

## Tight LP bounds for resource constrained project scheduling\*

Philippe Baptiste<sup>1</sup> and Sophie Demassez<sup>2</sup>

<sup>1</sup> CNRS, LIX, Ecole Polytechnique, 91128 Palaiseau, France  
(e-mail: philippe.baptiste@polytechnique.fr)

<sup>2</sup> LIA, Université d'Avignon, 84911 Avignon, France

**Abstract.** The best lower bound for the Resource Constrained Project Scheduling Problem is currently based on the resolution of several large Linear Programs (Brucker & Knust, EJOR, 107:272–288, 1998). In this paper, we show that (1) intensive constraint propagation can be used to tighten the initial formulation of the linear programs and (2) we introduce several sets of valid cutting planes. These improvements allow us to “close” 16 new instances of the PSPLIB with 60 activities and to improve the best known lower bounds of 64 instances.

**Key words:** Resource constrained project scheduling – Branch & bound – Constraint programming – Linear programming

### 1 Introduction

Many industrial scheduling problems are variants, extensions or restrictions of the “Resource-Constrained Project Scheduling Problem”. Given (i) a set of  $q$  resources  $\{R_1, \dots, R_q\}$  with given capacities  $c_1, \dots, c_q$ , (ii) a set of  $n$  non-interruptible activities  $\{A_1, \dots, A_n\}$  of given processing times  $p_1, \dots, p_n$ , (iii) a network of precedence constraints between the activities modeled as a dag  $G$ , and (iv) for each activity  $A_i$  and each resource  $R_r$  the amount  $c_{ir}$  of the resource required by the activity over its execution, the goal of the RCPSP is to find a schedule meeting all the constraints whose makespan (*i.e.*, the time at which all activities are finished) is minimal. The decision variant of the RCPSP, *i.e.*, the problem of determining whether there ex-

---

\* The authors would like to thank Peter Brucker and Sigrid Knust for providing their source code as well as Christian Artigues, Jacques Carlier and Philippe Michelon for enlightening discussions on project scheduling.

Correspondence to: P. Baptiste

ists a schedule of makespan smaller than a given deadline, is NP-hard in the strong sense [15].

The intractability of the RCPSP has motivated test of numerous optimization techniques and led to an extensive literature. Overviews of heuristic and exact procedures can be found in [18] and [12]. In order to improve the behavior of exact algorithms, like branch and bound, a wide variety of bounding techniques have been proposed. The techniques to compute lower bounds on the minimal makespan are roughly of three kinds: (1) the “specific” techniques, as called in [17], which rely on basic combinatorial arguments, (2) the relaxation of integer formulations (LP and Lagrangian relaxations) [4, 10, 22], and (3) the constraint-based techniques [2, 8, 13, 17].

Among the LP-based approaches, Mingozzi et al. [22] describe a new integer programming formulation for the RCPSP and propose some linear relaxations. Brucker and Knust [4] strengthen one of this relaxation by taking into account time windows for the activities and use column generation to deal with the large number of variables. Moreover, they preprocess the linear program, using constraint propagation, to reduce time windows, and compute their bound under a “destructive” approach: The lower bound is the maximal value  $T$  for which it can be proven that no solution exists with value lower than  $T$ . Recently, Demassey et al. [11] proposed a similar destructive lower bound including constraint propagation as preprocessing of another linear relaxation of the RCPSP. They use then some CP deductions to generate cutting planes for the linear program. These two last bounds are rather expensive in term of CPU time but they are currently the best one on the standard PSPLIB benchmark sets.

In this paper, we present an improvement of the Brucker and Knust bound: (1) We improve the preprocessing step thanks to intensive constraint propagation techniques and (2) we introduce several energetic reasoning-based cutting-planes for the linear program. These new bounds allow us to improve the best known lower bounds for several instances of the PSPLIB. Although this was not our initial purpose, the preprocessing step happened to improve a lot the behavior of a basic branch and bound procedure and we have improved the best known upper bounds of some instances of the PSPLIB. Altogether, we have been able to close 16 instances of the PSPLIB and improve the best known lower bounds of 64 instances. Compared to the previously best known bounds, the average gap (distance from the lower to the upper bound) reduction is 13.5%.

Table 1 provides a summary of the notation that will be used throughout this paper. The first seven lines correspond to the initial data of the instance while the three last ones (release date and deadline) change during the search.

The paper is organized as follows: In Section 2, we describe our naive branch and bound procedure relying on the standard mechanisms of Ilog Scheduler. In Section 3, we show how the amount of constraint propagation can be increased by adding “redundant machines” to the initial scheduling problem. To build these machines, we solve a Mixed Integer Program (MIP) also described in Section 3. Sections 4, 5, 6 and 7 are dedicated to the presentation of Brucker and Knust lower bound and to the cuts that we have added to tighten the linear formulation. Experimental results are reported in Section 8.

**Table 1.** Notation and definitions

$A_1, \dots, A_n$	activities of a project
$p_i$	processing time of activity $A_i$
$G$	acyclic digraph of precedence constraints between activities
$G(A_i)$	set of all successors of $A_i$ in $G$
$R_1, \dots, R_q$	renewable resources
$c_r$	capacity of resource $R_r$ (constant during the scheduling period)
$c_{ir}$	amount of resource $R_r$ used during the execution of $A_i$
$r_i$	release date of activity $A_i$
$d_i$	deadline of activity $A_i$
$[r_i, d_i]$	time window of activity $A_i$

## 2 A simple constraint programming framework

Constraint Programming is a paradigm aimed at solving combinatorial optimization problems. Often these combinatorial optimization problems are solved by defining them as one or several instances of the *Constraint Satisfaction Problem* (CSP). Informally speaking, an instance of the CSP is described by a set of *variables*, a set of possible values for each variable, and a set of *constraints* between the variables. The set of possible values of a variable is called the variable's *domain*. A constraint between variables expresses which combinations of values for the variables are allowed. Constraints can be stated either implicitly (also called intentionally), *e.g.*, an arithmetic formula, or explicitly (also called extensionally), where each constraint is expressed as a set of tuples of values that satisfy the constraint. An example of an implicitly stated constraint on the integer variables  $x$  and  $y$  is  $x < y$ . An example of an explicitly stated constraint on the integer variables  $x$  and  $y$  with domains  $\{1, 2, 3\}$  and  $\{1, 2, 3, 4\}$  is the tuple set  $\{(1, 1), (2, 3), (3, 4)\}$ . The question to be answered for an instance of the CSP is whether there exists an assignment of values to variables, such that all constraints are satisfied. Such an assignment is called a *solution* of the CSP.

One of the key ideas of constraint programming is that constraints can be used “actively” to reduce the computational effort needed to solve combinatorial problems. Constraints are thus not only used to test the validity of a solution, as in conventional programming languages, but also in an active mode to remove values from the domains, deduce new constraints, and detect inconsistencies. This process of actively using constraints to come to certain deductions is called *constraint propagation*. The specific deductions that result in the removal of values from the domains are called *domain reductions*. The set of values in the domain of a variable that are not invalidated by constraint propagation is called the *current domain* of that variable.

As the general CSP is NP-complete constraint propagation is usually incomplete. This means that some but not all the consequences of the set of constraints are deduced. In particular, constraint propagation cannot detect all inconsistencies.

Consequently, one needs to perform some kind of search to determine if the CSP instance at hand has a solution or not. Most commonly, search is performed by means of a *tree search* algorithm.

### *A constraint programming model for the RCPSP*

We use the following model to represent the RCPSP. One constrained integer variable is associated with each activity  $A_i$ . It represents the start time of the activity. Moreover, we introduce an additional activity  $A_{n+1}$  with  $p_{n+1} = 0$  that is constrained to start after the completion of any other activity. The makespan of the project is then exactly the starting time of  $A_{n+1}$ . The initial domains of all the  $n + 1$  start variables are  $\{0, \dots, \sum p_i\}$  where  $\sum p_i$  is a basic upper bound of the optimal project duration. In the following, we associate a time window  $[r_i, d_i]$  (release date, deadline) to each activity.  $r_i$  is the minimal value in the domain of the starting time variable and  $d_i$  is the maximal value in the domain of the starting time plus the processing time.

First, the RCPSP is an optimization problem. The goal is to determine a solution with minimal makespan and prove the optimality of the solution. A common technique to look for an optimal solution is to solve successive **decision variants** of the problem. Several strategies can be considered. One way is to iterate on the possible values, either from the lower bound of its domain up to the upper bound until one solution is found, or from the upper bound down to the lower bound determining each time whether there still is a solution. Another way is to use a dichotomizing algorithm, where one starts by computing an initial upper bound  $d_{n+1}$  and an initial lower bound  $r_{n+1}$  for the makespan. Then

1. Set  $T = \left\lfloor \frac{r_{n+1} + d_{n+1}}{2} \right\rfloor$ .
2. Constrain the starting time of  $A_{n+1}$  to be at most  $T$  and solve the resulting decision problem, *i.e.*, determine a solution with makespan lower than or equal to  $T$  or prove that no such solution exists. If a solution is found, set  $d_{n+1}$  to the value of makespan in the solution; otherwise, set  $r_{n+1}$  to  $T + 1$ .
3. Iterate steps 1 and 2 until  $d_{n+1} = r_{n+1}$ .

A branching procedure with constraint propagation at each node of the search tree is used to determine whether the problem with makespan at most  $T$  accepts a solution. The two important ingredients in this procedure are (1) the techniques used to propagate the constraints and (2) the heuristics used to build the search tree.

Performing more **constraint propagation** serves two purposes: first, detect that a partial solution at a given node cannot be extended into a complete solution with makespan lower than or equal to  $T$ ; second, reduce the domains of the start and end variables (*i.e.*, tighten the time windows), thereby providing useful information on which variables are the most constrained.

In our framework, a precedence between two activities  $A_i$  and  $A_j$  is modeled by a linear constraint between the starting time variables. Such constraints can be easily propagated using a standard arc-B-consistency algorithm [20]. In addition, a variant of Ford's algorithm (see for instance [16]) proposed by Cesta and Oddi [9] can be

used to detect any inconsistency between such constraints, in time polynomial in the number of constraints (and independent of the domain sizes).

Complex constraint propagation techniques have been proposed for cumulative resources [3,7,23] but the cost of these algorithms is usually not balanced by the subsequent reduction of search. In this paper, we have decided to use the simple Time-Table mechanism, widely used in Constraint-Based Scheduling tools, that allows to propagate the resource constraints in an incremental fashion. It relies on an explicit data structure called “Time-Table” to maintain information about resource utilization and resource availability over time. Resource constraints are propagated in two directions: from resources to activities, to update the time bounds of activities (earliest start times and latest end times) according to the availability of resources, and from activities to resources, to update the Time-Tables according to the time bounds of activities. We refer to [19] for a detailed description of Time-Tables.

Besides constraint propagation, a constraint-based scheduling procedure is also characterized by the way the **search tree** is built. We use the basic search procedure preimplemented in Ilog Scheduler [3, 19] that chronologically builds a schedule:

1. Initialize the set of *selectable* activities to the complete set of activities to schedule.
2. If all the activities have fixed start and end times, a solution is found, then exit. Otherwise, remove from the set of selectable activities those activities which have fixed start and end times.
3. If the set of selectable activities is not empty, select an activity from the set (one with smallest deadline), create a choice point (*i.e.*, a node in the search tree) for the selected activity to allow backtracking and schedule the selected activity from its earliest start time to its earliest end time. Then go to step 2.
4. If the set of selectable activities is empty, backtrack to the most recent choice point. (If there is no such choice point, report that there is no problem solution and exit.)
5. Upon backtracking, mark the activity that was scheduled at the considered choice point as not selectable as long as its earliest start and end times have not changed. Then go to step 2.

### 3 MIPs to build redundant single machines

Following the ideas of [2,5], we have decided to generate redundant machine constraints (a machine is a resource with unit capacity). Such redundant constraints are useful for at least two reasons:

- On a machine, edge-finding constraint propagation can be applied. It consists in determining whether an activity must, can, or cannot be the first or the last to execute among a set of activities that require the same machine [6]. This mechanism provides tightened time bounds for activities requiring the same machine. It is known to be extremely powerful and can be implemented in  $O(n \log n)$ .

- Specific branching schemes have been designed to schedule machines. Directly inspired by the large amount of work dedicated to job-shop scheduling, these schemes consist in ordering all activities that require the same single machine. Ordering the single machines before applying the branching scheme of Section 2 drastically improves the performance of the branch and bound. In this paper, we rely on the edge-finding implementation of Ilog Scheduler, already described in [1].

To generate redundant single machines, we look for sets of activities that are known not to overlap in any feasible solution. Note that two activities  $(A_i, A_j)$  never overlap in time (1) if there is a precedence constraint between  $A_i$  and  $A_j$  or (2) if there is a resource such that the total amount of capacity required by  $A_i$  and  $A_j$  on the considered resource exceeds its capacity. In the following, two activities meeting conditions (1) or (2) are said to be “compatible”. Any set of activities in which all activities are pairwise compatible is a candidate redundant machine.

We associate a binary variable  $X_i \in \{0, 1\}$  to each activity  $A_i$  ( $X_i$  equals 1 when  $A_i$  belongs to the single machine under construction, 0 otherwise). A vector  $X$  corresponds to a valid redundant machine if for all activities  $A_i, A_j$  that are not compatible,  $X_i + X_j \leq 1$ .

Since the edge-finding constraint propagation algorithm is costly in terms of CPU time, very few redundant machines can be generated. Hence, we have to heuristically select some of them. Our intuition is that “good” redundant machines are heavily loaded so, we try to find a vector  $X$  that maximizes  $\sum p_i X_i$ . The resulting problem is a MIP with  $n$  variables and at most  $n^2$  constraints (much less in practice). In [2], a greedy heuristic was used to build a solution to a similar MIP. Initial experiments have shown that, in terms of final reduction of time windows, it is much better to solve the MIP to optimality.

Following [2], we have decided to add several redundant machines to the problem. More precisely, we build one global redundant machine according to the above MIP and one redundant machine per cumulative resource. For each cumulative resource, we create a redundant machine in which we put all the activities requiring more than half of the resource (such activities never overlap in time). We then try to add some extra activities on this redundant machine. To do so, we modify the MIP by replacing  $X_u$  variables corresponding to activities  $A_u$  that are already on the redundant machine by 1 (the other  $X_u$  variables remain). We then have a “reduced” MIP that is solved to optimality.

#### 4 Brucker and Knust “destructive” bound

In this section, we precisely describe the lower bound of Brucker and Knust [4] evoked in the introduction. In [22], Mingozzi et al. have presented a new LP-formulation for a relaxation of the RCSP, where preemption is allowed and conjunctions (*i.e.*, precedence constraints  $A_j \in G(A_i)$ ) are treated as disjunctions ( $A_j \in G(A_i)$  or  $A_i \in G(A_j)$ ). This formulation is built on the notion of “feasible subsets”, *i.e.*, sets of activities that can be processed in parallel. Due to the large number of variables (one for each feasible subset), they approximate the optimal

value of this LP by computing heuristic solutions of the dual program. Brucker and Knust solve a strengthened version of Mingozi formulation. They divide the time horizon  $[0, T]$  and consider, for each time subinterval  $I$ , feasible subsets of activities that can effectively be processed within  $I$ , according to the time windows. In order to tighten the time windows of the activities, they first use constraint propagation techniques, including interval consistency, immediate selection, edge-finding and symmetric triples. Finally, they deal with the size of the linear program by using a column generation procedure. Following the destructive approach of Klein and Scholl [17], they do not compute directly a lower bound by solving such a LP-relaxation. Actually, they try to prove infeasibility of a given value  $T$  for the makespan, by first applying constraint propagation, then else by showing, with column generation, that the LP-relaxation has no solution. Hence, by a dichotomizing search, they look for the maximal value  $T$  such that  $T - 1$  is proved to be an infeasible makespan.

In the following, we recall the LP model used by Brucker and Knust (we follow the same notation and terminology). Given a trial makespan  $T$ , we assume that the constraint propagation has not detected infeasibility, *i.e.*, for each activity  $A_i$ , the time window  $[r_i, d_i]$  is larger than the duration  $p_i$ .

Let  $z_0 < z_1 < \dots < z_\tau$  denote the ordered sequence of all different  $r_i$  and  $d_i$  values. For all  $t \in \{1, \dots, \tau\}$ ,  $I_t$  denotes the interval  $[z_{t-1}, z_t]$  and  $F(t)$  is the set of activities  $A_i$  that can be scheduled in  $I_t$ , *i.e.*,  $r_i \leq z_{t-1}$  and  $z_t \leq d_i$ .

A subset  $X \subset F_t$  is said to be feasible if all activities in  $X$  can be simultaneously processed, *i.e.*,

$$\begin{aligned} &\forall A_i, A_j \in X, A_i \notin G(A_j) \text{ and } A_j \notin G(A_i) \\ &\forall r \in \{1, \dots, q\}, \sum_{A_i \in X} c_{ir} \leq c_r \end{aligned}$$

For any interval  $I_t$ ,  $q_t$  denotes the total number of feasible subsets of the interval and  $X_{jt}$  ( $1 \leq j \leq q_t$ ) denotes all feasible subsets of the interval. With each set  $X_{jt}$  is associated an incidence vector  $a^{jt} \in \{0, 1\}^n$  ( $a_i^{jt} = 1$  iff  $i \in X_{jt}$ ).

We have one variable  $x_{jt}$  per feasible subset in an interval  $t$ . It denotes the number of time units where all activities in  $X_{jt}$  are processed simultaneously. Non-negative artificial variables  $u_t$ ,  $t \in \{1, \dots, \tau\}$  are also introduced in order to turn the decision problem into an optimization problem. If precedence and non-preemption constraints are relaxed, it is easy to see that the scheduling problem is feasible if and only if the following linear program has the optimal value zero.

$$(LPBK) \min \sum_{t=1}^{\tau} u_t \tag{1}$$

subject to:

$$\sum_{t=1}^{\tau} \sum_{j=1}^{q_t} a_i^{jt} x_{jt} \geq p_i \quad \forall i \in \{1, \dots, n\} \tag{2}$$

$$\sum_{j=1}^{q_t} x_{jt} - u_t \leq z_t - z_{t-1} \quad \forall t \in \{1, \dots, \tau\} \tag{3}$$

$$x_{jt} \geq 0 \quad \forall t \in \{1, \dots, \tau\}, \forall j \in \{1, \dots, q_t\} \quad (4)$$

$$u_t \geq 0 \quad \forall t \in \{1, \dots, \tau\} \quad (5)$$

As shown in [4], (LPBK) is a large linear program (the total number of variables grows exponentially with  $n$ ) that can be efficiently solved with column generation. At each iteration, to generate improving columns, we have to solve a multidimensional knapsack problem. Brucker and Knust [4] describe a specific branch and bound algorithm for this purpose while we directly model the problem as a MIP solved by Cplex. In the following, we introduce several cuts to be added to the linear program. Still the same column generation scheme can be used.

## 5 Energetic cuts

We introduce a set of cuts based on “energetic reasoning” [14, 21]: Given a resource and an interval of time, the “required consumption” of all the activities over the interval is compared to the “provided” amount of resource during the same interval.

Given an activity  $A_i$  and a time interval  $[s, e]$ , we define  $p(A_i, s, e)$ , the required duration of  $A_i$  over  $[s, e]$ , as the minimum of

1.  $e - s$ , the length of the interval;
2.  $p_i^+(s) = \max(0, p_i - \max(0, s - r_i))$ , the number of time units during which  $A_i$  executes after time  $s$  if  $A_i$  is left-shifted, *i.e.*, scheduled as soon as possible;
3.  $p_i^-(e) = \max(0, p_i - \max(0, d_i - e))$ , the number of time units during which  $A_i$  executes before time  $e$  if  $A_i$  is right-shifted, *i.e.*, scheduled as late as possible.

Given this definition, it is easy to see that, in any feasible schedule, at least  $p(A_i, s, e)$  units of  $A_i$  are scheduled in  $[s, e]$  and hence its required energy consumption on a resource  $R_r$ , is  $p(A_i, s, e) * c_{ir}$ . So, the traditional energetic reasoning mechanism consists in checking that

$$\forall s, \forall e, \sum_i p(A_i, s, e) * c_{ir} \leq (e - s)c_r.$$

In our case, energetic reasoning is even more simple. We add a constraint, for any interval  $[s, e]$  ( $= [z_\sigma, z_\varepsilon]$ ), stating that at least  $p(A_i, s, e)$  units of  $A_i$  have to be processed. This leads to the following cutting planes:

$$\sum_{t=\sigma}^{\varepsilon} \sum_{j=1}^{q_t} a_i^{jt} x_{jt} = p(A_i, z_\sigma, z_\varepsilon) \quad \forall i \in \{1, \dots, n\}, \forall \sigma, \varepsilon \in \{0, \dots, \tau\}, \sigma < \varepsilon \quad (6)$$

## 6 Non-preemptive cuts

Inspired by energetic cuts, we have built more complex non-preemptive cuts. The basic idea is to consider a subset of non-overlapping time intervals and to compute a non-preemptive upper bound on the number of time units during which an activity can be processed in this subset.

For example, let  $A_i$  be an activity with  $r_i = 1$ ,  $d_i = 12$ , and  $p_i = 5$  and consider two intervals  $[1, 4]$  and  $[9, 11]$ . In a non-preemptive schedule,  $A_i$  cannot overlap with both  $[1, 4]$  and  $[9, 11]$ . Hence,  $A_i$  is processed during at most 3 time units in  $[1, 4] \cup [9, 11]$ .

More formally, given a subset of indices  $\Psi \subseteq \{1, \dots, \tau\}$ , and an activity  $A_i$ , we claim that if the distance  $z_{t'-1} - z_t$  between any two intervals  $I_t$  and  $I_{t'}$  ( $t, t' \in \Psi, t < t'$ ) is greater than or equal to the duration of  $A_i$  then, in any feasible non-preemptive schedule, activity  $A_i$  cannot be processed in more than one of the intervals of  $\Psi$ . For such an activity  $A_i$  and a subset  $\Psi$ , this leads to a new linear cutting plane for (LPBK):

$$\sum_{t \in \Psi} \sum_{j=1}^{q_t} a_i^{jt} x_{jt} \leq \max\{z_t - z_{t-1} \mid t \in \Psi\}. \quad (7)$$

Of course, many such constraints could be added and we consider only the subsets of constraints built as follows. For each activity  $A_i$  and each index  $t \in \{1, \dots, \tau\}$ , we define the subset of interval indices  $\Psi(t, i)$  as follows:

1. Initialize  $\Psi(t, i)$  to  $\{t\}$
2. Extend Forward: Let  $\theta = \max(\Psi(t, i))$  and let  $\theta'$  be the smallest integer greater than  $\theta$  such that  $z_{\theta+1} + p_i \leq z_{\theta'}$  and  $z_{\theta'+1} - z_{\theta'} \leq z_{t+1} - z_t$ . If no such  $\theta'$  exists, go to step 3. Otherwise add  $\theta'$  to  $\Psi(t, i)$  and iterate.
3. Extend Backward: Let  $\theta = \min(\Psi(t, i))$  and let  $\theta'$  the largest integer lower than  $\theta$  such that  $z_{\theta'+1} \leq z_{\theta} - p_i$  and  $z_{\theta'+1} - z_{\theta'} \leq z_{t+1} - z_t$ . If no such  $\theta'$  exists then the set  $\Psi(t, i)$  is completed, exit. Otherwise add  $\theta'$  to  $\Psi(t, i)$  and iterate.

With this definition, it is easy to see that the distance between any two intervals of  $\Psi(t, i)$  is greater than  $p_i$  and thus we can add a cutting plane as defined above for the corresponding subset  $\Psi(t, i)$ .

## 7 Precedence cuts

It is difficult to take into account the precedence constraints directly in the linear program (LPBK). Still, it happens to be useful to introduce a weak formulation of these constraints.

For each activity  $A_i$ , we add a new variable  $m_i$  that represents the mid-point of  $A_i$  ( $m_i$  equals the average of the starting time and the completion time of  $A_i$  on a non-preemptive schedule). Of course, we can easily express precedence constraints:

$$m_j - m_i \geq \frac{p_i + p_j}{2} \quad \forall i \in \{1, \dots, n\}, \forall A_j \in G(A_i). \quad (8)$$

We can also link the mid-point variables to the  $x_{jt}$  variables. To do so, we split an activity  $A_i$  into small pieces  $A_{i1}, A_{i2}, \dots, A_{i\tau}$ . The piece  $A_{it}$  corresponds to the part of  $A_i$  processed in interval  $I_t = [z_{t-1}, z_t]$ . Let then  $p_{it}$  and  $m_{it}$  denote respectively the duration of  $A_{it}$  and its mid-point. Hence we have

$$p_{it} = \sum_{j=1}^{q_t} a_i^{jt} x_{jt},$$

and,

$$m_i = \frac{1}{p_i} \sum_{t=1}^{\tau} m_{it} p_{it}.$$

Furthermore, assuming non-preemption, for each interval  $I_t$  during which  $A_i$  is in process, we have  $p_{it} \geq 1$ . Then  $m_{it}$  is at least  $z_{t-1} + \frac{1}{2}$  and at most  $\leq z_t - \frac{1}{2}$ . Hence we have  $\forall i \in \{1, \dots, n\}$

$$\sum_{t=1}^{\tau} (z_{t-1} + \frac{1}{2}) \sum_{j=1}^{q_t} a_i^{jt} x_{jt} \leq m_i p_i \leq \sum_{t=1}^{\tau} (z_t - \frac{1}{2}) \sum_{j=1}^{q_t} a_i^{jt} x_{jt}. \quad (9)$$

## 8 Experimental results

All experiments have been led on a HP Omnibook Pentium III running at 720 MHz. Our new bounds have been tested on the 480 instances of the standard PSPLIB benchmark with 60 activities. All experimental results are available online:

[http://www.lix.polytechnique.fr/baptiste/lb\\_rcpsp.html](http://www.lix.polytechnique.fr/baptiste/lb_rcpsp.html)

Given the large number of instances, we first decided to “remove” the easy ones by adding the redundant machines as described in Section 3 and by running the basic branch and bound procedure of Section 2 during at most 90.0 seconds of CPU time. The average CPU time required to solve the MIPs building redundant machines (Sect. 3) is low (2.4 seconds on the average with a maximum of 8.8 seconds). Surprisingly, 373 instances were solved by the branch and bound in an average CPU time of 0.7 seconds. Among these instances, 12 were still open in the PSPLIB.

Among the 107 “non-easy” remaining instances, we have tested the three lower bounds described in Sections 4, 5, 6 and 7.

- The “LB-BK” bound corresponds to the lower bound of Brucker and Knust as described in Section 4 (recall that, although this is the same lower-bound, the preprocessing is not the same).
- The “LB-BK-E” is obtained by adding the Energetic cutting planes to the linear program (Sect. 5).
- Finally, the “LB-BK-E-PREC” bound is obtained by adding the Energetic cutting planes, the Non-Preemptive cutting planes (Sect. 6) and the Precedence cutting planes (Sect. 7) to the linear program.

Recall that these bounds are “destructive”: the lower bound is the maximal value  $T$  for which it can be proven that no solution exists with value lower than  $T$ . To reach  $T$  as fast as possible, we use a dichotomizing algorithm for LB-BK. For the two remaining bounds, we just try to increase the lower bound one unit after another.

LB-BK required 172.7 seconds on the average. It is always at least as good as the lower bound provided by Brucker and Knust and improves on the best known lower bound for 34 of the 107 instances (the improvement is sometimes as large as 12%). We believe that this improvement comes from our preprocessing step that relies on the propagation on redundant machines (Sect. 3).

Compared to LB-BK, LB-BK-E improves the lower bound on 7 instances but the improvement is relatively small (1 or 2 units of makespan). Surprisingly, the LB-BK-E lower bounds were reached slightly faster than LB-BK (141.2 seconds on the average). This comes from the fact that while several iterations were required for LB-BK, few iterations were required for LB-BK-E.

LB-BK-E-PREC improves in turn on LB-BK-E. Indeed, 14 more lower bounds are improved in an average CPU time of 284.6 seconds. As before, the improvement is relatively small. LB-BK-E-PREC allows us to close 4 more instances.

Altogether, we have been able to close 16 instances of the PSPLIB and improve the best known lower bounds of 64 instances. The average gap reduction is 13.5%.

## References

1. Baptiste P, Le Pape C (1995) A theoretical and experimental comparison of constraint propagation techniques for disjunctive scheduling. In: Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence
2. Baptiste P, Le Pape C (2000) Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints* 5: 119–139
3. Baptiste P, Le Pape C, Nuijten W (2001) Constraint-based scheduling, vol 39 of ISOR. Kluwer, Amsterdam
4. Brucker P, Knust S (2000) A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research* 127: 355–362
5. Carlier J, Néron E (2001) A new lp based lower bound for the cumulative scheduling problem. Technical report, Université Technologique de Compiègne
6. Carlier J, Pinson E (1990) A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Annals of Operations Research* 26: 269–287
7. Carlier J, Pinson E (2002) Jackson's pseudo preemptive schedule and cumulative scheduling problems. Technical report, University of Compiègne
8. Caseau Y, Laburthe F (1996) Cumulative scheduling with task intervals. In: Maher M (ed) Proceedings of the Joint International Conference and Symposium on Logic Programming, JCPSP'96, pp 363–377. The MIT Press, Cambridge, MA
9. Cesta A, Oddi A (1996) Gaining efficiency and flexibility in the simple temporal problem. In: Proc. 3rd International Workshop on Temporal Representation and Reasoning
10. Christofides N, Alvarez-Valdés R, Tamarit JM (1987) Project scheduling with resource constraints: a branch and bound approach. *European Journal of Operational Research* 29(3): 262–273
11. Demasse S, Artigues C, Michelon P (2003) Constraint-propagation-based cutting planes: An application to the resource-constrained project-scheduling problem. *INFORMS Journal on Computing* (to appear)
12. Demeulemeester EL, Herroelen WS (2002) Project scheduling: a research handbook, vol 39 of ISOR. Kluwer, Amsterdam
13. Dorndorf U, Phan-Huy T, Pesch E (1999) A survey of interval capacity consistency tests for time- and resource-constrained scheduling, chap 10, pp 213–238. Kluwer, Amsterdam
14. Erschler J, Lopez P, Thuriot C (1991) Raisonnement temporel sous contraintes de ressources et problèmes d'ordonnancement. *Revue d'Intelligence Artificielle* 5: 7–32
15. Garey MR, Johnson DS (1979) Computers and intractability. A guide to the theory of NP-completeness. Freeman, New York

16. Gondran M, Minoux M (1984) *Graphs and algorithms*. Wiley, New York
17. Klein R, Scholl A (1999) Computing lower bound by destructive improvement: An application to resource-constrained project scheduling. *European Journal of Operational Research* 112: 322–346
18. Kolisch R, Hartmann S (1999) Algorithms for solving the resource-constrained project scheduling problem: Classification and computational analysis. In: Weglarz J (ed) *Handbook on recent advances in project scheduling*, chap 7. Kluwer, Amsterdam
19. Le Pape C (1994) Implementation of resource constraints in ILOG SCHEDULE: A library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering* 3(2): 55–66
20. Lhomme O (1993) Consistency techniques for numeric csp. In: *Proc. 13th International Joint Conference on Artificial Intelligence*
21. Lopez P, Erschler J, Esquirol P (1992) Ordonnancement de tâches sous contraintes: une approche énergétique. *R.A.I.R.O. APII* 26(5–6): 453–481
22. Mingozzi A, Maniezzo V, Ricciardelli S, Bianco L (1998) An exact algorithm for the multiple resource-constrained project scheduling problem based on a new mathematical formulation. *Management Science* 44: 714–729
23. Nuijten W (1994) *Time and resource constrained scheduling: A constraint satisfaction approach*. PhD thesis, Eindhoven University of Technology