Leveraging Constraint Scheduling: A Case Study to the Textile Industry

Alexandre Mercier-Aubin[1], Jonathan Gaudreault[1], and Claude-Guy Quimper[1]

Université Laval, Québec, QC, G1V 0A6, Canada

Abstract. Despite the significant progress made in scheduling in the past years, industrial problems with several hundred tasks remain intractable for some variants of the scheduling problems. We present techniques that can be used to leverage the power of constraint programming to solve an industrial problem with 800 non-preemptive tasks, 90 resources, and sequence-dependent setup times. Our method involves solving the traveling salesperson problem (TSP) as a simplification of the scheduling problem and using the simplified solution to guide the branching heuristics. We also explore large neighborhood search. Experiments conducted on a dataset provided by our partner from the textile industry show that we obtain non-optimal but satisfactory solutions.

Keywords: multi-resource · scheduling · constraint programming · traveling salesman problem

1 Introduction

Nowadays, most textiles are mass-produced using automated looms. In the context of the fourth industrial revolution [7, 8], the textile industry seeks to expand the automation to planning and scheduling tasks. While recent progress made in constraint programming allows tackling many NP-Hard scheduling problems, the size of the scheduling instances that can be solved for some variants of the problem remain small compared to what the industry needs. It is common to observe industrial instances with 800 tasks and limited resources. In this context, constraint programming can still be used to obtain good, but not optimal, schedules. However, extra work on branching heuristics and the use of local search is often required to obtain satisfactory results.

We present a study case of a scheduling problem encountered by our industrial partner, a textile company. More than 800 tasks need to be scheduled over 90 automated looms. A team of technicians needs to set up the looms before starting new tasks. The duration of each setup depends on the tasks that precede and succeed the setup and no more setups than technicians should be simultaneously scheduled.

We explain how we succeed in obtaining good, but possibly non-optimal, solutions for large instances. We achieve this goal by solving a simplification of the problem and using the simplification solution to find better solutions to the non-simplified problem. We also use a large neighborhood search to improve the

solution. Note that similar problems were studied in the literature [**?**], but the approach to solve the problem relies on different optimization techniques such as Mixed Integer Programs.

The paper is divided as follows. Section 2 presents the preliminary concepts about constraint scheduling, the connection between scheduling with setup times and the traveling salesman problem (TSP), and the large neighborhood search. Section 3 formally introduces the industrial scheduling problem. Section 4 presents the mathematical models. Section 5 shows how to solve the models. Section 6 presents the experimental results. Finally, Section 7 concludes this work.

2 Background

2.1 Constraint Scheduling

We present the main components of common scheduling problems. The actual industrial scheduling problem we will solve is formally defined in Section 3.

A *scheduling problem* is composed of a set of tasks $\mathcal{I}$ (or activities) that need to be positioned on a time line. A task $i \in \mathcal{I}$ has for parameters its *earliest starting time* est_i, its *latest completion time* lct_i, a *processing time* p_i, a *due date* d_i, and a resource consumption rate h_i also called *height*. We consider non-preemptive tasks, i.e. that when a task starts, it executes for exactly p_i units of time without interruption. A task must start no earlier than its earliest starting time and complete no later than its latest completion time. A task that completes after its due date incurs a penalty that depends on the objective function.

A *cumulative resource* r has capacity C_r. Multiple tasks can execute on a cumulative resource r as long as the sum of their heights is no greater than C_r. By fixing $C_r = h_i$ for all tasks i, we obtain a *disjunctive resource* that can only execute one task at a time. On such a resource, it could happen that a setup needs to be performed between the execution of two tasks. The *setup time* $t_{i,j}$ is a minimum lapse of time that must occur between the completion of task i and the starting time of task j, should task i executes before task j. The setup times satisfy the triangle inequality $t_{i,j} + t_{j,k} \geq t_{i,k}$.

Constraint programming can be used to solve scheduling problems. One can define a *starting time* variable S_i with domain $\text{dom}(S_i) = [\text{est}_i, \text{lct}_i - p_i]$. The constraints $\textsc{Cumulative}([S_1, \ldots, S_n], [p_1, \ldots, p_n], [h_1, \ldots, h_n], C_r)$ or $\textsc{Disjunctive}([S_1, \ldots, S_n], [p_1, \ldots, p_n])$ ensures that the cumulative or disjunctive resource is not overloaded. Extra constraints can easily be added to the model such as a *precedence constraint* $S_i + p_i + t_{i,j} \leq S_j$ that forces a task i to complete before a task j can start.

Combining the concepts presented above results in a large variety of scheduling problems. For instance, the Resource-Constrained Project Scheduling Problem (RCPSP) contains cumulative resources and non-preemptive tasks subject to precedence constraints. Constraint solvers can find optimal solutions to instances of the RCPSP with 120 tasks [15]. However, with setup times, constraint solvers can only solve instances up to 120 tasks but never to optimality [1]. In

order to improve the performances, search strategies and branching heuristics can be provided to the solver. It is also possible to use the constraint solver within a large neighborhood as explained in Section 2.2.

2.2 Local Search

A *local search* is a heuristic method that starts from a suboptimal and possibly unfeasible solution and tries to improve its feasibility and its objective value. At each iteration, an *operator* is applied, and the solution is modified. If the modified solution becomes more feasible or more optimal, it becomes the current solution. The operator modifies the values of a subset of the variables in the problem. When a large number of variables are modified, we say that the heuristic method is a *large neighborhood search*.

A constraint solver can be used as a large neighborhood search operator. One takes the current solution and forces a subset of variables to take the same values as the current solution. The remaining variables are let free. The result is a constraint satisfaction problem (CSP) with fewer variables to assign that is generally easier to solve. The solution of this CSP becomes the new current solution that can be further improved by selecting a different subset of variables.

There exists various strategies to select which variables to reassign in a scheduling problem. One could randomly select a fixed percentage of the tasks to reschedule [9]. It is also possible to select a time window from which all tasks whose execution is contained in this window are rescheduled [14]. Another option is to fix some precedences observed in the current solution between a subset of pairs of tasks and let the remaining pairs of tasks free from any precedence [13]. When minimizing the makespan in scheduling problems of 300 tasks, these techniques can significantly improve the objective function [4].

2.3 The Traveling Salesperson Problem

The traveling salesperson problem (TSP) is a classic optimization problem. In its directed variant, we have n cities and a distance matrix D such that $D_{i,j}$ is the distance to travel from i to j. The matrix satisfies the triangle inequality $D_{i,j} + D_{j,k} \geq D_{i,k}$. The salesperson plans on finding the shortest circuit visiting each city exactly once.

There is a strong connection between the TSP and the scheduling problem with setup times. For a scheduling problem with n tasks $\mathcal{I}$ and setup times $t_{i,j}$, one creates an instance of the TSP with $n+1$ cities and a distance matrix $D_{i,j} = t_{i,j}$ for $i, j \in \mathcal{I}$ and $D_{i,j} = 0$ otherwise. The extra city marks the beginning (and the end) of the schedule. The salesperson visits the cities (tasks) in order to minimize the sum of the distances (setup times). Figure 1 illustrates the reduction.

The solver Concorde [3] represents the state-of-the-art for solving the TSP. It can find and prove optimal solutions to instances with 85,000 cities. As it was designed solely for this type of problem, it cannot handle additional constraints. For example, if the tasks have earliest starting times and latest completion times,

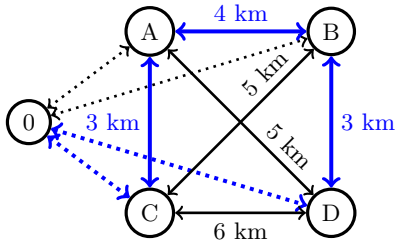

Fig. 1. Example of a scheduling problem with setup times reduced to a TSP problem. The node 0 is the dummy node. Dotted lines have null costs. The blue lines represent the optimal schedule: C, A, B, D.

the scheduling problem reduces to a TSP with time windows [11]. The solver Concorde cannot solve such problems.

Solving the TSP can help solving scheduling problems. For instance, Tran and Beck [17] use the TSP as the slave problem in a Benders decomposition to solve a problem with resource allocation and setup times.

3 Problem Description

We describe the industrial problem specified by our industrial partner. The parameters introduced in this section are summarized in Table 1. A task consists of weaving a textile on a loom. We therefore have a set of tasks $\mathcal{I}$ and a set of looms L. Each task $i \in \mathcal{I}$ is pre-assigned to a loom l_i and has for processing time p_i. A loom $l \in L$ becomes available at time a_l. Prior to this time, the loom is busy terminating a task not in $\mathcal{I}$ that can neither be interrupted nor rescheduled. Each task $i \in \mathcal{I}$ has a style z_i, a due date d_i, and a priority r_i. We wish to minimize the total tardiness weighted by priority, i.e. $\sum_{i \in \mathcal{I}} r_i \cdot \max(0, S_i + p_i - d_i)$ where S_i is the starting time of the task. The scheduling horizon spans from time 0 to time H. In practice, we have a horizon of 240 hours with a time step of 15 minutes resulting in $H = 240 \times \frac{60}{15} = 960$.

Major setups: Each loom $l \in L$ has an initial configuration c_l^{init} and a final configuration c_l^{final}. If $c_l^{\text{init}} \neq c_l^{\text{final}}$ then there is a *major setup* of duration p_l^{major} to change the configuration of the loom l. Only one major setup is possible during the scheduling horizon. A *specialized worker* is selected from a pool W to achieve this major setup. A task $i \in \mathcal{I}$ needs to be executed on a loom l_i when it has configuration $c_i \in \{c_{l_i}^{\text{init}}, c_{l_i}^{\text{final}}\}$. If $c_i = c_{l_i}^{\text{init}}$, the task needs to be executed prior to the major setup and after the major setup if $c_i = c_{l_i}^{\text{final}}$.

Minor setups: A *minor setup* needs to be performed between two consecutive tasks $i, j \in \mathcal{I}$ on a loom. While a major setup entails a configuration change, a minor setup only gets the loom ready for its next job. This setup is decomposed into several steps, each executed by a person of a different profession in P which

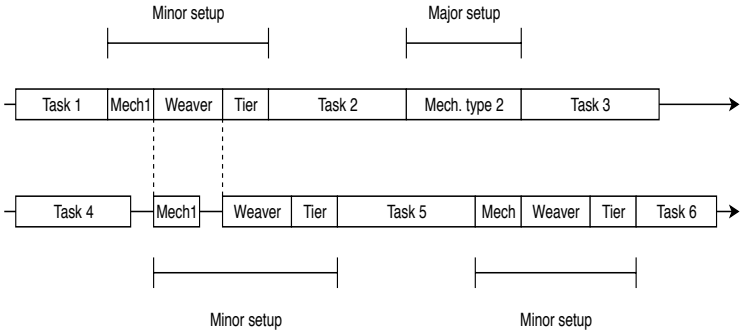

Fig. 2. Example of scheduling on two looms with one of each resource.

is disjoint from the set of workers W. The professions are sorted in order of execution and labeled with integers from 1 to $|P|$, i.e. that the first step of a minor setup is executed by profession 1, the second step is executed by profession 2, and so on. The order of execution is the same for all minor setups. The person of the profession $p \in P$ needs $t_{i,j,p}$ time to execute his/her part of the minor setup between task $i \in \mathcal{I}$ and $j \in \mathcal{I}$. There are q_p people of the profession p. Consequently, no more than q_p minor setup steps can be simultaneously executed by the people from the profession p.

Figure 2 shows a schedule on two looms with a worker of each category. We see conflicts delaying minor setups on the second loom.

$\mathcal{I}$	: Set of tasks.
L	: Set of looms.
P	: Set of professions for minor setups.
W	: Set of specialized workers for major setups.
r_i	: Priority of task i.
d_i	: Due date of task i.
z_i	: Style of task i.
l_i	: Loom assigned to task i.
c_l^{init}	: Initial configuration of loom l.
c_l^{final}	: Final configuration of loom l.
c_i	: Required configuration for task i.
p_i	: Processing time of task i.
p_l^{major}	: Major setup time of loom l.
$t_{i,j,p}$	: Minor setup time between tasks i and j for the profession p.
a_l	: Earliest available time of loom l.
q_p	: Number of workers of the profession p available for the minor setups.

Table 1. Parameters of the problem

4 Models

We present an optimization model that is later submitted to a constraint solver. The variables and domains are summarized in Table 2. Constraints are stated from (1) to (11).

There are three types of events to schedule: tasks, major setups, and minor setups between a task and its successor. Let $S_i \in [a_l, H)$, $S_l^{\text{major}} \in [a_{l_i}, H)$, and $S_{i,p}^{\text{minor}} \in [a_{l_i}, H)$ be their starting time variables for tasks $i \in \mathcal{I}$, loom $l \in L$, and profession $p \in P$. Their domains prevent the events from starting before their respective loom l becomes available.

The variable F_l encodes the first task to execute on loom l. The variable N_i encodes the task that succeeds task i on the loom. If i is the last task, its value is set to a sentinel. There is one sentinel per loom: $\sigma = \{\sigma_1, \ldots, \sigma_{|L|}\}$. The variable N_i is also defined when $i \in \sigma$ is a sentinel. The next task of a sentinel is the first task on the next loom. Consequently, the vector N is a permutation of $\mathcal{I} \cup \sigma$ with a single cycle.

Variable	Domain	Description
S_l^{major}	$[a_l, H)$	Start of the major setup on loom i
S_i	$[a_{l_i}, H)$	Start of task i
$S_{i,p}^{\text{minor}}$	$[a_{l_i}, H)$	Start of the minor setup for profession p between task i and its successor
N_i	$\{j \in \mathcal{I} \mid l_j = l_i\} \cup \{\sigma_{l_i}\}$	Next task after task i
F_l	$\{i \in \mathcal{I} \mid l_i = l\}$	First task on loom l
T	$[0, \infty)$	Total tardiness weighted by priority

Table 2. Variables and their domains

The model contains the constraints (1) to (11). The objective function (1) minimizes the tardiness of the tasks, weighted by priority. Constraints (2) and (3) ensure that a task requiring its loom's initial configuration executes before the major setup or else waits after the major setup to execute. Constraint (4) ensures that the first step of the minor setup following task i starts once task i is completed. Constraint (5) is a precedence constraint over the different steps of a minor setup. Constraint (6) makes the task N_i start immediately after the minor setup is completed. Indeed, once the loom is ready, there is no need to postpone the task. Constraints (7) and (8) ensure that the last task of a sentinel on a loom is the first task on the next loom. The loom that succeeds the last loom is the first loom. That creates a circuit visiting each task exactly once. This idea of a circuit is inspired from Focacci et al. [5] and led to the addition of constraint (9) to the model. This constraint [10] offers a strong filtering on the next variables N.

The model contains global constraints specialized for scheduling problems. We use the notation $[f(x) \mid x \in X]$ to represent the vector $[f(x_1), \ldots, f(x_n)]$ for $X = \{x_1, \ldots, x_n\}$. The constraint (10) limits to q_p the number of simultaneous

minor setups accomplished by a person of the profession p. The constraint (11) limits to $|W|$ the number of simultaneous major setups. The constraint (12) breaks a symmetry by forcing tasks producing the same product style on the same loom to execute in order of due dates.

Minimize T subject to:

$$\text{Minimize} \quad \sum_{\mathrm{i}\in\mathcal{I}} \mathrm{r_i} \cdot \max(0, \mathrm{S_i} + \mathrm{p_i} - \mathrm{d_i}) \tag{1}$$

$$c_i = c_{l_i}^{\text{init}} \implies S_i + p_i \leq S_{A_i}^{\text{major}} \qquad \forall i \in \mathcal{I} \tag{2}$$

$$c_i \neq c_{l_i}^{\text{init}} \implies S_{A_i}^{\text{major}} + p_l^{\text{major}} \leq S_i \qquad \forall i \in \mathcal{I} \tag{3}$$

$$S_i + p_i \leq S_{i,1}^{\text{minor}} \qquad \forall i \in \mathcal{I} \tag{4}$$

$$S_{i,p+1}^{\text{minor}} \geq S_{i,p}^{\text{minor}} + t_{i,N_i,p} \qquad \forall i \in \mathcal{I}, \forall p \in P \setminus \{|P|\} \tag{5}$$

$$S_{N_i} = S_{i,|P|}^{\text{minor}} + t_{i,N_i,|P|} \qquad \forall i \in \mathcal{I} \tag{6}$$

$$N_{\sigma_l} = F_{l+1} \qquad \forall l \in [1, |L| - 1] \tag{7}$$

$$N_{\sigma_{|L|}} = F_1 \tag{8}$$

$$\textsc{Circuit}(N) \tag{9}$$

$$\textsc{Cumulative}([S_{i,p}^{\text{minor}} \mid i \in \mathcal{I}], [t_{i,N_i,p} \mid i \in \mathcal{I}], 1, q_p) \qquad \forall p \in P \tag{10}$$

$$\textsc{Cumulative}([S_l^{\text{major}} \mid l \in L], [p_l^{\text{major}} \mid l \in L], 1, |W|) \tag{11}$$

$$S_a \leq S_b \qquad \forall a, b \in \mathcal{I}, z_a = z_b \wedge l_a = l_b \wedge d_a \leq d_b \tag{12}$$

5 Resolution

We present four methods to solve the model from the previous section. Some are pure heuristics or are *rules of thumb* used in the industry. These methods are either used as a point of comparison or are integrated as a branching heuristics in the constraint solver.

5.1 The Greedy Method Based on Due Dates

The first method, denoted Greedy, consists of executing the tasks on a loom in non-decreasing order of due dates. Ties are arbitrarily broken. If a resource is unavailable to either execute the minor or major setup that precedes a task, it delays the execution of the task until the resource becomes available and has time to complete the setup.

While this method might return a sub-optimal solution, it is nevertheless a rule of thumb used by people in the industry to generate an initial schedule that can be improved later. It is also a point of comparison for other methods.

5.2 The Circuit Method

The next approach denoted Circuit focuses on the Circuit constraint. We want to find the circuit that minimizes the sum of the setup times. While this

objective function is not the weighted tardiness, it is correlated. Indeed, shorter setups lead to shorter idle times and therefore earlier completion times.

We solve the TSP instance induced by the Circuit constraint using the solver Concorde [3]. Concorde is quick to solve TSP instances, especially for instances with fewer than 1000 cities. We sort the looms in non-decreasing amount of time available to execute the minor setups, i.e. for each loom $l \in L$, we compute $H - a_l - \sum_{i|l_i=l} p_i - p_l^{\text{major}}$. Loom by loom in the sorted order, we assign their tasks in the same order found by Concorde. We delay the minor and major setups until the resource is available. The resulting schedule minimizes the amount of time spent by the workers on the setups without considering tardiness.

5.3 The CP Method

The next approach denoted CP consists in coding the model with the MiniZinc [12] language and submitting the model to the constraint solver Chuffed [2].

As a branching heuristics, we generate a *template solution* before the search with either the method Greedy or Circuit. During the search, we choose a next variable N_i that can be assigned to the same value as in the template solution. That was implemented by declaring a vector B of Boolean variables connected to the model with the constraints $B_i = 1 \iff N_i = N_i^t$ where N_i^t is the successor of task i in the template solution. The heuristics branches on the vector B by setting the variables to the value 1. This has for effect to set $N_i = N_i^t$. In case the value 0 is selected for B_i (for instance, after a backtrack), that imposes the constraint $N_i! = N_i^t$ but this does not fix the variable N_i. Once the variables in B are assigned, the solver chooses the starting variables of a task, a minor setup, or a major set up with the smallest value in its domain and assigns this variable to this smallest value. This has for effect to set all next variables that were not already assigned to a value.

5.4 The LNS Method

The large neighborhood search, denoted LNS, starts with an initial solution that could be, for instance, the solution obtained from the methods Greedy and Circuit. It iteratively improves this solution by randomly selecting looms. The CP method is then called to reschedule the tasks on these looms while leaving the tasks on unselected looms untouched.

A note about our implementation. For each iteration, we generate a MiniZinc data file that contains the execution time of the unselected tasks. The model has unary constraints of the form $S_i = v$ to fix the variables to a value. A constraint states that the objective value must improve over the best solution found so far. We solve for satisfaction, i.e. we stop the search once we found an improving solution. An iteration of the LNS is given a timeout (we use 5 minutes) after which, if no solution is found, we pass to the next iteration without changing the current solution but by reseting 10% fewer looms. Whenever the solver returns unsatisfiable, we reset 10% more looms until a solution is found or infeasibility is proven (which implies that the current solution is optimal).

Since the time for compiling the MiniZinc code is significant and could be avoided if the local search was directly implemented in C++ calling the Chuffed solver, we do not count the MiniZinc compilation time in the solving time. The search stops when the computation time, that excludes MiniZinc compilation time, reaches a timeout.

In a context of a local search, we do not use the branching heuristics described in Section 5.3 since this heuristics aims at finding a solution similar to a template solution. In a local search, one rather wants to find a solution that is different from the current one. We simply randomly assign the next variables N.

6 Experiments

6.1 Instances

Our industrial partner shared 4 instances with 571, 592, 756, and 841 tasks. From each instance, we create a dataset of 10 instances by randomly selecting 10 %, 20 %, ..., 100 % of the tasks from the original instance. Once the tasks are selected, this selection is used for all the tests and all the solving methods. This allows to see how our algorithms scale with the number of tasks. The number of looms $|L| = 90$, the number of professions $|P| = 3$ with quantities $[q_1, q_2, q_3] = [5, 3, 2]$, and the number of workers $|W| = 1$ for the major setups remains constant for all instances of all datasets.

6.2 Experimental Setup

The CP model[1] was written in the MiniZinc language [12]. We use the solver Chuffed [2] with the free search parameter [16]. The LNS method is implemented in Python. We ran the experiments on a computer with the following configuration: Ubuntu 19.10, 16 GB ram, Processor Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz, 4008 Mhz, 4 Cores, 8 Logical Processors.

6.3 Methodology

We solved all instances using the Greedy and Circuit method. For the CP method, we tried three different branching heuristics: CP+Greedy assigns the next variables according to the solution of Greedy, CP+Circuit uses the solution from Circuit, and CP+Random randomly assigns the next variable and uses restarts with a Luby sequence with a scale of 250.

For the methods based on CP, we tried the three configurations with LNS and without LNS. At each iteration, 50 % of the looms are rescheduled. A specific iteration is given a timeout of 5 minutes to improve the solution.

All methods are given a timeout of 15 minutes. As the search goes, the CP methods (with or without LNS) keeps improving their best solution. We keep track of the objective value and time of the solutions as we find them during the search.

[1] The MiniZinc files are freely available on Claude-Guy Quimper's web site or directly at http://www2.ift.ulaval.ca/~quimper/publications/CPAIOR2020Submission.zip

6.4 Results

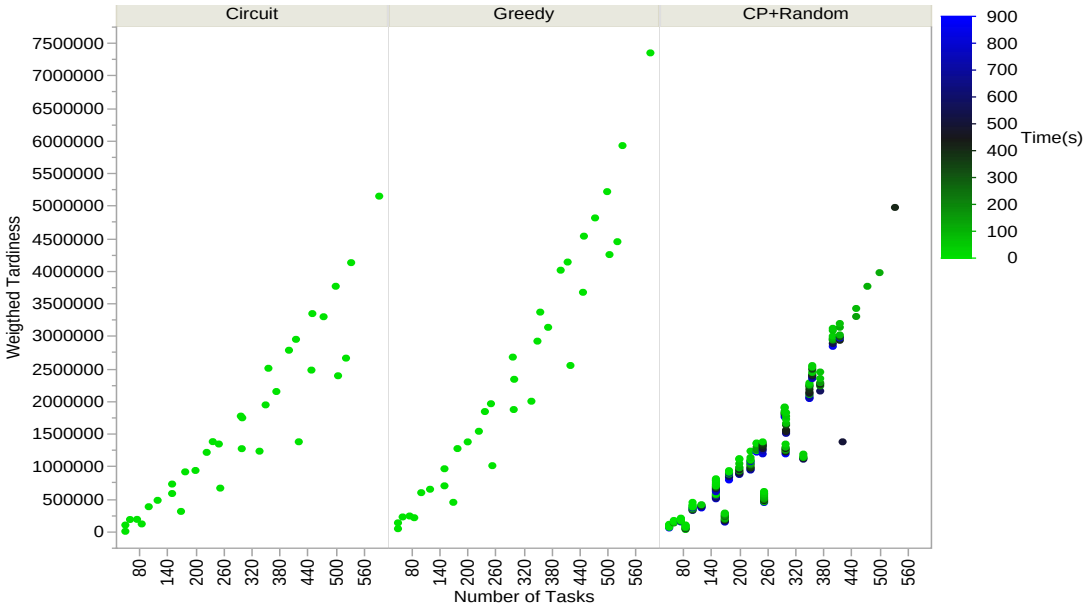

Fig. 3. Comparison between Circuit, Greedy, and one of the methods based on CP

Figure 3 shows the performances of the methods Greedy and Circuit. Each point represents a solution obtained for an instance: on the x-axis is the number of tasks in the instance and on the y-axis is the objective value of the solution returned by the method. The color indicates after how much time (in seconds) the solution was found.

As expected, instances with a larger number of tasks get a larger weighted tardiness. Indeed, since the resources and the due dates remain the constant among all instances, it gets harder to deliver products on time if we increase the number of orders without increasing the resources or delaying the due dates.

Both the methods Greedy and Circuit solve every instance in less than a second. The quality of the solution is not competitive with any method using CP. However, we will see that Greedy and Circuit are nevertheless useful to guide the branching heuristics of the CP solver.

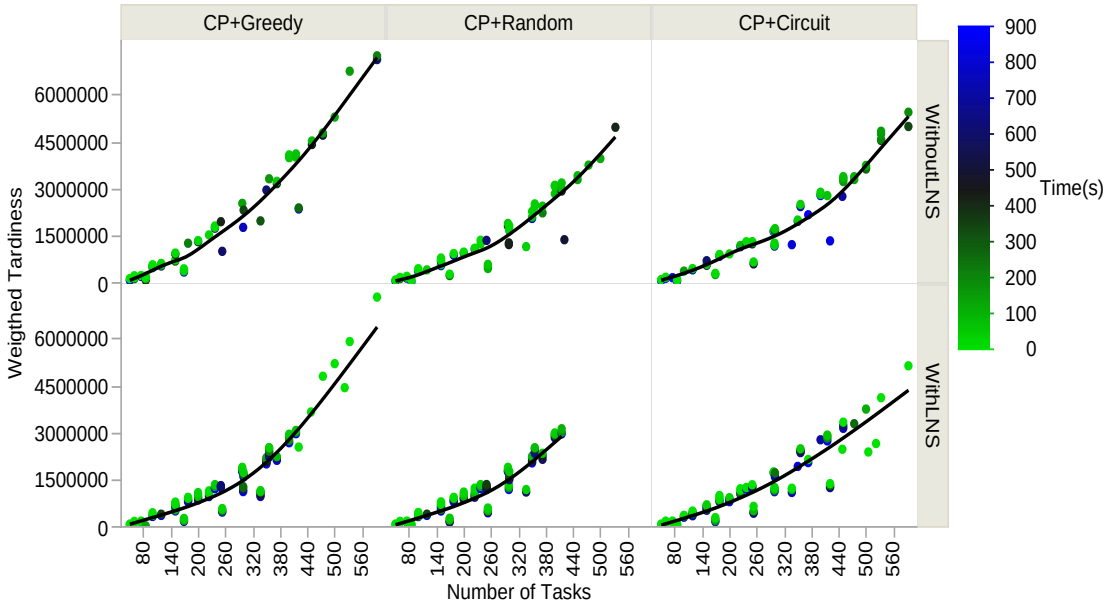

Fig. 4. Comparison of methods based on CP

Figure 4 presents a comparison of the six methods based on CP. The left, middle, and right graphs make the branching heuristics vary. It is either based on the Circuit, Greedy, or the random heuristics. The graphs on the top use the standard CP search while the graphs at the bottom use the LNS. For the LNS, the initial solutions are also based on Circuit, Greedy, and Random. For CP+Circuit+LNS and CP+Greedy+LNS, the next variables N_i are set in the initial solution according to the precedences in the solutions generated by Circuit and Greedy. For CP+Random+LNS, the initial solution is completely random.

The first thing to analyze on Figure 4 is how the methods behave on instances with more than 500 tasks. The methods CP+Random CP+Random+LNS are unable to solve all instances while other methods do. The CP+Greedy method is more stable than CP+Random while CP+Random obtains better solutions. CP+Circuit and CP+Circuit+LNS outperform CP+Random and CP+Random+LNS in both stability and objective value.

Globally, CP+Circuit and CP+Circuit+LNS offer the solutions with the smallest weighted tardiness. This might look surprising at first sight because Circuit aims at minimizing the amount of setups while Greedy directly minimizes tardiness. However, the solution that Circuit generates is optimal according to the amount of setup while the Greedy algorithm only returns an approximation for the weighted tardiness. Guiding the search towards a solution that is optimal, even according to a different but correlated criteria, provides the best solution.

The third observation is that CP+Circuit+LNS generally outputs better solutions than CP+Circuit. The same goes for CP+Greedy+LNS compared to CP+Greedy. While other methods perform better with our homemade LNS, the random heuristic offers poor results.

Finally, we observe that using LNS is beneficial. The poor results of the LNS on the random heuristic is most likely due to a bad initial solution.

7 Conclusion

We presented a model to solve an industrial instance presented by our partner. We showed how a solution from a simplification (the TSP) can guide the search to obtain better solutions. Our model is now able to find solutions to the expected range of tasks. The integration of our program to the operations planning team is in progress.

References

1. Chakrabortty, R.K., Sarker, R.A., Essam, D.L.: Resource constrained project scheduling with uncertain activity durations. Computers & Industrial Engineering **112**, 537 – 550 (2017). https://doi.org/https://doi.org/10.1016/j.cie.2016.12.040, http://www.sciencedirect.com/science/article/pii/S0360835216305186
2. Chu, G., Stuckey, P.J., Schutt, A., Ehlers, T., Gange, G., Francis, K.: Chuffed, a lazy clause generation solver (2018)
3. Cook, W.: In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation. Princeton University Press, Princeton University Press (2011)
4. Danna, E., Perron, L.: Structured vs. unstructured large neighborhood search: A case study on job-shop scheduling problems with earliness and tardiness costs. In: Rossi, F. (ed.) Principles and Practice of Constraint Programming – CP 2003. pp. 817–821. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
5. Focacci, F., Laborie, P., Nuijten, W.: Solving scheduling problems with setup times and alternative resources. In: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems. pp. 92–101 (2000)
6. Glass, H., Cooper, L.: Sequential search: A method for solving constrained optimization problems. J. ACM **12**(1), 71–82 (Jan 1965). https://doi.org/10.1145/321250.321256, https://doi.org/10.1145/321250.321256
7. Hermann, M., Pentek, T., Otto, B.: Design principles for industrie 4.0 scenarios. In: 49th Hawaii International Conference on System Sciences (HICSS). pp. 3928-–3937 (2016)
8. Kagermann, H., Helbig, J., Hellinger, A., Wahlster, W.: Recommendations for implementing the strategic initiative industrie 4.0: Securing the future of german manufacturing industry; final report of the industrie 4.0 working group. Tech. rep., Forschungsunion (2013)
9. Kajgård, E.: Route optimisation for winter road maintenance using constraint modelling (2015)
10. Lauriere, J.L.: A language and a program for stating and solving combinatorial problems. Artificial intelligence **10**(1), 29–127 (1978)

11. López-Ibáñez, M., Blum, C., Ohlmann, J.W., Thomas, B.W.: The travelling salesman problem with time windows: Adapting algorithms from travel-time to makespan optimization. Applied Soft Computing **13**(9), 3806–3815 (2013)
12. Nethercote, N., Stuckey, P.J., Becket, R., Brand, S., Duck, G.J., Tack, G.: Minizinc: Towards a standard cp modelling language. In: Principles and Practice of Constraint Programming – CP 2007. pp. 529–543 (2007)
13. Psaraftis, H.N.: k-interchange procedures for local search in a precedence-constrained routing problem. European Journal of Operational Research **13**(4), 391–402 (1983)
14. Savelsbergh, M.W.P.: Local search in routing problems with time windows. Annals of Operations Research **4**(1), 285–305 (Dec 1985)
15. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Why cumulative decomposition is not as bad as it sounds. In: Gent, I.P. (ed.) Principles and Practice of Constraint Programming - CP 2009. pp. 746–761. Springer Berlin Heidelberg, Berlin, Heidelberg (2009)
16. Shishmarev, M., Mears, C., Tack, G., Garcia de la Banda, M.: Learning from learning solvers. In: Rueher, M. (ed.) Principles and Practice of Constraint Programming. pp. 455–472. Springer International Publishing, Cham (2016)
17. Tran, T.T., Beck, J.C.: Logic-based benders decomposition for alternative resource scheduling with sequence dependent setups. In: Proceedings of the 20th European Conference on Artificial Intelligence ECAI'12. pp. 774–779 (2012)