Two Clause Learning Approaches for Disjunctive Scheduling

Mohamed Siala\textsuperscript{1,2,3}, Christian Artigues\textsuperscript{2,4}, and Emmanuel Hebrard\textsuperscript{2,4}

\textsuperscript{1} Insight Centre for Data Analytics, University College Cork, Ireland
\textsuperscript{2} CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
\textsuperscript{3} Univ de Toulouse, INSA, LAAS, F-31400 Toulouse, France
\textsuperscript{4} Univ de Toulouse, LAAS, F-31400 Toulouse, France
mohamed.siala@insight-centre.org
\{siala, artigues, hebrard\}@laas.fr

Abstract. We revisit the standard hybrid CP/SAT approach for solving disjunctive scheduling problems. Previous methods entail the creation of redundant clauses when lazily generating atoms standing for bounds modifications. We first describe an alternative method for handling lazily generated atoms without computational overhead. Next, we propose a novel conflict analysis scheme tailored for disjunctive scheduling. Our experiments on well known Job Shop Scheduling instances show compelling evidence of the efficiency of the learning mechanism that we propose. In particular this approach is very efficient for proving unfeasibility.

1 Introduction

Disjunctive scheduling refers to a large family of scheduling problems having in common the Unary Resource Constraint. This constraint ensures that a set of tasks run in sequence, that is, without any time overlap. The traditional constraint programming (CP) approaches for this problem rely on tailored propagation algorithms (such as Edge-Finding \cite{6,17,24}) and search strategies (such as Texture \cite{20}). The technique of Large Neighborhood Search \cite{22} (LNS) was also extensively used in this context \cite{7,25}.

A different type of approaches emerged recently, based on the so-called Conflict-Driven Clause Learning (CDCL) algorithm for SAT \cite{16}. This procedure uses resolution to learn a new clause for each conflict during search. Recent constraint programming approaches similarly trade off strong propagation-based inference for a way to learn during search. For instance, a hybrid CP/SAT method, Lazy Clause Generation (LCG) \cite{8,9,18} was shown to be extremely efficient on the more general Resource Constrained Project Scheduling Problem \cite{21} (RCPSP). Even simple heuristic weight-based learning was shown to be very efficient on disjunctive scheduling \cite{10-13}. The so called light model combines minimalist propagation with a slight variant of the weighted degree \cite{5} heuristic.

In this paper, we propose a hybrid CP/SAT method based on this light model. Similarly to LCG, our approach mimics CDCL However, it differs from
LCG in two main respects: First, as the time horizon can be large, literals representing changes in the bounds of the tasks domains should be generated “lazily” during conflict analysis as it was proposed in [8]. However, handling domain consistency through clauses entails redundancies and hence suboptimal propagation. We use a dedicated propagation algorithm, running in constant amortized time, to perform this task. The second contribution is a novel conflict analysis scheme tailored for disjunctive scheduling. This technique could be applied to any problem where search can be restricted to a predefined set of Boolean variables. This is the case of disjunctive scheduling since once every pair of tasks sharing the same resource is sequenced, we are guaranteed to find a complete solution in polynomial time. Most methods therefore only branch on the relative order of tasks sharing a common resource. We propose to use this property to design a different conflict analysis method where we continue resolution until having a nogood with only Boolean variables standing for task ordering. As a result, we do not need to generate domain atoms.

We compare the two methods experimentally and show that the benefit of not having to generate new atoms during search outweigh in many cases the more expressive language of literals available in traditional hybrid solvers. The novel approach is very efficient, especially for proving unfeasibility. We were able to improve the lower bounds on several well known Job Shop Scheduling Problem (JSSP) benchmarks. However, a method implemented within IBM CP-Optimizer studio recently found, in general, better bounds [25].

The rest of the paper is organized as follows: We give a short background on hybrid CP/SAT solving in Section 2. Next, we briefly describe the baseline CP model and strategies that we use in Section 3. We introduce in Section 4 our new lazy generation approach. In Section 5, we present our novel conflict analysis scheme. Last, we give and discuss the experimental results in Section 6.

2 Hybrid CP/SAT Solving

In this section, we briefly describe the basic mechanisms and notations of hybrid CP/SAT used in modern Lazy Clause Generation solvers [8,18,9].

Let \( \mathcal{X} = \{x_1, \ldots, x_n\} \) be a sequence of variables. A domain \( \mathcal{D} \) maps every variable \( x \in \mathcal{X} \) to a finite set \( \mathcal{D}(x) \subseteq \mathbb{Z} \). \( \mathcal{D}(x) \) is a range if \( \max(\mathcal{D}(x)) - \min(\mathcal{D}(x)) + 1 = |\mathcal{D}(x)| \). A constraint \( C \) is defined by a relation over the domains of a sequence of variables. In this context, each constraint is associated to a propagator to reduce the domains with respect to its relation. Propagators are allowed to use only domain operations of a given type (often \( [x = v], [x \neq v], [x \leq v], \) and \( [x \geq v] \)). Every domain operation is associated to a literal \( p \) that can have an atom (i.e. Boolean variable) generated “eagerly” before starting the search, or “lazily” when learning a conflict involving this change. Let \( p \) be a literal corresponding to a domain operation. \( \text{level}(p) \) is the number of nodes in the current branch of the search tree when \( p \) becomes true (i.e., when this domain operation takes place). Moreover, \( \text{rank}(p) \) is the number of such domain operations that took place at the same node of the search tree before \( p \). Last, a literal \( p \) is as-
sociated to a logical implication of the form \( \varphi \Rightarrow p \) where \( \varphi \) is a conjunction of literals sufficient for a given propagator to deduce \( p \). We call \( \varphi \) an explanation of \( p \). We assume that the function \( \text{explain}(p) \) returns the explanation \( \varphi \) for \( p \). The explanations can be computed at the moment of propagation (i.e., clausal and forward explanations in [8]), or on demand during conflict analysis (backward). The notion of explanation is extended to a failure \( \perp \) in the natural way.

Whenever a failure occurs during search, a conflict analysis procedure is used to compute a nogood, that is, an explanation of the failure which is not yet logically implied by the model. Algorithm 1 depicts the conflict analysis procedure based on the 1-UIP scheme [27]. The nogood \( \Psi \) is initialized in Line 1 to the explanation for the failure. It is returned once it contains a single literal that has been propagated at the current level (Line 2). When this is not the case, the last propagated literal is selected and its explanation is resolved with the current nogood in Line 3. The final nogood \( \Psi \) can be seen as \( \bigwedge_{i \in [1,n]} p_i \Rightarrow p \). The solver then backtracks to the level \( l = \max(\text{level}(p_i)) \).

Algorithm 1: 1-UIP-with-Propagators

1. \( \Psi \leftarrow \text{explain}(\perp) \);
2. while \( |\{ q \in \Psi \mid \text{level}(q) = \text{current level} \}| > 1 \) do
   \( p \leftarrow \text{arg max}_{q \in \Psi} (\text{rank}(q) \wedge \text{level}(q) > 0) \) \( \setminus \{ p \} \);
3. \( \Psi \leftarrow \Psi \cup \{ q \mid q \in \text{explain}(p) \wedge \text{level}(q) > 0 \} \);
4. return \( \Psi \);

Consider now the lazy generation of atoms (i.e. after computing \( \Psi \)). When \( \llbracket x \leq u \rrbracket \) has to be generated, the clauses \( \neg \llbracket x \leq a \rrbracket \lor \llbracket x \leq u \rrbracket ; \neg \llbracket x \leq u \rrbracket \lor \llbracket x \leq b \rrbracket \) where \( a \) and \( b \) are the nearest generated bounds to \( u \) with \( a < u < b \) must be added to maintain the consistency of \( \llbracket x \leq u \rrbracket \) with respect to previously generated atoms. In this case, the clause \( \neg \llbracket x \leq b \rrbracket \lor \llbracket x \leq u \rrbracket \) becomes redundant.

3 A Simple CP Model for Job Shop Scheduling

A JSSP consists in sequencing the tasks from \( n \) jobs on \( m \) machines. The jobs \( \mathcal{J} = \{ J_i \mid 1 \leq i \leq n \} \) define sequences of tasks to be scheduled in a given order. Moreover, each of the \( m \) tasks of a job must be processed by a different machine. We denote \( T_{ik} \) the task of job \( J_i \) requiring machine \( M_k \). Each task \( T_{ik} \) is associated to a processing duration \( p_{ik} \). Let \( t_{ik} \) be the variable representing the starting time of task \( T_{ik} \). The processing interval of a task \( T_{ik} \) is \( [t_{ik}, t_{ik} + p_{ik}] \). For all \( k \in [1, m] \), the Unary Resource Constraint for machine \( M_k \) ensures that for any \( i \neq j \in [1, n] \), there is no overlap between the processing interval of \( T_{ik} \) and \( T_{jk} \). We use a simple decomposition with \( O(n^2) \) Boolean variables \( \delta_{kij} \) (\( i < j \in [1, n] \)) per machine \( M_k \) using the following DISJUNCTIVE constraints:

\[
\delta_{kij} = \begin{cases}
0 & \iff t_{ik} + p_{ik} \leq t_{jk} \\
1 & \iff t_{jk} + p_{jk} \leq t_{ik}
\end{cases}
\]
A JSSP requires also a total order on the tasks of each job. We enforce this order with simple precedence constraints. Last, the objective is to minimize the total scheduling duration, i.e., the makespan. To this purpose, we have an objective variable $C_{\text{max}}$ subject to precedence constraints with the last task of each job.

We use the same search strategy as that proposed in [10, 13]. First, we compute a greedy upper bound on the makespan. Then we use a dichotomic search to reduce the gap between lower and upper bounds. Each step of this dichotomic phase corresponds to the decision problem with the makespan variable $C_{\text{max}}$ fixed to the mid-point value. Each step is given a fixed time cutoff, and exceeding it is interpreted as the instance being unsatisfiable. Therefore, the gap might not be closed at the end of this phase, so a classic branch and bound procedure is used until either closing the gap or reaching a global cutoff.

We branch on the Boolean variables of the Disjunctive constraints following [13] using the solution guided approach [3] for value ordering. We use the weighed degree heuristic $\text{taskDom/tw}$ in a similar way to [10, 13] in addition to VSIDS [16]. The former branches first on variables occurring most in constraints triggering failures. The latter favors variables involved in conflict analysis.

4 Lazy Generation of Atoms

In this section we describe an efficient propagator to maintain the consistency of lazily generated atoms. Recall that domain clauses are likely to be redundant in this case (Section 2). Instead of generating clauses to encode the different relationships between the newly generated atoms, we propose to encode such relations through a dedicated propagator in order to avoid this redundancy and hence the associated overhead. We introduce the DomainFaithfulness constraint. Its role is twofold: firstly, it simulates Unit-Propagation (UP) as if the atoms were generated eagerly; secondly it performs a complete channeling between the range variable and all its generated domain atoms.

We consider only the lazy generation of atoms of the type $[x \leq u]$ since all propagators in our models perform only bound tightening operations. Nevertheless, the generalization with $[x = v]$ is quite simple and straightforward. Let $x$ be a range variable, $[v_1, \ldots, v_n]$ be a sequence of integer values, and $[b_1 \ldots b_n]$ be a sequence of lazily generated Boolean variables s.t. $b_i$ is the atom $[x \leq v_i]$. We define DomainFaithfulness$(x, [b_1 \ldots b_n], [v_1, \ldots, v_n])$ as follows: $\forall i, b_i \leftrightarrow x \leq v_i$.

We describe now the data structures and the procedures that allow to maintain Arc Consistency [4] (AC) with a constant amortized time complexity down a branch of the search tree. We assume without loss of generality that $n \geq 1$ and that $v_i < v_{i+1}$. The filtering is then organized in two parts:

1. Simulating UP as if the atoms were eagerly generated with all domain clauses. That is, whenever an atom $b_i$ becomes assigned to 1 (respectively 0), the atom $b_{i+1}$ (respectively $b_{i-1}$) is assigned to 1 (respectively 0).
2. Performing a complete channeling between $x$ and $b_1, \ldots, b_n$: We sketch the process related to the upper bound of $x$. A similar process in applied with the lower bound. There are two cases to distinguish:
(a) Changing the upper bound of $x$ w.r.t. newly assigned atoms: When an atom $b_i \leftrightarrow [x \leq v_i]$ becomes assigned to 1, we check if $v_i$ can be the new upper bound of $x$. Note that a failure can be triggered if $v_i < \min(x)$.

(b) In the case of an upper bound propagation event, one has to assign some atoms to be consistent with the new upper bound $u$ of $x$. Every atom $b_i$ such that $v_i \geq u$ has to be assigned to 1. Let $i_{ub} = \arg \max_k (v_k \mid D(b_k) = \{1\})$. Observe first that the part simulating UP (1) guarantees that all atoms $b_i$ where $i > i_{ub}$ are already assigned to 1. Consider now the set $\varphi = \{b_{i_{ub}}, b_{i_{ub}-1}, b_{i_{ub}-2}, \ldots, b_{last_{ub}}\}$ where $last_{ub} = \arg \min_k (v_k \mid v_k \geq u)$). It is now sufficient to assign every atom in $\varphi$ to 1 in order to complete the filtering.

Finally, since down a branch of the search tree the upper bound can only decrease, we can compute the current value of $i_{ub}$, that is, $\arg \max_k (v_k \mid D(b_k) = \{1\})$ by exploring the sequence of successive upper bounds from where we previously stopped. Therefore, each atom is visited at most once down a branch. This filtering can thus be performed in constant amortized time, that is, in $O(n)$ time down a branch, however, we must store $i_{ub}$ and $i_{lb}$ as “reversible” integers.

5 Learning Restricted to Task Ordering

Here we introduce a learning scheme as an alternative to lazy generation for disjunctive scheduling problems. Recall that we branch only on Boolean variables coming from the Disjunctive constraints. It follows that every bound literal $p$ s.t. $\text{level}(p) > 0$ has a non-empty explanation. We can exploit precisely this property in order to avoid the generation of any bound atom.

The first step in this framework is to perform conflict analysis as usual by computing the 1-UIP nogood $\Psi$. Next, we make sure that the latest literal in $\Psi$ is not a bound literal. Otherwise, we keep explaining the latest literal in $\Psi$ until having such UIP. We know that this procedure terminates because the worst case would reach the last decision which corresponds to a UIP that is not a bound literal. Let $\Psi^*$ be the resulting nogood.

Consider now $\mathcal{I}$ to be the set of bound literals in $\Psi^*$. Instead of generating the corresponding atoms, we start a second phase of conflict analysis via the procedure Substitute($\mathcal{I}, \Psi$) with ($\mathcal{I}, \Psi^*$) as input. Algorithm 2 details this procedure. In each step of the main loop, a bound literal $p$ from $\mathcal{I}$ is chosen (Line 1) and replaced in $\Psi$ with its explanation (Line 2). $\mathcal{I}$ is updated later at each step in Line 3. The final nogood $\Psi$ involves only Boolean variables. Note that the backjump level remains the same as in $\Psi^*$ since for every $p$ there must exists a literal in $\text{explain}(p)$ with the same level of $p$.

The advantage of this approach is that since no atom need to be generated during search, we do not need to maintain the consistency between tasks’ domains and bounds atoms. In particular, it greatly reduces the impact that the size of the scheduling horizon has on the space and time complexities. Note, however, that there may be a slight overhead in the conflict analysis step, compared to the lazy generation mode, since there are more literals to explain. Moreover,
Algorithm 2: Substitute(\(\mathcal{I}, \Psi\))

\[
\text{visited} \leftarrow \emptyset ; \\
\text{while } |\mathcal{I}| > 0 \text{ do} \\
\quad p \leftarrow \text{choose } p \in \mathcal{I} ; \\
\quad \text{visited} \leftarrow \text{visited} \cup \{p\} ; \\
\quad \Psi \leftarrow \Psi \cup \{q \mid q \in \text{explain}(p) \land \text{level}(q) > 0 \} \setminus \text{visited} ; \\
\quad \mathcal{I} \leftarrow \{q \mid q \in \Psi \land q \text{ is a bound literal}\}; \\
\text{return } \Psi ;
\]

since the language of literal is not as rich in this method, shorter proofs may be possible with the standard approach.

6 Experimental Results

We first compare the two approaches described in this paper, that is, the implementation of lazy generation using DomainFaithfulness (lazy) as well as the new learning scheme (disj) against the CP model described in [11] on two well known benchmarks for the JSSP: Lawrence [14] and Taillard [23]. Then, we compare the lower bounds found by our approaches with the best known lower bounds. All models are implemented within Mistral-2.0 and are tested on Intel Xeon E5-2640 processors. The source code, the detailed results, and the experimental protocol are available at http://siala.github.io/jssp/details.pdf.

We denote CP(task) the CP model using the taskDom/tw heuristic. For the hybrid models, we use the notation H(\(\theta, \sigma\)) where \(\theta \in \{\text{vsids}, \text{task}\}\) is the variable ordering and \(\sigma \in \{\text{lazy, disj}\}\) indicates the type of learning used.

6.1 Empirical Evaluation on the Job Shop Scheduling Problem

We ran every method 10 times using randomized geometric restarts [26]. Each dichotomy step is limited to 300s and \(4 \times 10^6\) propagation calls. The total runtime for each configuration is limited to 3600s. The results are summarized in Table 1.

Table 1 is divided in two parts. The first part concerns instances that are mostly proven optimal by our experiments. We report for these instances the average percentage of calls where optimality was proven \(\%O\), the average CPU time \(T\), and the number of nodes explored by second (nds/s). The second part concerns the rest of the instances (i.e. hard instances). We report for each data set the speed of exploration (nds/s) along with the average percentage relative deviation (PRD) of each model. The PRD of a model \(m\) for an instance \(C\) is computed with the formula: \(100 \times \frac{C_m - C_{\text{best}}}{C_{\text{best}}}\), where \(C_m\) is the minimum makespan found by model \(m\) for this instance ; and \(C_{\text{best}}\) is the minimum makespan known in the literature [1, 2, 25]. The bigger the PRD, the less efficient a model is.

Consider first small and medium sized instances, i.e., la-01-40, tai-01-10, and tai-11-20. Table 1 shows clearly that the learning scheme that we introduced (i.e. H(vsids/task, disj)) dominates the other models on these instances. For example H(vsids, disj) proves 91.5\% of Lawrence instances to optimality whereas
CP(\textit{task}) and H(\textit{vsids, lazy}) achieve a ratio of 87\% and 88\%, respectively. Moreover, VSIDS generally performs better than weighted degree on these instances, although this factor does not seem as important as the type of learning.

Consider now the rest of instances (tai-21-30 to tai-61-70). The impact of clause learning is more visible on these instances. For example, with tai-31-40, CP(\textit{task}) has a PRD of 5.4149 while the worst hybrid model has a PRD of 4.0536. The choice of the branching strategy seems more important on the largest instances. For instance, on tai-51-60, the PRD of H(\textit{task, disj}) is 1.1156 while H(\textit{vsids, disj}) has a PRD of 3.2449.

Table 1 shows also that the CP model is often faster than any hybrid model (w.r.t. nds/s). This is expected because of the overhead of propagating the learnt clauses in the hybrid models. Lazy generation (\textit{lazy}) slows the search down considerably compared to the mechanism we introduced (\textit{disj}).

Regarding the clauses size, we observed that the new method constantly learns shorter clauses. For example when both methods use the heuristic VSIDS, the average size on tai-11..20 is 31 for the disjunctive learning whereas the standard approach has an average size of 43. This may seem surprising, but several bounds literals may share part of their explanations.

Overall, our hybrid models outperform the CP model in most cases. The novel conflict analysis shows promising results especially for small and medium sized instances. It should be noted that we did not find new upper bounds for hard instances. However, our experiments show that our best model deviates only of a few percents from the best known bounds in the literature. Note that the state-of-the-art CP method [25] uses a timeout of 30000s (similarly to [19]) with a maximum of 600s per iteration in the bottom-up phase for improving the lower bound. Moreover, they use the best known bounds as an extra information along with a double threading phase. In our experiments, we use a single thread and a fixed time limit of 3600s per instance.

### Table 1. Summary of the results

<table>
<thead>
<tr>
<th>Instances</th>
<th>CP(\textit{task})</th>
<th>H(\textit{vsids, disj})</th>
<th>H(\textit{vsids, lazy})</th>
<th>H(\textit{task, disj})</th>
<th>H(\textit{task, lazy})</th>
</tr>
</thead>
<tbody>
<tr>
<td>la-01-40</td>
<td>87.522</td>
<td>87.560</td>
<td>91.543</td>
<td>88.632</td>
<td>88.732</td>
</tr>
<tr>
<td>la-01-10</td>
<td>89.768</td>
<td>58.775</td>
<td>90.317</td>
<td>88.106</td>
<td>88.634</td>
</tr>
</tbody>
</table>

**Mostly proven optimal**

<table>
<thead>
<tr>
<th>Instances</th>
<th>%O T nds/s</th>
<th>%O T nds/s</th>
<th>%O T nds/s</th>
<th>%O T nds/s</th>
<th>%O T nds/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>la-01-40</td>
<td>87.522</td>
<td>87.560</td>
<td>91.543</td>
<td>88.632</td>
<td>88.732</td>
</tr>
<tr>
<td>la-01-10</td>
<td>89.768</td>
<td>58.775</td>
<td>90.317</td>
<td>88.106</td>
<td>88.634</td>
</tr>
</tbody>
</table>

**Hard instances**

<table>
<thead>
<tr>
<th>Instances</th>
<th>PRD nds/s</th>
<th>PRD nds/s</th>
<th>PRD nds/s</th>
<th>PRD nds/s</th>
<th>PRD nds/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>tai-11-20</td>
<td>1.8432</td>
<td>4908</td>
<td>1.446</td>
<td>3584</td>
<td>1.3725</td>
</tr>
<tr>
<td>tai-21-30</td>
<td>1.6131</td>
<td>3244</td>
<td>0.9150</td>
<td>2361</td>
<td>1.0841</td>
</tr>
<tr>
<td>tai-31-40</td>
<td>5.4149</td>
<td>3501</td>
<td>4.0210</td>
<td>2623</td>
<td>3.7350</td>
</tr>
<tr>
<td>tai-41-50</td>
<td>7.0439</td>
<td>2234</td>
<td>4.8362</td>
<td>1615</td>
<td>4.6800</td>
</tr>
<tr>
<td>tai-51-60</td>
<td>3.0346</td>
<td>1688</td>
<td>3.2449</td>
<td>2726</td>
<td>3.7809</td>
</tr>
<tr>
<td>tai-61-70</td>
<td>6.8598</td>
<td>1432</td>
<td>6.5890</td>
<td>2414</td>
<td>5.4264</td>
</tr>
</tbody>
</table>
6.2 Lower Bound Computation

We ran another set of experiments with a slightly modified dichotomy phase. Here we assume that the outcome of step $i$ is satisfiable instead of unsatisfiable when the cutoff is reached. Each dichotomy step is limited to 1400s and the overall time limit is 3600s. We used the set of all 22 open Taillard instances before the results of [25] and we found 7 new lower bounds (cf. Table 2). However, most of these bounds were already improved by Vilím et al. [25]. The main difference is that their algorithm uses the best known bounds as an additional information before starting search. Our models, however, are completely self-contained in the sense where search is started from scratch (see Section 3).

<table>
<thead>
<tr>
<th>Instance</th>
<th>New Bound</th>
<th>Old Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>tai13</td>
<td>1305</td>
<td>1282</td>
</tr>
<tr>
<td>tai21</td>
<td>1613</td>
<td>1573</td>
</tr>
<tr>
<td>tai23</td>
<td>1514</td>
<td>1474</td>
</tr>
<tr>
<td>tai25</td>
<td>1544</td>
<td>1518</td>
</tr>
<tr>
<td>tai26</td>
<td>1561</td>
<td>1558</td>
</tr>
<tr>
<td>tai29</td>
<td>1576</td>
<td>1525</td>
</tr>
<tr>
<td>tai30</td>
<td>1515</td>
<td>1485</td>
</tr>
</tbody>
</table>

We computed the average PRD w.r.t. the best known lower bound including those reported in [25]. Excluding the set tai40-50, the best combination, $H(vsids, disj)$ using Luby restarts [15], finds lower bounds that are 2.78% lower, in average, than the best known bound in the literature.

Finally, in order to find new lower bounds, we launched again the model $H(vsids, disj)$ with 2500s as a timeout for each dichotomy step and 7200s as a global cutoff. We found a new lower bound of 1583 for tai-29 (previously 1573) and one of 1528 for tai-30 (previously 1519).

7 Conclusion

We introduced two hybrid CP/SAT approaches for solving disjunctive scheduling problems. The first one avoids a known redundancy issue regarding lazy generation without computational overhead, whereas the second constitutes a ‘hand-crafted’ conflict analysis scheme for disjunctive scheduling.

These methods fit naturally the CDCL framework and, in fact, similar techniques appear to have been implemented in existing hybrid solvers, such as Chuffed, although they have never been published nor documented. Experimental evaluation shows the promise of these techniques. In particular, the novel learning scheme is extremely efficient for proving unfeasibility. Indeed, we improved the best known lower bounds of two Taillard instances.

Acknowledgments

This work is funded by CNRS and Midi-Pyrénées region (grant number 11050449). The authors would like to thank the Insight Centre for Data Analytics for kindly granting us access to its computational resources.
References