Using column generation to solve parallel machine scheduling problems with minmax objective functions

J.M. van den Akker · J.A. Hoogeveen ·
J.W. van Kempen

Published online: 11 August 2010
© The Author(s) 2010. This article is published with open access at Springerlink.com

Abstract In this paper we consider the parallel machine scheduling problem of minimizing an objective function of the minmax type, like maximum lateness, subject to release dates, deadlines, and/or generalized precedence constraints. We use a destructive strategy to compute a lower bound. Here we test the feasibility of a decision problem by applying column generation to compute a bound on the number of machines that we need to feasibly accommodate all jobs.

After having derived the lower bound, we try to find a matching upper bound by identifying a feasible schedule with objective function value equal to this lower bound. Our computational results show that our lower bound is so strong that this is almost always possible. We are able to solve problems with up to 160 jobs and 10 machines in 10 minutes on average.

Keywords Column generation · Maximum lateness · Release dates · Precedence constraints · Time-indexed formulation

1 Introduction

In this paper we consider one of the basic problems in scheduling and project management. We are given m parallel, identical machines, which are continuously available from time zero onwards and can process no more than one job at a time; these machines have to process n jobs, which are denoted by $J_1, \dots, J_n$. Processing J_j requires one, arbitrary processor during an uninterrupted period of length p_j , which period must start at or after the given release date r_j and must be completed by the given deadline $\bar{d}_j$. Given a schedule σ , we denote the completion time of job J_j by $C_j(\sigma)$, and hence, we need for all jobs J_j that $r_j + p_j \leq C_j(\sigma) \leq \bar{d}_j$ for σ to be feasible. Moreover, the jobs may be subject to generalized precedence constraints, which prescribe that for a pair of jobs J_i and J_j the difference in completion time $C_j(\sigma) - C_i(\sigma)$ should be at least (at most, or exactly) equal to some given value q_{ij} . The quality of the schedule is measured by some objective function of minmax type, which is assumed to be nondecreasing in the completion times, like maximum lateness or maximum cost. Here, the maximum lateness is defined as $\max_j L_j(\sigma)$, where $L_j(\sigma) = C_j(\sigma) - d_j$; d_j signals the due date, by which the job preferably should be completed. A special case occurs when all due dates are equal to zero; in this case, the objective function becomes equal to minimizing the maximum completion time, that is, the makespan of the schedule.

We solve these problems by applying the technique of column generation. This approach has been shown to work very well for the problem of minimizing total weighted completion time on a set of identical parallel machines (see van den Akker et al. 1999 and Chen and Powell 1999). Since the appearance of these papers, the technique of column generation has been applied to many parallel machine problems

J.M. van den Akker · J.A. Hoogeveen (✉)
Department of Information and Computing Sciences, Utrecht
University, P.O. Box 80089, 3508 TB Utrecht, The Netherlands
e-mail: slam@cs.uu.nl

J.M. van den Akker
e-mail: marjan@cs.uu.nl

J.W. van Kempen
KLM Royal Dutch Airlines, Pricing and Revenue Management
(AMS/RX), P.O. Box 7700, 1117 ZL Schiphol Airport,
The Netherlands
e-mail: Jules-van.Kempen@klm.com

with a sum type criterion in which the jobs are known to follow a specific order on the individual machines; we refer to van den Akker et al. (2005) for an overview. One notable exception is by Brucker and Knust (2000, 2002, 2003), who apply column generation to a number of resource-constrained project scheduling problems in which the goal is to minimize the makespan. They first formulate the problem as a decision problem and then use linear programming to check whether there exists a feasible schedule in which preemption is allowed; here the decision variables refer to the length of a time slice during which a given set of jobs is executed simultaneously.

We use the three-field notation scheme introduced by Graham et al. (1979) to denote scheduling problems. The remainder of this chapter is organized as follows. In Sect. 2, we describe the basic approach for the relatively simple problem of minimizing $L_{\max}$ without release dates and generalized precedence constraints. We explain the column generation approach and the derivation of an intermediate lower bound in Sects. 3 and 4. In Sects. 5 and 6, we add release dates and generalized precedence constraints. In Sect. 7, we describe our local search algorithm to solve the pricing problem approximately, and in Sect. 8 we formulate the pricing problem as a time-indexed integer linear programming problem that can be used to find (an upper bound on) the solution of the pricing algorithm. Finally, in Sect. 9 we describe our computational experiments and draw some conclusions.

Our contribution We give the first algorithm for solving these kinds of problems using column generation. Our approach is comparable to the approach by Brucker and Knust, but they work with sets of jobs that are executed at the same time, whereas we work with sets of jobs that are executed by the same machine. We further describe an efficient way to use an intermediate lower bound such that we conclude infeasibility of the decision problem without having to solve the LP-relaxation to optimality.

2 The basic approach

In this section, we sketch the basic approach, which we illustrate on the $P \parallel L_{\max}$ problem, that is, we assume that all jobs are available at time zero. It is well known that this optimization problem can be solved by answering a set of decision problems, which are obtained by putting an upper bound L on the value of the objective function. Since the restriction $L_{\max} \leq L$ is equivalent to the constraint that $L_j = C_j - d_j \leq L$ for each job J_j , we find that $C_j \leq d_j + L \equiv \bar{d}_j$; the decision problem is then to determine whether there exists a feasible schedule that meets all deadlines, where we take the minimum of the original deadline and the deadline induced by the constraint $L_{\max} \leq L$. Hence, we can solve

the optimization problem by determining the smallest value L that allows a feasible schedule.

Since the machines are identical, 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 that meets all deadlines?* Checking the feasibility of a subset is easy by executing the jobs in order of earliest deadline and see whether these are all met (Jackson 1955); we call this the Jackson order. Note that it is identical to the earliest due date (EDD) order if the original deadlines are no smaller than the induced deadlines. We solve this decision problem by answering the question: what is the minimum number of machines that we need to get a feasible schedule? Or equivalently, into how many feasible single-machine schedules do we have to partition the jobs?

Given a subset of jobs, it is easy to find the corresponding single-machine schedule by putting the jobs in Jackson order. Therefore, we also call a subset of jobs that allows a feasible single-machine schedule a *machine schedule*. The above question is then to find the minimum number of mutually distinct machine schedules that contain all jobs. We can formulate the above problem as an integer linear programming problem as follows. Let S be the set containing all machine schedules. We introduce binary variables x_s ($s = 1, \dots, |S|$) that take value 1 if machine schedule s is selected and 0 otherwise. Each machine schedule s is encoded by a vector $a_s = (a_{1s}, \dots, a_{ns})$, where $a_{js} = 1$ if machine schedule s contains job J_j and $a_{js} = 0$, otherwise. We have to minimize the number of machine schedules that we select, such that each job is contained in one machine schedule. Hence, we have to determine values x_s that solve the problem

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

subject to

$$\sum_{s \in S} a_{js} x_s = 1, \quad \text{for each } j = 1, \dots, n, \quad (1)$$

$$x_s \in \{0, 1\}, \quad \text{for each } s \in S. \quad (2)$$

We obtain the linear programming relaxation by replacing conditions (2) by the conditions $x_s \geq 0$ for all $s \in S$; we do not need to enforce the upper bound of 1 for x_s , since this follows immediately from the conditions (1). We solve the LP-relaxation using column generation.

3 Column generation

We first solve the linear programming relaxation for a small initial subset of the columns. Given the solution to the linear

programming problem with the current set of variables, it is well known from the theory of linear programming (see for instance Bazaraa et al. 1990) that the reduced cost of a variable x_s is given by

$$c'_s = c_s - \sum_{j=1}^n \lambda_j a_{js} = 1 - \sum_{j=1}^n \lambda_j a_{js},$$

where $\lambda_1, \dots, \lambda_n$ are the dual multipliers corresponding to the constraints (1) of the solution of the current LP. If for each variable x_s we have that $c'_s \geq 0$, then the solution with the current set of variables solves the linear programming problem with the complete set of variables as well. To check whether this condition is fulfilled, we minimize the reduced cost over all machine schedules. Therefore, we must pick the subset of the jobs with maximum total dual multiplier value among all subsets of jobs that lead to a feasible single-machine schedule, that is, we must solve the problem

$$\max_{s \in S} \sum_{j=1}^n \lambda_j a_{js}. \quad (3)$$

We use $\hat{c}$ to denote the outcome value of this problem; hence, we have that the minimum reduced cost, which we denote by c^* , is equal to

$$c^* = 1 - \hat{c}.$$

This maximization problem is equivalent to the problem of minimizing the total weight of the jobs that are not selected, which is known as the problem of minimizing the weighted number of tardy jobs, where the weight of a job is equal to the dual multiplier λ_j and the due date for each job is equal to the deadline $\bar{d}_j$. Note that the constraint that each weight is nonnegative in this scheduling problem is not restrictive, since a job with negative weight will never be selected in the maximization problem. This problem, which is denoted as $1 \parallel \sum w_j U_j$ in the three-field notation scheme, is solvable in $O(n \sum p_j)$ time by the dynamic programming algorithm of Lawler and Moore (1969). Hence, in this situation we solve the pricing problem to optimality. If $c^* \geq 0$, then we have solved the linear programming relaxation; otherwise, we add the variable with minimal reduced cost value to the LP and solve it again. In this way, we solve the linear programming relaxation to optimality. If the outcome value is more than m , then we know that the answer to the decision problem is ‘no’; if the outcome value is no more than m , and we have not identified a feasible solution yet that uses m (or fewer) machines, then we solve the integer linear programming problem to optimality using the branch-and-bound algorithm developed by Van den Akker et al. (1999) for the problem $P \parallel \sum w_j C_j$.

4 An intermediate lower bound

In the above implementation we have to apply column generation to the bitter end, that is, until we have concluded that the linear programming relaxation has been solved to optimality, before we have found a valid lower bound. Since we only need to know whether the outcome value is more than m or no more than m , we are not interested in the exact outcome value, as long as it allows us to decide the decision problem. We therefore compute an *intermediate lower bound*. This procedure works as follows (see also Bazaraa et al. 1990 for a general description of this principle).

Since c^* is the outcome value of the pricing problem, we know that for each $s \in S$ we have

$$1 = c_s = c'_s + \sum_{j=1}^n \lambda_j a_{js} \geq c^* + \sum_{j=1}^n \lambda_j a_{js}.$$

We use this expression to find a lower bound for the objective function of the linear programming relaxation as follows

$$\begin{aligned} \sum_{s \in S} x_s &\geq \sum_{s \in S} \left(c^* + \sum_{j=1}^n \lambda_j a_{js} \right) x_s \\ &= c^* \sum_{s \in S} x_s + \sum_{s \in S} \sum_{j=1}^n \lambda_j a_{js} x_s \\ &= c^* \sum_{s \in S} x_s + \sum_{j=1}^n \lambda_j \left[\sum_{s \in S} a_{js} x_s \right] \\ &= c^* \sum_{s \in S} x_s + \sum_{j=1}^n \lambda_j, \end{aligned}$$

where we use constraint (1). Therefore, we find that

$$(1 - c^*) \sum_{s \in S} x_s \geq \sum_{j=1}^n \lambda_j.$$

Since $1 - c^* = \hat{c}$, which value is larger than 1, as $c^* < 0$, we find that

$$\sum_{s \in S} x_s \geq \sum_{j=1}^n \lambda_j / \hat{c}.$$

This gives the desired intermediate lower bound, which we can use to decide whether m machines are sufficient. If this lower bound has value smaller than or equal to m , then we continue with solving the LP-relaxation.

Solving the problem $P \parallel L_{\max}$ was relatively simple, since each machine schedule can be represented by just listing the indices of the jobs that it contains, and since the column generation problem can be solved by applying dynamic programming. We can use the same methodology to

solve any problem for which putting an upper bound on the objective function results in a set of deadlines. Hence, we can solve the more general $P \parallel f_{\max}$ problem in the same fashion, where $f_{\max}$ denotes maximum cost. This is defined as $\max_j f_j(C_j)$, where $f_j(t)$ is the cost function of job J_j , which is assumed to be nondecreasing in t . In this case, the deadline $\bar{d}_j$ for each job J_j ($j = 1, \dots, n$) is the largest integral value of t such that $f_j(t) \leq F$. This value t can easily be computed if the function $f_j(t)$ has an inverse, and we can find it using binary search otherwise. Since the problem $P|r_j|C_{\max}$ is the mirror problem of $P \parallel L_{\max}$, we can solve this problem through our basic approach as well.

5 Including release dates: $P|r_j|L_{\max}$

In this section we assume that next to the deadlines, which may come from putting an upper bound on $L_{\max}$, there are release dates as well. When we follow the basic approach that we worked out for $P \parallel L_{\max}$, then we need to find a set of at most m machine schedules that contain all jobs and that obey all release dates and deadlines. Since we have both release dates and deadlines, we cannot easily define the corresponding machine schedule if we know the set of jobs it consists of. Therefore, we represent a machine schedule s by listing the completion times of the jobs that it contains next to the vector a_s , which has $a_{js} = 1$ if job J_j is contained and $a_{js} = 0$, otherwise.

The problem of minimizing the number of machines that we need can then be formulated as an integer linear programming problem, which is the same as the one in Sect. 2. Hence, when we solve the problem using column generation, we get the same pricing problem, but now we have to construct a machine schedule with maximum weight that obeys the release dates and deadlines. This boils down to the well-known scheduling problem $1|r_j|\sum w_j U_j$, which is $\mathcal{NP}$ -hard in the strong sense. We do not want to solve this to optimality, and therefore we introduce in Sect. 7 a local search procedure to find an approximately optimal solution to the pricing problem, which we use in our column generation. Note that the introduction of the release dates in the problem does not destroy the validity of the intermediate lower bound, but to compute it, we need the optimum $\hat{c}$ of the maximization problem (3), and not just a lower bound on this, which we find using local search. To be able to use the intermediate lower bound, we describe in Sect. 8 how to compute an upper bound on $\hat{c}$, which value we can fill in for $\hat{c}$ to find a feasible lower bound on the number of machines that we need to feasibly accommodate all jobs. Before showing the details of the local search and the upper bound, we first discuss how to tackle the problem in which there are generalized precedence constraints as well.

6 The full problem

In this section, we assume that there are release dates, deadlines, and generalized precedence constraints. We again translate the problem into one of minimizing the number of machine schedules that are needed. Since two jobs that are connected through a precedence constraint do not have to be executed by the same machine, we assume that the machine schedules obey the release dates and deadlines, and we include a constraint in the integer linear programming formulation for each of the generalized precedence constraints. We define $A1$ as the arc set containing all pairs (i, j) such there exists a precedence constraint of the form $C_j - C_i \geq q_{ij}$; similarly, we define $A2$ and $A3$ as the arc sets that contain an arc for each pair (i, j) , for which $C_j - C_i \leq q_{ij}$ and $C_j - C_i = q_{ij}$, respectively. To avoid modeling the same relation in both $A1$ and $A2$, we assume that $q_{ij} \geq 0$. Note that the intersection of $A1$ and $A2$ does not have to be empty, since there can be a lower and upper bound on the difference of two completion times. We denote the union of $A1$, $A2$, and $A3$ by the multiset A . This leads to the following integer linear programming formulation:

$$\begin{aligned} \min \quad & \sum_{s \in S} x_s \\ \text{subject to} \quad & \sum_{s \in S} a_{js} x_s = 1, \quad \text{for each } j = 1, \dots, n, \\ & \sum_{s \in S} C_{js} x_s - \sum_{s \in S} C_{is} x_s \geq q_{ij}, \quad \text{for each } (i, j) \in A1; \\ & \sum_{s \in S} C_{js} x_s - \sum_{s \in S} C_{is} x_s \leq q_{ij}, \quad \text{for each } (i, j) \in A2; \\ & \sum_{s \in S} C_{js} x_s - \sum_{s \in S} C_{is} x_s = q_{ij}, \quad \text{for each } (i, j) \in A3; \\ & x_s \in \{0, 1\}, \quad \text{for each } s \in S. \end{aligned}$$

Here C_{js} denotes the completion time of job J_j in column s , which we define to be equal to 0 if J_j is not contained in s . If we want to solve the LP-relaxation by applying column generation, then we find that the reduced cost of a machine schedule s is equal to

$$c'_s = c_s - \sum_{j=1}^n a_{js} \lambda_j - \sum_{j=1}^n \left[\sum_{h \in P_j} \delta_{hj} C_{hs} - \sum_{k \in S_j} \delta_{jk} C_{ks} \right].$$

Here the sets P_j and S_j are defined as the sets containing all predecessors and successors of job J_j in A , respectively. Hence, we must solve the maximization problem

$$\sum_{j=1}^n a_{js} \lambda_j + \sum_{j=1}^n \left[\sum_{h \in P_j} \delta_{hj} C_{hs} - \sum_{k \in S_j} \delta_{jk} C_{ks} \right]$$

over all machine schedules $s \in S$. We solve this problem approximately using local search (see Sect. 7). Again, we can compute an intermediate lower bound. Let $\hat{c}$ denote the outcome value of the maximization problem, and let c^* denote the minimum reduced cost. Hence, we find that

$$c_s \geq c^* + \sum_{j=1}^n a_{js} \lambda_j + \sum_{j=1}^n \left[\sum_{h \in P_j} \delta_{hj} C_{js} - \sum_{k \in S_j} \delta_{jk} C_{js} \right].$$

We plug this in, and use that

$$\begin{aligned} & \sum_{j=1}^n \left[\sum_{h \in P_j} \delta_{hj} C_{js} x_s - \sum_{k \in S_j} \delta_{jk} C_{js} x_s \right] \\ &= \sum_{(j,k) \in A} \delta_{jk} [C_{ks} x_s - C_{js} x_s]. \end{aligned}$$

We then obtain

$$\begin{aligned} \sum_{s \in S} c_s x_s &\geq c^* \sum_{s \in S} x_s + \sum_{j=1}^n \lambda_j \left[\sum_{s \in S} a_{js} x_s \right] \\ &\quad + \sum_{(j,k) \in A} \delta_{jk} \left[\sum_{s \in S} C_{ks} x_s - C_{js} x_s \right] \\ &\geq c^* \sum_{s \in S} x_s + \sum_{j=1}^n \lambda_j + \sum_{(j,k) \in A} \delta_{jk} q_{jk}, \end{aligned}$$

where we use that in case of a constraint with $\geq$ sign the dual multiplier has value ≥ 0 , whereas in case of a constraint with $\leq$ sign the dual multiplier is ≤ 0 . Hence, we may conclude that

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

is an intermediate lower bound on the number of machines that we need.

If we get stuck, that is, the outcome of the LP-relaxation does not lead to ‘no’ on the decision problem, then we assume for the time-being that the decision problem is feasible, and we decrease the upper bound L on $L_{\max}$ that we want to test. If the upper bound $L - 1$ yields an infeasible decision problem and, if we cannot decide the feasibility of the decision problem with upper bound L , then we apply branch-and-bound. Initially, we used the branching strategy developed by Carlier (1987) of splitting the execution interval $[r_j, \bar{d}_j]$ into two disjunct intervals $[r_j, Q]$ and $[Q - p_j + 1, \bar{d}_j]$, where Q is some time point in the middle of the interval $[r_j, \bar{d}_j]$. It turned out in our experiments,

however, that it is better to solve an integer linear programming formulation in which we request that $L_{\max} = L$; we work this out in Sect. 9.

7 Generating new columns by local search

In this section, we describe the local search algorithm that we have implemented to solve the pricing problem, which is the problem of finding the feasible single-machine schedule s that minimizes

$$1 - \sum_{j=1}^n a_{js} \lambda_j - \sum_{j=1}^n \left[\sum_{h \in P_j} \delta_{hj} C_{js} - \sum_{k \in S_j} \delta_{jk} C_{js} \right]. \quad (4)$$

Solving the original pricing problem is equivalent to finding the single-machine schedule that obeys the release dates and deadlines and maximizes

$$\sum_{j=1}^n \lambda_j a_{js} + \sum_{j=1}^n C_{js} \left[\sum_{h \in P_j} \delta_{hj} - \sum_{k \in S_j} \delta_{jk} \right]. \quad (5)$$

If we define $Q_j = \sum_{h \in P_j} \delta_{hj} - \sum_{k \in S_j} \delta_{jk}$, then we have to maximize

$$\sum_{j=1}^n \lambda_j a_{js} + \sum_{j=1}^n Q_j C_{js}. \quad (6)$$

Looking at this formula we see that, if job J_j gets selected, then this Q_j value determines whether it is more profitable to execute the job as late as possible ($Q_j > 0$) or as early as possible ($Q_j < 0$). In a preprocessing step, we can even determine the time interval during which we must complete job J_j to make its selection profitable.

In our local search we use a two-phase procedure. In the first phase, we determine the jobs that are selected and the order in which they are executed. In the second phase, we then determine the optimal set of completion times. We first discuss the procedure used in the second phase. Since the values Q_j are fixed, we can solve this subproblem in linear time using the following shifting procedure, which resembles the procedure for a similar problem given by Garey et al. (1988).

Procedure: Optimize completion times

1. Consider the selected jobs in the given order and place all jobs as early as possible, that is, each job is started at the maximum of its release date and the completion time of its predecessor.
2. Look at the jobs in reversed order. Suppose that we are currently working on job J_j . If $Q_j > 0$, we will delay the job until either
 - (a) Job J_j reaches its deadline: we put $C_j = \bar{d}_j$.

- (b) Job J_j hits its successor: J_j is *glued* to its successor J_k (which may already have been glued to some of its successors) to form a new job J_{new} with $Q_{\text{new}} = Q_j + Q_k$ and the new deadline is the deadline that is reached first when this job J_{new} is shifted towards the end of the schedule.
- If $Q_{\text{new}} > 0$, we try to shift J_{new} further to the right as described before.
 - If $Q_{\text{new}} \leq 0$, J_{new} stays at this position.

Note that if and only if there is no placement possible for the given selection and order, step (1) will be forced to end a job after its deadline. It is straightforward to show that the above procedure finds the optimum set of completion times for the given selection of jobs and the given order in which these are scheduled. Since in each step we either take a job on an earlier position into consideration, or we glue together some jobs, we are done after $O(n)$ steps, where n is the number of jobs that have been selected. Since each step can be implemented to run in constant time, the algorithm runs in linear time.

We now describe the first step of the local search procedure. We define a solution in our local search as a selection of the jobs and the order in which they should be processed, after which we find the value of this solution by solving the second step. Our local search uses the following methods to exploit the solution space:

- Remove a random job from the set of selected jobs;
- Add a random, yet unselected job at a random place in the order of selected jobs;
- Replace a random job from the current selection by a random, yet unselected job;
- Swap the positions of two random jobs in the set of selected jobs.

In our computational experiments, we added up to 50 columns with negative reduced cost per iteration.

8 Time-indexed formulation

The last item on the list is to find an upper bound for $\hat{c}$, which is defined as the outcome of the maximization problem (5), such that we can compute the intermediate lower bound. To this end, we formulate the problem as an integer linear programming problem using a time-indexed formulation. Here we have variables x_{jt} ($j = 1, \dots, n; t = r_j, \dots, \bar{d}_j - p_j$), which get value 1 if job J_j is started at time t , and value 0 otherwise. The corresponding cost coefficient c_{jt} is easily determined. We need constraints to enforce that each job is started at most once and that the machine executes at most

one job at a time. The corresponding ILP-formulation is

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

subject to

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

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

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

Here T denotes the latest point in time at which at least two jobs can be executed. In our experiments, we computed this upper bound every 50 iterations, or when our local search algorithm could not find any column with negative reduced cost. It is well known (see for instance de Sousa and Wolsey (1992) and van den Akker (1994)) that solving the LP-relaxation gives a very strong upper bound if *each job must be executed*. In our case, however, we can (partly) select jobs. In some cases, the possibility of executing a job partly decreases the quality of the upper bound, which disables the intermediate lower bound to decide the feasibility problem. If in this situation our local search algorithm cannot find a column with negative reduced cost, then we get stuck. To avoid this, we then turn to the original ILP-formulation of the pricing problem. We compute for which value of the objective function of the pricing problem we find an intermediate lower bound equal to m . We then ask our ILP solver CPLEX whether there exists a solution to the pricing problem with this value or larger. If the answer to this decision problem is ‘no’, then we can conclude that m machines are not enough; if the answer is ‘yes’, then we add the corresponding column and continue with solving the LP-relaxation by column generation. Therefore, we do not have to solve the ILP-formulation of the pricing problem to optimality, which is convenient, since CPLEX has the option of putting the emphasis on feasibility. It turns out that the increased running time of solving the ILP instead of the LP does not add significantly to the total running time, for the number of ILP solves needed is limited.

9 Computational results

9.1 Comparing methods

Since we could not find any computational results for the problem $P|r_j, \text{prec}|L_{\max}$ in the literature, we have compared our method to the rather straightforward approach of

solving the following time-indexed ILP-formulation of this problem, which very closely resembles the one stated in Sect. 8.

min L

subject to

$$\begin{aligned} \sum_{t=r_j}^{\bar{d}_j-p_j} x_{jt} &= 1, \quad \text{for each } j = 1, \dots, n; \\ \sum_{j=1}^n \sum_{t=t'-p_j+1}^{t'} x_{jt'} &\leq m, \quad \text{for each } t' = 0, \dots, T; \\ \sum_{t=r_j}^{\bar{d}_j-p_j} x_{jt} + p_j - d_j &\leq L, \quad \text{for each } j = 1, \dots, n; \\ \sum_{t=r_j}^{\bar{d}_j-p_j} x_{jt} - \sum_{t=r_i}^{\bar{d}_i-p_i} x_{it} &\geq q_{ij}, \quad \text{for each } (i, j) \in A1; \\ \sum_{t=r_j}^{\bar{d}_j-p_j} x_{jt} - \sum_{t=r_i}^{\bar{d}_i-p_i} x_{it} &\leq q_{ij}, \quad \text{for each } (i, j) \in A2; \\ \sum_{t=r_j}^{\bar{d}_j-p_j} x_{jt} - \sum_{t=r_i}^{\bar{d}_i-p_i} x_{it} &= q_{ij}, \quad \text{for each } (i, j) \in A3; \\ x_{jt} &\in \{0, 1\}, \\ &\text{for each } j = 1, \dots, n \text{ and each } t = 0, \dots, T. \end{aligned}$$

To reduce the number of variables in the above formulation, we only allow L to be smaller than or equal to some value of L' for which a feasible solution has been found through some heuristic; this is achieved by using deadlines $\bar{d}_j$ that are equal to $d_j + L'$. Moreover, we check the number of jobs that are being executed at time t' only until we reach the time point at which we are sure that no more than m jobs can be executed simultaneously.

Originally, we had in mind to apply the column generation lower bound in a branch-and-bound algorithm with a branching rule based on splitting the execution intervals. When we compared this method to the method of solving the time-indexed ILP-formulation through CPLEX, we noticed that these methods have different strengths and weaknesses. The lower bound is very strong (it almost always coincided with the optimum), but the branch-and-bound algorithm had problems to find a schedule with $L_{\max}$ value equal to the lower bound. CPLEX, on the other hand, can ‘easily’ find an optimal solution, but showing optimality is a big problem. Therefore, we tried to exploit the best of both worlds in defining a *hybrid method*. We first compute our lower bound LB on $L_{\max}$ through column generation. We then pass this

information to the time-indexed formulation by adding the constraint $L = LB$ and ask CPLEX for a solution. As a nice side-effect, we can ignore a big part of the variables in our ILP-formulation.

9.2 Results

In our experiments we compare our hybrid algorithm 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 13 scenarios; for each scenario we ran ten test instances. The scenarios are described in Table 1; n denotes the number of jobs, m the number of machines, and #prec denotes the number of precedence constraints.

The first 8 scenarios are used to compare our hybrid algorithm to letting CPLEX solve the ignorant ILP. The scenarios 8–10 are used to find out the influence of the number of machines, whereas in the last two scenarios the influence of a doubling of the times value is measured. The results of the experiments are summarized in Table 2. The results of the hybrid algorithm are denoted in the row starting with Hi, where i denotes the number of the scenario; the results of applying CPLEX (Version 9.0) to the ignorant ILP-formulation appear in the row starting with Ci. The algorithms were encoded in Java (Version 1.4.2_05) and the experiments were run on a Dell Optiplex GX270 P4 2,8 Ghz computer. For each instance we let each algorithm run for at most 30 minutes. We keep track of 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 have gathered some more information. By (‘#LB=OPT’) we denote the number of times that we could prove that the lower bound equalled the optimum; we denote the proven difference between the optimum and the lower bound (‘Max dif’). 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’). Finally, 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.

We also tested the performance of our local search algorithm on the pricing problem by comparing it to the method of generating columns by solving the ILP of the pricing problem in each iteration. For the scenarios 0, 5, 11 and 12 we ran 10 instances each. We determined the lower bound on these instances by always using our local search algorithm

Table 1 Test scenarios

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[0,80]	U[80,150]	80	7	30
3	U[1,20]	U[0,40]	U[60,80]	100	9	40
4	U[1,20]	U[0,60]	U[80,110]	120	9	50
5	U[1,20]	U[0,60]	U[80,110]	140	10	50
6	U[1,20]	U[0,60]	U[80,110]	160	10	50
7	U[1,20]	U[0,60]	U[80,110]	180	10	60
8	U[1,20]	U[0,60]	U[40,80]	60	3	30
9	U[1,20]	U[0,60]	U[40,80]	60	5	30
10	U[1,20]	U[0,60]	U[40,80]	60	7	30
11	U[1,20]	U[0,60]	U[50,80]	30	3	15
12	U[1,40]	U[0,120]	U[100,160]	30	3	15

Table 2 Results of comparing CPLEX and the hybrid algorithm

	# success	Avg t	Max t	#LB=OPT	Max dif	Avg t LB	Max t LB	Avg #ILP	Max #ILP
H0	10	30	62	10	0	26	60	8	43
C0	8	145	1005						
H1	10	191	336	10	0	107	155	16	45
C1	0	–	–						
H2	10	28	34	10	0	8	12	0	0
C2	10	2	5						
H3	9	183	302	9	0	117	217	16	68
C3	0	–	–						
H4	10	372	503	10	0	246	423	70	177
C4	0	–	–						
H5	9	391	497	9	0	237	340	46	126
C5	0	–	–						
H6	10	620	864	10	0	331	452	36	62
C6	0	–	–						
H7	9	1033	1579	9	0	533	640	45	78
C7	0	–	–						
H8	9	471	840	9	0	258	560	64	169
C8	0	–	–						
H9	10	54	173	10	0	41	153	16	92
C9	8	74	466						
H10	10	17	39	10	0	9	35	0	0
C10	10	4	24						
H11	9	26	77	9	0	24	75	12	76
C11	10	57	195						
H12	8	229	1149	8	0	86	183	45	87
C12	5	729	1489						

and by using the optimal solutions to the ILP. The results are depicted in Table 3. Here H_i denotes the hybrid algorithm run on scenario i , and I_i denotes the results obtained on sce-

nario i by the algorithm in which the pricing algorithm is solved by the ILP. Scenario 0 is used to show the difference for *easy* instances, whereas scenario 5 is used to investigate

Table 3 Results of comparing LS to only ILP solving

	# success	Average time	Maximum time
H0	10	26	60
I0	10	81	189
H5	10	237	340
I5	0	–	–
H11	9	24	75
I11	9	59	137
H12	8	86	183
I12	8	360	637

difficult instances. Finally scenarios 11 and 12 are used to investigate the influence of the doubling of the time values on the results.

9.3 Evaluation of our experiments

Our results clearly show that our hybrid algorithm outperforms the method of letting CPLEX solve the ignorant ILP by far. CPLEX is not able to solve the ignorant ILP in less than 30 minutes for most of the tested instances, whereas our hybrid algorithm easily solves nearly all instances. Looking at scenario 3 we already see that CPLEX solved no instance with 100 jobs and 9 machines within 30 minutes, whereas our hybrid algorithm solves all instances we tested up to 160 jobs and 10 machines (scenarios 3–6).

Our results also show that 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. It seems reasonable to conjecture that for some of the instances this is 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. If we for instance stopped an instance of C5, then the best upper bound found so far in general was way off from the optimum. This may be explained partly by the reduction in size of the model, but it is most certainly also due to the preprocessing steps performed by CPLEX. 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.

The time needed to solve an instance seems to depend mostly on the number of jobs per machine: if we look at scenarios 8–10, then we can see that problems with 20 jobs per machine get really difficult. However, doubling the time values (scenarios 11 and 12) adds a relatively little increase to

the average time needed to solve an instance, but one problem becomes unsolvable for our hybrid algorithm. But also here our hybrid algorithm shows its merit in comparison to letting CPLEX solve the ignorant ILP, for doubling the times makes CPLEX incapable to solve any of the instances: it does not even find any solution for these instances, which is of course due to the large increase in variables in the ILP model.

Table 2 already shows the quality of our local search algorithm since the number of (costly) ILP solves of the pricing problem is limited and does not seem to depend much on the size or difficulty of the problem. Table 3 further validates that our local search algorithm performs very well, for without the local search algorithm it takes already 3 times as much time to compute the lower bound for *easy* instances. And *difficult* instances even become unsolvable within 30 minutes, whereas we need only a little more than 3 minutes on average to compute the lower bound for these instances when using the local search method. Finally, doubling the time values also doubles the time needed to compute the lower bound when local search is used to solve the pricing problem, whereas solving the pricing problem by ILP quadruples the average time for 4 instances and leads to a failure to compute a lower bound for one instance within 30 minutes.

10 Conclusion and future research

We have described a column generation approach for the parallel machine scheduling problem with minmax objective subject to release dates and precedence constraints. We have tested the algorithm for maximum lateness subject to release dates, but with minimum delay precedence constraints only. We expect (but this is only a guess) that problems with maximum cost are harder to solve, since changing the upper bound on the cost with a small amount does not necessarily change the deadlines of all jobs. We expect that the nature of the precedence constraints does not change the effectiveness of the algorithm. It is an interesting, nontrivial step to extend this algorithm to the case with *uniform*, or even *unrelated* machines.

The next step in the research will be to investigate the natural connection with constraint satisfaction, which for instance can be used to tighten the release dates and deadlines (see for instance the book by Baptiste et al. 2001). This looks like a very promising direction to improve the effectiveness of the algorithm, as already witnessed by the success of the preprocessing phase of the CPLEX algorithm.

Open Access This article is distributed under the terms of the Creative Commons Attribution Noncommercial License which permits any noncommercial use, distribution, and reproduction in any medium, provided the original author(s) and source are credited.

References

- Baptiste, P., Le Pape, C., & Nuijten, W. (2001). *Constraint-based scheduling: applying constraint programming to scheduling problems*. Dordrecht: Kluwer Academic.
- Bazaraa, M. S., Jarvis, J. J., & Sherali, H. D. (1990). *Linear programming and network flows*. New York: Wiley.
- 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.
- Brucker, P., & Knust, S. (2002). Lower bounds for scheduling a single robot in a job-shop environment. *Annals of Operations Research*, 115, 147–172.
- Brucker, P., & Knust, S. (2003). Lower bounds for resource-constrained project scheduling problems. *European Journal of Operational Research*, 149, 302–313.
- Carlier, J. (1987). Scheduling jobs with release dates and tails on identical machines to minimize the makespan. *European Journal of Operational Research*, 29, 298–306.
- Chen, Z. L., & Powell, W. B. (1999). Solving parallel machine scheduling problems by column generation. *INFORMS Journal on Computing*, 11, 78–94.
- de Sousa, J. P., & Wolsey, L. A. (1992). A time-indexed formulation of non-preemptive single-machine scheduling problems. *Mathematical Programming*, 54, 353–367.
- Garey, M. R., Tarjan, R. E., & Wilfong, G. T. (1988). One-processor scheduling with symmetric earliness and tardiness penalties. *Mathematics of Operations Research*, 13, 330–348.
- Graham, R. L., Lawler, E. L., Lenstra, J. K., & Rinnooy Kan, A. H. G. (1979). Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5, 287–326.
- Jackson, J. R. (1955). *Scheduling a production line to minimize maximum tardiness* (Research Report 43). Management Sciences Research Project, UCLA.
- Lawler, E. L., & Moore, J. M. (1969). A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16, 77–84.
- van den Akker, J. M. (1994). *LP-based solution methods for single-machine scheduling problems*. PhD thesis, Eindhoven University of Technology.
- van den Akker, J. M., Hoogeveen, J. A., & Van de Velde, S. L. (1999). Parallel machine scheduling by column generation. *Operations Research*, 47, 862–872.
- van den Akker, J. M., Hoogeveen, J. A., & Van de Velde, S. L. (2005). Applying column generation to machine scheduling. In G. Desaulniers, J. Desrosiers, & M. M. Solomon (Eds.), *Column generation* (pp. 303–330). Berlin: Springer.