A Column Generation Based Destructive Lower Bound for Resource Constrained Project Scheduling Problems[⋆]

J.M. van den Akker, G. Diepen, and J.A. Hoogeveen

Department of Information and Computing Sciences
Utrecht University
P.O. Box 80089, 3508 TB Utrecht, The Netherlands
`marjan@cs.uu.nl, diepen@cs.uu.nl, slam@cs.uu.nl`

Abstract. In this paper we present a destructive lower bound for a number of resource constrained project scheduling (RCPS) problems, which is based on column generation. We first look at the problem with only one resource. We show how to adapt the procedure by Van den Akker et al. [1] for the problem of minimizing maximum lateness on a set of identical, parallel machines such that it can be used to solve these RCPS problems. We then consider a number of variants of the RCPS problem with one or more resources and show how these can be solved by our approach. Because of the close relation between RCPS and the cumulative constraint in constraint programming, our method can be used as an efficient filtering algorithm for the cumulative constraint as well.

1980 Mathematics Subject Classification (Revision 1991): 90B35.
Keywords and Phrases: resource constrained project scheduling, cumulative constraint, linear programming, column generation, generalized precedence constraints.

1 Introduction

In this paper we consider a number of basic problems from project scheduling; we refer to the survey paper by Brucker, Drexl, Möhring, Neumann, and Pesch[6] for an overview of this area. We further refer to Van den Akker, Hoogeveen, and Van de Velde [2], Baptiste, Le Pape, and Nuijten [3], and Bazaraa, Jarvis, and Sherali [4] for an overview of the application of column generation in scheduling, for an overview of the application of constraint programming in scheduling, and for an overview of linear programming in general, respectively.

The basic resource constrained scheduling problem we are looking at is defined as follows. We are given a set of n jobs, which we denote by $J_1, \ldots, J_n$. For each job J_j we are given its processing time p_j, its release date r_j, its deadline

[⋆] Supported by BSIK grant 03018 (BRICKS: Basic Research in Informatics for Creating the Knowledge Society)

$\bar{d}_j$, and its resource consumption pattern, which gives the amount of resource needed during its execution; for the time being, we assume that there is only one kind of resource. For each job J_j we are asked to find a valid starting time S_j and completion time $C_j = S_j + p_j$ such that job J_j does not start before its release date ($S_j \geq r_j$), it is completed by its deadline ($C_j \leq \bar{d}_j$), and such that the total resource consumption of the jobs at any time t does not exceed the amount of resources available at that time. Moreover, between each pair of jobs J_i and J_j, there can be generalized precedence constraints, which define a lower bound and/or upper bound on $S_i - S_j$. In case the upper and lower bound are equal, we say that there is a *no-wait* constraint between J_i and J_j. The goal is to minimize either the makespan $C_{\max}$ or the maximum lateness $L_{\max} = \max_j L_j$, where the lateness L_j of job J_j is defined as the difference between the completion time C_j and the due date d_j, which denotes a target completion time. In fact, our approach can easily be generalized further to deal with any regular minimax function.

The resource constrained project scheduling problem has received attention from both operations research and constraint programming. We only discuss a few contributions. Brucker and Knust ([7], [8]) have applied column generation to a number of resource constrained project scheduling problems in which the goal is to minimize the makespan. Here they first formulate the problem as a decision problem and then use linear programming to check whether it is possible to execute all jobs in a feasible preemptive schedule; here the decision variables refer to the length of a time slice during which a given set of jobs is executed simultaneously. Cesta, Oddi, and Smith [9] have applied constraint programming to the makespan problem. The key here is to determine a schedule that is feasible for all constraints except for the resource consumption. Then resource conflicts are determined and resolved.

Van den Akker, Hoogeveen, and van Kempen [1] have looked at the special case of the above model in which the available amount of resources is constant over time (say m) and each job has a constant resource consumption pattern of one, that is, at any time during its execution, it consumes one unit of resource. This problem is then equivalent to the parallel machine scheduling with m parallel, identical machines. Van den Akker et al. [1] have presented a column generation based method to solve it, which yields a lower bound that turned out to be tight in all their computational experiments. They further gave a method to find a feasible solution with value equal to the lower bound. We will briefly discuss their method in Section 2. In Section 3 we will describe how we can extend their method to solve a number of basic RCPS problems. In Section 4 we consider two other extensions, concerning change-over times and machine maintenance. Finally, we draw some conclusions and present directions for future research in Section 5.

2 Reviewing the basic method

Here, we briefly review the column generation approach presented in [1] for the problem of minimizing $L_{\max}$ on a set of m parallel, identical machines. Each job needs exactly one machine during its processing. Furthermore, there are release dates, deadlines, and generalized precedence constraints. The minimization problem is turned into a feasibility problem by putting an upper bound L on $L_{\max}$; this is equivalent to adding deadlines $\bar{d}_j \leftarrow d_j + L$ $(j = 1, \ldots, n)$. Since a feasible schedule corresponds to a collection of at most m feasible, single-machine schedules containing all n jobs, the decision problem can be reformulated as: *is it possible to partition the jobs in at most m subsets such that for each subset we can find a feasible single-machine schedule*? Finally, the latter decision problem is solved by answering the question: what is the minimum number of feasible single-machine schedules that are needed to accommodate all jobs?

This problem is formulated as an integer linear programming problem as follows. We call a subset of jobs that allow a feasible single-machine schedule with respect to the release dates and deadlines a *machine schedule*. Let S be the set containing all machine schedules. We introduce binary variables x_s $(s = 1, \ldots, |S|)$ that take value 1 if machine schedule s is selected and 0 otherwise. For each machine schedule s we encode whether job J_j is included (then $a_{js} = 1$) or not ($a_{js} = 0$), and we encode the starting times S_{js} of the jobs with $a_{js} = 1$ $(j = 1, \ldots, n)$. Since two jobs that are connected through a precedence constraint do not have to be executed by the same machine, the generalized precedence constraints are not included in the feasibility of the machine schedules, and we include a constraint in the integer linear programming formulation for each of the generalized precedence constraints. We define A^1 as the arc set containing all pairs (i, j) such there exists a precedence constraint of the form $S_j - S_i \geq q_{ij}$; similarly, we define A^2 and A^3 as the arc sets that contain an arc for each pair (i, j), for which $S_j - S_i \leq q_{ij}$ and $S_j - S_i = q_{ij}$, respectively. Note that the intersection of A^1 and A^2 does not have to be empty. We denote the union of A^1, A^2, and A^3 by the multiset A. This leads to the following integer linear programming formulation

$$\min \quad \sum_{s \in S} x_s$$

subject to

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n, \tag{1}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s \geq q_{ij} \text{ for each } (i, j) \in A^1; \tag{2}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s \leq q_{ij} \text{ for each } (i, j) \in A^2; \tag{3}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s = q_{ij} \text{ for each } (i, j) \in A^3; \tag{4}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S.$$

We relax the integrality constraints to $x_s \geq 0$; the upper bound $x_s \leq 1$ follows from the other constraints. The LP-relaxation is solved by applying column generation. To start, we give each job its own machine. Given the outcome of the current LP, we find dual multipliers λ_j for Constraints 1 and δ_{ij} for the Constraints 2-4 in which jobs J_i and J_j are involved. The reduced cost of machine schedule s is then equal to

$$c_s' = 1 - \sum_{j=1}^{n} a_{js}\lambda_j - \sum_{j=1}^{n}\left[\sum_{h \in Prec_j} \delta_{hj}S_{js} - \sum_{k \in Suc_j} \delta_{jk}S_{js}\right],$$

where $Prec_j$ and Suc_j are defined as the sets containing all predecessors and successors of job J_j in A, respectively. The pricing problem is then to find a machine schedule with minimum reduced cost. Since solving the LP-relaxation by column generation only renders us a lower bound when the column generation procedure has finished, we compute an intermediate lower bound as

$$\sum_{s \in S} x_s \geq \left[\sum_{j=1}^{n} \lambda_j + \sum_{(j,k) \in A} \delta_{jk} q_{jk}\right] / (1 - c^*),$$

where c^* denotes the outcome value of the pricing problem.

Since the pricing problem is $\mathcal{NP}$-hard to solve, we do not solve it to optimality in each iteration. We apply a two-phase Simulated Annealing procedure to find a good solution. In the first phase, we decide which jobs are included in the machine schedule and in which order. In the second step, we find the optimal starting times of the included jobs. After that, we change the choices made in phase 1, etc. We mostly use the local search procedure to find good solutions to the pricing problem, but after 50 iterations, or when we cannot find any improving column, we turn to a time-indexed linear programming formulation of the pricing problem. Here we use binary variables x_{jt} to indicate whether job J_j starts at time t or not; the corresponding cost coefficients c_{jt} are easily determined. The ILP-formulation (ignoring the constant) then becomes

$$\min \sum_{j=1}^{n} \sum_{t=r_j}^{\bar{d}_j - p_j} c_{jt} x_{jt}$$

subject to

$$\sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt} \leq 1 \quad \forall j = 1, \ldots, n; \tag{5}$$

$$\sum_{j=1}^{n} \sum_{s=t-p_j+1}^{t} x_{js} \leq 1 \quad \forall t = 0, \ldots, T-1; \tag{6}$$

$$x_{jt} \in \{0,1\} \quad \forall j = 1, \ldots, n; \forall t = r_j, \ldots, \bar{d}_j - p_j.$$

Here T denotes the latest point in time at which at least two jobs can be executed. Constraint 5 decrees that each job can be chosen at most once, and Constraint 6 states that at most one job should be executed at any time.

Since we need to find out whether there exists a solution using at most m machines, we stop as soon as the outcome of the LP-relaxation has hit m. If the outcome of the current LP is bigger than m, and we cannot find an improving column, then we can compute the outcome value of the pricing problem that we need such that the intermediate lower bound equals m. We can then ask the ILP-solver whether there exists a solution to the pricing problem with that value or less. If it does not exist, then we have proven that m is not achievable, and we are done; if we can find it, then this is an improving column that we add, etc. In this way, we do not have to solve the time-indexed formulation to optimality.

Finally, when we have found the smallest upper bound L on $L_{\max}$ that cannot be proven impossible, then we try to construct a feasible schedule with $L_{\max}$ equal to L by formulating the problem as a time-indexed ILP. Solving this ILP from scratch only works for small instances. But when we insert our knowledge of the lower bound by adding the constraint $LMAX = L$, then our ILP-solver CPLEX finds a feasible solution rather quickly, if it exists. So far (and we have run a lot of experiments), we have not found an instance in which the optimum solution is not equal to the lower bound.

In the remainder of this section, we discuss the computational experiments by Van den Akker et al.[1]. Note that in these experiments, we did not include any no-wait precedence constraints. In our experiments we compared our hybrid algorithm, i.e. column generation and then for the identified lower bound L solving the time-indexed ILP with $LMAX = L$, to the approach of letting CPLEX solve the time-indexed ILP formulation without knowing the value of the lower bound; from now on, we will refer to this as the *ignorant ILP*. We have applied both algorithms on 6 scenarios; for each scenario we ran ten test instances. The scenarios are described in Table 1. The algorithms were encoded

Number	p_j	r_j	d_j	n	m	# prec
0	U[1,20]	U[0,60]	U[50,80]	40	4	20
1	U[1,20]	U[0,40]	U[30,60]	70	5	35
2	U[1,20]	U[0,40]	U[60,80]	100	9	40
3	U[1,20]	U[0,60]	U[80,110]	180	10	60
4	U[1,20]	U[0,60]	U[40,80]	60	5	30
5	U[1,20]	U[0,60]	U[50,80]	30	3	15

Table 1. Test scenarios

in Java and the experiments were run on a Pentium 4, 3 Ghz PC with 1 GB memory. For each instance we let each algorithm run for at most 30 minutes.

Our results clearly showed that our hybrid algorithm outperformed the method of letting CPLEX solve the ignorant ILP by far. For all instances we managed to solve, the derived lower bound was equal to the optimal value. There are some instances for which we could not check whether optimum and lower bound coincided, for we could not solve them within 30 minutes. This may be due to a gap between the lower bound and the optimum. However, we were never able to show that the lower bound differed from the optimum for any instance. Altogether we may draw the conclusion that our lower bound is extremely strong. If we compare solving the ignorant ILP with the second part of the hybrid algorithm, then we see that specifying the optimum makes a lot of difference. Most likely the preprocessing steps performed by CPLEX play an important role in this. Therefore, we may expect the technique of constraint satisfaction to work very well to find a solution of value L if such a solution exists.

3 The RCPS problem

3.1 One resource

Unit resource consumption
We first look at the case that the resource consumption pattern is constantly equal to 1 for each job J_j, but the available amount of nonrenewable resources is not constant over time. We capture this situation in the general framework of [1] by issuing dummy jobs, which 'eat up' the missing resources. This is achieved in the following way. We define the number of machines m to be equal to the maximum amount of resource available at any time. Now we determine the amount of resource that is missing over time with respect to m; this will yield a figure with a number of piles of blocks on top of each other, where the higher you come, the smaller the block is. For each block, we introduce a dummy job with the following characteristics: it has processing time equal to the length of the block; release date equal to the left time point of the block; and deadline equal to the right time point of the block. In the example of Figure 1, we have a pile with three blocks, which lead to the three jobs called D_1, D_2, D_3. The release dates and deadlines of these jobs are r_i and $\bar{d}_i$; the processing times are equal to $\bar{d}_i - r_i$ $(i = 1, \ldots, 3)$. We further assume that each dummy job has a due date that is unrestrictively large to prevent any interference with the $L_{\max}$ value. Note that the choice of dummy jobs is not unique. We can, for example, mingle the dummy jobs D_1 and D_2 to obtain D'_1 and D'_2 by swapping the deadlines and adjusting the processing times: a solution with these two dummy jobs can be translated into a solution with the two original dummy jobs by applying a 'cross-over' operation of the two involved machine schedules at time $\bar{d}_2$. Similarly, dummy jobs can be split. Anyway, it is easily seen that each feasible solution for this instance that uses no more than m machines corresponds to a feasible solution for the RCPS with equal objective value.
Arbitrary integral resource consumption
We now assume that the resource consumption pattern is constant for each job, but it can be any arbitrary integral value greater than or equal to 1. Suppose that J_j is some job that needs a constant amount of $k \geq 2$ units of resource during its execution. We capture this situation in the general framework by replacing job J_j by job J'_j and $k - 1$ additional dummy jobs. Here J'_j is identical to J_j, except for its resource consumption, which we put equal to one. Furthermore, each dummy job has processing time equal to p_j, but it has no release date and

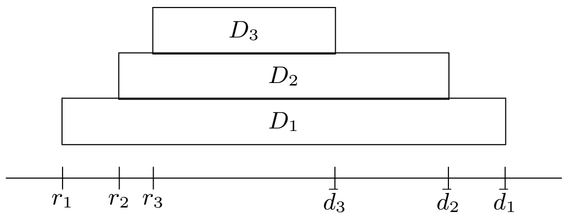

Fig. 1. Example of the dummy jobs

deadline, and it is independent of all other jobs, except for J'_j: we force that all these dummy jobs and J'_j are started at the same time by means of a no-wait constraint. It is easily seen that solving the resulting instance with unit resource consumption is equivalent to solving the original instance.

If the resource consumption of job J_j is not constant over time, but can attains arbitrary integral values, then we replace J_j by a set of new jobs with a constant resource consumption pattern equal to 1, and we glue these together by no-wait constraints, such that their joint resource consumption pattern is equivalent to that of the original job J_j.

We can now use the approach of [1] to find the lower bound. Furthermore, we can use the time-indexed formulation of [1] to look for a schedule with value equal to the lower bound. Since the dummy jobs that replace an original job J_j are glued together by no-wait constraints, and since the time-indexed formulation uses variables x_{jt} indicating whether job J_j starts at time t, we can restrict ourselves to the original jobs (with their varying resource consumption patterns) in the time-indexed formulation. Obviously, we must then adjust Constraints 6 to deal with the consumption patterns.

Note the close connection between the above RCPS problem and the cumulative constraint (see the on line Global Constraint Catalog by Beldiceanu and Demassey [5]. The cumulative constraint decrees that we should find for a given set of jobs starting times, which obey the release dates and deadlines, such that the total resource consumption should never exceed the available amount of resource. To filter this constraint, we must check whether a feasible schedule exists for the above resource constraint project scheduling problem without initial precedence constraints. Note that fixing the start time of some job can be easily included in the model by adjusting the available amount resource.

3.2 Multiple resources

We assume in this subsection that there are only two resources involved, but each model can be easily generalized to deal with any number of resources. We first transform it to an instance in which each job consumes only one resource during its execution: this is easy to achieve by replacing a job that needs both resources with two copies that only need one of the individual resources but are identical otherwise. These copies are then tied together by no-wait constraints such that they start at the same time. Next, we use the transformations described above to achieve that each job uses exactly one amount of resource (either resource 1 or

2) at any time during its execution. We now have transformed the problem into a parallel machine scheduling problem in which there are two different sets of identical machines; we assume that there are m_1 (m_2) machines corresponding to resource 1 (2).

We apply the same solution strategy as Van den Akker et al. [1] We divide the jobs and the machines into two groups, where jobs are only assigned to machines of the right group, which is easily incorporated in the pricing problem. We again minimize the total number of machines that is used, but we add the constraint that we use at least m_1 (m_2) machines of group 1 (2): if we then find a solution using no more than $m_1 + m_2$ machines, then we know that we do not use too much of resources 1 and 2 separately. Note that we could have added constraints decreeing that we use no more than m_1 (m_2) machines of group 1 (2) instead, but then we run into problems when we look for a feasible solution of the LP-relaxation to start with. Finally, we add some 'empty' columns, such that these two constraints can always be met.

As an illustration, we work things out for the case in which there are two resources, and each job J_j has a constant resource consumption pattern, requiring either 0 or 1 unit of resource 1 and 2. We assume for the ease of exposition that initially there are no precedence constraints. We denote the set of jobs requiring resource 1 only by R_1; similarly, we use R_2 to denote the jobs requiring resource 2 only. We denote the set of jobs that need both resources by $R_{1,2}$; these jobs will be split into two operations. These two operations need only one resource and are connected by a no-wait constraint. We use S and V to denote the set of machine schedules for resources 1 and 2, and we use x_s and y_v as binary decision variables. Moreover, we use a_{js} and b_{jv} to indicate whether job J_j is included in machine schedules s and v for resource 1 and 2, respectively. This leads to the following ILP-formulation, where with a little abuse of notation, a job $J_j \in R_{1,2}$ in fact consists of its two operations.

$$\min \quad \sum_{s \in S} x_s + \sum_{v \in V} y_v$$

subject to

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j \in R_1 \cup R_{12} \tag{7}$$

$$\sum_{v \in V} b_{jv} y_v = 1, \text{ for each } j \in R_2 \cup R_{12} \tag{8}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{v \in V} S_{jv} y_v = 0, \text{ for each } j \in R_{1,2} \tag{9}$$

$$\sum_{s \in S} x_s \geq m_1 \tag{10}$$

$$\sum_{v \in V} y_v \geq m_2 \tag{11}$$

$$x_s, y_v \in \{0, 1\}, \text{ for each } s \in S \text{ and } v \in V.$$

When we solve the LP-relaxation by column generation, we find that the reduced cost of a schedule $s \in S$ is equal to

$$c'_s = 1 - \lambda_0 - \sum_{j \in R_1 \cup R_{12}} a_{js}\lambda_j - \sum_{j \in R_{1,2}} \delta_j S_{js};$$

here λ_0 is the dual multiplier corresponding to Constraint 10, λ_j $(j \in R_1)$ are the dual multipliers corresponding to the Constraints 7, and δ_j $(j \in R_{1,2})$ are the dual multipliers corresponding to the Constraints 10. The reduced cost of machine schedule $v \in V$ is computed in an equivalent way. It is readily verified that the pricing problem is similar to the one of [1], which implies that the local search procedure and time-indexed formulation to solve it can still be applied. Furthermore, we can compute an intermediate lower bound as follows. Let c_1^* denote the optimal value of the pricing problem for resource 1. If we fill in $c'_s \geq c_1^*$ in the formula of the reduced cost, then we find that

$$1 \geq c_1^* + \lambda_0 + \sum_{j \in R_1 \cup R_{12}} a_{js}\lambda_j + \sum_{j \in R_{1,2}} \delta_j S_{js}.$$

Hence,

$$\begin{gathered}\sum_{s \in S} x_s \geq \sum_{s \in S} \left[c_1^* + \lambda_0 + \sum_{j \in R_1 \cup R_{12}} a_{js}\lambda_j + \sum_{j \in R_{1,2}} \delta_j S_{js} \right] x_s = \\ (c_1^* + \lambda_0) \sum_{s \in S} x_s + \sum_{j \in R_1 \cup R_{12}} \lambda_j \sum_{s \in S} [a_{js} x_s] + \sum_{j \in R_{1,2}} \delta_j \sum_{s \in S} S_{js} x_s = \\ (c_1^* + \lambda_0) \sum_{s \in S} x_s + \sum_{j \in R_1 \cup R_{12}} \lambda_j + \sum_{j \in R_{1,2}} \delta_j \sum_{s \in S} S_{js} x_s.\end{gathered}$$

Similarly, we find that

$$\sum_{v \in V} y_v \geq (c_2^* + \mu_0) \sum_{v \in V} y_v + \sum_{j \in R_2 \cup R_{12}} \mu_j - \sum_{j \in R_{1,2}} \delta_j \sum_{v \in V} S_{js} y_v;$$

here μ_0 is the dual multiplier corresponding to Constraint 11, μ_j $(j \in R_1 \cup R_{12})$ is the dual multiplier corresponding to Constraint 8, and c_2^* is the outcome value of the pricing problem for resource 2. If we add these two inequalities up, then the terms containing S_{js} cancel out, because of Constraints 9. Rearranging the terms, we find that

$$(1 - c_1^* - \lambda_0) \sum_{s \in S} x_s + (1 - c_2^* - \mu_0) \sum_{v \in V} y_v \geq \sum_{j \in R_1 \cup R_{12}} \lambda_j + \sum_{j \in R_2 \cup R_{12}} \mu_j.$$

If $1 - c_1^* - \lambda_0 = 1 - c_2^* - \mu_0$, then we can divide by this term and find a lower, provided that $1 - c_1^* - \lambda_0 > 0$, which issue we discuss later. Suppose that $1 - c_1^* - \lambda_0 > 1 - c_2^* - \mu_0$; the other case can be dealt with in the same way.

Then we add to this inequality $(c_2^* - c_1^* + \mu_0 - \lambda_0)$ times inequality 11, and we find the intermediate lower bound

$$\sum_{s\in S} x_s + \sum_{v\in V} y_v \geq \frac{((c_2^* - c_1^* + \mu_0 - \lambda_0)m_2 + \sum_{j\in R_1\cup R_{12}} \lambda_j + \sum_{j\in R_2\cup R_{12}} \mu_j}{(1 - c_1^* - \lambda_0)}.$$

What is left to show is that $(1 - c_1^* - \lambda_0) > 0$. We know that $c_1^* \leq 0$, since any column that is used in the current LP solution has zero reduced cost. Moreover, if both λ_0 and μ_0 are positive, then both constraints are binding, which implies that we have found a solution with value $m_1 + m_2$, which means that we can stop. Hence, at least one of λ_0 and μ_0 is zero, which implies that the maximum of $1 - c_1^* - \lambda_0$ and $1 - c_2^* - \mu_0$ is positive.

Finally, we look at the problem of finding a feasible solution with this value. It is easily verified that the time-indexed formulation of [1] to find a feasible solution can be used, but we must split the m machines into two sets representing the m_1 and m_2 units of resources 1 and 2, respectively.

Machine scheduling with operators

A special case of the above is the situation in which each job needs an operator to start it up, which takes 1 time unit per job. Hence, we should not start more jobs at any moment than there are operators available. We can model the operators as a second resource, but alternatively we can add the starting times to the machine schedules and force the restriction on the number of operators by adding constraints. Here, we work out the second option, which has the additional advantage that we can model a varying number of available operators. We again assume without loss of generality that there are no additional precedence constraints. We use o_{st} to indicate whether a job starts at time t in machine schedule s; we use Op_t to denote the number of operators available at time t. We then arrive at the ILP-formulation:

$$\min \quad \sum_{s\in S} x_s$$

subject to

$$\begin{aligned} \sum_{s\in S} a_{js} x_s &= 1, \text{ for each } j = 1, \ldots, n \\ \sum_{s\in S} o_{st} x_s &\leq Op_t, \text{ for all } t = 0, \ldots, T-1 \qquad (12) \\ x_s &\in \{0,1\}, \text{ for each } s \in S, \end{aligned}$$

where T denotes a given time horizon. The reduced cost of a machine schedule s is then equal to

$$c'_s = 1 - \sum_{j=1}^{n} a_{js}\lambda_j - \sum_{t=0}^{T} o_{st}\pi_t,$$

where π_t denotes the dual variable corresponding to Constraints 12. The corresponding pricing problem can be minimized using the local search procedure and the time-indexed formulation of [1]. Furthermore, since $\pi_t \leq 0$ $(t = 0, \ldots, T)$, it is readily determined that

$$\left[\sum_{j=1}^{n} \lambda_j + \sum_{t=0}^{T} \pi_t Op_t\right] / (1 - c^*)$$

is an intermediate lower bound on the outcome of the LP-relaxation.

Finally, we can use the time-indexed formulation of [1] to find a solution with value equal to the lower bound, but we have to add constraints to ensure that the required number of operators is no more than the available number at any time

$$\sum_{j=1}^{n} x_{jt} \leq Op_t, \text{ for all } t = 0, \ldots, T-1.$$

3.3 Computational experiments

We tested our hybrid algorithm for the case with one type of resource, unit resource consumption, and variable resource availability over time. We consider the instances from Table 1. Besides the basic scenario with full resource availability, we consider two scenarios for each instance. In the first scenario, there is one pile of dummy jobs (reflecting the resource unavailability) where the pile is located around half of the estimated makespan of the schedule. In the second scenario there are two shorter piles around one third and two third of the estimated makespan, respectively. In both scenarios the maximum amount of unavailable resources is about $\lceil \frac{m}{2} \rceil$. The first scenario is denoted by Hi-T1 and the second by Hi-T2. We report the number of times out of 10 that an optimum was found ('# success'), and we report the average and maximum amount of time in seconds needed for the successful runs ('Avg t' and 'Max t'). For the hybrid algorithm, we denote by ('#LB=OPT') the number of times that we could prove that the lower bound equalled the optimum. Next, we report the average and maximum time needed to find the lower bound for the successful runs ('Avg t LB' and 'Max t LB'). By ('Avg #ILP' and 'Max #ILP'), we denote the number of times that we solved the ILP formulation of the pricing problem; this was conducted after each series of 50 runs of the local search algorithm, since we wanted to find out whether the intermediate lower bound could decide the problem already, and whenever the local search algorithm could not find an improving column. Finally, we report on the increase of the lateness because of resource unavailability (Avg incL and Max incL, both in percentages). Again, the maximal running time is 30 minutes. The results are given in Table 2. Our computational results indicate that the resource unavailability increases the running time of the algorithm but that in most cases the algorithm is still able to solve the problem within 30 minutes. For the largest instances (of type 3), we were able to compute the lower bound but could not complete the ILP within 30 minutes. In most cases the scenario with one pile is more difficult than the one with two piles. Finally, most cases were solved and moreover, for all these

	# success	Avg t	Max t	#LB =OPT	Avg t LB	Max t LB	Avg #ILP	Max #ILP	Avg incL	Max incL
H0	10	30	62	10	27	60	8	43		
H0-T1	9	41	81	9	35	72	20	94	42	69
H0-T2	10	39	71	10	32	62	19	79	27	54
H1	10	191	336	10	108	156	16	45		
H1-T1	9	166	207	9	83	119	13	38	37	45
H1-T2	10	437	1238	10	190	926	15	30	42	52
H2	9	183	302	9	117	217	16	68		
H2-T1	9	297	497	9	137	383	14	20	87	107
H2-T2	10	340	582	10	112	158	11	16	125	155
H3	9	1033	1579	9	534	640	45	78		
H3-T1	6	1393	1736	6	578	730	55	75	29	33
H3-T2	9	1288	1473	9	642	927	52	88	35	40
H4	10	54	173	10	42	153	16	92		
H4-T1	9	76	121	9	56	103	29	91	13	28
H4-T2	9	84	165	9	60	139	34	97	18	37
H5	9	26	77	9	24	76	13	76		
H5-T1	9	61	139	9	47	135	17	46	112	179
H5-T2	10	77	214	10	59	205	10	31	144	258

Table 2. Results of the hybrid algorithm with resource unavailability

cases the lower bound equals the optimum, which emphasizes the strength of our lower bound.

We further have tested the suitability of using the destructive lower bounding technique for filtering the cumulative constraint. Hereto, we conducted some experiments to find out the time it take to test whether a schedule with $L_{\max} \leq L$ can exist for a specific value of L. Given the optimum L^* of the instance, we checked for the first two instances of Table 2 whether a schedule can exist with $L_{\max} \leq L$, where $L = L^*-8, L^*-4, L^*-2, L^*-1, L^*, L^*+1, L^*+2, L^*+4, L^*+8$. Note that we computed each test from scratch. We further have added the time needed to establish the lower bound of L^* in the column with header $LB-time$. The average running times (in milliseconds) are displayed in Table 3.

4 Other extensions

4.1 Set-up times and change-over times

So far, we have assumed that as soon as a machine has finished a job, it can start the next one. In many applications, however, there can be a mandatory delay, which is called a *set-up time* or a *change-over time*. A set-up time just depends on the job that is to be started; the change-over time depends on both the job that is to be started and the job that has just been completed. Here we assume that the change-over times obey the triangle inequality.

We first deal with the set-up times, since this is fundamentally easier than the case with change-over times. The basic idea is to add the set-up time to the

Number	type	L*-8	L*-4	L*-2	L*-1	L*	L*+1	L*+2	L*+4	L*+8	LB-time
0	T1	5743	7646,5	7322,5	10507,5	3664,5	2520	1961,5	1621,5	1273,5	27470
0	T2	5808	4719,5	6689	7419	5734,5	2689	2375,5	1649,5	1314,5	28490,5
1	T1	15536,5	15170,5	17069,5	19793,5	8498	5782,5	4810	3670,5	3059,5	78295
1	T2	13803,5	12979,5	13958	17875,5	14053,5	7781,5	5618,5	3874	3172	127785
2	T1	22630	20362	30830,5	25904,5	13403	8937,5	7379,5	6286,5	3389,5	106328,5
2	T2	25798	24991,5	43552,5	26895,5	10881,5	9245	8698,5	5823,5	4661,5	114240,5
3	T1	123023	407135,5	171078,5	229253,5	60221,5	35801	24943,5	19858	11638	558123
3	T2	85441,5	125920,5	144617,5	137556,5	161999	36871,5	27514	17564,5	11821,5	456557,5
4	T1	135,5	10	9	10	5240	3519	2516,5	1949	1482,5	11430,5
4	T2	670,5	10	9	12875,5	15498,5	3372,5	2822,5	2368	1666	45079
5	T1	6038,5	6278,5	6654,5	15213	7634,5	2759,5	2053	1807	1323	42497
5	T2	6211,5	6720,5	8016	8405,5	2889	2650	2232	1430	1191	36518

Table 3. Running times for testing feasibility

processing time; we then consider the first part of processing the job as setting it up. We must then update the release date by subtracting the set-up time from it, which might lead to a negative release date. We may further have to update the right-hand-sides of the generalized precedence constraints, but this is simply a matter of administration. An optimal solution for the problem with set-up times is then readily obtained from the optimal solution for the adjusted instance without set-up times.

Sequence-dependent change-over times are much harder. We incorporate this type of constraint in the column generation: we look for single machine schedules that obey the release dates, deadlines, and the change-over times. This implies that the ILP formulation remains the same; we only must add another constraint to the pricing problem. It is easily dealt with in the local search procedure that Van den Akker et al. use to solve the pricing problem approximately, but it cannot be incorporated in the time-indexed formulation to solve the pricing problem. If we want to solve the pricing problem then, we might use branch-and-bound. Moreover, we cannot use the time-indexed formulation of [1] to find an optimal solution. Very recently, Pereira Lopes and Valério de Carvalho [10] have presented a branch-and-price algorithm for this problem, but with an additive objective function.

4.2 Machine unavailability and planned maintenance

Machine unavailabilities are similar to varying resource availabilities, but they are more restrictive, since we put a label on a machine with its unavailability pattern instead of aggregating the capacities of all machines. One way to tackle this problem is to label the machines and determine for each one a separate set of machine schedules, from which we must select one. An alternative and quicker way is to add dummy jobs to the instance which correspond to unavailabilities. In a correct solution, we will have for each unavailability pattern that a feasible machine schedule will be selected that contains the dummy jobs corresponding to this unavailability pattern, which gives us a schedule for the corresponding machine. In case of a planned maintenance, we know that the machine is being repaired for a given time, but we do not know when this time period starts: we then give the dummy job a release date and deadline corresponding to the earliest start time and the latest completion time of the repair. The only difficulty left is to ensure that a given set of dummy jobs corresponding to the unavailabilities and repairs of a given machine all end up in the same, selected machine schedule. Just like in the previous subsection, we put these constraints in the pricing problem. These additional constraints to a machine schedule are easily being dealt with in the local search procedure. When we want to solve the pricing problem to optimality, we can use the time-indexed formulation, but we must add a constraint for each pair of jobs that must be executed on the same machine or on different machines: if J_i and J_j are to be executed on the same machine, then we add the constraint

$$\sum_{t=r_i}^{\bar{d}_i - p_i} x_{it} = \sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt};$$

if J_i and J_j must go on different machines, then we require

$$\sum_{t=r_i}^{\bar{d}_i - p_i} x_{it} + \sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt} \leq 1.$$

Note that we do not have to solve a pricing problem for each machine separately. Since each job has to be executed, there will be one machine 'executing' the set of dummy jobs that we introduced to mimic the unavailability pattern of this machine. Unfortunately, after having determined the lower bound, we cannot straightaway use the time-indexed formulation of [1] to look for a solution with equal value, since we must force the set of dummy jobs representing the machine unavailability pattern on one machine that does not execute any other dummy job. We can use a similar formulation in which we distinguish between the machines by using variables x_{ijt} indicating that job J_j starts at time t on machine i, but this will blow up the model tremendously, since we cannot aggregate the machines and require that at most m are used then anymore.

5 Conclusions and future research

We have described how the framework by Van den Akker et al. [1] can be used to solve a number of basic resource project scheduling problems. We further have shown how to incorporate change-over times and machine maintenance. Except for the case with change-over times, we can use the same tool kit as in [1] to compute the lower bound. This lower bound always coincided with the optimum in the computational experiments conducted in [1], and we found the same phenomenon in our experiments for the case of the strongly related problem with a varying amount of resources available. We are working on more elaborate computational experiments including other cases. When it comes to finding a solution with value equal to the lower bound, we can in many cases use the time-indexed formulation of [1] in which we specify the wanted optimum beforehand. Van den Akker et al. conjectured that this is presumably due to the preprocessing step within CPLEX, which suggest that the technique of constraint programming should be able to find such a solution more quickly, or show that it does not exist. Constraint programming seems to be the most eminent candidate to look for a solution with value equal to the lower bound for the problems with machine unavailabilities and change-over times. This is one of the directions that we work on.

References

1. J.M. van den Akker, J.A. Hoogeveen, and J.W. van Kempen (2006). Parallel machine scheduling through column generation: minimax objective functions (extended abstract). Y. Azar and T. Erlebach (Eds.) *ESA 2006*. LNCS 4168, Springer, 648–659.
2. J.M. van den Akker, J.A. Hoogeveen, and S.L. Van de Velde (2005). Applying column generation to machine scheduling. G. Desaulniers, J. Desrosiers, and M.M. Solomon (eds.). *Column Generation*, Springer, 303–330.
3. P. Baptiste, C. Le Pape, and W. Nuijten (2001). *Constraint-based scheduling*: Applying constraint programming to scheduling problems. Kluwer Academic Publishers, Dordrecht, The Netherlands.
4. M.S. Bazaraa, J.J. Jarvis, and H.D. Sherali (1990). *Linear Programming and Network Flows*, Wiley, New York.
5. N. Beldiceanu and S. Demassey (2007). *Global Constraint Catalog* www.emn.fr/x-info/sdemasse/gccat/index.html
6. P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch (1999). resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research 112*, 3–41.
7. P. Brucker and S. Knust (2000). A linear programming and constraint propagation-based lower bound for the RCPSP. *European Journal of Operational Research 127*, 355–362.
8. P. Brucker and S. Knust (2003). Lower bounds for resource-constrained project scheduling problems. *European Journal of Operational Research 149*, 302–313.
9. A. Cesta, A. Oddi, and S.F. Smith (2002). A constraint-based method for project scheduling with time windows. *Journal of Heuristics 8*, 109–136.
10. M.J. Pereira Lopes and J.M. Valério de Carvalho (2007). A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times. *European Journal of Operational Research 176*, 1508–1527.