pheromone has a strong influence on the solution process). When $\alpha = 0$, we skip the pheromone updating step and, therefore, more solutions are constructed within a same time limit. However, nothing is learned from a construction to another and better results are obtained with larger values for

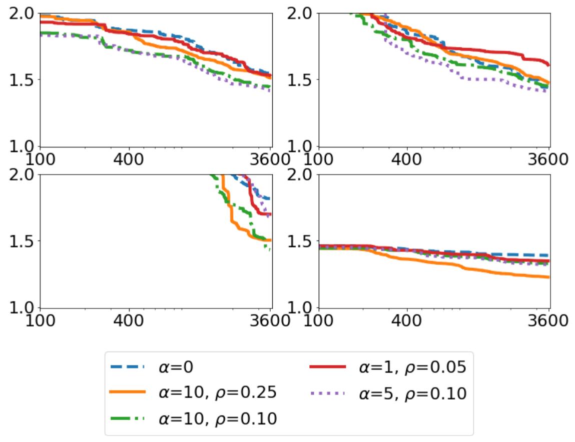

Figure 2: Influence of α and ρ

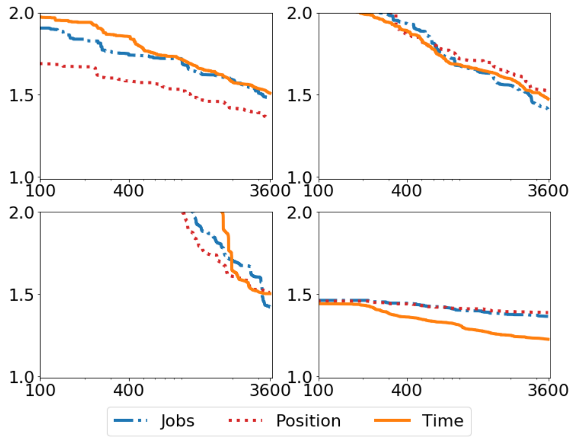

Figure 3: Influence of $\tau_{structure}$

α, for the four considered instances. However, the best setting for α and ρ is different from one instance to the other: for instances corresponding to the two top plots, the best results are obtained when $\alpha = 5$ and $\rho = 0.1$ and when $\alpha = 10$ and $\rho = 0.1$. However, for the two other instances (and more particularly for the bottom right one), the best results are obtained with the setting computed by paramILS ($\alpha = 10$ and $\rho = 0.25$).

Figure 3 displays results when changing the value of $\tau_{structure}$, *i.e.*, changing the definition of pheromone trails. Again, the best setting changes from an instance to the other. For example, for the top left (resp. bottom right) instance, the best results are obtained with *Position* (resp. *Time*).

Figure 4 displays results when changing the values of *stepSize* and Std_{dev}, which are the two extra parameters used when $\tau_{structure}$ is set to *Time*. Again, the best setting varies from an instance to the other: on the top left instance, the configuration computed by paramILS obtains the worst results whereas on the bottom right it obtains the best results.

Finally, Figure 5 displays results when changing the values of q_{LS} and d_{LS}, which are the two parameters related to the Local Search

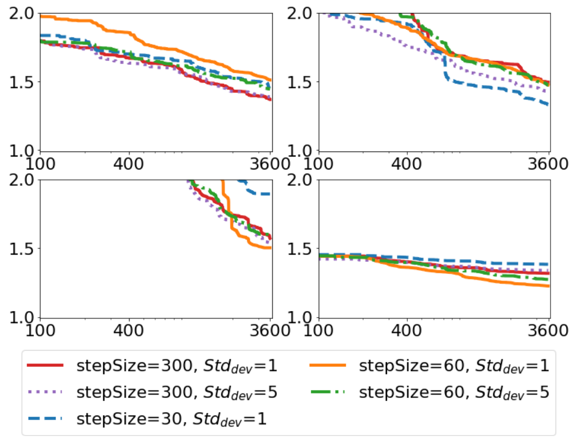

Figure 4: Influence of StepSize and std_{dev}

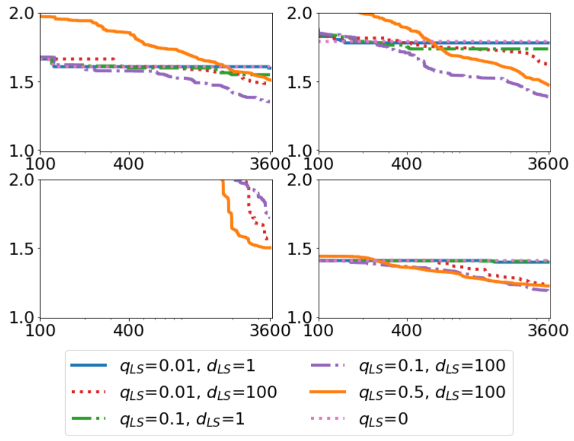

Figure 5: Influence of d_{LS} and q_{LS}

step. When $q_{LS} = 0$, local search is never triggered and, for the four considered instances, better results are obtained with larger values of q_{LS}, showing the interest of using local search to improve solutions. However, for the two top (resp. bottom) instances, better results are obtained when $q_{LS} = 0.1$ (resp. $q_{LS} = 0.5$), *i.e.*, when local search is triggered every 10 (resp. 2) cycles, on average.

Hence, the main conclusion of this section is that the best parameter configuration is very different from an instance to another.

5 PER INSTANCE PARAMETER SELECTION

As the best parameter configuration strongly varies from an instance to another, we propose to automatically choose a different configuration for each new instance to solve. To this aim, we could dynamically tune ACO parameters as proposed, for example, in [11, 14, 22]). Another solution is to use a per-instance algorithm selectors which selects from an algorithm portfolio the algorithm expected to perform best on a given problem instance. One of the most prominent systems that employs this approach is SATzilla [31],

Table 2: Configurations used by Llama: For each configuration, we give its parameter values, and #Best gives the number of instances for which this configuration has obtained the best results (for *closed*, *open*, and all instances, repectively).

Number (name)	α	β	N_{ants}	q_{LS}	d_{LS}	ρ	τ_{max}	τ_{min}	Structure	$stepSize$	std_{dev}	#Best		
												Closed	*Open*	Total
1 (ACO paramILS)	10	10	50	0.5	100	0.25	1	0.1	Time	60	1	227	215	442
2 (ACO Single Best)	3	10	40	0.01	100	0.05	4	0.1	Jobs			188	270	458
3	3	10	40	0		0.05	4	0.1	Jobs			171	126	297
4	3	10	40	0.01	100	0.05	4	0.1	Position			201	200	401
5	3	10	40	0		0.05	4	0.1	Position			239	171	410
6	1	10	40	0.01	100	0.05	4	0.1	Time	180	5	92	213	305
7	3	10	40	0.01	100	0.05	4	0.1	Time	60	3	46	192	238
8	5	10	40	0.01	100	0.05	4	0.1	Time	180	5	92	171	263
9 (Tabu search)	0	10	40	1	100							160	186	346

which defined the state of the art in SAT solving for a number of years (see [12] for additional information on algorithm selection).

In this section, we use the R package Llama [13] to learn an algorithm selection model. Llama supports many selection approaches. We performed a set of preliminary experiments to determine the approach that works best for our application, *i.e.*, regularized random forest. We use 10-fold cross-validation to evaluate the performance of Llama (*i.e.*, our benchmark is randomly partitioned into 10 subsets, and we repeat 10 experiments where 9 subsets are used to train a selection model which is evaluated on the remaining subset).

Llama uses features (extracted from instances) to learn the selection model. Hence, each instance must be described by a set of features. In our experiments, we have considered the following features: the number of jobs $\#J$, the number of machines $\#M$, the number of orders $\#O$, the capacity L, job release dates r_j, due dates d_j, durations p_j, and setup-times s_{ij} (for r_j, d_j, p_j, and s_{ij}, we compute the average, minimal, maximal, and standard deviations among all jobs). We also extract some features from breaks, and the due date range, the due date tightness, the job-machine factor and the setup-time severity as defined in [17].

The model is trained to select one algorithm within a given portfolio of algorithms. In our context, this portfolio contains different instances of our ACO algorithm, corresponding to different parameter configurations. We have selected 9 complementary parameter configurations which are described in Table 2. The last column of Table 2 gives the number of instances (among the 1459 instances of our benchmark) for which the configuration has obtained the best results (the smallest CPU time for *closed* instances, and the smallest objective function value for *open* instances). Note that the total of this column is greater than 1459 because for some instances several approaches may have the same results.

Configuration 1 corresponds to *ACO paramILS*, and it obtains the best results on 442 instances (227 *closed* and 215 *open*). This shows us that the automatic search performed by paramILS on a subset of 10 instances generalizes quite well on the complete benchmark as it is the best configuration for more than 30% of the instances.

However, Configuration 2 is slightly better than *ACO paramILS* (it obtains the best results on 16 more instances). If these two configurations have close results, they have very different parameter settings: in Configuration 2, α and ρ are much smaller (which means that the influence of pheromone is less important), q_{LS} is 5 times as small (which means that local search is triggered 5 times less often), and $\tau_{structure} = Jobs$ (which means that pheromone is used to learn job precedence relations instead of job time relations). Configuration 2 is called *ACO Single Best* as it is the best performing configuration (among the 9 considered configurations).

Configuration 9 is called *Tabu Search* because it actually corresponds to a multi-start Tabu search approach as $\alpha = 0$ (and pheromone is not updated), and the probability of applying local search to improve a constructed solution is $q_{LS} = 1$.

We call *Llama ACO* the approach which first uses the model learned by Llama to select one configuration (among the 9 configurations listed in Table 2), and then solves the instance with the selected configuration. We compare this approach with 4 other approaches, *i.e.*, *ACO paramILS* (corresponding to configuration 1), *ACO Single Best* (corresponding to configuration 2), *Tabu Search* (corresponding to configuration 9), and IBM CPOptimizer. We also report results of a Virtual Best Solver (denoted *VBS ACO*) which selects for each instance the best performing ACO configuration (among the 9 configurations listed in Table 2). This approach is purely virtual as it can be designed only if we have an oracle able to predict what is the best configuration without any error.

In figure 6, we plot the cumulative number of solved instances with respect to time for the 674 *closed* instances. CP Optimizer has the worst performance. This was expected as it is a complete approach, which is able to prove optimality whereas all other approaches are heuristic approaches. *Tabu search* quickly finds the optimal value on many instances, but after one hour it has been able to find the optimal solution for only 584 instances. *ACO paramILS*, *ACO Single Best*, and *Llama ACO* solve less instances than *Tabu search* for short CPU time limits (smaller than 20 seconds), but they solve more instances when increasing the time limit. When the time limit is equal to one hour, *ACO paramILS* (resp. *ACO Single Best* and *Llama ACO*) have been able to solve 617 (resp. 626 and 640) instances. Hence, on *closed* instances, using Llama allows us to slightly improve results. *VBS ACO* shows us what could be expected if we had a perfect oracle, *i.e.*, if the model learned by Llama were perfect.

Let us now consider the 785 *open* instances, for which we do not know the optimal solution. In Figure 7, we plot the evolution of the ratio between the best solution found within t seconds and the reference solution (on average for the 785 instances) when

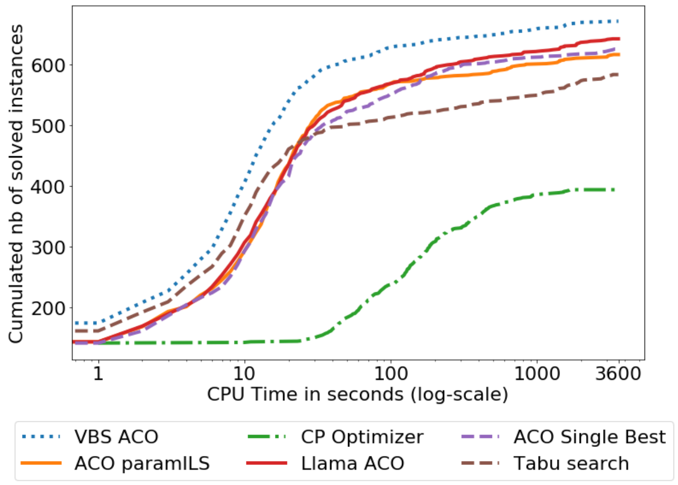

Figure 6: Results for the 674 *closed* instances: Evolution of the cumulative number of solved instances with respect to time

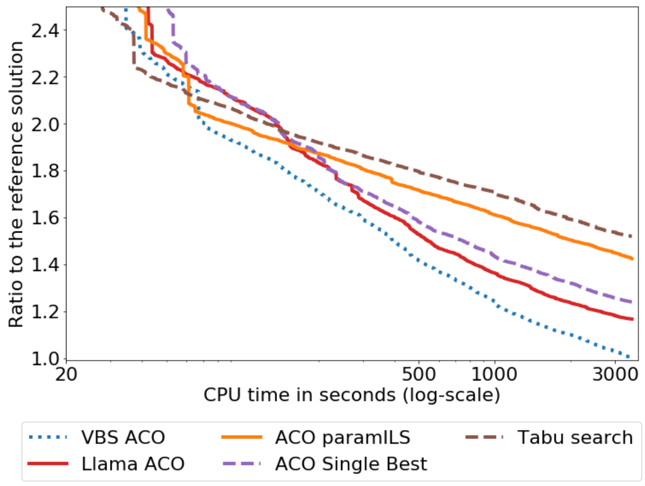

Figure 7: Results for the 785 *open* instances: Evolution of the average ratio between the best solution found within t seconds and the reference solution when increasing the time limit t.

increasing the time limit t. We do not plot results for CP Optimizer as it is not able to find any solution on the hardest instances. Again, for short time limits (smaller than 80 seconds), *Tabu Search* obtains the best results. However, for longer time limits, all ACO variants outperform *Tabu Search*: for a time limit of one hour, the ratio to the reference solution is equal to 1.17 (resp. 1.24, 1.42, and 1.52) for *Llama ACO* (resp. *ACO Single Best*, *ACO paramILS*, and *Tabu Search*). In other words, the best solution it found is 17% (resp. 24%, 42%, and 52%) larger than the reference solution. Hence, on these instances, using LLAMA allows us to clearly improve results compared to a single configuration.

6 CONCLUSION

We have introduced a new scheduling problem, corresponding to a real industrial problem where jobs are partitioned into groups and the number of active groups must never exceed a given capacity. We have introduced a new ACO algorithm for this problem, and compared three different kinds of pheromone trails: two of them are classical ones whereas the third one is a new one. We have shown that the best parameter setting strongly varies from one instance to another, and we have shown how to use a per-instance algorithm selector to dynamically choose a parameter configuration for each new instance to solve. We have experimentally evaluated our approach on a large benchmark of 1459 instances, and we have shown that our ACO algorithm with a static parameter setting clearly outperforms a Tabu search algorithm, and that even better results are obtained when using machine learning to select the parameter configuration for each instance separately.

When comparing the results obtained by *Llama ACO* with the results obtained by a virtual best solver (which always selects the best configuration for each instance), we notice that there is still room for improving the model used to select configurations. Hence, further work will aim at improving this model. In particular, we will study the interest of using other features to describe instances, and other machine learning algorithms to learn the model.

REFERENCES

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

[2] Philippe Baptiste and Nicolas Bonifas. 2018. Redundant cumulative constraints to compute preemptive bounds. *Discrete Applied Mathematics* 234 (Jan. 2018), 168–177. https://doi.org/10.1016/j.dam.2017.05.001

[3] Nicolas Bonifas. 2016. A O(n2 log(n)) propagation for the Energy Reasoning. (2016), 4.

[4] Peter Brucker, Andreas Drexl, Rolf Möhring, Klaus Neumann, and Erwin Pesch. 1999. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research* 112, 1 (Jan. 1999), 3–41. https://doi.org/10.1016/S0377-2217(98)00204-5

[5] Alberto Colorni, Marco Dorigo, and Vittorio Maniezzo. 1991. Distributed Optimization by Ant Colonies. In *Proceedings of the First European Conference on Artificial Life.*

[6] Marco Dorigo and Thomas Stützle. 2004. *Ant colony optimization.* MIT Press, Cambridge, Mass. OCLC: 834298732.

[7] Fred Glover, James P. Kelly, and Manuel Laguna. 1996. New advances and applications of combining simulation and optimization. In *Proceedings of the 28th conference on Winter simulation - WSC '96.* ACM Press, Coronado, California, United States, 144–152. https://doi.org/10.1145/256562.256595

[8] Michael Guntsch and Martin Middendorf. 2002. Applying Population Based ACO to Dynamic Optimization Problems. In *Ant Algorithms (Lecture Notes in Computer Science).* Springer, Berlin, Heidelberg, 111–122. https://doi.org/10.1007/3-540-45724-0_10

[9] Sönke Hartmann and Rainer Kolisch. 2000. Experimental evaluation of state-of-the-art heuristics for the resource-constrained project scheduling problem. *European Journal of Operational Research* 127, 2 (Dec. 2000), 394–407. https://doi.org/10.1016/S0377-2217(99)00485-3

[10] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stuetzle. 2009. ParamILS: An Automatic Algorithm Configuration Framework. *Journal of Artificial Intelligence Research* 36 (Oct. 2009), 267–306. https://doi.org/10.1613/jair.2861

[11] Madjid Khichane, Patrick Albert, and Christine Solnon. 2009. An ACO-Based Reactive Framework for Ant Colony Optimization: First Experiments on Constraint Satisfaction Problems. In *Learning and Intelligent Optimization.* Vol. 5851. Springer Berlin Heidelberg, Berlin, Heidelberg, 119–133. https://doi.org/10.1007/978-3-642-11169-3_9

[12] Lars Kotthoff. 2014. Algorithm Selection for Combinatorial Search Problems: A Survey. *AI Magazine* 35, 3 (Sept. 2014), 48–60. https://doi.org/10.1609/aimag.v35i3.2460

[13] Lars Kotthoff. 2014. LLAMA: Leveraging Learning to Automatically Manage Algorithms. *arXiv:1306.1031 [cs]* (April 2014). http://arxiv.org/abs/1306.1031

arXiv: 1306.1031.

[14] Kuan Yew Wong and Komarudin. 2008. Parameter tuning for ant colony optimization: A review. In *2008 International Conference on Computer and Communication Engineering*. IEEE, Kuala Lumpur, Malaysia, 542–545. https://doi.org/10.1109/ICCCE.2008.4580662

[15] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím. 2018. IBM ILOG CP optimizer for scheduling: 20+ years of scheduling with constraints at IBM/ILOG. *Constraints* 23, 2 (April 2018), 210–250. https://doi.org/10.1007/s10601-018-9281-x

[16] Jae-Ho Lee, Jae-Min Yu, and Dong-Ho Lee. 2013. A tabu search algorithm for unrelated parallel machine scheduling with sequence- and machine-dependent setups: minimizing total tardiness. *The International Journal of Advanced Manufacturing Technology* 69, 9-12 (Dec. 2013), 2081–2089. https://doi.org/10.1007/s00170-013-5192-6

[17] Young Hoon Lee and Michael Pinedo. 1997. Scheduling jobs on parallel machines with sequence-dependent setup times. *European Journal of Operational Research* 100, 3 (Aug. 1997), 464–474. https://doi.org/10.1016/S0377-2217(95)00376-2

[18] Pengpeng Lin, Jun Zhang, and Marco A. Contreras. 2015. Automatically configuring ACO using multilevel ParamILS to solve transportation planning problems with underlying weighted networks. *Swarm and Evolutionary Computation* 20 (Feb. 2015), 48–57. https://doi.org/10.1016/j.swevo.2014.10.006

[19] Manuel López-Ibáñez and Thomas Stützle. 2010. Automatic Configuration of Multi-Objective ACO Algorithms. In *Swarm Intelligence*. Vol. 6234. Springer Berlin Heidelberg, Berlin, Heidelberg, 95–106. https://doi.org/10.1007/978-3-642-15461-4_9

[20] D. Merkle, M. Middendorf, and H. Schmeck. 2002. Ant colony optimization for resource-constrained project scheduling. *IEEE Transactions on Evolutionary Computation* 6, 4 (Aug. 2002), 333–346. https://doi.org/10.1109/TEVC.2002.802450

[21] Klaus Neumann and Christoph Schwindt. 2003. Project scheduling with inventory constraints. *Mathematical Methods of Operations Research (ZOR)* 56, 3 (Jan. 2003), 513–533. https://doi.org/10.1007/s001860200251

[22] Héctor Neyoy, Oscar Castillo, and José Soria. 2013. Dynamic Fuzzy Logic Parameter Tuning for ACO and Its Application in TSP Problems. In *Recent Advances on Hybrid Intelligent Systems*. Vol. 451. Springer Berlin Heidelberg, Berlin, Heidelberg, 259–271. https://doi.org/10.1007/978-3-642-33021-6_21

[23] Pierre Ouellet and Claude-Guy Quimper. 2013. Time-Table Extended-Edge-Finding for the Cumulative Constraint. In *Principles and Practice of Constraint Programming*. Vol. 8124. Springer Berlin Heidelberg, Berlin, Heidelberg, 562–577. https://doi.org/10.1007/978-3-642-40627-0_42

[24] Michael L. Pinedo. 2016. *Scheduling*. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-26580-3

[25] Mauricio G. C. Resende and Celso C. Ribeiro. 2003. Greedy Randomized Adaptive Search Procedures. In *Handbook of Metaheuristics*. Springer US, Boston, MA, 219–249. https://doi.org/10.1007/0-306-48056-5_8

[26] J.M.J. Schutten. 1996. List scheduling revisited. *Operations Research Letters* 18, 4 (Feb. 1996), 167–170. https://doi.org/10.1016/0167-6377(95)00057-7

[27] Krzysztof Socha and Marco Dorigo. 2008. Ant colony optimization for continuous domains. *European Journal of Operational Research* 185, 3 (March 2008), 1155–1173. https://doi.org/10.1016/j.ejor.2006.06.046

[28] Wouter Souffriau, Pieter Vansteenwegen, Greet Vanden Berghe, and Dirk Van Oudheusden. 2008. Automated Parameterisation of a Metaheuristic for the Orienteering Problem. In *Adaptive and Multilevel Metaheuristics*. Vol. 136. Springer Berlin Heidelberg, Berlin, Heidelberg, 255–269. https://doi.org/10.1007/978-3-540-79438-7_13

[29] T. Stützle and H. Hoos. 1998. Improvements on the Ant-System: Introducing the MAX-MIN Ant System. In *Artificial Neural Nets and Genetic Algorithms*. Springer Vienna, Vienna, 245–249. https://doi.org/10.1007/978-3-7091-6492-1_54

[30] R.F. Tavares Neto and M. Godinho Filho. 2013. 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 (Jan. 2013), 150–161. https://doi.org/10.1016/j.engappai.2012.03.011

[31] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. 2008. SATzilla: Portfolio-based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research* 32 (July 2008), 565–606. https://doi.org/10.1613/jair.2490