ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230278

Enhancing Hybrid CP-SAT Search for Disjunctive Scheduling

Arthur Bit-Monnot

LAAS-CNRS, Université de Toulouse, CNRS, INSA, Toulouse, France

Abstract. Disjunctive scheduling problems such as the job shop and open shop are at the heart of many real world scheduling instances. In this paper, we frame such problems as disjunctive temporal networks associated with a makespan minimization objective. For those, we propose a hybrid approach between SAT/SMT and CP solvers. In particular, we keep from SMT solvers the aggregated constraint propagation in decision procedures as well as the explanations and clause learning mechanisms upon conflict. However, like all CP solvers, we maintain an explicit domain representation of integer variables, tightly integrated with clause learning. Automated search exploits explanations to derive activity-based heuristics combined with the more classical value-based heuristics of CP solvers. The resulting solver is shown to be competitive with state-of-the-art exhaustive search solvers on the classical benchmarks of job shop and open shop problems.

1 Introduction

Disjunctive scheduling problems such as the Job Shop are some of most emblematic scheduling problems. In essence, they require finding a schedule of minimal duration (makespan) of a set of operations where some operations are mutually exclusive: both require access to a unique resource that prevents any temporal overlapping in their executions.

Resolution of disjunctive scheduling problems have historically been the domain of specialized local-search methods such as Tabu Search for Job Shop scheduling [26] or particle swarm optimization for Open Shop scheduling [33]. Constraint programming approaches have however remained competitive by leveraging with strong inference methods such as edge-finding [8] and search techniques such as Large-Neighborhood Search (LNS, [34]). Two of the most impactful constraint programming approaches remain however quite general. Grimes and Hébrard [15] propose to encode disjunctive scheduling problems as disjunctive temporal networks and achieve state-of-the-art performance on a number of job shop variants when using *weighted degree* heuristic for a straighforward CP model. On the other hand the introduction of Failure-Directed-Search [39] in CPOptimizer allowed to close many instances when combined with strong propagation methods and LNS [21].

In this paper, we propose a new solver for disjunctive scheduling problems, that exploits their representations as disjunctive temporal networks in a constraint programming framework [15]. The proposed solver is designed from the ground up with explanations and conflict-driven clause-learning (CDCL, [24]) in mind with the aim of leveraging some key capabilities from SAT solvers. In particular, we show that a careful integration of literals in the solver avoids some pitfalls of previous hybrid CP/SAT solvers for scheduling [31, 35]. The solver is completed by a dedicated global propagator for reified difference constraints and generic search strategies that leverage the solver's explanation mechanisms to guide the search. Despite its simple and generic design, we show the resulting solver to achieve state-of-the-art performance over the standard benchmarks for the Job Shop (JSP) and Open Shop (OSP) problems.

2 Simple Model for Disjunctive Scheduling

Disjunctive Scheduling Problems Disjunctive scheduling problems are characterized by a set of activities A. For each activity $a \in A$, we are given its duration d_a and must determine its start time t_a. The schedule is subject to precedence constraints (e.g. activity a_1 must precede activity a_2) and no-overlap constraints (e.g. activities a_1 and a_2 must not be executing at the same time).

In JSP and OSP all activities are associated with a machine for which they require exclusive usage (i.e. they must not overlap with tasks requiring the same machine). The activities are further grouped in jobs with two tasks of the same job not being allowed to overlap. In addition, JSP imposes a predefined total order over the activities of a job. The objective is to find a valid schedule that minimizes the makespan, i.e., the end time of the latest finishing task.

Constraint Programming Model To model the problem in CP, we use a discrete domain variable t_a with a finite domain $\mathcal{D}(t_a) \subset \mathbb{N}$ for each activity $a \in A$. Precedences are encoded as difference constraints:

$$t_{a_1} + d_{a_1} \leq t_{a_2}$$

On the other hand, no-overlap constraints are expressed as disjunctions of difference constraints:

$$t_{a_1} + d_{a_1} \leq t_{a_2} \vee t_{a_2} + d_{a_2} \leq t_{a_1}$$

Following the approach of Grimes and Hébrard [15], such no-overlap expression are further simplified into a pair of reified difference constraints with the introduction of a new binary variable b_{a_1,a_2} that is equal to 1 iff a_1 comes first and 0 iff a_2 comes first:

$$b_{a_1,a_2} = 1 \Leftrightarrow t_{a_1} + d_{a_1} \leq t_{a_2}$$
$$b_{a_1,a_2} = 0 \Leftrightarrow t_{a_2} + d_{a_2} \leq t_{a_1}$$

The makespan can be materialized as an additional integer variable t_e such that

$$\forall a \in A, t_a + d_a \leq t_e.$$

Let us refer to $\mathcal{X}$ as the set of all variables encoding the start-times (integer), the order (boolean) and the makespan (integer). The objective is to find an assignment to all variables in $\mathcal{X}$ that satisfies all constraints and minimizes the value of t_e. Note the particular homogeneity of this encoding where all constraints are (reified) difference constraints.

3 Anatomy of a solver

Let us first provide a high level overview of the organization of the solver and of its search mechanism. As in most constraint programming solver, the central component of a solver is a data-structure that holds the current domain of each variable alongside a complete history of all changes that were applied to these domains, the *trail*.

A set of *reasoners* is in charge of maintaining the consistency of the *domains* with respect to a set of constraints. As in the theories of SMT solvers, a reasoner is typically in charge of a set of homogenous constraints. In particular the two reasoners we consider for disjunctive scheduling problems are:

- a reasoner for (reified) difference constraints of the form $\ell \Leftrightarrow x + d \leq x'$.
- a SAT-like reasoner, in charge of disjunctive constraints of the form $l_1 \vee \cdots \vee l_n$, where each l_i is a boolean literal.

The *search component* is in charge of orchestrating a backtracking search until an optimal solution is found, or until the entire search space has been explored. At a high level, search follows the Conflict-Driven Clause-Learning (CDCL) scheme [24]: every time the search runs into a conflict, a clause is produced that prevents the search from running into the same conflict.

3.1 Literals and Signed Variables

A literal represents a lower bound or upper bound on a variable $x \in \mathcal{X}$ and is typically denoted as $[\![x \geq v]\!]$ and $[\![x \leq v]\!]$ for a constant integer v.

For each variable x_i in $\mathcal{X}$, let us introduce two virtual *signed variables* y_i^+ and y_i^- defined such that $y_i^+ = x_i$ and $y_i^- = -x_i$. We call the corresponding set of signed variables $\mathcal{Y}$. This notation trick allows us to encode the lower bound and upper bound of x_i as the upper bounds of y_i^- and y_i^+ respectively. Note that, with this notation, any literal can be expressed in the form $[\![y \leq v]\!]$ with $y \in \mathcal{Y}$.

3.2 Domains

The current domain of a variable is represented with its upper and lower bounds. All domains are stored in a datastructure $\mathcal{B}$ that stores the upper bound of signed variables and provides access to them through the functions:

- $\text{GETUB}(y)$, where y is a signed variable in $\mathcal{Y}$, returns the current upper-bound of y.
- $\text{SETUB}(y, v)$ where y is a signed variable in $\mathcal{Y}$ and v is an integer, enforces the upper bound of y to be at least as tight as v and:
  - raises a conflict if applying this change would result in an empty domain for y,
  - returns *true* if the domain of y was tightened, or
  - returns *false* otherwise (no-op).

The *domains* structure is associated to a trail $\mathcal{T}$ that records all events affecting $\mathcal{B}$. Each time a domain is modified (i.e., $\text{SETUB}(y, v_{new})$ is called), an event (y, v_{new}, v_{prev}) is appended to the trail where v_{prev} is the previous upper bound of y. The trail serves the purpose of *(i)* letting independent modules scan through the changes that affected the domains and *(ii)* restore a domain to a previous value.

3.3 Efficiently Handling Literals

Given a literal $\ell = [\![y \leq v]\!]$ we say that ℓ is entailed ($\mathcal{B} \vdash \ell$) if $\text{GETUB}(y) \leq v$. We also use the notation $\text{SET}(\ell)$ as a shorthand for $\text{SETUB}(y, v)$.

As it should be obvious from this definition, a literal has no existence of its own but acts as compact representation for statements on the underlying variable. This is unlike previous hybrid SAT/CP approaches where each literal requires the creation of a dedicated boolean variable and explicit propagation to keep the domains of the integer variable in sync with the domain of the boolean variables of its literals [31, 35].

Very concretely, our $[\![y \leq v]\!]$ literal is encoded as the product of the signed variable's identifier and of an integer value. On modern hardware, this can be represented on 64 bits: 32 for the signed variable (31 for variable ID and 1 for the sign) and 32 bits for the integer value. This has some immediate benefits on the computational cost of some operations. First, "creating" a literal only requires concatenating two values on the stack. Second, the existence of a literal does not induce any synchronization process among several variables.

This literal representation does have some drawbacks. In particular, this is a pervasive change that impacts most aspects of a clause-learning solver and is certainly hard to retrofit in an existing code base. Very pragmatically, it also increases the size of a literal (typically from 32 bits in most SAT solvers to 64 bits) which might in turn negatively impact memory consumption and cache usage.

3.4 Conflict-Driven Clause-Learning Search

The solver adopts a common backtracking search mechanism with clause learning (Algorithm 1). This approach corresponds to the CDCL algorithm [24] of which we give a high-level overview.

In a nutshell, the solver iteratively propagates all constraints until a fixed point is reached. This process will typically tighten the domains of the variables and might fail (i.e. result in a contradiction). If the propagation failed, the conflict is analyzed to produce a clause which is recorded in order to avoid the search to run into the same conflict. The search then backtracks to the *asserting level* at which the clause is not violated and proceeds with a new round of propagations. Once a propagation round succeeds, the solver checks if the current assignment is complete, and if not branches by setting an arbitrary literal to true. The process is repeated until either a complete assignment is found, or the search space is exhausted.

4 Difference Logic Reasoner

Let us now focus on the reasoner that is dedicated to handling the set of reified difference constraints, each of the form:

$$\ell \Leftrightarrow x_i + \delta \leq x_j$$

where l is a literal, x_i and x_j are variables in $\mathcal{X}$ and δ is an integer constant.

Algorithm 1 Search procedure overview (CDCL)

```
1:  procedure SEARCH($\mathcal{B}, \mathcal{T}$)
2:      while true do
3:          $res \leftarrow$ PROPAGATE()
4:          if $res$ is conflict then
5:              $clause \leftarrow$ EXPLAIN($res$)
6:              if clause is empty then
7:                  return UNSAT
8:              else
9:                  lvl $\leftarrow$ ASSERTINGLEVEL($clause$)
10:                 BACTRACK($lvl$)
11:                 ADDASSERTINGCLAUSE($clause$)
12:         else if all variables are bound then
13:             return solution($\mathcal{B}$)
14:         else
15:             $\ell \leftarrow$ NEXTDECISION()
16:             SAVESTATE()
17:             SET($\ell$)
```

4.1 Elementary Propagators

We start by decomposing this constraint into four *elementary propagators*. Let us first state the relations that should be enforced in order to maintain bound consistency in the positive case ($\mathcal{B} \vdash l$) and the negative case ($\mathcal{B} \vdash \neg l$):

$$\begin{aligned} l &\begin{cases} x_i \geq v \;\rightarrow\; x_j \geq v + \delta \\ x_j \leq v \;\rightarrow\; x_i \leq v - \delta \end{cases} \\ \neg l &\begin{cases} x_j \geq v \;\rightarrow\; x_i \geq v - \delta + 1 \\ x_i \leq v \;\rightarrow\; x_j \leq v + \delta - 1 \end{cases} \end{aligned} \tag{1}$$

where v is universally quantified (the above and below expression should hold for any value of v). These rules can be reformulated in terms of signed variables, obtaining:

$$\begin{aligned} l &\begin{cases} y_i^- \leq -v \;\rightarrow\; y_j^- \leq -v - \delta \\ y_j^+ \leq v \;\rightarrow\; y_i^+ \leq v - \delta \end{cases} \\ \neg l &\begin{cases} y_j^- \leq -v \;\rightarrow\; y_i^- \leq -v + \delta - 1 \\ y_i^+ \leq v \;\rightarrow\; y_j^+ \leq v + \delta - 1 \end{cases} \end{aligned} \tag{2}$$

The direct benefit of this reformulation is that all four rules are now expressed in a homogenous form, where an upper bound change on a signed variable induces a restriction on another signed-variable's upper bound:

$$\ell : y \leq v \;\rightarrow\; y' \leq v + d. \tag{3}$$

where ℓ is a literal, y and y' and signed variables in $\mathcal{Y}$ and d is an integer constant; We call a rule of this form an *elementary propagator*, noted as:

$$\ell : y \xrightarrow{d} y'$$

Such a propagator needs to be triggered anytime ℓ becomes true or the upper bound of y changes. When this occurs, the propagator will either update the uppper bound of y' or deactivate the edge (i.e. enforcing $\neg\ell$) according to the rules of Table 1.

4.2 Organized propagation

In the previous subsection, we have shown how a reified difference constraint can be broken down into four elementary propagators to enforce consistency. Instead of considering all elementary propagators independently, we propose to orchestrate their propagation. The idea of organized propagation has been previously proposed in the context of SMT [32] and CP [12, 20] solvers, without our notion of elementary propagators.

The detailed procedure for propagation is presented in Algorithm 2. The PROPAGATEBOUNDS(y) function runs an incremental Bellman-Ford algorithm over the active edges. The algorithm is an adaptation of the incremental propagation algorithm for *Simple Temporal Networks (STN)* of Cesta and Oddi [9]. The algorithm is substantially simplified as a result of our unified handling of upper and lower bounds with signed variables. Compared to independent propagation of difference constraints as often done in CP solvers (e.g. [15]) a particularly interesting feature of the algorithm is its early detection of negative cycles (line 31).

The PROPAGATEALL($\mathcal{T}$) algorithm organizes the propagation by

Watched	Condition	Enforced literal
$[\![y \leq v]\!]$	ℓ	$[\![y' \leq v + d]\!]$
ℓ	$[\![y \leq v]\!]$	$[\![y' \leq v + d]\!]$
$[\![y \leq v]\!]$	$\neg[\![y' \leq v + d]\!]$	$\neg\ell$

Table 1. Propagation rules for an elementary propagator $\ell : y \xrightarrow{d} y'$. Propagation is triggered when the *watched* literal becomes true if the condition literal is entailed by the current domains.

Algorithm 2 Propagation of the difference constraints.

```
1:  procedure PROPAGATEALL($\mathcal{T}$)
2:      $i \leftarrow$ index of next unprocessed event in trail
3:      $ActivationQueue \leftarrow \{\}$
4:      while $i < |\mathcal{T}|$ or $ActivationQueue \neq \emptyset$ do
5:          while $i < |\mathcal{T}|$ do
6:              // extract next event in trail
7:              $[\![y \leq v]\!] \leftarrow \mathcal{T}[i]$ ; $i \leftarrow i + 1$
8:              $l = [\![y \leq v]\!]$
9:              for each propagator $p$ activated by $l$ do
10:                 $push(ActivationQueue, p)$
11:             for each inactive propagator $l' : y \xrightarrow{\delta} y'$ do
12:                 // activating would enforce $y' \leq v + \delta$
13:                 if $\mathcal{B} \vdash \neg[\![y' \leq v + \delta]\!]$ then
14:                     SET($\neg l'$)
15:             PROPAGATEBOUNDS($y$)
16:
17:         // no bound changes left, activate pending edges
18:         while $ActivationQueue \neq \emptyset$ do
19:             $y \xrightarrow{\delta} y' \leftarrow pop(ActivationQueue)$
20:             MARKACTIVE($y \xrightarrow{\delta} y'$)
21:             if SETUB($y'$, GETUB($y$) $+ \delta$) then
22:                 PROPAGATEBOUNDS($y'$)
23:
24: procedure PROPAGATEBOUNDS($y_{start}$)
25:     $Queue \leftarrow \{y_{start}\}$
26:     while $Queue \neq \emptyset$ do
27:         $y \leftarrow pop(Queue)$
28:         for each active propagator $y \xrightarrow{\delta} y'$ do
29:             $updated \leftarrow$ SETUB($y', ub(y) + \delta$)
30:             if $updated \wedge y' = y_{start}$ then
31:                 raise Err(cycle)
32:             else if $updated$ then
33:                 $push(Queue, y')$
```

processing each pending event ℓ in the trail and:

- marking activated propagators for later processing (line 10),
- setting as impossible the propagators that could not be activated after the event (line 14),
- propagating the variable whose bound is updated by ℓ by running the incremental bellman-ford algorithm (line 15).

Once all events have been processed, the edges marked for activation are activated one-by-one to maintain the condition of validity of the incremental Bellman-Ford algorithm.

Note that the case where an inconsistency is detected as a result of a domain update is not explicitly covered in Algorithm 2. Instead, we assume that an inconsistency raised by SETUB/SET would be immediately returned by both algorithms together with sufficient information to lazily derive an explanation (covered in the next section).

5 Explanation and Clause Learning

5.1 Background

When facing a conflict during search, CDCL solvers will derive a new *clause* (i.e. a disjunction of literal) whose primary objective is to avoid running into the same conflict later into the search [24]. The clause is derived through an explanation process for which we here give the essential steps.

Explanation occurs when a propagator deduces that a literal ℓ must hold (i.e. it invokes SET(ℓ)) while $\neg\ell$ is already entailed at the current decision level. From this contradiction, the objective is to learn an asserting clause of the form

$$c_1 \wedge \cdots \wedge c_n \implies l_{ass}$$

such that there is a decision level where all c_i literals are entailed and $\neg l_{ass}$ is not.

The key mechanism for building explanations is the following. Provided a contradiction of the form

$$c_1 \wedge \cdots \wedge c_n \implies \bot$$

which is initially the last contradiction we faced ($\ell \wedge \neg\ell \implies \bot$):

1. Select the conjunct c_i that was asserted last.
2. Demand an explanation for c_i, of the form
$$c_i^1 \wedge \cdots \wedge c_i^n \implies c_i$$
3. Apply the resolution rule: substitute c_i in the conflict, obtaining
$$c_1 \wedge \cdots \wedge (c_i^1 \wedge \cdots \wedge c_i^n) \wedge \cdots \wedge c_n \implies \bot$$

The process is repeated until a *single* literal c_j from the current decision level remains in the conflict: the first unique implication point (1UIP). We thus obtain the asserting clause:

$$c_1 \wedge \cdots \wedge c_{j-1} \wedge c_{j+1} \cdots \wedge c_n \implies \neg c_j$$

At this point, the search algorithm would backtrack to the asserting level of the clause and record the clause in a *clause database*. At the following propagation, the clause would be unit and the literal $\neg c_j$ would be set.

5.2 Clause Learning Details

Incorporating explanations and clause learning in our solver is fairly straightforward, as a result of the first-hand support for bound literals and of the very homogenous nature of the difference constraints. Following a lazy explanation schema [14], clause learning requires three elements: *(i)* identification for each entailed literal of the propagator that enforced it, *(ii)* the capability for each propagator to explain the inferences made and *(iii)* a module for storing and propagating clauses.

Point *(i)* is made possible by storing, alongside the upper-bounds in $\mathcal{B}$ and the events in $\mathcal{T}$, the source of the inference (e.g. the identifier of the difference logic propagator from which the update was made). This metadata is provided as an additional parameter in the SETUB and SET functions.

Generating explanations When an explanation is needed (step 2), the propagator that asserted the literal is queried for the implying conjuncts. The explanation process is unique for each constraint and global constraints in particular may have elaborate explanation techniques [30] but is immediate for difference logic in general [35]. For our elementary propagators in particular, Table 1 gives sufficient information to explain any inference made by Algorithm 2. For instance, if the literal $[\![y' \leq 10]\!]$ was asserted by the propagator $\ell : y \xrightarrow{6} y'$, the explanation would be the conjunction $\ell \wedge [\![y \leq 4]\!]$.

A literal that is always true (entailed at the first decision level) is not added to the conflict set. As a result, the learned clause might be empty which implies that the problem is UNSAT.

Clause Learning While several clause learning variants exists, SAT solvers typically favor First Unique Implication Point (1UIP [24]) which corresponds to the one described in the previous background section. It is possible to further refine the clause beyond the first UIP but, at least in SAT solvers, doing so appear to deteriorate the quality of the learned clauses [3].

Note that, regardless of the branching strategy, this clause learning approach will produce clauses that mix literals involving the binary precedence variables as well imposing lower and upper bounds on the start-time numeric variables.

In SMT solvers, the difference logic theory would be required to provide a clause only involving the boolean variables, which is a consequence of the design of SMT solvers where the SAT solver is isolated from the details of the theory and in particular of the domains of numeric variables [25]. A similar choice is made is the context of hybrid CP/SAT solvers to restrict learning to boolean variables [35, 28]. In that case, the choice appeared to be driven by efficiency considerations as including non-boolean variables required them to create and manage bound-literals, which came at very high runtime cost [35]. On the other hand, our design with first-hand support for numeric domains and bound literals allows us to stick to a 1UIP learning scheme without any artificial overhead.

5.3 Disjunctive Reasoner and Clause Database

Once a clause is learned, it needs to be stored and propagated. The disjunctive reasoner is a module in charge of maintaining the consistency of a set of (learned) clauses, each of the form

$$l_1 \vee l_2 \vee \cdots \vee l_n$$

where each l_i is a literal of the form $[\![y_j^{\pm} \leq v]\!]$.

The disjunctive reasoner acts as a SAT solver: it maintains a database of clauses and performs unit propagation. In essence, for each clause cl in the database, we look for distinct literals that are not falsified by the current domains:

- if all literals are false, the clause is violated and a conflict is reported, which will cause the search to backtrack;
- if all but one literal ℓ are false, then the clause is unit and ℓ is enforced ($\textsc{Set}(\ell)$);
- otherwise, two non-false literals are selected and added to a watch-list. Once one of these literal is set, the clause will be reevaluated.

For each second of CDCL search, the solver will typically learn thousands of clauses. Keeping all learnt clause would thus quickly overload the database and dramatically slow down unit propagation. We adopt the database management of MiniSat [11]: the solver is given an initial limit of learned clauses. When this limit is reached, half the learned clauses are removed. The maximum size of the database is regularly scaled up during search, resulting in an exponential growth of the database over time. As in MiniSat, the clauses to keep are selected based on their recent participation in conflicts.[1]

6 Search Strategy

Our objective in this section is to define a general purpose search strategy that *(i)* is not overly tight to the peculiarities of disjunctive scheduling problems and *(ii)* is capable of quickly converging to high quality solutions and *(iii)* is capable of proving the unsatifiability or optimality of a solution. On the other hand, the state-of-the-art approach for jobshop scheduling, relies on a combination of Large-Neighborhood-Search (LNS) whose primary role is to provide high quality solutions and of Failure-Directed-Search (FDS) whose role is to quickly exhaust the search space to prove optimality or run into hard to find solutions [39].

Decision Variables Like other CP approaches that rely on difference constraints to encode disjunctive scheduling problems, we select as decision variables the boolean variables that impose an order between two activities [15, 35]. Intuitively, branching on ordering variables will construct a partial order schedule. It should be noted that, once all ordering variables have been set, the remaining constraints form a Simple Temporal Network [10]. If successfully propagated by Algorithm 2, all remaining numeric variable can then be assigned their lower bound (defining the earliest starting time schedule of the associated STN).

Learning Rate Branching For variable selection, we rely on the Learning Rate Branching (LRB) approach from SAT solvers that aims at maximizing the number conflicts (and thus learned clauses) per decision [23]. The core idea is the following. Once a decision variable is set, either from a direct decision or as a result of propagation, it can participate in conflicts. Following the definition of LRB, we consider that a decision variable participates in a conflict if it (1) appears in the learned clause, or (2) appears in an explanation that was used to produce the learned clause, or (3) appears in the explanation of a literal of the learned clause.

When a decision variable x is unset (i.e. the solver backtracks to a lower decision level) we estimate its local learning rate has the ratio of the number of conflicts it was involved in and of the total number of conflicts that occurred since it was set. This measure is used to update its learning rate estimation lr_x with an exponentially moving average update:

$$lr_x \leftarrow (1-\alpha) \times lr_x + \alpha \times \frac{\text{Num conflicts involved}}{\text{Num conflicts}}$$

At each search step, the decision variable with the highest learning rate is selected.

Solution Guidance For value selection, we adopt a solution guided-approach: for any decision variable selected by LRB, we prefer the value it had in the best solution found so far. This strategy notably builds on the insight that JSP scheduling benefits from intensification of the search around the incumbent solution [4].

Restart Strategy We use a geometric restart strategy with a hundred conflict initially allowed. At each restart, the number of conflicts is increased by a factor 1.2. It is worth noting that, with solution guidance, the solver will attempt exactly the same assignment before or after the restart but will be deviated from it by the learned clauses, that persist across restarts.

Search Initialization Both our variable and value selection strategies rely on information that is not initially available to the solver. In order to bootstrap the search, we thus adopt a greedy strategy that is applied until the first conflict is encountered, with the objective of finding a solution of reasonable quality.

In this setting, the branching variable is selected among start-time variables that are not assigned yet. The selected variable is the one with the smallest lower bound (i.e. earliest start time), with the smallest domain size for tie breaking. As a decision, the variable is assigned its lower bound. This corresponds to common priority rules of greedy methods for the jobshop scheduling problem (First-Come-First-Served with Least-Remaining-Slack as tie breaking [6]).

7 Experiments

We compare the performance of the solver with three state-of-the-art approaches on a number of OSP and JSP instances.

Openshop There are three sets of OSP instances which are widely studied in the literature, 60 instances of [38]; 52 instances of [7]; and 80 instances of [16]. All instances involve “square” problems, with the same number of jobs and machines. The instances range in size from 3x3 to 20x20.

Jobshop There is a large number of JSP benchmarks, stretching back to the 3 instances proposed by [13]. The other benchmarks we consider here are: 40 instances of [22], 5 instances proposed by [1], 10 instances proposed by [2], 4 instances proposed by [40], 20 instances proposed by [36], and finally 70 instances of [38]. Instances range in size from 6x6 to 50x20.

7.1 Compared systems

Aries refers to our own system. It is an open source (MIT-licensed) constraint programming library targeting planning and scheduling problems[2].

[1] We also experimented with *Literals Block Distance* (LBD [3]) as a metric to select which clauses to retain. Despite its success in pure SAT solvers, in our implementation LBD appeared to negatively impact the performance on jobshop and openshop problems.

[2] Available at https://github.com/plaans/aries

CPOPTIMIZER is a state-of-the-art commercial solver for scheduling [21]. It features a highly performant automatic search that notably exploits on a combination Large Neighborhood Search (LNS, [34]) and Failure-Directed Search [39]. In our experiments, we use CPOPTIMIZER version 22.1 and rely on the JSP and OSP models provided in the distribution. The solver is configured to run a single worker.

MISTRAL refers to a solver specifically targetting disjunctive scheduling [15] based on the Mistral constraint programming library [17]. It relies on CP models analogous to ours, where all constraints are (potentially reified) difference constraints. Search exploits two phases: at first, a dichotomic search is used to quickly provide a reasonable initial solution. After some maximum time, a standard branch and bound approach is used, with a variant of the weighted-degree heuristic. We use the implementation provided by the authors in our experiments.[3]

CPSAT is a state-of-the-art CP solver bundled in Google's OR-TOOLS package [28]. It relies on Lazy Clause Generation (LCG [27]) and has been the best contender of the latest MINIZINC challenge [37]. We use the JSP model provided in the distribution of CPSAT, with minor adaptations for OSP.

We do not directly compare with other solvers beside constraint programming as we are not aware of any competitive approach or encoding. The disjunctive encoding has been notably experimented in SMT and MILP solvers [29] but despite being an improvement, remains far from the performance of, e.g., CPOPTIMIZER. SAT solvers have seen some success in proving lower bound in hard JSP instances but so far remain limited to small instances and with very high runtimes that make them unsuitable in an optimization context [18, 19].

7.2 Results

All benchmarks are run on an Intel® Xeon® E-2146G processor with a timeout of 900 seconds and restricted to use a single CPU core. We use the following metrics to evaluate the results:

- Solved: number of solved instances, for which the returned solution was proved optimal.
- APRD: Averaged Percentage Relative Deviation. The PRD of a solver s for an instance is measured as $\frac{c_s - c^*}{c^*} \times 100$ where c_s is the cost of the solution found by s and c^* is cost of the best found solution by any of the compared solvers. The APRD is the average PRD over all considered instances.
- VBS: Virtual Best Solver. Measures the number of times a solver is the *virtual best solver* that returned the best quality solution with ties broken by minimal runtime. If a tie remains between n solvers on an instance, the contribution of the instance to the VBS score is split between tied solvers (increasing their score by $\frac{1}{n}$)
- VBS$_{\text{QUAL}}$: Same as VBS but only considers the solution quality to determine the best solver.

Open Shop Results As it can be seen in Table 2 and Figure 1, on the OSP instances, ARIES and MISTRAL largely dominate. Both solvers prove optimally of their solutions in all instances, the whole

[3] Despite our efforts, we were not able to run the clause-learning variant of mistral [35]. The authors indicated that that performance should be comparable with the one of the baseline mistral. Short experiments also showed that we were able to easily improve on each lower bound they reported.

Table 2. Results for the 195 openshop problems.

	Solved	Total runtime (s)	APRD	VBS	VBS$_{\text{QUAL}}$
ARIES	**195**	**634**	**0**	72.5	**49.2**
CPOPTIMIZER	193	3290	0.005	4.0	48.4
CPSAT	188	7939	0.005	3.0	48.1
MISTRAL	**195**	1624	**0**	**115.5**	**49.2**

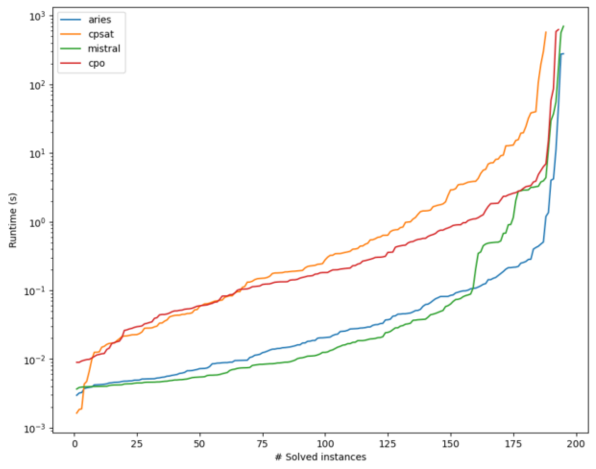

Figure 1. Openshop: Runtime necessary to find proven optimal solutions.

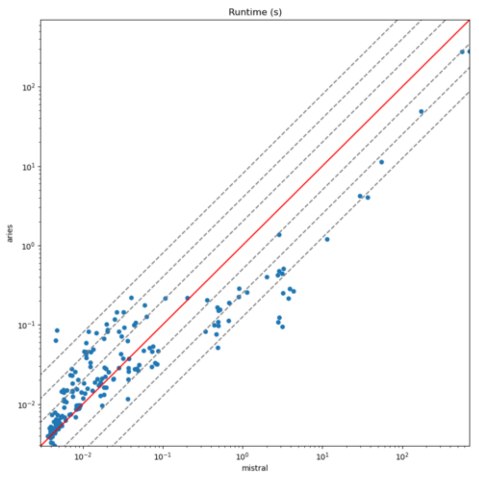

Figure 2. Openshop: Runtime comparison between ARIES and MISTRAL. Dashed lines materialize powers of two runtime differences.

process requiring 634 seconds for ARIES and 1624 seconds for MISTRAL. It can be observed in Figure 2 that MISTRAL performs extremely well on simple instances but is notably slower for all instances requiring more than 300 milliseconds to solve.

CPOPTIMIZER fails to solve two instances (J8-PER10-0 and J8-PER10-2) but overall performs drastically better than in the comparison of [15], before the introduction of Failure-Directed-Search.

Job Shop Results On the JSP instances, CPOPTIMIZER achieves an impressive number of 102 solved instances and strictly dominates in that area: no other solver closed an instance that CPOPTIMIZER didn't. The results are more balanced in terms of solution quality where the results of ARIES and CPOPTIMIZER are very close with both an APRD below 0.5%. The plot of the PRD over individual instances in Figure 4 indicates that there is no strict dominance in this

Table 3. Results for 162 jobshop instances.

	Solved	Total runtime (s)	APRD	VBS	VBS$_{\text{QUAL}}$
ARIES	77	78 214	0.459	**92.3**	61.4
CPOPTIMIZER	**102**	**62 441**	**0.230**	60.8	**63.9**
CPSAT	65	93 208	3.529	2.3	19.2
MISTRAL	63	92 760	2.951	6.5	17.4

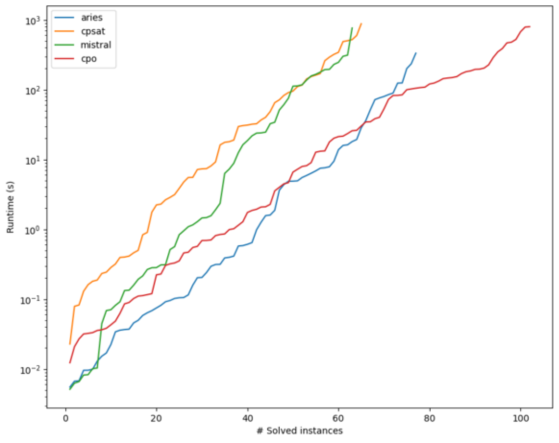

Figure 3. Jobshop: Runtime necessary to find proven optimal solutions.

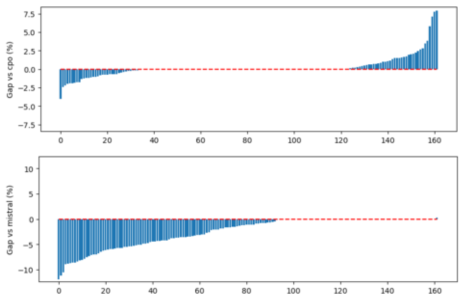

Figure 4. Solution quality variation on jobshop instances. The gap is computed as $\frac{(m_1-m_2)\times 100}{min(m_1,m_2)}$ where m_1 is the makespan of the solution found by ARIES and m_2 is the makespan of the solution found by CPOPTIMIZER (resp. MISTRAL). A negative value indicates that ARIES returned a strictly better solution.

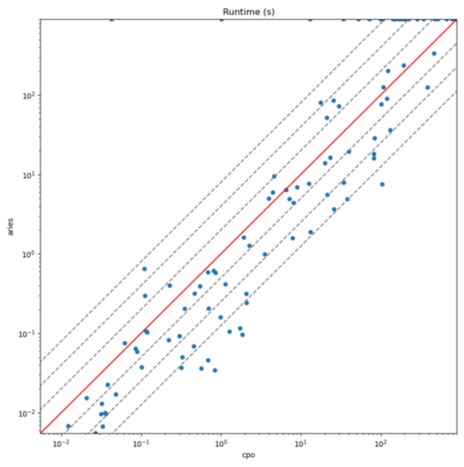

Figure 5. Jobshop: Runtime comparison between ARIES and CPOPTIMIZER. Dashed lines materialize powers of two runtime differences.

matter with both solvers contributing the best solution on a number of instances. This results in comparable VBS$_{\text{QUAL}}$ scores.

In terms of runtime, the results are split between instances where ARIES times-out and the instances where both solvers prove optimality. As it can be seen in Figure 5, for the latter, ARIES generally outperforms CPOPTIMIZER. The combination of high quality solutions with excellent runtime on most instances leads ARIES to win on the *virtual best solver* score by a substantial margin.

7.3 Ablation Study

To identify the contribution of the various features described in the paper, we perform an ablation study based on the 40 Lawrence JSP instances and a timeout of 60 seconds. We highlight below the most striking contributions of each feature, compared to a solver configuration without them.

- **Clause learning**: storing the learned clause in the database is impactful on harder instances (requiring more than 10 seconds). Overall beneficial (average 1.42 speed-up) but detrimental on some problems. Impact increases with runtime.
- **Search initialization**: consistently improves runtime, by up to 8 seconds. Benefits shrink on hard instances.
- **LRB**: switching to the more common VSIDS [24] decreases the decisions/conflict ratio and is highly detrimental on hard instances: on average 2.2 times slower on instances that require more than 10 seconds to solve. Impact increases with runtime.
- **Extended conflicting variables**: ignoring the category (3) of variables from the conflict in LRB makes the solver 1.4 times slower, almost regardless of instance hardness.
- **Solution guidance**: removal is very detrimental, resulting in only 28 problems solved (vs 36 with solution guidance)
- **implicit literals**: cannot be removed from our implementation. In a similar setting, Siala et al. [35] suggest that creating bound literals explicitly would slow down their solver by a factor 10. This lead them to choose a strategy that would not require bound-literals in the learned clause. We do not observe any such impact in our implementation.

8 Conclusion

In this paper, we showed how a carefully designed solver, leveraging advancements in the CP and SAT fields, can be made to compete with the established state-of-the-art on perhaps the most fundamental of scheduling problems (CPOptimizer, since FDS in 2015 [39]).

Novel and critical in this integration are the implicit representation of literals and the exploitation of signed variables for uniform representation. As literals (or boolean variables in CP) are such a fundamental part of solvers, these representations cannot be easily retrofitted in an existing codebase. It allows us to greatly simplify the integration of SAT and CP solvers, removing the need to (1) create boolean variables for bound-literals, (2) having dedicated propagators to synchronize their domains with the one of the original variable, and (3) managing their lifetime. All while reducing their overhead and streamlining implementation.

As it is built from the ground up with a CP-SAT integration in mind, the resulting solver remains extremely simple in its conception and implementation. Despite its simplicity and excellent performance, very little is actually specific to disjunctive scheduling in the solver which opens up its application to more challenging use-cases. We notably wish to pursue development towards the needs of temporal planning, where we have already seen promising results [5].

Acknowledgments

This work has been partially supported by AIPlan4EU, a project funded by EU Horizon 2020 research and innovation program under GA n.101016442.

References

[1] Joseph Adams, Egon Balas, and Daniel Zawack, 'The shifting bottleneck procedure for job shop scheduling', *Management Science*, (1988).

[2] David Applegate and William Cook, 'A computational study of the job-shop scheduling problem', *ORSA Journal of Comuting*, (1991).

[3] Gilles Audemard and Laurent Simon, 'Predicting learnt clauses quality in modern sat solvers', in *International Joint Conference on Artificial Intelligence (IJCAI)*, (2009).

[4] J. Christopher Beck, 'Solution-guided multi-point constructive search for job shop scheduling', *Journal of Artificial Intelligence Research (JAIR)*, (2007).

[5] Arthur Bit-Monnot, 'Experimenting with Lifted Plan-Space Planning as Scheduling: Aries in the 2023 IPC', in *Proceedings of the 2023 International Planning Competition (IPC)*, (2023).

[6] Jacek Blazewicz, Wolfgang Domschke, and Erwin Pesch, 'The job shop scheduling problem: Conventional and new solution techniques', *European Journal of Operational Research*, (1996).

[7] Peter Brucker, Johann Hurink, Bernd Jurisch, and Birgit Wöstmann, 'A branch & bound algorithm for the open-shop problem', *Discrete Applied Mathematics*, (1997).

[8] Jacques Carlier and Eric Pinson, 'An algorithm for solving the job-shop problem', *Management Science*, (1989).

[9] Amedeo Cesta and Angelo Oddi, 'Gaining Efficiency and Flexibility in the Simple Temporal Problem.', in *International Symposium on Temporal Representation and Reasoning (TIME)*, (1996).

[10] Rina Dechter, Itay Meiri, and Judea Pearl, 'Temporal constraint networks', in *Artificial Intelligence*, (1989).

[11] Niklas Eén and Niklas Sörensson, 'An extensible sat-solver', in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, (2003).

[12] Thibaut Feydy, Andreas Schutt, and Peter J. Stuckey, 'Global difference constraint propagation for finite domain solvers', in *International Symposium on Principles and Practice of Declarative Programming (PPDP)*, (2008).

[13] Henry Fisher, 'Probabilistic learning combinations of local job-shop scheduling rules', *Industrial scheduling*, (1963).

[14] Ian P. Gent, Ian Miguel, and Neil C. A. Moore, 'Lazy explanations for constraint propagators', in *International Symposium on Practical Aspects of Declarative Languages*, (2010).

[15] Diarmuid Grimes and Emmanuel Hébrard, 'Solving variants of the job shop scheduling problem through conflict-directed search', *INFORMS Journal on Computing*, (2015).

[16] Christelle Guéret and Christian Prins, 'A new lower bound for the open-shop problem', *Annals of Operations Research*, (1999).

[17] Emmanuel Hebrard, 'Mistral, a constraint satisfaction library', *Third International CSP Solver Competition*.

[18] Hong Huang and Shaohua Zhou, 'An efficient sat algorithm for complex job-shop scheduling', in *International Conference on Manufacturing Science and Engineering*, (2018).

[19] Miyuki Koshimura, Hidetomo Nabeshima, Hiroshi Fujita, and Ryuzo Hasegawa, 'Solving Open Job-Shop Scheduling Problems by SAT Encoding', *IEICE Trans. Inf. Syst.*, (2010).

[20] Philippe Laborie and Jerôme Rogerie, 'Reasoning with Conditional Time-Intervals', in *International Florida Artificial Intelligence Research Society Conference (FLAIRS)*, (2008).

[21] Philippe Laborie, Jérôme Rogerie, Paul Shaw, and Petr Vilím, 'IBM ILOG CP optimizer for scheduling', *Constraints*, (2018).

[22] S. Lawrence, 'Resource constrained project scheduling: An experimental investigation of heuristic scheduling techniques (supplement)', (1984).

[23] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and K. Czarnecki, 'Learning rate based branching heuristic for sat solvers', in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, (2016).

[24] Joao Marques-Silva, Inês Lynce, and Sharad Malik, 'Conflict-driven clause learning sat solvers', in *Handbook of Satisfiability*, (2009).

[25] Robert Nieuwenhuis and Albert Oliveras, 'Dpll(t) with exhaustive theory propagation and its application to difference logic', in *International Conference on Computer Aided Verification*, (2005).

[26] Eugeniusz Nowicki and Czeslaw Smutnicki, 'An advanced tabu search algorithm for the job shop problem', *Journal of Scheduling*, (2005).

[27] Olga Ohrimenko, Peter James Stuckey, and Michael Codish, 'Propagation via lazy clause generation', *Constraints*, (2009).

[28] Laurent Perron and Vincent Furnon, 'OR-Tools'.

[29] Sabino Francesco Roselli, Kristofer Bengtsson, and Knut Åkesson, 'SMT solvers for job-shop scheduling problems: Models comparison and performance evaluation', in *International Conference on Automation Science and Engineering (CASE)*, (2018).

[30] Andreas Schutt, Thibaut Feydy, and Peter James Stuckey, 'Explaining time-table-edge-finding propagation for the cumulative resource constraint', (2013).

[31] Andreas Schutt, Thibaut Feydy, Peter James Stuckey, and Mark Wallace, 'Solving rcpsp/max by lazy clause generation', *Journal of Scheduling*, (2013).

[32] Roberto Sebastiani, 'Lazy satisability modulo theories', *Journal on Satisfiability, Boolean Modeling and Computation*, (2007).

[33] D. Y. Sha and Cheng-Yu Hsu, 'A new particle swarm optimization for the open shop scheduling problem', *Computers & Operations Research*, (2008).

[34] Paul Shaw, 'Using constraint programming and local search methods to solve vehicle routing problems', in *International Conference on Principles and Practice of Constraint Programming (CP)*, (1998).

[35] Mohamed Siala, Christian Artigues, and Emmanuel Hebrard, 'Two clause learning approaches for disjunctive scheduling', in *International Conference on Principles and Practice of Constraint Programming (CP)*, (2015).

[36] Robert H. Storer, S. David Wu, and Renzo Vaccari, 'New search spaces for sequencing problems with application to job shop scheduling', *Management Science*, (1992).

[37] Peter J. Stuckey, Thibaut Feydy, Andreas Schutt, Guido Tack, and Julien Fischer, 'The minizinc challenge 2008–2013', *AI Magazine*, (2014).

[38] E. Taillard, 'Benchmarks for basic scheduling problems', *European Journal of Operational Research*, (1993).

[39] Petr Vilím, Philippe Laborie, and Paul Shaw, 'Failure-directed search for constraint-based scheduling', in *Integration of AI and OR Techniques in Constraint Programming (CPAIOR)*, (2015).

[40] Takeshi Yamada and Ryohei Nakano, 'A genetic algorithm applicable to large-scale job-shop problems', in *Parallel Problem Solving from Nature*, (1992).