An Incomplete Constraint-Based System for Scheduling with Renewable Resources

Abstract. In this paper, we introduce a new generic 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 so as to solve resource usage conflicts. In this context, we define a new 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 generic local search neighborhoods and search strategies mixing iterated tabu search and inconsistency repair. The experiments performed lead to new best upper bounds on several scheduling benchmarks.

1 Introduction

In scheduling, renewable resources correspond to 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, for which a given duration can be required between successive activities 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 [32]), Job Shop Scheduling Problems with Sequence-Dependent Setup Times (SDST-JSSPs[2]), and Resource Constrained Project Scheduling Problems (RCPSPs [9]).

In the constraint programming community, the requirement to efficiently deal with renewable resources led to the definition of specific global constraints like the *disjunctive* [11] and the *cumulative* constraints [1], together with efficient propagation mechanisms such as edge-finding [12, 35] or timetable edge-finding [36]. Following these developments, constraint programming is nowadays one of the best systematic approach for solving scheduling problems with renewable resources [21, 33, 18].

In parallel, several incomplete search techniques were developed in the scheduling community to quickly produce good-quality solutions on large instances. One of them is *list scheduling*. It manipulates a priority list between activities, and at each step it 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 [13] or genetic algorithms applying crossover operations between priority lists [19]. *In this paper, we propose a combination between list scheduling and constraint programming. More specifically, we combine list scheduling*

with constraint-based local search [20], 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 containing three layers: (1) a *constraint-based incremental evaluation layer*, used for estimating very quickly the impact of local modifications of a given priority list, (2) a *neighborhood layer*, which consists of a catalog of generic neighborhoods usable for updating priority lists, and (3) a *search strategy layer*, which implements tabu search and conflict-repair techniques to escape local minima and plateaus.

The paper is organized as follows. Sect. 2 and 3 present the new scheduling framework considered and a lazy schedule generation scheme for this framework. Sect. 4, 5, 6 detail the three layers mentioned above. Sect. 7 shows the efficiency of the approach on standard benchmarks. Sect. 8 discusses extensions of the techniques proposed.

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

For covering both 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).

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 $m_{0,r} \in \mathcal{M}_r$. Moreover, 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 for making 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 Eq. 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).

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

$$\forall r \in \mathcal{R}, \forall a \in \mathcal{A}_r,\ \sum_{b \in \mathcal{A}_r \mid \sigma_b \leq \sigma_a < \sigma_b + p_b} q_{b,r} \leq K_r \tag{2}$$

$$\forall r \in \mathcal{R}, \forall a, b \in \mathcal{A}_r \text{ s.t. } m_{a,r} \neq m_{b,r}, \tag{3}$$
$$(\sigma_a \geq \sigma_b + p_b + \Delta_{r,m_{b,r},m_{a,r}}) \vee (\sigma_b \geq \sigma_a + p_a + \Delta_{r,m_{a,r},m_{b,r}})$$

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

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

The SDST-RCPSP framework brings a unifying view of several renewable resources types. The more complex part of this view (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.

Precedence graphs for SDST-RCPSP Another convenient way of representing a solution schedule is the standard concept of *precedence graph*. Such a graph contains nodes which are labeled by activities and arcs $a \rightarrow b$ which are labeled by the duration of a. These arcs correspond to precedence constraints "b can start only once a is finished". See Fig. 1 (upper part) for an illustration. 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 precedence constraint posted to prevent resources from being overused, also called *resource precedences* (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 built also contains setup activities $setup_{a,r}$ to be used for changing the current running mode m of a resource r just before realizing an activity a if needed. The duration of $setup_{a,r}$ is then given by setup duration $\Delta_{r,m,m_{a,r}}$. In Fig. 1, a setup activity is used for resource r just before realizing activity E.

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 $d_{D,t} = 9$). Then, the earliest start time of a is given by $est_a = d_{s,a}$, the makespan mk of the schedule is given by $mk = d_{s,t}$, and the latest start time of a is given by $lst_a = mk - d_{a,t}$. An activity is said to be *critical* iff $est_a = lst_a$, and its temporal flexibility is given by $mk - (d_{s,a} + d_{a,t})$.

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 $\phi^r_{a \rightarrow b}$ which are 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) for an illustration, where for instance activity D receives one resource unit from both A and B. For the flow-network to be consistent, 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), in order to guarantee that every resource has a unique running mode at any time. Last, all resource flows $\phi^r_{a \rightarrow b}$ 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 $\phi^r_{a \rightarrow b}$ between two activities $a, b \in \mathcal{A}_r$ such that $m_{a,r} \neq m_{b,r}$.

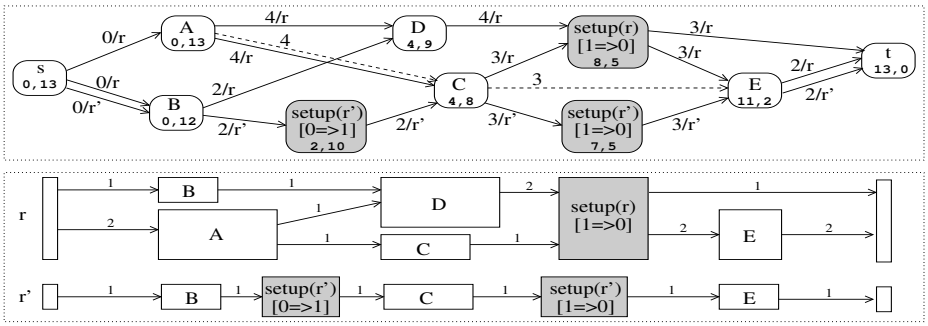


Fig. 1. Precedence graph (top) and flow-network (bottom) for a SDST-RCPSP where $\mathcal{R}$=$\{r, r'\}$, $\mathcal{A}$=$\{A, B, C, D, E\}$, $\mathcal{P}$=$\{(A, C), (C, E)\}$, p_A=p_D=4, p_B=p_E=2, p_C=3, $\mathcal{R}_A = \mathcal{R}_D = \{r\}$, $\mathcal{R}_B = \mathcal{R}_C = \mathcal{R}_E = \{r,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

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 $\mathcal{O} = [a_1, \ldots, a_n]$ 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 $\mathcal{O}$. In the following, we denote by $\mathcal{O}_r$ the sequence of successive activities which consume r. On the example of Fig. 1, $\mathcal{O}_r = [A, B, C, D, E]$ 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_i in priority list $\mathcal{O}$ and it tries to insert this activity into the current schedule at the earliest possible time t such that starting a_i at t is feasible in terms of project precedences and resource consumptions, and such that activities already placed in the schedule are not delayed. During the process, the generation scheme computes, for each activity a, its earliest start time σ_a and 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

$$\Phi_{a,r} = [(\phi_1, a_1), \ldots, (\phi_h, a_h)]$$

which represents that for every $i \in [1..h]$, a resource amount ϕ_i is released at the end of activity a_i and is available for future activities. In Fig. 1, the set of pending flows over resource r after the insertion of activity A would be $\Phi_{A,r} = [(1, s), (2, A)]$, meaning that one resource unit is still available from the source node while two resource units are released at the end of A. After the insertions of B, C, D, E, the pending flows would be

$\Phi_{B,r} = [(1, B), (2, A)]$, $\Phi_{C,r} = [(1, B), (1, A), (1, C)]$, $\Phi_{D,r} = [(1, C), (2, D)]$, and $\Phi_{E,r} = [(1, setup_{r,1,0}), (2, E)]$. *In the following, we assume that the pending flows are ordered by increasing release time, that is* $\sigma_{a_1} + p_{a_1} \leq \ldots \leq \sigma_{a_h} + p_{a_h}$.

We now come back to the generation process. For every resource r, the initial list of pending flows after source activity s is $\Phi_{s,r} = [(K_r, s)]$, which means that 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 $\Phi_{prev_{a,r},r} = [(\phi_1, a_1), \ldots, (\phi_h, a_h)]$ be the pending flows obtained after the realization 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 required 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).

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

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

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:

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

Activity a is then introduced in the precedence graph. To do this, 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 $a_i \rightarrow setup_{a,r}$ with flow ϕ_i for each pending flow (ϕ_i, a_i) in $\Phi_{prev_{a,r},r}$, and one arc $setup_{a,r} \rightarrow a$ pointing to the activity introduced, with flow $q_{a,r}$. 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 get the pending flow (ϕ_i, a_i) in $\Phi_{prev_{a,r},r}$ which is released at the latest time but before σ_a. In the precedence graph, we add arc $a_i \rightarrow a$ to represent that resource units released by a_i are transmitted to a. The number of resource units transferred is $\min(\phi_i, q_{a,r})$. If this does not suffice to cover the total amount of resource required by a (case $\phi_i < q_{a,r}$), we continue with pending flows $(\phi_{i-1}, a_{i-1}), (\phi_{i-2}, a_{i-2}) \ldots$ until resource consumption $q_{a,r}$ is fully covered. The new pending flows after the introduction of a are obtained from these updates. Note that systematically selecting the pending flow that has the latest release time allows to reduce resource idle periods. In the example of Fig. 1, activity C, which can only start after the end of A due to precedence constraint $(A, C) \in \mathcal{P}$, consumes resource units released by A but not resource units released by B, whereas the latter are available earlier.

These operations are realized until all activities in priority list $\mathcal{O}$ have been considered. The generation scheme defined is *lazy* because for inserting an activity in the schedule, 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 which induces an optimal schedule. In another direction, all schedules produced are *semi-active*, meaning that for every activity a, there is no date $t < \sigma_a$ such that all start times $t, t+1, \ldots, \sigma_a$ 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 onto 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. In particular, for a disjunctive resource, our lazy serial 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 quality solutions, a possible 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 were constraint programming comes into play, since we use Constraint-Based Local Search (CBLS [20]) 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 so-called *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 $x \leftarrow sum(i \in [1..N])\, y_i$. 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 allows $O(1)$ querying time (ordering query between two activities), $O(1)$ time for activity deletion, and $O(1)$ amortized time for activity insertion. For each element a in the list, it maintains the immediate predecessor and the immediate successor of a. It also assigns to a an integer tag_a such that if a is located before b in the list, then $tag_a < tag_b$. The tags used are not necessarily consecutive integers, instead they are spread over $[0..2^{31}-1]$. When inserting an element c between two successive elements a and b, the tag of c is given by $\frac{tag_a + tag_b}{2}$ if $tag_a + 1 < tag_b$, and otherwise operations are used to retag some elements in the order to make some place 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 deactivated. 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 is useful to get at some steps a better view of the promising insertion positions for an activity (see Sect. 5).

Invariant outputs The invariant builds the precedence graph and returns, for each activity a, the distance from the source node to a ($d_{s,a}$, also denoted by 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 immediate predecessor $prev_{a,r}$ and the immediate 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 $\Phi_{a,r}$ obtained after the insertion of a in the precedence graph. For each resource and each activity, the space complexity required to maintain these pending flows is $O(M)$, with M the maximum number of resource consumptions that might be performed in parallel on a resource.

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, with on one hand moves of some activities in the priority list, and on the other hand 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 ($\mathcal{O}_r$). To do this, all activities for which changes occurred are removed from sequences $\mathcal{O}_r$. All activities whose consumptions are still activated ($u_a = true$) are then sorted by increasing tag and reintroduced in sequences $\mathcal{O}_r$ 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}, which contains activities for which some revisions must be made; to ensure that revisions are realized in the right order (in a topological order with relation to the precedence graph), activities are ordered in Q^{rev} in the same order as their tag in $\mathcal{O}$;
- for every activity a, a *precedence revision set* $\mathcal{P}_a^{rev}$, which contains activities b such that (b, a) is a precedence in $\mathcal{P}$ and some updates made on b might have an impact on the earliest start time of a;
- for every activity a, a *resource revision set* $\mathcal{R}_a^{rev}$, which contains a subset of resources $r \in \mathcal{R}_a$ such that the impact of the consumption of resource r by activity 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

Algorithm 1: Incremental revision procedure for the CBLS invariant introduced

```
1  {O_r | r ∈ R} ← UPDATEORDERS()
2  (Q^rev, {P_a^rev | a ∈ A}, {R_a^rev | a ∈ A}) ← INITREVISIONS()
3  while Q^rev ≠ ∅ do
4      a ← EXTRACTMIN(Q^rev)
5      ρ ← max_{b∈P_a^rev}(σ_b^old + p_b); if u_a^old then ρ ← max(ρ, max_{r∈R_a^rev} σ_{a,r})
6      if u_a then σ_{a,r} ← EARLIESTSTART(a, r, Φ_{prev_{a,r},r}, m_{prev_{a,r},r}) for each r ∈ R_a^rev
7      ρ' ← max_{b∈P_a^rev}(σ_b + p_b); if u_a then ρ' ← max(ρ', max_{r∈R_a^rev} σ_{a,r})
8      if ρ' ≥ σ_a then σ_a ← ρ'
9      else if ρ = σ_a^old then
10         σ_a ← max_{(b,a)∈P}(σ_b + p_b); if u_a then σ_a ← max(σ_a, max_{r∈R_a} σ_{a,r})
11     if σ_a ≠ σ_a^old then
12         foreach (a, b) ∈ P do
13             if b ∉ Q^rev then ADD(⟨b, tag_b⟩, Q^rev)
14             ADD(a, P_b^rev)
15         if u_a then R_a^rev ← R_a
16     if u_a then
17         foreach r ∈ R_a^rev do
18             Φ^old ← Φ_{a,r}; Φ_{a,r} ← APPLY(a, r, σ_a, Φ_{prev_{a,r},r}, m_{prev_{a,r},r})
19             if (next_{a,r} ≠ t) ∧ ((Φ_{a,r} ≠ Φ^old) ∨ (∃(φ, b) ∈ Φ_{a,r} s.t. σ_b ≠ σ_b^old)) then
20                 if next_{a,r} ∉ Q^rev then ADD(⟨next_{a,r}, tag_{next_{a,r}}⟩, Q^rev)
21                 ADD(r, R^rev_{next_{a,r}})
```

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 that has the lowest tag in ordering $\mathcal{O}$ are considered (line 4). After that, the algorithm computes the contribution ρ 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 Eq. 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 be 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 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.

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, old values 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 defined 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. Moreover, the invariant can be part of a larger CBLS model containing other expressions.

5 Generic Local Search Neighborhoods

We now present neighborhoods which are used in our CBLS system for performing local search over priority lists. We detail some techniques used for quickly estimating the quality of neighbor priority lists. These techniques make the CBLS invariant defined in the previous section a so-called *differentiable invariant* [20].

REINSERTION *neighborhood* One of the most common neighborhood used in scheduling consists in reinserting a given activity a at another place in the schedule. To do this efficiently, we first deactivate resource consumptions for activity a and we evaluate the new schedule using the CBLS invariant. In this new schedule, only project precedences associated with a are taken into account. 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 in order 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 in order 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, that is it will not create precedence cycles. However, not all insertion positions deserve to be tested. It suffices to test the insertion of a just after the activities c which share a common resource with a.

Let c be a relevant insertion position located between b_1 and b_2. To get an estimation of the quality of the schedule obtained if a is inserted just after c, we first simulate the insertion of a over resources $r \in \mathcal{R}_a$. To do this, we compute the new earliest start time of a, denoted by $\hat{\sigma}_a$, and the new pending flows $\hat{\Phi}_{a,r}$ that would be obtained after the insertion of a. We then evaluate the quality of the resulting schedule by estimating *the contribution to the makespan associated with the part of the schedule modified by the insertion of* a. More precisely, we estimate the length of the longest path through one of the activities involved in the lists of pending flows $\hat{\Phi}_{a,r}$ obtained after the insertion of a. In the case of disjunctive resources, as the pending flows obtained after inserting a are

always reduced to $\hat{\Phi}_{a,r} = [(1, a)]$, the definition adopted coincides with the length of the longest path through a in the precedence graph. In the case of cumulative resources, it gives a more global evaluation.

More formally, for every resource $r \in \mathcal{R}_a$, let $\hat{prev}_{a,r}$ and $\hat{next}_{a,r}$ be the activities that would immediately precede and follow a on resource r if a is inserted at the chosen position. For every activity z, in addition to the set of *forward* pending flows $\Phi_{z,r}$ obtained after the insertion of z, it is possible to incrementally maintain a set of *backward* pending flows $\Psi_{z,r}$. The latter corresponds to a sequence $[(\psi_1, a_1), \ldots, (\psi_h, a_h)]$ such that for every $i \in [1..h]$, activity a_i waits for a flow ψ_i which is released by activities placed strictly before z in $\mathcal{O}_r$. On Fig. 1, the backward pending flows for C and B on r are respectively given by $\Psi_{C,r} = [(C, 1), (D, 2)]$ and $\Psi_{B,r} = [(B, 1), (C, 1), (D, 1)]$. Intuitively, $\Phi_{z,r}$ and $\Psi_{z,r}$ describe the resource usage frontiers obtained when cutting the flow-network respectively just after and just before the realization of z.

Coming back to insertion evaluation issues, if no setup is required between a and $\hat{next}_{a,r}$, we simulate the merging of frontiers $\hat{\Phi}_{a,r}$ and $\Psi_{\hat{next}_{a,r},r}$ to get an estimation of the length of the longest path through an activity involved in $\hat{\Phi}_{a,r}$. To do this, we simulate the creation of a new set of flows $\hat{F}$ coming out of the pending flows in $\hat{\Phi}_{a,r}$ and going into the pending flows in $\Psi_{\hat{next}_{a,r},r}$. To define the flows in $\hat{F}$, we order the flows (ψ, z) in $\Psi_{\hat{next}_{a,r},r}$ 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. By realizing these operations, a new set of flows $\hat{F}$ is obtained. By defining $\hat{d_{s,x}}$ as "$\hat{d_{s,x}} = \hat{\sigma}_a$ if $x = a$ and $\hat{d_{s,x}} = d_{s,x}$ otherwise", the evaluation associated with resource r for the insertion of a just after c is:

$$evalInsert_{a,c,r} = \max_{x \xrightarrow{\phi} y \in \hat{F}} (\hat{d_{s,x}} + p_x + d_{y,t}) \tag{8}$$

On the other hand, if the activity $\hat{next}_{a,r}$ that would follows a on r does not use the same resource mode as a, then a setup operation is required between a and $\hat{next}_{a,r}$. The estimation associated with the insertion of a just after c in $\mathcal{O}$ is then:

$$evalInsert_{a,c,r} = \max_{(\phi,z) \in \hat{\Phi}_{a,r}} (\hat{d_{s,z}} + p_z) + \Delta_{r,m_{a,r},m_{\hat{next}_{a,r},r}} + \max_{(\psi,z) \in \Psi_{\hat{next}_{a,r},r}} d_{z,t} \tag{9}$$

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

$$evalInsert_{a,c} = \max(\max_{r \in \mathcal{R}_a} evalInsert_{a,c,r}, \hat{\sigma}_a + p_a + \max_{(a,b) \in \mathcal{P}} d_{b,t}) \tag{10}$$

When choosing a particular insertion position for an activity a, we also automatically realize updates 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 $\mathcal{O}$.

Note that in the case of disjunctive resources, the evaluation adopted generalizes classical formulas used in job shop scheduling and the evaluation gives the exact value of the length of the longest source-to-sink path through a in the schedule that would be obtained [27]. Compared to [4], which also uses resource flows for guiding the way an activity should be inserted into an RCPSP schedule, testing one insertion in our case has a lower complexity (complexity $O(nMm)$ with n the number of activities m the number of resources, and M the maximum number of activities that might be performed in parallel over a resource, instead of $O(n^2m)$).

Other neighborhoods Several other neighborhoods are used in our scheduling system. We describe them with fewer details to leave some place for the other parts.

- OR-OPT. This neighborhood is inspired by the *or-opt-k* neighborhood [5] used for traveling salesman problems (TSPs). For a TSP, or-opt-k searches for a better positioning for a block of k cities successively traversed. For SDST-RCPSP, such a neighborhood can be useful for optimizing setup times. In the implementation of this neighborhood, we use techniques for considering only or-opt moves which are *consistent* with regards to the project precedences and *relevant* with regards to resource consumptions. To evaluate the reinsertion of a block $B = [a_1, a_2, \ldots, a_k]$ of k successive activities, we apply activities in B one by one to compute the pending flows $\hat{\Phi}_{r,B}$ that would be obtained after the insertion of B. We then use the same flow merging techniques as before to evaluate the quality of a move.
- 2-OPT. This neighborhood is inspired by the *2-opt* moves [14] used for TSPs. In a TSP, 2-opt considers a subset of successive cities $c_i, c_{i+1}, \ldots, c_j$ and tries to traverse these cities in the reverse order $c_j, \ldots, c_{i+1}, c_i$. Such a neighborhood can be useful for scheduling with setup times. Again, we use techniques to find successive activities whose ordering can be safely reversed without creating precedence cycles. For evaluating a 2-opt move, the activities reversed are applied one by one, and we use the same flow merging techniques as before.
- SWAP. This neighborhood considers two activities a and b such that swapping a and b does not create a precedence cycle. Using this neighborhood is more expensive because for evaluating one move, we explicitly make the swap and incrementally compute its effect using the evaluation function of the invariant.
- REORDERBYDISTTOSINK. This neighborhood 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 [34].
- REINSERTIONDEEP. This last neighborhood is a large neighborhood which considers an activity a and which tries to reinsert a at the best possible position. To enlarge the set of possible insertion positions for a, it also moves project ancestors of a at their leftmost position in the order and project descendants of a at their rightmost position in the order without increasing the current makespan.

6 Search Strategies

The last part of the CBLS scheduling system built allows search strategies to be defined. Several CBLS invariants are used to implement these search strategies efficiently, such as invariants which incrementally maintain the set of critical activities of the schedule.

We describe here the strategy which is used in the experiments. One of the main component of this strategy is Variable Neighborhood Search (VNS [29]). This component successively considers the neighborhoods defined (or a subset of them), and applies these neighborhoods to critical activities until reaching a solution which is locally optimal. When exploring each neighborhood, we maintain, for each activity a, a tabu status which is set to true when the last move made on a did not improve the makespan, and which is reset to false if any move improves the makespan. The exploration of a given neighborhood stops when all critical activities are marked as tabu. At each step, the

move selected for an activity a is either one move that has the best makespan evaluation (for instance according to Eq. 10 for the REINSERTION neighborhood), or one move for which the makespan evaluation is strictly lower than the current makespan. The first option is used 70% of the time in the experiments and corresponds to a deepest descent strategy. The second one is used the rest of the time (if possible) and allows to diversify search.

On top of VNS, two search phases are successively used. In the first one, we start from an empty schedule (i.e. $u_a = false$ for every activity a) and at each step, we randomly select one activity which is not in the schedule yet and for which all project predecessors are already contained in the schedule. We then insert this activity at the best possible position in priority list $\mathcal{O}$ according to the REINSERTION neighborhood. Once the schedule is full, it is optimized using VNS. This schedule construction process is iterated N times and the best solution found is systematically recorded. As a result, we use here a kind of Greedy Randomized Adaptive Search Procedure [15]. In the experiments, $N = 10$ or 50 depending on the problem size.

The second search phase starts from the best solution found after the first phase and uses tabu search to escape local optima and plateaus [16]. During tabu search, VNS only considers moves which are not tabu according to the tabu list. We use here a very simple form of tabu list of limited length which contains *makespan values*. One advantage of this choice is that it is generic (independent of the resource types manipulated), and if an optimal solution has not been found yet then we are sure that it is never deemed as tabu. One drawback is that if all schedules but the optimal one have the same makespan, it is very unlikely that the tabu list will allow to find this optimal solution. However, we observed that such a choice works quite well on the benchmarks considered. After each application of VNS, the makespan of the current solution is added to the tabu list, the oldest tabu makespan being removed if some place is needed. The tabu search phase ends when the makespan value of the current solution has not been improved during a certain number of applications of VNS. Then,

- If after tabu search the current makespan is not strictly better than the best makespan known $mkBest$, we try to repair the inconsistency by working on the bottleneck of the problem. To do this, we randomly choose one non-critical activity a, we deactivate all its resource consumptions ($u_a = false$), we clear the tabu list, and we apply the VNS-based tabu search again.
- Otherwise, if the algorithm finds a solution whose current makespan is strictly less than $mkBest$, we choose one activity whose resource consumptions have been deactivated, we reactivate its resource consumptions, we clear the tabu list, and we apply the VNS-based tabu search again. If resource consumptions are active for all activities, then this means that a better solution has been found and the search goes on with an updated best makespan value.

During this process, we use heuristics for choosing the activity whose consumptions must be activated or deactivated. These heuristics maintain a conflict-weight for each activity, which is higher for activities which are often critical (kind of *wdeg* heuristic [8]). Given an activity, its selection probability for activation (resp. deactivation) is proportional (resp. inversely proportional) to its conflict-weight, so as to insert (resp.

remove) activities which are the more likely (resp. the less likely) to belong to the bottleneck of the problem. For avoiding spending too much time on conflict resolution in place of search over full schedules, each activity can be removed at most once and we never remove activities which have been critical at some step.

When no activity is candidate for being removed from the schedule, we use a perturbation phase in order to diversify search: we select $x\%$ of the activities and we randomly move each of them in priority list $\mathcal{O}$, while making additional updates to ensure that the new priority list does not induce precedence cycles. For all the experiments, the default perturbation percentage is x = 10%. In the end, by combining tabu search and perturbation, we get an *iterated tabu search* metaheuristic. As it is combined with the inconsistency repair mechanism, we call the global search strategy an *Iterated Tabu Search with Repair* (**ITSR**).

Last, from time to time, we perform a restart either from a new solution or from the best solution known (uniform distribution between the two options). For the experiments, a restart is realized 20% of the time instead of perturbing the current solution. Globally, the search strategy obtained mixes local search and global search, and it makes a trade-off between search intensification and search diversification.

7 Experiments

Experiments are performed on clusters composed of 20 Intel Xeon 2.8GHz processors with a shared memory of 20GB 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.

SDST-JSSP Table 1 summarizes the results obtained on the SDST-JSSP instances proposed by [10]. 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 hour. All neighborhoods defined in Sect. 5 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-start: 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 [25] and tabu search [26], and the CP approach defined in [17]. In the table, these methods are respectively referred to as AF08, BSV08, GVV08, GVV09, and GH10. The latter uses the same 1h time limit as ITSR on a similar processor. As in [17], 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 bound found by ITSR, which have also been verified using a separate checker. Globally, ITSR finds 5 new upper bounds and for all other instances, it finds all best known upper bounds, making it an efficient and robust optimization technique.

MJSSP Fig. 2a summarizes results obtained for the Multi-Capacity Job Shop Scheduling Problem (MJSSP). This problem involves cumulative resources and activities which

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

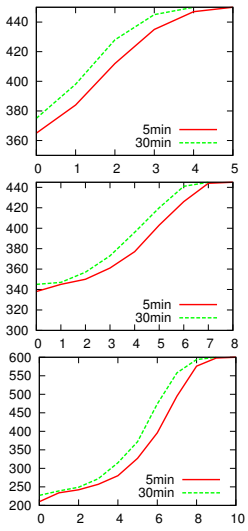
Table 1. Experiments for SDST-JSSP: comparison with the state-of-the-art for makespan minimization (best & mean values over 10 runs, with a 1 hour time limit for each run)

all consume one resource unit. As in [31, 28], which introduces *iterative flattening* (a precedence constraint posting approach), the instances considered are derived from standard JSSP instances by duplicating jobs and by 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 [30] (column NA96 in Fig. 2a). The instances selected are the 21 instances for which [28] provides new best upper bounds (column MV04 in Fig. 2a). In [28], these bounds are presented as the best ones found during the work on iterative flattening (no computation time specified). Over the 21 instances, MV04 finds a best upper bound for 6 instances, whereas ITSR manages to find 12 new best upper bounds. To get these results, the or-opt and 2-opt neighborhoods are deactivated. Fig. 2a also shows the results obtained when changing the length of the tabu list tl and the perturbation percentage x of the iterated tabu search.

RCPSP Figure 2b reports results obtained on RCPSP instances taken from the PSPLIB (`http://www.om-db.wi.tum.de/psplib`). These instances are split into four categories: j30, j60, j90, and j120, where jx means that the instance contains x 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. Given the number of instances, results are averaged over 5 runs. Fig. 2b shows the results obtained when the time limit is set to 5 and 30 minutes. Results on j30 instances are not presented because ITSR always manages to find the best known (optimal) solution within 5 minutes. For the other instances, Fig. 2b provides the distribution of the relative distance between the average makespan found by ITSR and the best makespan reported in the PSPLIB. For example, for j60 with a time limit of 5 minutes, the graph expresses that among the 450 instances considered, the distance to the best solution known is 0% for approximately 365 instances, it is $\leq 1\%$ for approximately 380 instances, $\leq 2\%$ for a bit more

Instance	#jobs x #res	NA96 lb	NA96 ub	MV04 Best	ITSR,tl=2,x=10 Best	ITSR,tl=2,x=10 Avg	ITSR,tl=4,x=10 Best	ITSR,tl=4,x=10 Avg	ITSR,tl=4,x=30 Best	ITSR,tl=4,x=30 Avg
la04-2	20 x 5	572	590	577	**576***	576.0	**576***	576.0	**576***	576.1
la04-3	30 x 5	570	590	584	**577***	579.7	579	580.7	**577***	580.6
la16-2	20 x 10	888	935	**929**	933	938.4	933	939.2	936	939.6
la16-3	30 x 10	717	935	**927**	939	947.1	933	946.2	942	946.5
la17-2	20 x 10	750	765	**756**	**756**	757.8	757	758.6	757	758.3
la17-3	30 x 10	646	765	**761**	763	765.7	765	767.0	764	766.8
la18-2	20 x 10	783	844	818	**814***	818.1	**814***	818.2	815	819.2
la18-3	30 x 10	663	844	**813**	818	825.4	817	823.2	819	823.9
la19-2	20 x 10	730	840	803	**792***	799.9	798	800.9	797	802.6
la19-3	30 x 10	617	840	801	799	802.5	800	803.2	**798***	801.2
la20-2	20 x 10	829	902	864	**859***	867.2	**859***	864.6	**859***	866.4
la20-3	30 x 10	756	902	863	**862***	871.5	868	872.3	872	873.9
la24-2	30 x 10	704	935	932	911	917.9	**909***	915.4	914	917.9
la24-3	45 x 10	704	935	929	**917***	923.2	**917***	922.9	**917***	922.9
la25-3	45 x 10	723	977	**965**	977	983.2	977	984.4	975	982.0
la38-2	30 x 15	943	1196	1185	**1180***	1188.6	1181	1191.4	1187	1192.4
la38-3	45 x 15	943	1196	1195	1197	1204.9	**1190***	1206.1	1197	1211.2
ft10-2	20 x 10	835	930	913	906	908.9	902	907.8	**901***	907.6
ft10-3	30 x 10	655	930	**912**	915	921.8	918	924.3	920	928.1
ft20-2	40 x 5	1165	1165	1186	1172	1175.0	1171	1172.9	*1169*	1173.7
ft20-3	60 x 5	387	1165	1205	1182	1190.1	*1181*	1187.4	*1181*	1188.0

(a)



(b)

Fig. 2. Experiments with cumulative resources; (a) MJSSP instances: comparison with the iterative flattening approach (best & mean values over 10 runs with a 1 hour time limit for each run); (b) RCPSP instances from the PSPLIB: j60 (top), j90 (middle), and j120 (bottom); representation of the average number of instances (y-axis) for which the deviation percentage with regards to the best upper bound is less than the value on the x-axis (average results over 5 runs)

than 410 instances, and so on. Globally, for j60 (resp. j90 and j120), ITSR manages to find schedules which are always within 5% (resp. 7% and 9%) from the best upper bounds. To give an idea of the evolution of the best solution found along time, for a single run over one of the hardest j120 instances (instance j120-36-6), ITSR produces a first solution after 40 milliseconds; after half a second, the best solution available has a makespan equal to 255, to be compared with the best upper bound 225 given in the PSPLIB; after 10 seconds, the best makespan known is 250, after 1 minute it is equal to 246, after 5 minutes it is equal to 245, and after 10 minutes it is equal to 244, therefore at a distance of 19 time units from the best makespan known.

8 Extensions and Future Work

In this paper, we gave a global view of CBLS techniques adapted to SDST-RCPSP. In our current implementation, these techniques are actually extended to a wider class of problems involving release and due dates for activities, time-dependent processing times, resource availability windows, or decisions on the resources used by activities. Resources with time-varying availability profiles, activities with time-varying resource consumptions, and maximum distance constraints are not covered yet. Taking these aspects into account requires to develop of a more complex CBLS invariant. Another research direction is to get more insight into the contribution of each component of the search strategy defined, and to work on automatic parameter tuning during search.

References

1. Aggoun, A., Beldiceanu, N.: Extending CHIP in order to solve complex scheduling and placement problems. Mathematical and Computer Modelling 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. European Journal of Operational Research 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. Annals OR 159(1), 249–267 (2008)
4. Artigues, C., Michelon, P., Reusser, S.: Insertion techniques for static and dynamic resource constrained project scheduling. European Journal of Operational Research 149, 249–267 (2003)
5. Babin, G., Deneault, S., Laporte, G.: Improvements to the or-opt heuristic for the symmetric traveling salesman problem. Journal of the Operational Research Society 58, 402–407 (2007)
6. Balas, E., Simonetti, N., Vazacopoulos, A.: Job shop scheduling with setup times, deadlines and precedence constraints. J. Scheduling 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: Proc. of the 10th European Symposium on Algorithms (ESA). pp. 152–164. Rome, Italy (2002)
8. Boussemart, F., Hemery, F., Lecoutre, C., Saïs, L.: Boosting systematic search by weighting constraints. In: Proc. of the 16th European Conference on Artificial Intelligence (ECAI'04). pp. 146–150 (2004)
9. Brucker, P., Drexl, A., Möring, R., Neumann, K., Pesch, E.: Resource-constrained project scheduling: Notation, classification, models, and methods. European Journal of Operational Research 112(1), 3–41 (1999)
10. Brucker, P., Thiele, O.: A branch and bound method for the general-shop problem with sequence dependent setup-times. Operations Research Spektrum 18(3), 145–161 (1996)
11. Carlier, J.: The one machine sequencing problem. European Journal of Operational Research 11, 42–47 (1982)
12. Carlier, J., Pinson, E.: Adjustment of heads and tails for the job-shop problem. European Journal of Operational Research 78, 146–161 (1994)
13. Croce, F.D.: Generalized pairwise interchanges and machine scheduling. European Journal of Operational Research 83(2), 310–319 (1995)
14. Croes, G.A.: A method for solving traveling salesman problems. Operations Research 6, 791–812 (1958)
15. Feo, T., Resende, M.: Greedy randomized adaptive search procedures. Journal of Global Optimization 6, 109–133 (1995)
16. Glover, F., Laguna, M.: Tabu Search. In: Modern Heuristic Techniques for Combinatorial Problems, pp. 70–141. Blackwell Scientific Publishing (1993)
17. Grimes, D., Hebrard, E.: Job shop scheduling with setup times and maximal time-lags: A simple constraint programming approach. In: Proc. of CPAIOR'10. pp. 147–161 (2010)
18. Grimes, D., Hebrard, E.: Solving variants of the job shop scheduling problem through conflict-directed search. INFORMS Journal on Computing 27(2), 268–284 (2015)
19. Hartmann, S.: A competitive genetic algorithm for resource-constrained project scheduling. Naval research logistics 45, 733–750 (1997)
20. Hentenryck, P.V., Michel, L.: Constraint-based Local Search. MIT Press (2005)
21. Ilog: IBM ILOG CPLEX and CpOptimizer, http://www-03.ibm.com/software/products/
22. Katriel, I., Michel, L., Hentenryck, P.V.: 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, chap. 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. European Journal of Operational Research 174(1), 2337 (2006)
25. M. A. González, C. R. Vela, R.V.: A new hybrid genetic algorithm for the job shop scheduling problem with setup times. In: Proc. of the 18th International Conference on Automated Planning and Scheduling (ICAPS'08). pp. 116–123 (2008)
26. M. A. González, C. R. Vela, R.V.: Genetic algorithm combined with tabu search for the job shop scheduling problem with setup times. In: Proc. of IWINAC (1). pp. 265–274 (2009)
27. Mastrolilli, M., Gambardella, L.: Effective neighborhood functions for the flexible job shop problem. Journal of Scheduling 3(1), 3–20 (2000)
28. Michel, L., Hentenryck, P.V.: Iterative relaxations for iterative flattening in cumulative scheduling. In: Proc. of the 14th International Conference on Automated Planning and Scheduling (ICAPS'04). pp. 200–208 (2004)
29. Mladenovíc, N., Hansen, P.: Variable neighborhood search. Computers and Operations Research 24(11), 10971100 (1997)
30. Nuitjen, W.P.M., Aarts, E.H.L.: A computational study of constraint satisfaction for multiple capacitated job shop scheduling. European Journal of Operational Research 90(2), 269–284 (1996)
31. Oddi, A., Cesta, A., Policella, N., Smith, S.F.: Iterative flattening search for resource constrained scheduling. Journal of Intelligent Manufacturing 21, 17–30 (2010)
32. Pinedo, M.: Scheduling: Theory, Algorithms, and Systems. Springer (2012)
33. 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)
34. Valls, V., Ballestn, F., Quintanilla, S.: Justification and RCPSP: A technique that pays. European Journal of Operational Research 165(2), 375386 (2005)
35. Vilím, P.: Edge finding filtering algorithm for discrete cumulative resources in O(k n log n). In: Proc. of CP'09. pp. 802–816 (2009)
36. Vilím, P.: Timetable edge finding filtering algorithm for discrete cumulative resources. In: Proc. of CPAIOR'11. pp. 230–245 (2011)