

An Incomplete Constraint-Based System for Scheduling with Renewable Resources

Cédric Pralet^(✉)

ONERA – The French Aerospace Lab, 31055 Toulouse, France
`cedric.pralet@onera.fr`

Abstract. In this paper, we introduce a new framework for managing several kinds of renewable resources, including disjunctive resources, cumulative resources, and resources with setup times. In this framework, we use a list scheduling approach in which a priority order between activities must be determined to solve resource usage conflicts. In this context, we define a new differentiable constraint-based local search invariant which transforms a priority order into a full schedule and which incrementally maintains this schedule in case of change in the order. On top of that, we use multiple neighborhoods and search strategies, and we get new best upper bounds on several scheduling benchmarks.

1 Introduction

In scheduling, renewable resources are resources which are consumed during the execution of activities and released in the same amount at the end of these activities. Such resources are present in most scheduling problems, and various types of renewable resources were extensively studied in the literature, such as *disjunctive resources*, which can perform only one activity at a time, *disjunctive resources with setup times*, which can require some time between activities successively realized by the resource, or *cumulative resources*, which can perform several activities in parallel up to a given capacity. These three kinds of renewable resources are respectively present in Job Shop Scheduling Problems (JSSPs [30]), Job Shop Scheduling Problems with Sequence-Dependent Setup Times (SDST-JSSPs [2]), and Resource Constrained Project Scheduling Problems (RCPSPs [8]).

In the constraint programming community, specific global constraints were defined to efficiently deal with renewable resources, like the *disjunctive* and *cumulative* constraints [1, 10], together with efficient propagators based on edge-finding [11, 36], timetable edge-finding [37], or on mechanisms to deal with setup times [34]. Following these developments, constraint programming is nowadays one of the best systematic approach for solving scheduling problems with renewable resources [19, 21, 32].

In parallel, several incomplete search techniques were developed in the scheduling community to quickly produce good-quality solutions on large instances. One of these is *list scheduling*. It manipulates a priority list between

activities, and at each step considers the next activity in the list and inserts it at the earliest possible time without delaying activities already placed in the schedule (so-called *serial schedule generation scheme*). For RCPSP, such a list scheduling approach is used for heuristic search [23, 24] but also for designing local search [12] or genetic algorithms [20].

In this paper, we propose a combination between list scheduling and constraint programming. More specifically, we combine list scheduling with constraint-based local search [35], with the goal of being able to deal with various types of renewable resources (disjunctive or cumulative, with or without setup times). To get such a combination, we define a new constraint-based scheduling system composed of three layers: (1) an *incremental evaluation layer*, used for estimating very quickly the impact of local modifications on a given priority list, (2) a *neighborhood layer*, containing a catalog of neighborhoods usable for updating priority lists, and (3) a *search strategy layer*, which implements various techniques for escaping local minima and plateaus.

The paper is organized as follows. Sections 2–3 present the new framework considered and a lazy schedule generation scheme for this framework. Sections 4–5 detail the first layer mentioned above. Sections 6–7 give a brief overview of the second and third layers, and Sect. 8 shows the performance of the approach on standard benchmarks. In our current implementation, the techniques proposed are actually applied to a wider class of problems involving release and due dates for activities, time-dependent processing times, resource availability windows, and choices on the resources used by activities. We present here a simplified version for readability issues.

2 RCPSP with Sequence-Dependent Setup Times (SDST-RCPSP)

To simultaneously cover cumulative resources and disjunctive resources with setup times, we introduce a new framework called *Resource Constrained Project Scheduling Problem with Sequence-Dependent Setup Times* (SDST-RCPSP). The more complex part of this unifying framework (cumulative resources with setup times) can also be useful in practice. For instance, in manufacturing, a painting machine might be able to simultaneously paint several items with the same color, while requiring a setup time to change the color used by the machine when needed. In space applications, satellites can be equipped with several communication channels allowing them to transmit several data files in parallel to a given ground reception station, while requiring a setup time to change the pointing of the satellite to download data to another station.

Formally, an SDST-RCPSP is defined by a set of renewable resources \mathcal{R} and by a set of activities \mathcal{A} to be realized. Each resource $r \in \mathcal{R}$ has a maximum capacity K_r (equal to 1 for disjunctive resources), a set of possible running modes \mathcal{M}_r (reduced to a singleton for resources without setup times), and an initial running mode $m_{0,r} \in \mathcal{M}_r$. For each pair of distinct resource modes $m, m' \in \mathcal{M}_r$,

a setup time $\Delta_{r,m,m'}$ is introduced to represent the duration required by resource r to make a transition from mode m to mode m' .

Each activity $a \in \mathcal{A}$ has a duration (or processing time) p_a and consumes a set of resources $\mathcal{R}_a \subseteq \mathcal{R}$. We assume that $p_a > 0$ when activity a consumes at least one resource. With each activity a and each resource $r \in \mathcal{R}_a$ are associated the quantity of resource $q_{a,r} \in [1..K_r]$ consumed by a over r , and the resource mode $m_{a,r} \in \mathcal{M}_r$ required for realizing a . In the following, for every resource r , we denote by \mathcal{A}_r the set of activities a which consume r (i.e. such that $r \in \mathcal{R}_a$). Activities are also subject to an acyclic set of *project precedence constraints* $\mathcal{P} \subseteq \mathcal{A} \times \mathcal{A}$, which contains pairs of activities (a, b) such that b cannot start before the end of a .

A *solution* to an SDST-RCPSP assigns a start time $\sigma_a \in \mathbb{N}$ to each activity $a \in \mathcal{A}$. The end time of a is then given by $\sigma_a + p_a$. A solution is said to be *consistent* when constraints in Eqs. 1 to 4 hold. These constraints impose that all project precedences must be satisfied (Eq. 1), that the capacity of resources must never be exceeded (Eq. 2), that there must be a sufficient setup time between activities requiring distinct resource modes (Eq. 3), and a sufficient setup time with regards to the initial modes (Eq. 4).

$$\forall (a, b) \in \mathcal{P}, \sigma_a + p_a \leq \sigma_b \quad (1)$$

$$\forall r \in \mathcal{R}, \forall a \in \mathcal{A}_r, \sum_{b \in \mathcal{A}_r \mid \sigma_b \leq \sigma_a + p_a} q_{b,r} \leq K_r \quad (2)$$

$$\forall r \in \mathcal{R}, \forall a, b \in \mathcal{A}_r \text{ s.t. } m_{a,r} \neq m_{b,r}, \quad (3)$$

$$(\sigma_a \geq \sigma_b + p_b + \Delta_{r,m_b,r,m_{a,r}}) \vee (\sigma_b \geq \sigma_a + p_a + \Delta_{r,m_{a,r},m_b,r})$$

$$\forall r \in \mathcal{R}, \forall a \in \mathcal{A}_r \text{ s.t. } m_{a,r} \neq m_{0,r}, (\sigma_a \geq \Delta_{r,m_{0,r},m_{a,r}}) \quad (4)$$

A solution is said to be *optimal* iff it minimizes the makespan, defined as the end time of the last activity ($\max_{a \in \mathcal{A}} (\sigma_a + p_a)$).

Precedence Graphs. Another way of defining a solution schedule is the standard concept of *precedence graph*. Such a graph contains nodes labeled by activities, and arcs $a \rightarrow b$ labeled by the duration of a (see Fig. 1, upper part). These arcs correspond to precedence constraints “ b can start only once a is finished”. Each arc $a \rightarrow b$ corresponds either to a *project precedence* $(a, b) \in \mathcal{P}$ given in the initial specification (dotted lines in Fig. 1), or to a *resource precedence* posted to prevent resources from being overused (continuous lines in Fig. 1). A precedence graph must be acyclic, and it also contains two dummy activities of null duration called the *source node* s and the *sink node* t , which respectively represent the start and the end of the schedule. The precedence graph G contains arcs $s \rightarrow a$ and $a \rightarrow t$ that guarantee that the source and the sink activities respectively precede and follow every activity in \mathcal{A} . In the case of SDST-RCPSP, the precedence graph also contains setup activities $setup_{a,r}$ for changing the current running mode m of a resource r just before realizing an activity a requiring another running mode $m_{a,r} \neq m$. The duration of $setup_{a,r}$ is then given by $\Delta_{r,m,m_{a,r}}$.

From this, it is possible to compute, for every activity a , the length of the longest path from the source node to a in G , denoted by $d_{s,a}$, and the length of

the longest path from a to the sink node, denoted by $d_{a,t}$. These distances are given inside each activity node in Fig. 1 (e.g., $d_{s,D} = 4$ and $d_{D,t} = 7$). Then, the earliest start time of a is given by $est_a = d_{s,a}$, the makespan mk of the schedule corresponds to the distance $d_{s,t}$ from the source to the sink, and the latest start time of a is given by $lst_a = mk - d_{a,t}$. The resulting earliest and latest time schedules are given in Fig. 1 (middle part). An activity is said to be *critical* iff $est_a = lst_a$, and its temporal flexibility is given by $mk - (d_{s,a} + d_{a,t})$. In Fig. 1, activities A , C , and E are critical.

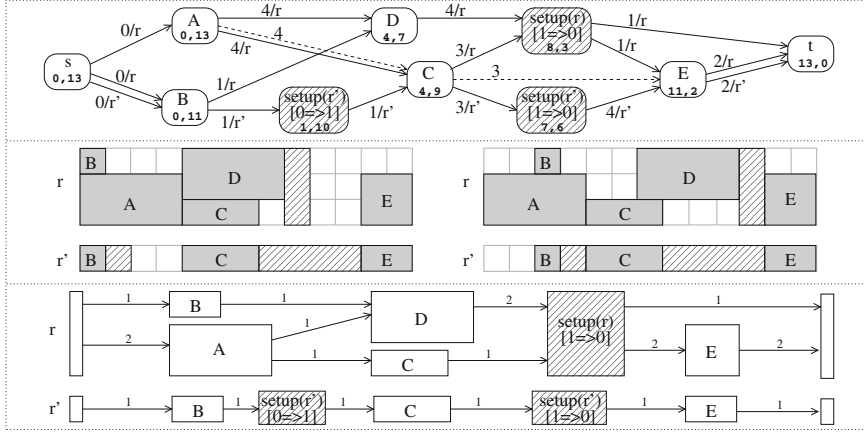


Fig. 1. Precedence graph (top), resulting earliest schedule (middle left) and latest schedule (middle right), and flow-network (bottom) for a SDST-RCPSP where $\mathcal{R} = \{r, r'\}$, $\mathcal{A} = \{A, B, C, D, E\}$, $\mathcal{P} = \{(A, C), (C, E)\}$, $p_A = p_D = 4$, $p_B = 1$, $p_E = 2$, $p_C = 3$, $\mathcal{R}_A = \mathcal{R}_D = \{r\}$, $\mathcal{R}_B = \mathcal{R}_C = \mathcal{R}_E = \{r, r'\}$, $K_r = 3$, $K_{r'} = 1$, $q_{A,r} = q_{D,r} = q_{E,r} = 2$, $q_{B,r} = q_{C,r} = 1$, $q_{x,r'} = 1$, $\mathcal{M}_r = \mathcal{M}_{r'} = \{0, 1\}$, $m_{0,r} = m_{A,r} = m_{B,r} = m_{C,r} = m_{D,r} = 1$, $m_{E,r} = 0$, $\Delta_{r,0,1} = \Delta_{r,1,0} = 1$, $m_{0,r'} = m_{B,r'} = m_{E,r'} = 0$, $m_{C,r'} = 1$, $\Delta_{r',0,1} = 1$, $\Delta_{r',1,0} = 4$. In the precedence graph, an arc $a \rightarrow b$ label by r/x is used when the duration of a is x and the precedence link is introduced because of resource r . The priority list used is $[A, B, C, D, E]$.

To deal with cumulative resources, as in [4], it is also possible to represent, for each resource precedence $a \rightarrow b$, the number of resource units $\phi_{a \rightarrow b}^r$ released at the end of activity a and used by activity b . These resource transfers are represented in a flow-network (see Fig. 1, bottom part). To get a consistent flow-network, the sum of all resource flows associated with a resource r and which respectively point to and come out of an activity node a must be equal to the amount of resource $q_{a,r}$ required by a . For setup activities, the sum of these flows must be equal to the total capacity of the resource (K_r), to guarantee that every resource has a unique running mode at any time. Last, all resource flows $\phi_{a \rightarrow b}^r$ used in the flow-network must be consistent with the resource modes associated with activities. This means for instance that there cannot be a direct flow $\phi_{a \rightarrow b}^r$ between two activities $a, b \in \mathcal{A}_r$ such that $m_{a,r} \neq m_{b,r}$.

3 A Lazy Precedence Graph Generation Scheme

As said in the introduction, instead of directly searching for activity start times σ_a or for precedence graphs, we use a list scheduling approach in which we search for a priority list $\mathcal{O} = [a_1, \dots, a_n]$ containing all activities in \mathcal{A} . This priority list is then transformed into a precedence graph through a serial schedule generation scheme. In the following, we manipulate only *consistent priority lists* \mathcal{O} , which are such that for every project precedence (a, b) in \mathcal{P} , activity a is placed before activity b in \mathcal{O} . Also, we denote by \mathcal{O}_r the sequence of successive activities which consume r . On the example of Fig. 1, $\mathcal{O} = [A, B, C, D, E]$, $\mathcal{O}_r = [A, B, C, D, E]$ and $\mathcal{O}_{r'} = [B, C, E]$.

The generation scheme that we use starts from a precedence graph containing only the source node. At each step, it considers the next activity a in priority list \mathcal{O} and tries to insert it into the current schedule at the earliest possible time σ_a so that starting a at time σ_a is feasible in terms of project precedences and resource consumptions, and that activities already placed in the schedule are not delayed. The generation scheme also computes the so-called *pending resource flows* obtained for r after the insertion of a , denoted by $\Phi_{a,r}$. Formally, these pending resource flows correspond to a list

$$\Phi_{a,r} = [(\phi_1, z_1), \dots, (\phi_h, z_h)]$$

such that for every $i \in [1..h]$, a resource amount ϕ_i is released at the end of activity z_i and is available for future activities. Figure 2 (upper part) gives the set of pending flows over resource r obtained after the insertion of each activity. For example, the set of pending flows after the insertion of C would be $\Phi_{C,r} = [(1, B), (1, A), (1, C)]$, and the set of pending flows after the insertion of E would be $\Phi_{E,r} = [(1, \text{setup}_{r,1,0}), (2, E)]$. In the following, we assume that the pending flows are ordered by increasing release time, that is $\sigma_{z_1} + p_{z_1} \leq \dots \leq \sigma_{z_h} + p_{z_h}$.

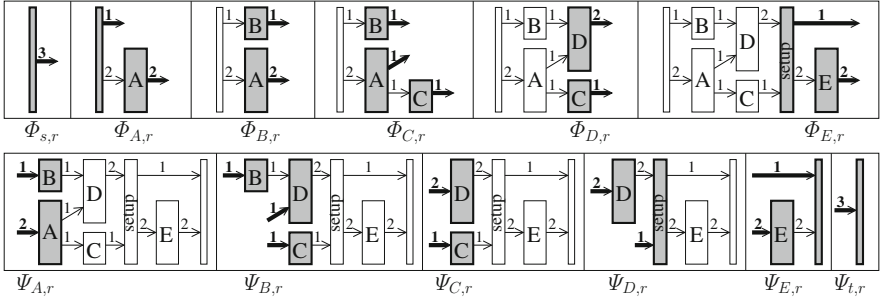


Fig. 2. Forward pending flows (upper part) and backward pending flows (bottom part)

We now come back to the generation process. For every resource r , the initial list of pending flows after source activity s is $\Phi_{s,r} = [(K_r, s)]$, since

initially, the whole capacity of the resource is available. Then, let a be the next activity to consider in the priority list. For each resource $r \in \mathcal{R}_a$, let $\Phi_{prev_{a,r},r} = [(\phi_1, z_1), \dots, (\phi_h, z_h)]$ be the pending flows obtained after the insertion of the activity $prev_{a,r}$ that immediately precedes a on r . If the resource mode $m_{prev_{a,r},r}$ associated with this predecessor is distinct from the resource mode $m_{a,r}$ required by a , a setup is needed before a and the earliest time $\sigma_{a,r}$ at which r can support the realization of a is given by Eq. 5. Otherwise, $\sigma_{a,r}$ corresponds to the earliest time at which resource level $q_{a,r}$ can be made available according to the pending resource flows in $\Phi_{prev_{a,r},r}$ (Eq. 6).

$$\sigma_{a,r} \leftarrow \sigma_{z_h} + p_{z_h} + \Delta_{r, m_{prev_{a,r},r}, m_{a,r}} \text{ if } m_{prev_{a,r},r} \neq m_{a,r} \quad (5)$$

$$\sigma_{z_k} + p_{z_k} \text{ otherwise, with } k = \min\{k' \in [1..h] \mid \sum_{i \in [1..k']} \phi_i \geq q_{a,r}\} \quad (6)$$

From these elements, the start time of a produced by the schedule generation scheme is given by the maximum between the end time of project predecessors of a , and the earliest start times of resource consumptions associated with a :

$$\sigma_a \leftarrow \max(\max_{(b,a) \in \mathcal{P}} (\sigma_b + p_b), \max_{r \in \mathcal{R}_a} \sigma_{a,r}) \quad (7)$$

In Fig. 1, the earliest start time of C on resource r is $\sigma_{C,r} = 1$ (the single resource unit required for C can be available just after the end of activity B), its earliest start time on r' is $\sigma_{C,r'} = 2$ (requirement to change the resource running mode), and its earliest start time according to its project predecessors equals 4 (end time of A). As a result, the earliest start time of C is $\sigma_C \leftarrow \max(4, 1, 2) = 4$.

The activity a considered is then truly introduced in the precedence graph. If the running mode of resource r before the introduction of a is distinct from the resource mode $m_{a,r}$ required by a , then a setup activity $setup_{a,r}$ is added to the precedence graph, together with one arc $z_i \rightarrow setup_{a,r}$ with flow ϕ_i for each pending flow (ϕ_i, z_i) in $\Phi_{prev_{a,r},r}$, and one arc $setup_{a,r} \rightarrow a$ with flow $q_{a,r}$ pointing to a . The new pending flows after the introduction of a are obtained from these updates.

Otherwise, if the current state of resource r is equal to the resource state $m_{a,r}$ required by a , we compute the maximum index k such that pending flow (ϕ_k, z_k) in $\Phi_{prev_{a,r},r}$ releases its resource units *before* σ_a (i.e. $\sigma_{z_k} + p_{z_k} \leq \sigma_a$). This specific index is chosen for reducing resource idle periods (minimization of the length of the potential idle period created between the end of z_k and the start of a). In the precedence graph, we add arc $z_k \rightarrow a$ to represent that resource units released by z_k are transmitted to a . The number of resource units transferred is $\min(\phi_k, q_{a,r})$. If this does not suffice to cover the total amount of resource required by a (case $\phi_k < q_{a,r}$), we continue with pending flows $(\phi_{k-1}, z_{k-1}), (\phi_{k-2}, z_{k-2}) \dots$ until resource consumption $q_{a,r}$ is fully covered. The new pending flows after the introduction of a are obtained from these updates. For resource r in Fig. 1, activity C , for which $\sigma_C = 4$, consumes resource units released by A , while activity D consumes resource units released by both A and B .

These operations are successively realized for each activity in priority list \mathcal{O} . The generation scheme defined is *lazy* because it only considers pending

flows, which are located at the end of the resource usage profile. It does not exploit potential valleys present in this profile. In terms of scheduling, this entails that the generation scheme does not necessarily generate *active schedules*, which means that some activities might be started earlier without delaying other activities. Nevertheless, for every active schedule \mathcal{S} , there exists a priority list which generates this schedule (it suffices to order activities by increasing start times in \mathcal{S}). This implies that for regular performance measures such as makespan minimization, there exists a priority list inducing an optimal schedule. Also, all schedules produced are *semi-active*, meaning that for every activity a , there is no date $t < \sigma_a$ such that all start times $t, t+1, \dots, \sigma_a$ lead to a consistent schedule.

One advantage of the lazy schedule generation scheme proposed is its low time complexity compared to serial schedule generation schemes used for RCPSP, which maintain a global resource usage profile. Indeed, each insertion into a given resource has a *worst-case* time complexity which is linear in M , the maximum number of resource consumptions that might be performed in parallel on r , instead of a complexity linear in the number of activities in \mathcal{A} . In particular, for a disjunctive resource, our lazy schedule generation scheme takes a constant time for each consumption insertion.

4 Incremental Schedule Maintenance Techniques

The techniques defined in the previous section can be used to automatically derive a full schedule from a priority list \mathcal{O} . To progressively get better solutions, a strategy is to perform local search in the space of priority lists, for instance by changing the position of one activity in the list, by swapping the positions of two activities, or by moving a block of successive activities.

This is where constraint programming comes into play, since we use Constraint-Based Local Search (CBLS [35]) for realizing these updates efficiently. As in standard constraint programming, CBLS models are defined by decision variables, constraints, and criteria. One distinctive feature is that in CBLS, all decision variables are assigned when searching for a solution. The search space is then explored by performing local moves which reassign some decision variables, and it is explored more freely than in tree search with backtracking. One specificity of CBLS models is that they manipulate *invariants*, which are one-way constraints $x \leftarrow exp$ where x is a variable and exp is a functional expression of other variables of the problem, such as $x \leftarrow \text{sum}(i \in [1..N]) y_i$. During local moves, these invariants are efficiently maintained thanks to specific procedures that incrementally reevaluate the output of invariants (left part) in case of change in their inputs (right part).

Invariant Inputs. To achieve our goal, we introduce a new CBLS invariant which takes as an input a priority list \mathcal{O} . In our CBLS solver, priority list \mathcal{O} is implemented based on the data structure defined in [7] for encoding total orders. This data structure maintains the predecessor and the successor of each element a in the list. It also assigns to a an integer $tag_a \in [0..2^{31} - 1]$ such that if a is located

before b in the list, then $tag_a < tag_b$. When inserting an element c between two successive elements a and b , the tag of c is set to $\frac{tag_a + tag_b}{2}$ if $tag_a + 1 < tag_b$, and otherwise operations are used to retag elements so as to allow some space between tag_a and tag_b .

The invariant introduced also takes as an input one boolean parameter u_a for each activity $a \in \mathcal{A}$, specifying whether resource consumptions associated with a are activated or not. When u_a takes value false, only the impact of project precedence constraints is taken into account for a , and in a full schedule, u_a must take value true for all activities. Adding such inputs to the invariant can bring a better view of the promising insertion positions for an activity (see Sect. 5).

Invariant Outputs. The invariant built maintains the precedence graph and returns, for each activity a , the distance from the source node to a ($d_{s,a}$, equal to σ_a), and the distance from a to the sink node ($d_{a,t}$). It also maintains, for each resource r , the sequence \mathcal{O}_r of activities which successively use r . Each sequence \mathcal{O}_r is represented as a linked list defining the predecessor $prev_{a,r}$ and the successor $next_{a,r}$ of a in \mathcal{O}_r . For incremental computation reasons, the invariant also maintains, for each activity a and each resource $r \in \mathcal{R}_a$, the pending resource flows $\Phi_{a,r}$ after the insertion of a . For each resource and each activity, the space complexity required to record these flows is $O(M)$, with M the maximum number of resource consumptions that might be performed in parallel on a resource. In the end, the *lazySGS* invariant defined takes the form:

$$(\{d_{s,a} \mid a \in \mathcal{A}\}, \{d_{a,t} \mid a \in \mathcal{A}\}, \{\mathcal{O}_r \mid r \in \mathcal{R}\}) \leftarrow lazySGS(\mathcal{O}, \{u_a \mid a \in \mathcal{A}\}) \quad (8)$$

Even if such an invariant is rather *large*, it can be used in a CBLS model containing other invariants. For example, output variables $d_{s,a}$ and $d_{a,t}$ are usually used as inputs for invariants that incrementally compute the temporal flexibility of each activity ($flex_a \leftarrow d_{s,t} - (d_{s,a} + d_{a,t})$), and then for computing the set of critical activities using a set invariant ($CriticalSet \leftarrow \{a \in \mathcal{A} \mid flex_a = 0\}$). Also, when considering extensions of the invariant in which there is a choice on the resources used by activities, the associated resource allocation decisions can be connected to other invariants managing other constraints such as limitations on non-renewable resources.

Incremental Evaluation. The incremental evaluation function of the CBLS invariant corresponds to Algorithm 1. This algorithm is inspired by the incremental longest paths maintenance algorithms introduced in [22]. The main difficulty is that in our case, we need more than incrementally computing distances in a precedence graph: we also need to incrementally manage the construction of the precedence graph itself, which depends on the pending resource flows successively obtained.

Two kinds of changes must be considered: (1) moves of some activities in the priority list; (2) activation/deactivation of resource consumptions for some activities. To handle these changes, the first step consists in updating sequences of resource consumptions applied to resources (\mathcal{O}_r). All activities for which changes

occurred are removed from sequences \mathcal{O}_r , and all activities whose consumptions are still activated ($u_a = \text{true}$) are sorted by increasing tag and reintroduced in \mathcal{O}_r via a single forward traversal. These updates are realized by a call to `UPDATEORDERS` at line 1.

For performing forward revisions, Algorithm 1 uses several data structures:

- a *global revision queue* Q^{rev} containing activities for which some revisions must be made; to ensure that revisions are realized in a topological order with relation to the precedence graph, activities are ordered in Q^{rev} as their tags in \mathcal{O} ;
- for every activity a , a *precedence revision set* \mathcal{P}_a^{rev} containing activities b such that (b, a) is a precedence in \mathcal{P} and the end time of b has been updated;
- for every activity a , a *resource revision set* \mathcal{R}_a^{rev} containing resources $r \in \mathcal{R}_a$ such that the impact of the consumption of resource r by a might not be up-to-date.

Data structures Q^{rev} , \mathcal{P}_a^{rev} , \mathcal{R}_a^{rev} are initialized at line 2 by a call to `INITREVISIONS`. The latter adds to revision queue Q^{rev} all activities a which have been activated or deactivated since the last evaluation of the invariant. For these activities, all resources in \mathcal{R}_a are added to revision set \mathcal{R}_a^{rev} . For every other activity a and every resource $r \in \mathcal{R}_a$, if activity a has a new predecessor in \mathcal{O}_r , then a is added to Q^{rev} and resource r is added to \mathcal{R}_a^{rev} . Last, \mathcal{P}_a^{rev} is initially empty for every activity a .

After these initialization steps, while there are some revisions left in Q^{rev} , the revisions associated with the activity a that has the lowest tag in ordering \mathcal{O} are considered (line 4). After that, the algorithm computes the contribution ρ^{old} of all elements in \mathcal{P}_a^{rev} and \mathcal{R}_a^{rev} to the previous value of the start time of a , denoted by σ_a^{old} (line 5). The computations performed take into account the value of u_a at the last evaluation of the invariant (value u_a^{old}). Then, the algorithm computes the new contribution ρ of these same elements to the new value of the start time of a (lines 6–7), by using function `EARLIESTSTART` which recomputes terms $\sigma_{a,r}$ seen in Eqs. 5 and 6.

If the new contribution ρ is greater than or equal to the current value of σ_a , this means that temporal constraints have been strengthened since the last evaluation of the invariant, hence the new value of σ_a is ρ (line 8). Otherwise, if the old contribution ρ^{old} was a support for σ_a^{old} , then σ_a is recomputed from scratch (lines 9–10). Otherwise, there is a weakening of the temporal constraints associated with the elements revised, but these elements did not support the previous value of σ_a , hence σ_a is up-to-date.

If the new value obtained for σ_a is distinct from its old value σ_a^{old} , revisions are triggered for project successors of a (lines 11–15). Last, resource consumptions associated with a and which require a revision are handled, by recomputing the pending flows after the insertion of a for these resources (function `APPLY`), and by triggering new resource revisions over successor activities in case of a change in these pending flows (lines 16–21). Note that it is more likely to get at some point an equality between two lists of pending resource flows than between two full resource usage profiles.

Algorithm 1. Incremental revision procedure for the CBLS invariant introduced

```

1   $\{\mathcal{O}_r \mid r \in \mathcal{R}\} \leftarrow \text{UPDATEORDERS}()$ 
2   $(Q^{rev}, \{\mathcal{P}_a^{rev} \mid a \in \mathcal{A}\}, \{\mathcal{R}_a^{rev} \mid a \in \mathcal{A}\}) \leftarrow \text{INTREVISIONS}()$ 
3  while  $Q^{rev} \neq \emptyset$  do
4     $a \leftarrow \text{EXTRACTMIN}(Q^{rev})$ 
5     $\rho^{old} \leftarrow \max_{b \in \mathcal{P}_a^{rev}} (\sigma_b^{old} + p_b)$ ; if  $u_a^{old}$  then  $\rho^{old} \leftarrow \max(\rho^{old}, \max_{r \in \mathcal{R}_a^{rev}} \sigma_{a,r})$ 
6    if  $u_a$  then  $\sigma_{a,r} \leftarrow \text{EARLIESTSTART}(a, r, \Phi_{prev_{a,r},r}, m_{prev_{a,r},r})$  for each
        $r \in \mathcal{R}_a^{rev}$ 
7     $\rho \leftarrow \max_{b \in \mathcal{P}_a^{rev}} (\sigma_b + p_b)$ ; if  $u_a$  then  $\rho \leftarrow \max(\rho, \max_{r \in \mathcal{R}_a^{rev}} \sigma_{a,r})$ 
8    if  $\rho \geq \sigma_a$  then  $\sigma_a \leftarrow \rho$ 
9    else if  $\rho^{old} = \sigma_a^{old}$  then
10      $\sigma_a \leftarrow \max_{(b,a) \in \mathcal{P}} (\sigma_b + p_b)$ ; if  $u_a$  then  $\sigma_a \leftarrow \max(\sigma_a, \max_{r \in \mathcal{R}_a} \sigma_{a,r})$ 
11   if  $\sigma_a \neq \sigma_a^{old}$  then
12     foreach  $(a, b) \in \mathcal{P}$  do
13       if  $b \notin Q^{rev}$  then  $\text{ADD}(\langle b, tag_b \rangle, Q^{rev})$ 
14        $\text{ADD}(a, \mathcal{P}_b^{rev})$ 
15     if  $u_a$  then  $\mathcal{R}_a^{rev} \leftarrow \mathcal{R}_a$ 
16   if  $u_a$  then
17     foreach  $r \in \mathcal{R}_a^{rev}$  do
18        $\Phi^{old} \leftarrow \Phi_{a,r}$ ;  $\Phi_{a,r} \leftarrow \text{APPLY}(a, r, \sigma_a, \Phi_{prev_{a,r},r}, m_{prev_{a,r},r})$ 
19       if  $(next_{a,r} \neq t) \wedge ((\Phi_{a,r} \neq \Phi^{old}) \vee (\exists (\phi, z) \in \Phi_{a,r} \text{ s.t. } \sigma_z \neq \sigma_z^{old}))$ 
         then
20         if  $next_{a,r} \notin Q^{rev}$  then  $\text{ADD}(\langle next_{a,r}, tag_{next_{a,r}} \rangle, Q^{rev})$ 
21          $\text{ADD}(r, \mathcal{R}_{next_{a,r}}^{rev})$ 

```

Backward revisions, which compute distances from every activity to the sink node, are performed similarly. The main differences are that revisions are triggered in the direction of predecessors of activities, and that backward revisions do not update the precedence graph but just follow the graph produced by the forward revisions. After all forward and backward revisions, the old values which need to be recorded are updated following the changes made ($u_a^{old} \leftarrow u_a$, $\sigma_a^{old} \leftarrow \sigma_a$, $d_{a,t}^{old} \leftarrow d_{a,t}$).

As a last remark, note that the CBLS invariant obtained can deal with planning horizons containing many time-steps, since the complexity of the algorithm defined does not depend on the number of possible values of temporal distances. Further analyses would be required to get, as in [22], the complexity of the incremental evaluation function in terms of changes in the inputs and outputs of the invariant. Finally, it would be possible to define another version of the invariant taking as inputs directly individual priority orders \mathcal{O}_r over each resource, provided that these orders are compatible (invariant $(\{d_{s,a} \mid a \in \mathcal{A}\}, \{d_{a,t} \mid a \in \mathcal{A}\}) \leftarrow \text{lazySGS}'(\{\mathcal{O}_r \mid r \in \mathcal{R}\}, \{u_a \mid a \in \mathcal{A}\})$).

5 Differentiability of the Invariant

In addition to incremental reevaluation issues, one key feature of invariants in CBLS is their *differentiability* [35], which allows to quickly *estimate* the quality of local moves instead of fully evaluating them. To get such a differentiability, in addition to the set of *forward* pending flows $\Phi_{a,r}$ obtained after the insertion of an activity a , the invariant also maintains a list of *backward* pending flows $\Psi_{a,r} = [(\psi_1, z_1), \dots, (\psi_h, z_h)]$ obtained just before each activity a . This list contains pairs (ψ_i, z_i) such that activity z_i waits for a flow ψ_i which is released by activities placed before a in \mathcal{O}_r . See the bottom part of Fig. 2 for an illustration. On this figure, the backward pending flows for C and B on r are $\Psi_{C,r} = [(C, 1), (D, 2)]$ and $\Psi_{B,r} = [(B, 1), (C, 1), (D, 1)]$. Intuitively, $\Phi_{a,r}$ and $\Psi_{a,r}$ describe the resource usage frontiers obtained when cutting the flow-network respectively just after and just before the realization of a .

Let us now detail the techniques used for quickly estimating the quality of the possible reinsertions of an activity a in priority list \mathcal{O} . To do this efficiently, we first deactivate resource consumptions for a , to take into account only project precedences for a , and we evaluate the new schedule using the incremental evaluation function of the CBLS invariant. Then, from the current activity list \mathcal{O} , it is possible to compute all relevant insertion positions for a that do not lead to a precedence cycle. More precisely, we traverse the ancestors of a in the current precedence graph to find the greatest-tag ancestor activity b_1 such that there is a common resource used by a and b_1 ($\mathcal{R}_a \cap \mathcal{R}_{b_1} \neq \emptyset$). We also traverse the descendants of a to find the lowest-tag descendant activity b_2 such that there is a common resource used by a and b_2 . Reinserting a in priority list \mathcal{O} anywhere between b_1 and b_2 is consistent from the project precedences point of view. However, not all insertion positions deserve to be tested. It suffices to test the insertion of a just after activities c which share a common resource with a .

Let c be a relevant insertion position in \mathcal{O} located between b_1 and b_2 . To estimate the quality of the schedule obtained by inserting a just after c , we first simulate the insertion of a over resources $r \in \mathcal{R}_a$. For each resource $r \in \mathcal{R}_a$, we compute the activities $\mathit{prev}_{a,r}$ and $\mathit{next}_{a,r}$ that would immediately precede and follow a on r if a is inserted at the chosen position. We then simulate the merging of the forward pending flows in $\Phi_{\mathit{prev}_{a,r},r}$, followed by a , followed by the backward pending flows in $\Psi_{\mathit{next}_{a,r},r}$. To simulate this merging, we compute a set of additional resource flows and nodes that allow to get a consistent flow network (see Fig. 3). The quality of the resulting schedule is then estimated by *the contribution to the makespan associated with the part of the schedule modified by the insertion of a* , or in other words by the length of the longest source-to-sink path which traverses flow arcs added for merging the forward and backward pending flows (longest path through the gray area in Fig. 3).

In the case of disjunctive resources, the evaluation adopted generalizes classical formulas used in job shop scheduling and gives the exact value of the length of the longest source-to-sink path through a in the schedule that would be obtained [25]. In the case of cumulative resources, it gives a more global evaluation. To efficiently compute the length of such longest paths, we use the

(already known) distances $d_{s,b}$ from the source node to activities b contained in the forward pending flows, and the (already known) distances $d_{b,t}$ from activities b contained in the backward pending flows to the sink node.

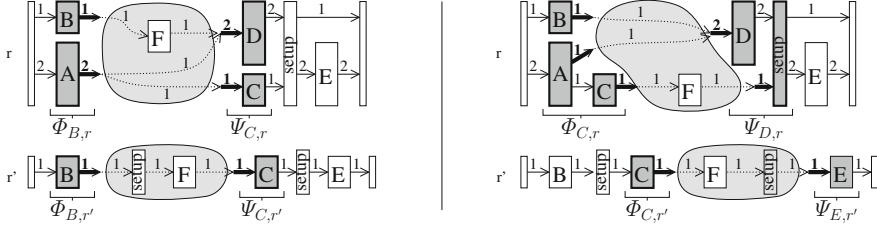


Fig. 3. Flow merging for estimating the quality of the insertion of activity F into the schedule given in Fig. 1: simulation of the insertion between B and C (left), and between C and D (right)

More formally, we first compute the earliest start time $\hat{\sigma}_a$ of a for the chosen insertion position. Next, for each resource $r \in \mathcal{R}_a$, we compute the new pending flows $\hat{\Phi}_{a,r}$ that would be obtained just after the insertion of a , and for each pending flow (ϕ, z) in $\hat{\Phi}_{a,r}$ we denote by $\hat{\sigma}_z$ the earliest start time obtained for z .

If no setup is required between a and $\hat{n}ext_{a,r}$ (case $m_{a,r} = m_{\hat{n}ext_{a,r},r}$), we compute a set of flows Γ which allow to merge forward pending flows in $\hat{\Phi}_{a,r}$ with backward pending flows in $\Psi_{\hat{n}ext_{a,r},r}$. To define Γ , flows (ψ, z) in $\Psi_{\hat{n}ext_{a,r},r}$ are ordered by increasing activity-tag, and for each of them we deliver from $\hat{\Phi}_{a,r}$ the resource flow ψ required by z , by using the same principles as in the schedule generation scheme of Sect. 3. For resource r , the quality of the insertion of a just after c in \mathcal{O} is estimated by:

$$length_{a,c,r} = \max_{x \rightarrow y \in \Gamma} (\hat{\sigma}_x + p_x + d_{y,t}) \quad (9)$$

Otherwise (case $m_{a,r} \neq m_{\hat{n}ext_{a,r},r}$), a setup operation is required just after a , as for the insertion of F on resource r' on the right part of Fig. 3. In this case, the estimation associated with the insertion of a just after c in \mathcal{O} is:

$$length_{a,c,r} = \max_{(\phi,z) \in \hat{\Phi}_{a,r}} (\hat{\sigma}_z + p_z) + \Delta_{r,m_{a,r},m_{\hat{n}ext_{a,r},r}} + \max_{(\psi,z) \in \Psi_{\hat{n}ext_{a,r},r}} d_{z,t} \quad (10)$$

Finally, the global estimation associated with the insertion of a just after c in \mathcal{O} is:

$$length_{a,c} = \max((\max_{r \in \mathcal{R}_a} length_{a,c,r}), (\hat{\sigma}_a + p_a + \max_{(a,b) \in \mathcal{P}} d_{b,t})) \quad (11)$$

This estimation takes into account not only the estimation of the quality of the insertion over each resource, but also the project successors of the activity

inserted. Compared to [4], which uses resource flows for guiding the way an activity should be inserted into an RCPSP schedule, testing the insertion at one position in our case has a lower complexity (complexity $O(Mm + P)$ with m the number of resources, M the maximum number of activities that might be performed in parallel over a resource, and P the maximum number of project predecessors and successors for an activity, instead of complexity $O(nm)$ with n the number of activities in \mathcal{A}).

6 Generic Local Search Neighborhoods

As said in the introduction, the other parts of our scheduling system are described with fewer details. Several neighborhoods are used when searching for good priority lists \mathcal{O} :

- REINSERTION: reinserts a given activity a at one of the best positions in the schedule according to the estimations provided by the differentiation techniques of the previous section. When choosing a particular insertion position for a , updates are automatically made in \mathcal{O} to guarantee that all ancestors (resp. descendants) of a in the precedence graph are still located on the left (resp. on the right) of a in \mathcal{O} .
- REINSERTIONDEEP: considers an activity a and tries to reinsert a at the best possible position. To enlarge the set of candidate insertion positions, it also moves project ancestors of a at their leftmost position in the order and project descendants of a at their rightmost position, without increasing the current makespan.
- OR-OPT: consists in searching for a better positioning for a block B of k successive activities. It is inspired by the *or-opt-k* neighborhood [5] used for traveling salesman problems (TSPs), and it can be useful for scheduling with setup times for SDST-RCPSP. The candidate reinsertion positions for B are efficiently explored by adapting the differentiation techniques seen previously.
- 2-OPT: considers a set of successive activities a_i, a_{i+1}, \dots, a_j and tries to realize them in the reverse order. Such a neighborhood is inspired by the *2-opt* moves [13] used for TSPs and can be useful for scheduling with setup times. It is efficiently explored by extending the differentiation techniques seen in the previous section.
- SWAP: considers two activities a, b such that swapping a and b does not create a precedence cycle. Using this neighborhood is more expensive since for evaluating a swap, we explicitly make it and compute its exact effect.
- REORDERBYDISTTOSINK: transforms the current priority list \mathcal{O} by ordering elements by decreasing distance to the sink node. For RCPSP, such a strategy is known as the *Forward-Backward Improvement* algorithm [33].

7 Search Strategy

The last part of our CBLs scheduling system allows search strategies to be defined. For space limitation reasons, we only give a brief overview of the strategy

used for the experiments. One of the main component of this strategy is Variable Neighborhood Search (VNS [27]), which successively considers the neighborhoods defined and applies these neighborhoods to critical activities until reaching a locally optimal solution.

Then, tabu search is used to escape local optima and plateaus [14]. More precisely, we maintain a tabu list which contains *forbidden makespan values*, and schedules that have a forbidden (real or estimated) makespan cannot be selected anymore during VNS. Once VNS converges to a new locally optimal (and non-tabu) solution, the makespan of this solution is added to the tabu list, the oldest tabu makespan is removed if some place is needed, and VNS is launched again. Tabu search ends when the makespan of the current solution has not been improved during a certain number of VNS applications.

Another technique is used for focusing search on the bottleneck of the problem: (a) if after tabu search the current makespan is not strictly better than the best makespan known, we randomly choose one non-critical activity a , deactivate its resource consumptions ($u_a = false$), clear the tabu list, and apply tabu search again on the problem containing one less activity; (b) otherwise, we select an activity whose resource consumptions have been deactivated, reactivate these consumptions, clear the tabu list, and apply tabu search again on the problem containing one more activity. Each time a full schedule with a better makespan is found, it is recorded as the best solution. We call the obtained metaheuristic *Tabu Search with Repair* (TSR).

Last, to avoid spending too much time on conflict resolution rather than on search over full schedules, each non-critical activity can be removed at most once during each call to TSR. When no activity is candidate for being removed from the schedule, we select $x\%$ of the activities ($x = 10\%$ in the experiments), randomly move each of them in priority list \mathcal{O} while preserving the satisfaction of all project precedences, and call TSR on the resulting schedule. From time to time, we also perform restarts. In the end, by combining tabu search and perturbation, we get a metaheuristic that we call *Iterated Tabu Search with Repair* (ITSR). It combines local and global search, and makes a trade-off between search intensification and diversification.

8 Experiments

Experiments are performed on clusters composed of 20 Intel Xeon 2.6 GHz processors with a shared memory of 65 GB of RAM. Each run used for solving one problem instance is performed on a single processor (no parallel solving). For each cluster, the 20 CBLS models together with the 20 search engines easily fit onto the available memory. The invariant defined is integrated into the *InCELL* CBLS library [31].

SDST-JSSP. Table 1 summarizes the results obtained on SDST-JSSP instances [9]. Each instance contains n jobs and m resources, leading to nm activities to schedule. For ITSR, the results presented are obtained based on

10 runs, each run having a time limit of 1 h. All neighborhoods defined in Sect. 6 are used except for the swap neighborhood, and the or-opt-k neighborhood is limited to $k = 2$. The length of the tabu list which contains makespan values is set to 15, and for tabu search the number of iterations allowed without improvements is set to 15 as well. ITSR is compared against techniques of the state-of-the-art: the branch-and-bound method defined in [3], the method defined in [6] which adapts to SDST-JSSP the shifting bottleneck procedure used for JSSP, two methods based on genetic algorithms hybridized with local search [16] and tabu search [17], and the CP approach defined in [18]. In the table, these methods are respectively referred to as AF08, BSV08, GVV08, GVV09, and GH10. The latter uses the same 1 h time limit as ITSR on a similar processor. As in [18], the bold font in the table is used for best makespan values, underlined values are used when the solver proves optimality, and values marked with a star denote new best upper bounds found by ITSR, which have also been verified using a separate checker. *Globally, ITSR finds 6 new upper bounds and for all other instances it always finds the best known upper bound.* Table 1 also shows the impact of the incremental computation techniques and of the differentiability of the *lazySGS* invariant. Compared to ITSR, the version which fully evaluates local moves instead of using differentiability (column ITSR-D in Table 1) typically performs 10 times less local moves, while the version which uses neither differentiability nor incremental evaluations (column ITSR-D-I in Table 1), typically performs 20 times less local moves. The solutions produced after one hour by these versions have higher makespans on average.

MJSSP. Figure 4a summarizes results obtained for the Multi-Capacity Job Shop Scheduling Problem (MJSSP). This problem involves cumulative resources and activities which all consume one resource unit. As in [26, 29], which introduces *iterative flattening* (a precedence constraint posting approach), the instances considered are derived from standard JSSP instances by duplicating jobs and increasing the capacity of resources accordingly. The advantage of doing this is that results over the initial JSSP instances provide lower and upper bounds for the MJSSP instances [28] (column NA96 in Fig. 4a). The instances selected are the 21 instances for which [26] provides new best upper bounds (column MV04 in Fig. 4a). In [26], these bounds are presented as the best ones found during the work on iterative flattening (no computation time specified). These bounds have been improved in [15], which uses randomized large neighborhood search (column GLN05). *Over the 21 instances, ITSR manages to find 12 best upper bounds compared to MV04, but only one compared to GLN05.* To get these results, the or-opt and 2-opt neighborhoods are deactivated because they are less relevant without setup times, and the length of the tabu list is set to 2. As for SDST-JSSP, Fig. 4a shows that deactivating the differentiability and the incremental computation capabilities of the *lazySGS* invariant degrades the average quality of the solutions produced.

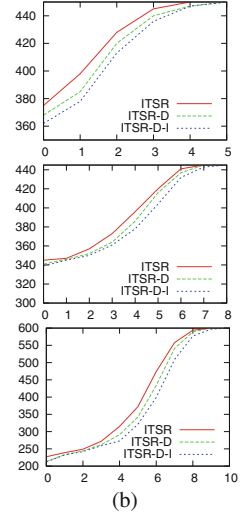
RCPSP. Figure 4b reports the results obtained on RCPSP instances j60, j90, and j120 (www.om-db.wi.tum.de/psplib), which respectively contain 60, 90, and 120

Table 1. Experiments for SDST-JSSP: comparison with the state-of-the-art for makespan minimization (best & mean values over 10 runs, with a 1 h time limit for each run)

Instance	#jobs x #res	AF08	BSV08	GVV08		GVV09		GH10		ITSR		ITSR-D		ITSR-D-I	
		Best	Best	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg	Best	Avg
t2-ps06	15 x 5	1009	1018	1026	1026			1009	1009.0	1009	1009.2	1009	1009.7	1009	1010.3
t2-ps07	15 x 5	970	1003	970	971			970	970.0	970	970.0	970	970.0	970	970.0
t2-ps08	15 x 5	963	975	963	966			963	963.0	963	963.0	963	963.0	963	963.0
t2-ps09	15 x 5	1061	1060	1060	1060			1060	1060.0	1060	1060.0	1060	1060.0	1060	1060.0
t2-ps10	15 x 5	1018	1018	1018	1018			1018	1018.0	1018	1018.0	1018	1018.0	1018	1018.0
t2-ps11	20 x 5	1494	1470	1438	1439	1438	1441	1443	1463.8	1437*	1437.6	1438	1442.1	1438	1441.9
t2-ps12	20 x 5	1381	1305	1269	1291	1269	1277	1269	1322.2	1269	1269.0	1269	1270.8	1269	1271.1
t2-ps13	20 x 5	1457	1439	1406	1415	1415	1416	1415	1428.8	1404*	1406.4	1406	1414.1	1406	1414.5
t2-ps14	20 x 5	1483	1485	1452	1489	1452	1489	1452	1470.5	1452	1452.0	1452	1452.0	1452	1452.0
t2-ps15	20 x 5	1661	1527	1485	1502	1485	1496	1486	1495.8	1479*	1485.9	1485	1492.9	1485	1496.0
t2-pss06	15 x 5		1126					1114	1114.0	1114	1117.7	1114	1120.6	1114	1122.0
t2-pss07	15 x 5		1075					1070	1070.0	1070	1070.0	1070	1070.0	1070	1070.0
t2-pss08	15 x 5		1087					1072	1073.0	1072	1073.1	1072	1073.6	1072	1074.2
t2-pss09	15 x 5		1181					1161	1161.0	1161	1161.0	1161	1161.0	1161	1161.0
t2-pss10	15 x 5		1121					1118	1118.0	1118	1118.0	1118	1118.0	1118	1118.0
t2-pss11	20 x 5		1442					1412	1425.9	1409*	1412.4	1412	1417.6	1414	1420.6
t2-pss12	20 x 5		1290			1258	1266	1269	1287.6	1257*	1260.5	1258	1266.7	1260	1269.3
t2-pss13	20 x 5		1398			1361	1379	1365	1388.0	1361	1364.7	1361	1371.9	1367	1373.9
t2-pss14	20 x 5		1453					1452	1453.0	1452	1452.0	1452	1452.0	1452	1452.0
t2-pss15	20 x 5		1435					1417	1427.4	1410*	1411.1	1410*	1421.1	1417	1421.8

Instance	#jobs x #res	NA96	MV04	GLN05	ITSR		ITSR-D		ITSR-D-I	
		lb/ub	Best	Best	Best	Avg	Best	Avg	Best	Avg
la04-2	20 x 5	572/590	577	576	576	576.0	584	587.3	582	588.7
la04-3	30 x 5	570/590	584	573	577	579.7	594	603.7	596	605.0
la16-2	20 x 10	888/935	929	915	933	938.4	932	936.6	930	937.6
la16-3	30 x 10	717/935	927	918	939	947.1	941	951.9	945	955.3
la17-2	20 x 10	750/765	756	755	756	757.8	760	763.3	760	763.4
la17-3	30 x 10	646/765	761	755	763	765.7	773	777.7	772	779.0
la18-2	20 x 10	783/844	818	811	814	818.1	828	833.7	832	836.6
la18-3	30 x 10	663/844	813	808	818	825.4	854	861.4	840	858.4
la19-2	20 x 10	730/840	803	795	792*	799.9	818	826.2	822	827.5
la19-3	30 x 10	617/840	801	787	799	802.5	841	848.5	842	852.3
la20-2	20 x 10	829/902	864	859	859	867.2	872	878.9	871	878.6
la20-3	30 x 10	756/902	863	854	862	871.5	879	894.5	888	901.4
la24-2	30 x 10	704/935	932	903	911	917.9	966	976.6	963	975.3
la24-3	45 x 10	704/935	929	898	917	923.2	1000	1010.8	1000	1014.4
la25-3	45 x 10	723/977	965	945	977	983.2	1027	1037.6	1035	1043.9
la38-2	30 x 15	943/1196	1185	1175	1180	1188.6	1249	1260.8	1223	1261.5
la38-3	45 x 15	943/1196	1195	1168	1197	1204.9	1286	1304.6	1297	1311.3
ft10-2	20 x 10	835/930	913	891	906	908.9	921	931.3	922	934.2
ft10-3	30 x 10	655/930	912	879	915	921.8	946	957.7	955	962.3
ft20-2	40 x 5	1165/1165	1186	1182	1172	1175.0	1195	1215.6	1213	1226.0
ft20-3	60 x 5	387/1165	1205	1179	1182	1190.1	1239	1254.8	1220	1255.7

(a)



(b)

Fig. 4. Results on cumulative resources: (a) results on MJSSP instances (best & mean values over 10 runs with a 1 h time limit per run); (b) results on RCPSP instances j60 (top), j90 (middle), and j120 (bottom); representation of the mean number of instances (y-axis) for which the deviation percentage with regards to the best known upper bound is less than the value on the x-axis (average results over 5 runs with a 30 min time limit per run)

activities to schedule. This time, the length of the tabu list is set to 4 as well as the maximum number of iterations without improvement for tabu search. Figure 4b gives the distribution of the relative distance between the average makespan found by ITSR after 30 minutes and the best makespan reported in the PSPLIB. For example, for j60, the graph expresses that among the 450 instances of j60 considered, the distance to the best solution known is 0% for approximately 375 instances, it is $\leq 1\%$ for approximately 400 instances, and so on. Globally, for j60 (resp. j90 and j120), ITSR finds schedules which are within 5% (resp. 7% and 9%) from the best upper bounds. Figure 4b shows the degradation of the search efficiency when deactivating the differentiation techniques (ITSR-D), and then both differentiability and incremental computations (ITSR-D-I).

9 Conclusion and Future Work

In this paper, we gave a global view of CBLs techniques adapted to SDST-RCPSp, with a focus on a new CBLs invariant capable of dealing with a large class of renewable resources. With regards to existing CBLs systems, this invariant is rather large in the sense that it does not decompose the management of renewable resources into several smaller invariants which are then dynamically ordered in the graph of invariants. We believe that this allows us to get more powerful differentiation techniques, however additional experiments should be performed to confirm this point. For future work, it would be useful to extend the invariant introduced to take into account resources with time-varying availability profiles, activities with time-varying resource consumptions, or maximum distance constraints between activities, and to get more insight into the contribution of each component of the search strategy defined.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. *Math. Comput. Modell.* **17**(7), 57–73 (1993)
2. Allahverdi, A., Ng, C., Cheng, T., Kovalyov, M.Y.: A survey of scheduling problems with setup times or costs. *Eur. J. Oper. Res.* **187**(3), 985–1032 (2008)
3. Artigues, C., Feillet, D.: A branch and bound method for the job-shop problem with sequence-dependent setup times. *Ann. Oper. Res.* **159**(1), 135–159 (2008)
4. Artigues, C., Michelon, P., Reusser, S.: Insertion techniques for static and dynamic resource constrained project scheduling. *Eur. J. Oper. Res.* **149**(2), 249–267 (2003)
5. Babin, G., Deneault, S., Laporte, G.: Improvements to the Or-opt heuristic for the symmetric traveling salesman problem. *J. Oper. Res. Soc.* **58**, 402–407 (2007)
6. Balas, E., Simonetti, N., Vazacopoulos, A.: Job shop scheduling with setup times, deadlines and precedence constraints. *J. Sched.* **11**(4), 253–262 (2008)
7. Bender, M.A., Cole, R., Demaine, E.D., Farach-Colton, M., Zito, J.: Two simplified algorithms for maintaining order in a list. In: Möhring, R., Raman, R. (eds.) *ESA 2002*. LNCS, vol. 2461, pp. 152–164. Springer, Heidelberg (2002). doi:[10.1007/3-540-45749-6_17](https://doi.org/10.1007/3-540-45749-6_17)

8. Brucker, P., Drexl, A., Möring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: notation, classification, models, and methods. *Eur. J. Oper. Res.* **112**(1), 3–41 (1999)
9. Brucker, P., Thiele, O.: A branch and bound method for the general-shop problem with sequence dependent setup-times. *Oper. Res. Spekt.* **18**(3), 145–161 (1996)
10. Carlier, J.: The one machine sequencing problem. *Eur. J. Oper. Res.* **11**, 42–47 (1982)
11. Carlier, J., Pinson, E.: Adjustment of heads and tails for the job-shop problem. *Eur. J. Oper. Res.* **78**, 146–161 (1994)
12. Croce, F.D.: Generalized pairwise interchanges and machine scheduling. *Eur. J. Oper. Res.* **83**(2), 310–319 (1995)
13. Croes, G.A.: A method for solving traveling salesman problems. *Oper. Res.* **6**, 791–812 (1958)
14. Glover, F., Laguna, M.: Tabu search. In: *Modern Heuristic Techniques for Combinatorial Problems*, pp. 70–141. Blackwell Scientific Publishing (1993)
15. Godard, D., Laborie, P., Nuijten, W.: Randomized large neighborhood search for cumulative scheduling. In: *Proceedings of the 15th International Conference on Automated Planning and Scheduling (ICAPS 2005)*, pp. 81–89 (2005)
16. González, M.A., Vela, C.R., Varela, R.: A new hybrid genetic algorithm for the job shop scheduling problem with setup times. In: *Proceedings of the 18th International Conference on Automated Planning and Scheduling (ICAPS 2008)*, pp. 116–123 (2008)
17. González, M.A., Vela, C.R., Varela, R.: Genetic algorithm combined with tabu search for the job shop scheduling problem with setup times. In: Mira, J., Ferrández, J.M., Álvarez, J.R., Paz, F., Toledo, F.J. (eds.) *IWINAC 2009. LNCS*, vol. 5601, pp. 265–274. Springer, Heidelberg (2009). doi:[10.1007/978-3-642-02264-7_28](https://doi.org/10.1007/978-3-642-02264-7_28)
18. Grimes, D., Hebrard, E.: Job shop scheduling with setup times and maximal time-lags: a simple constraint programming approach. In: Lodi, A., Milano, M., Toth, P. (eds.) *CPAIOR 2010. LNCS*, vol. 6140, pp. 147–161. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-13520-0_19](https://doi.org/10.1007/978-3-642-13520-0_19)
19. Grimes, D., Hebrard, E.: Solving variants of the job shop scheduling problem through conflict-directed search. *INFORMS J. Comput.* **27**(2), 268–284 (2015)
20. Hartmann, S.: A competitive genetic algorithm for resource-constrained project scheduling. *Naval Res. Logist.* **45**, 733–750 (1997)
21. Ilog: IBM ILOG CPLEX and CpOptimizer. <http://www-03.ibm.com/software/products/>
22. Katriel, I., Michel, L., Van Hentenryck, P.: Maintaining longest paths incrementally. *Constraints* **10**(2), 159–183 (2005)
23. Kolisch, R., Hartmann, S.: Heuristic algorithms for solving the resource-constrained project scheduling problem: classification and computational analysis. In: *Handbook on Recent Advances in Project Scheduling*, pp. 147–178. Kluwer Academic Publishers, Dordrecht (1999)
24. Kolisch, R., Hartmann, S.: Experimental investigation of heuristics for resource-constrained project scheduling: an update. *Eur. J. Oper. Res.* **174**(1), 23–37 (2006)
25. Mastrolilli, M., Gambardella, L.: Effective neighborhood functions for the flexible job shop problem. *J. Sched.* **3**(1), 3–20 (2000)
26. Michel, L., Van Hentenryck, P.: Iterative relaxations for iterative flattening in cumulative scheduling. In: *Proceedings of the 14th International Conference on Automated Planning and Scheduling (ICAPS 2004)*, pp. 200–208 (2004)

27. Mladenović, N., Hansen, P.: Variable neighborhood search. *Comput. Oper. Res.* **24**(11), 1097–1100 (1997)
28. Nuijten, W.P.M., Aarts, E.H.L.: A computational study of constraint satisfaction for multiple capacitated job shop scheduling. *Eur. J. Oper. Res.* **90**(2), 269–284 (1996)
29. Oddi, A., Cesta, A., Policella, N., Smith, S.F.: Iterative flattening search for resource constrained scheduling. *J. Intell. Manuf.* **21**, 17–30 (2010)
30. Pinedo, M.: *Scheduling: Theory, Algorithms, and Systems*. Springer, New York (2012). doi:[10.1007/978-1-4614-2361-4](https://doi.org/10.1007/978-1-4614-2361-4)
31. Pralet, C., Verfaillie, G.: Dynamic online planning and scheduling using a static invariant-based evaluation model. In: *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS 2013)*, pp. 171–179 (2013)
32. Schutt, A., Feydy, T., Stuckey, P.J., Wallace, M.G.: Solving the resource constrained project scheduling problem with generalized precedences by lazy clause generation. *CoRR* abs/1009.0347 (2010)
33. Valls, V., Ballestín, F., Quintanilla, S.: Justification and RCPSP: a technique that pays. *Eur. J. Oper. Res.* **165**(2), 375–386 (2005)
34. Van Cauwelaert, S., Dejemeppe, C., Monette, J.N., Schaus, P.: Efficient filtering for the unary resource with family-based transition times. In: *Proceedings of the 22nd International Conference on Principles and Practice of Constraint Programming (CP 2016)*, pp. 520–535 (2016)
35. Van Hentenryck, P., Michel, L.: *Constraint-Based Local Search*. MIT Press, Cambridge (2005)
36. Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in $O(k n \log n)$. In: *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP 2009)*, pp. 802–816 (2009)
37. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Achterberg, T., Beck, J.C. (eds.) *CPAIOR 2011*. LNCS, vol. 6697, pp. 230–245. Springer, Heidelberg (2011). doi:[10.1007/978-3-642-21311-3_22](https://doi.org/10.1007/978-3-642-21311-3_22)