ORSA Journal on Computing
Vol. 5, No. 2, Spring 1993

0899-1499/93/501-192 $01.25

Duality-Based Algorithms for Scheduling Unrelated Parallel Machines

S. L. VAN DE VELDE / *Department of Mechanical Engineering, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands, phone: +31-53-892583; Email: s.l.vandevelde@wb.utwente.nl*

(Received: May 1990; final revision received: May 1991, January 1992; accepted: January 1992)

We consider the following parallel machine scheduling problem. Each of n independent jobs has to be scheduled on one of m unrelated parallel machines. The processing of job J_j on machine M_i requires an uninterrupted period of positive length p_{ij}. The objective is to find an assignment of jobs to machines so as to minimize the maximum job completion time. The objective of this paper is to design practical algorithms for this $\mathcal{NP}$-hard problem. We present an optimization algorithm and an approximation algorithm that are both based on surrogate relaxation and duality. The optimization algorithm solves quite large problems within reasonable time limits. The approximation algorithm is based upon a novel concept for iterative local search, in which the search direction is guided by surrogate multipliers.

We consider the following machine scheduling problem. There are m parallel machines available for processing a set of n independent jobs $\mathcal{J} = \{J_1, \ldots, J_n\}$. Each of these machines can handle at most one job at a time. The processing of job J_j $(j = 1, \ldots, n)$ on machine M_i $(i = 1, \ldots, m)$ requires an uninterrupted period of positive length p_{ij}. We may assume that these processing times are integral. Each job has to be scheduled on one of the machines. A *schedule* is an assignment of each of the jobs to exactly one machine. The length of the schedule, also referred to as the *makespan*, is the maximum job completion time; by definition, the makespan is also equal to the maximum machine completion time. The objective is to find a schedule of minimum length.

This problem has many applications. It arises in computer system scheduling, where the machines are processors of a distributed computing environment with varying capabilities across the tasks. It also arises in scheduling flexible manufacturing systems. For instance, a cluster of parallel machines may form a single or bottleneck stage in the production process. The problem also occurs in the context of machine load balancing, where machines have to be equipped with the appropriate tools for the jobs assigned to them. If production follows a cyclic pattern, and if the *system set-up time* (the time to load the machine with the appropriate tools) is costly relative to production time, then an obvious objective is to minimize the cycle time; note that cycle time minimization is equivalent to throughput maximization. Berrada and Stecke[2] consider such a problem with limited capacities of the tool magazines of the machines.

In case $p_{ij} = p_j$ for each J_j and M_i, the machines are said to be *identical*. If $p_{ij} = p_j/s_i$, where s_i denotes the speed of M_i, then the machines are *uniform*. In the general case, the machines are *unrelated*. Following the notation of Graham, Lawler, Lenstra, and Rinnooy Kan,[14] we refer to these problems as $P\|C_{max}$, $Q\|C_{max}$, and $R\|C_{max}$, respectively.

Since the problem is already $\mathcal{NP}$-hard in case of two identical machines, no polynomial-time algorithm for $R\|C_{max}$ exists unless $\mathcal{P} = \mathcal{NP}$. The traditional dilemma is then to balance solution quality with running time. An optimal solution may only be found at the expense of an exponential amount of computation time; a polynomial-time algorithm cannot be guaranteed to produce an optimal solution.

Two attempts have been made to solve the $R\|C_{max}$ problem to optimality. Stern[25] presents a branch-and-bound algorithm; Horowitz and Sahni[19] develop a dynamic programming procedure. In either case, no computational results are reported.

Much research effort has been invested in the development of approximation algorithms with a guaranteed accuracy. An approximation algorithm that asymptotically never delivers a schedule length of more than ρ times the optimal length is referred to as a ρ-approximation algorithm; such an approximation algorithm is said to have *performance guarantee* ρ. Ibarra and Kim[20] and Davis and Jaffe[5] propose various approximation algorithms with worst-case performance ratios that increase with the number of machines. For *fixed* m (i.e., the number of machines is specified as part of the problem type and not of the problem instance), Horowitz and Sahni[19] give a fully polynomial approximation scheme with time and space complexity

Subject classifications: Production/scheduling: approximation/heuristic; Sequencing: Deterministic; Multiple machine.
Other key words: Programming, Integer, Algorithms, Branch-and-bound, Relaxation.

$O(nm(nm/(\rho-1))^{m-1})$. A *polynomial approximation scheme* is a family of algorithms that contains for any $\rho > 1$ a ρ-approximation algorithm with a running time that is bounded by a polynomial in the problem size; this running time may depend on ρ. A family of algorithms is called a *fully polynomial approximation scheme* if it contains for any $\rho > 1$ a ρ-approximation algorithm for which the running time is bounded by a polynomial in the problem size as well as in $1/(\rho-1)$.

Potts[24] presents a 2-approximation algorithm. Its time requirement is polynomial only for fixed m; its space requirement, however, is polynomial in m. For the 2-machine case, Potts improves the worst-case ratio to $(1+\sqrt{5})/2$. The algorithm is a two-phase procedure. In the first phase, linear programming is used to assign at least $n-m+1$ jobs; in the second phase, complete enumeration is used to schedule the remaining jobs. Using Potts' algorithm as the basis, Lenstra, Shmoys, and Tardos[22] present a 2-approximation algorithm that is polynomial in m. They also present a polynomial approximation scheme for a fixed number of machines, requiring space bounded by a polynomial in the problem size and $\log(1/(\rho-1))$. In addition, they prove a notable negative result: unless $\mathcal{P} = \mathcal{NP}$, no polynomial ρ-approximation algorithm exists for any $\rho < 3/2$.

Two papers consider $R||C_{max}$ from an empirical point of view. De and Morton[6] present several hybrid list scheduling algorithms and perform a large-scale computational testing. Hariri and Potts[18] propose several two-phase heuristics that proceed in the spirit of Potts' 2-approximation algorithm. The first phase is identical: linear programming is used to schedule at least $n-m+1$ jobs. The second phase proceeds differently: a heuristic is used as a substitute for complete enumeration to schedule the remaining jobs. Note that Potts' 2-approximation algorithm dominates such two-phase heuristics in terms of quality but not in terms of speed. Hariri and Potts also consider several constructive heuristics, using them in conjunction with iterative local improvement procedures.

In spite of the considerable attention that the $R||C_{max}$ problem has received, there is still a lack of practical algorithms and computational insight. We address this issue here. We are concerned with methods that solve $R||C_{max}$ satisfactorily from a practical standpoint. We develop an optimization algorithm and an approximation algorithm that are both based on surrogate relaxation and duality. The optimization algorithm, of the branch-and-bound type, solves large problems within reasonable time limits. The approximation algorithm is based upon a novel concept for iterative local search, where the search direction is guided by surrogate multipliers.

The organization of this paper is as follows. In Section 1, we formulate the $R||C_{max}$ problem as an integer linear program, derive the surrogate problem, and examine the surrogate dual problem; this problem is the basis for the optimization algorithm and for the approximation algorithm. In Section 2, we present the approximation algorithm and discuss the relation to Potts' 2-approximation algorithm (Potts[24]). A complete description of the branch-and-bound algorithm is given in Section 3. Some computational results are presented in Section 4. Conclusions are given in Section 5.

1. Minimizing Makespan and Its Dual Problem

Good lower and upper bounds on the minimal schedule length are necessary for a branch-and-bound algorithm to be effective. We derive lower bounds from the surrogate dual problem of $R||C_{max}$, obtained from a 0-1 linear programming formulation. Analyzing the surrogate dual problem, we will show that the search for a good lower bound can almost be integrated with the search for a good upper bound.

Evidently, there is an optimal solution in which the jobs are processed without delay. In addition, the ordering of the jobs on the machines is irrelevant for the length of the schedule. We are therefore actually looking for an *assignment* of jobs to machines. Accordingly, we introduce assignment variables x_{ij} $(i = 1,\ldots,m,\ j = 1,\ldots,n)$ that take the value 1 if J_j is scheduled on M_i, and 0 otherwise. If we let C_i denote the completion time of machine M_i, then we have $C_i = \sum_{j=1}^{n} p_{ij}x_{ij}$. The maximum value of the machine completion times, denoted by C_{max}, is then the length of the schedule.

The $R||C_{max}$ problem, hereafter referred to as problem (P), is to determine values x_{ij} that minimize

$$C_{max} \qquad \text{(P)}$$

subject to

$$\sum_{j=1}^{n} p_{ij}x_{ij} \leqslant C_{max}, \qquad \text{for } i = 1,\ldots,m, \qquad (1)$$

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad \text{for } j = 1,\ldots,n, \qquad (2)$$

$$x_{ij} \in \{0,1\}, \qquad \text{for } i = 1,\ldots,m,\ j = 1,\ldots,n. \qquad (3)$$

The conditions (1) ensure that the completion time of each machine is less than or equal to the length of the schedule; the conditions (2) guarantee that each job is assigned. The conditions (2) and (3) ensure that each job is scheduled on *exactly one* machine, thereby precluding preemption.

Any $\mathcal{NP}$-hard combinatorial optimization problem can be seen as an "easy-to-solve" problem complicated by a number of "nasty" side constraints. A common strategy for lower bound computation is to relax a set of nasty constraints; the resulting problem will be easier to solve and its optimal solution provides a lower bound for the original problem. For instance, replacing the integrality constraints (3) with the weaker conditions $x_{ij} \geqslant 0$ $(i = 1,\ldots,m,\ j = 1,\ldots,n)$ yields the *linear programming relaxation* $(\overline{P})$; this problem is solvable in polynomial time.

Lagrangian relaxation and *surrogate relaxation* are more involved techniques. The idea behind Lagrangian relaxation is to remove a set of nasty side constraints from the set of constraints, and to put them into the objective function, each weighted by a given Lagrangian multiplier. This

is what is referred to as *dualizing* the nasty constraints. The resulting problem is the *Lagrangian problem*. The idea behind surrogate relaxation is to replace a set of nasty constraints with a single condition that is a weighted aggregation of these constraints. The resulting problem is the *surrogate relaxation problem*. In theory, the best surrogate bound is at least as good as the best Lagrangian bound; in practice, the former is much harder to obtain. Greenberg and Pierskalla[15] and Karwan and Rardin[21] compare both relaxation methods.

In our application, we identify the constraints (1) as the nasty constraints. Due to the special structure of problem (P), the resulting Lagrangian problem and the resulting surrogate relaxation problem are exactly the same. In this case, however, surrogate relaxation is easier to perform. Introducing a vector of multipliers $\lambda = (\lambda_1, \ldots, \lambda_m) \geqslant 0$ with $\lambda_i > 0$ for at least one i ($i = 1, \ldots, m$), we replace the conditions (1) with

$$\sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} p_{ij} x_{ij} \leqslant \sum_{i=1}^{m} \lambda_i C_{\max}, \tag{1a}$$

or, equivalently,

$$C_{\max} \geqslant \sum_{i=1}^{m} \sum_{j=1}^{n} \lambda_i p_{ij} x_{ij} \Big/ \sum_{i=1}^{m} \lambda_i. \tag{1b}$$

The surrogate relaxation problem, referred to as problem (S_λ), is then to determine $S(\lambda)$, which is the minimum of

$$\sum_{i=1}^{m} \sum_{j=1}^{n} \lambda_i p_{ij} x_{ij} \Big/ \sum_{i=1}^{m} \lambda_i \tag{S_λ}$$

subject to

$$\sum_{i=1}^{m} x_{ij} = 1, \qquad \text{for } j = 1, \ldots, n, \tag{2}$$

$$x_{ij} \in \{0, 1\}, \qquad \text{for } i = 1, \ldots, m,\ j = 1, \ldots, n. \tag{3}$$

It is a matter of writing out to verify that the Lagrangian problem, obtained by dualizing the conditions (1), gives a worthless lower bound unless $\sum_{i=1}^{m} \lambda_i = 1$. The Lagrangian objective function subsequently boils down to the surrogate objective function; the Lagrangian problem is then exactly the same as the surrogate relaxation problem.

We make now some observations concerning the structure, the properties, and the solution of the surrogate relaxation problem. In the remainder, we let $v(\cdot)$ denote the optimal solution value of problem $(\cdot)$.

Observation 1. Problem (S_λ) provides a lower bound on v(P), since any solution that satisfies (1) also satisfies (1b) (but not necessarily vice versa). We have therefore that $S(\lambda) \leqslant v$(P) for any vector $\lambda = (\lambda_1, \ldots, \lambda_m) \geqslant 0$ of surrogate multipliers with $\lambda_i > 0$ for at least one i ($i = 1, \ldots, m$).

Observation 2. Problem (S_λ) is solvable in $O(nm)$ time by assigning each job J_j to a machine M_h for which $\lambda_h p_{hj} = \min_{1 \leqslant i \leqslant m} \lambda_i p_{ij}$. Ties may be settled arbitrarily.

Note that $S(\lambda) = \sum_{j=1}^{n} \min_{1 \leqslant i \leqslant m} \lambda_i p_{ij} / \sum_{i=1}^{m} \lambda_i$. We refer to $\lambda_i p_{ij}$ as the *dual processing time* of J_j on M_i. The conditions (3) of the surrogate relaxation problem can be replaced with the conditions $x_{ij} \geqslant 0$ ($i = 1, \ldots, m$, $j = 1, \ldots, n$) without affecting the optimal value $S(\lambda)$. Problem (S_λ) is said to have the *integrality property*, since it can be solved as a linear programming problem.

Observation 3. Any solution to (S_λ) is also a feasible solution to the primal problem (P), for any vector $\lambda = (\lambda_1, \ldots, \lambda_m) \geqslant 0$ of surrogate multipliers with $\lambda_i > 0$ for at least one i ($i = 1, \ldots, m$).

The constraints (2) and (3) enforce the assignment of each job to exactly one machine. For a specific optimal solution of problem (S_λ), let $C_i(\lambda)$ denote the completion time of M_i. The approximate solution value is then $C_{\max}(\lambda) = \max_{1 \leqslant i \leqslant m} C_i(\lambda)$. The way we settle ties when solving problem (S_λ) affects $C_{\max}(\lambda)$.

Observation 4. The objective value $S(\lambda)$ is a *convex* combination of the machine completion times. This implies that $\min_{1 \leqslant i \leqslant m} C_i(\lambda) \leqslant S(\lambda) \leqslant \max_{1 \leqslant i \leqslant m} C_i(\lambda)$.

Consider the following instance of the $R\|C_{\max}$ problem, where eight jobs are to be scheduled on three machines with the processing times given in Table I. Let $\lambda = (1, 1, 1)$ be the vector of surrogate multipliers. The surrogate problem (S_λ) is solved by assigning each job to the machine with the smallest processing time for it. The resulting schedule is represented by the Gantt chart of Figure 1. The initial choice $\lambda = (1, 1, 1)$ gives an elementary lower bound: it is the sum of the minimum processing times divided by the number of machines. The lower bound is $S(\lambda) = 18\frac{1}{3}$; the upper bound is $C_{\max}(\lambda) = 33$.

The best surrogate lower bound is found by solving the *surrogate dual problem*, referred to as problem (D). It is defined as

$$v(\mathrm{D}) = \max\{S(\lambda) \mid \lambda \geqslant 0\}. \tag{D}$$

In the remainder, we let λ^* denote the vector of optimal surrogate multipliers; furthermore, we call problem (P) the *primal* problem.

Table I. Processing Time Matrix

	J_1	J_2	J_3	J_4	J_5	J_6	J_7	J_8
M_1	6	3	10	12	11	14	8	6
M_2	10	∞	15	6	6	11	14	7
M_3	11	9	14	14	∞	10	10	9

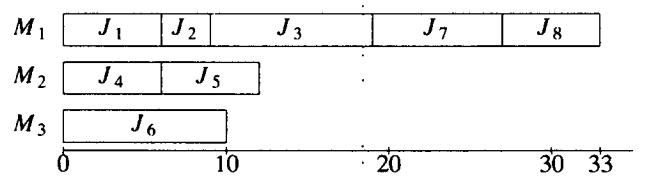

Figure 1. Gantt chart for $\lambda = (1, 1, 1)$; the dotted line indicates the lower bound $S(\lambda)$.

Observation 5. Since (S_λ) possesses the integrality property, we have $v(\bar{P}) = v(D)$: the surrogate dual yields the same lower bound as the linear programming relaxation $(\bar{P})$ (Greenberg and Pierskalla[15]; Karwan and Rardin[21]).

Observation 5 implies that $v(D)$ and λ^* can be found in polynomial time by solving problem $(\bar{P})$. Nonetheless, we develop a so-called *ascent direction algorithm* to approximate λ^*. In general, an ascent direction algorithm is an iterative approximation algorithm that generates a series of monotonically increasing lower bounds on the optimal solution value of the dual problem. Ascent direction algorithms have been shown to be successful for a wide range of combinatorial optimization algorithms, including plant location problems (Bilde and Krarup[3]; Erlenkotter[8]; Guignard and Spielberg[17]), the traveling salesman problem (Balas and Christofides[1]; Christofides[4]), the generalized assignment problem (Fisher, Jaikumar and Van Wassenhove[10]), and the set covering problem (Fisher and Kedia[11]). For these applications, the gain in speed compensates the loss in lower bound quality more than sufficiently.

In our application, the advantage is twofold. First, the ascent direction algorithm is fast, requiring only $O(mn)$ time per iteration; second, in each iteration it produces also a feasible solution to the primal problem.

The key feature of an ascent direction algorithm is the adjustment of only a limited number of multipliers per iteration; if the consequences of particular multiplier adjustments can be evaluated, then one guaranteeing an improved lower bound is chosen. An ascent direction method is problem-specific: it exploits the special structure of the problem and of the formulation. Guignard and Rosenwein[16] present an application-oriented guide for designing Lagrangian ascent direction algorithms. For a discussion of a theoretical nature, refer to Van de Velde.[26]

The shape of the function $S:\lambda \to S(\lambda)$ and the *directional derivatives* are important for evaluating the consequences of multiplier adjustments. To specify the form of S, we first observe that there are no more than m^n feasible solutions for any problem (S_λ), since each of the n jobs can be assigned to no more than m machines. Let X be the set containing these solutions. Hence, X can be represented as $X = \{x^{(1)}, \ldots, x^{(K)}\}$, where $K \leqslant m^n$. Problem (D) is then equivalent to the following problem: maximize

$$z$$

subject to

$$z \leqslant \lambda A x^{(k)} \Big/ \sum_{i=1}^{m} \lambda_i, \qquad \text{for} \quad k = 1, \ldots, K,$$

$$\lambda \geqslant 0 \quad \text{and} \quad \sum_{i=1}^{m} \lambda_i > 0,$$

where A represents the processing time matrix. Hence, the function $S:\lambda \to S(\lambda)$ is the lower envelope of a finite set of functions of the type $\lambda \to \lambda A x^{(k)} / \Sigma_{i=1}^{m} \lambda_i$; it implies that S is continuous in λ and everywhere differentiable except at all λ where (S_λ) has multiple optimal solutions. We call such λ the *points of non-differentiability*. We have depicted

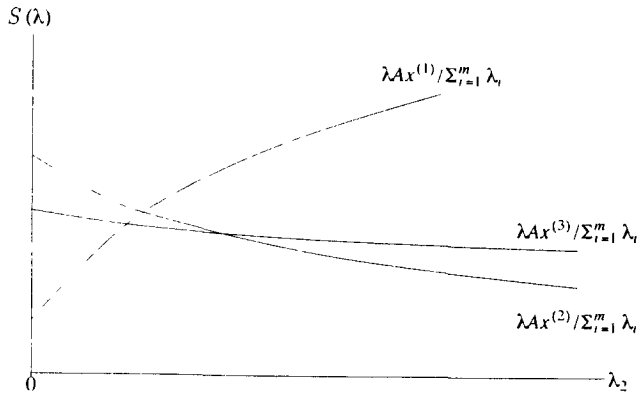

Figure 2. The form of $S(\lambda)$.

in Figure 2 the shape of S for $m = 2$ and $K = 3$ with λ_1 fixed. The form of the function S is unusual; this is because of the term $\Sigma_{i=1}^{m} \lambda_i$ appearing in the denominator. If the term $\Sigma_{i=1}^{m} \lambda_i$ were absent, then the function S would be the lower envelope of a finite set of *linear* functions (cf. Fisher[9]).

The *directional derivative* of the function S at λ is defined as

$$S_u(\lambda) = \lim_{\epsilon \downarrow 0} \frac{S(\lambda + \epsilon u) - S(\lambda)}{\epsilon}, \tag{4}$$

for any vector $u \in \mathbb{R}^m$. Hence, a necessary condition for λ to be optimal is that

$$S_u(\lambda) \leqslant 0, \quad \text{for all} \quad u \in \mathbb{R}^m; \tag{5}$$

later on, we show that condition (5) is also sufficient for optimality. If $S_{\bar{u}}(\lambda) > 0$ for some $\bar{u} \in \mathbb{R}^m$, then $\bar{u}$ is called an *ascent direction* of L at λ: we get an improved lower bound by moving some scalar step size Δ along $\bar{u}$. In general, it is difficult to compute directional derivatives. However, it is easy to compute them for the *primitive* vectors. A vector $u = (u_1, \ldots, u_m)$ is called primitive if all $u_i = 0$ for all i but one. Hence, there are at most $2m$ different primitive directional derivatives at any λ.

We show that the primitive directional derivatives reduce to simple expressions. For $i = 1, \ldots, m$, let $l_i^+(\lambda)$ be the directional derivative at λ for any primitive vector with the ith component positive; hence, $l_i^+(\lambda)$ is the derivative for increasing λ_i. For $i = 1, \ldots, m$, let $l_i^-(\lambda)$ be the directional derivative at λ for any primitive vector with the ith component negative; hence, $l_i^-(\lambda)$ is the derivative for decreasing λ_i. The first step is to rewrite definition (4) for the primitive directional derivatives; accordingly, we get

$$l_i^+(\lambda) = \lim_{\epsilon \downarrow 0} \frac{S(\lambda + \epsilon e^{(i)}) - S(\lambda)}{\epsilon},$$

and

$$l_i^-(\lambda) = \lim_{\epsilon \downarrow 0} \frac{S(\lambda - \epsilon e^{(i)}) - S(\lambda)}{\epsilon},$$

where $e^{(i)}$ is the ith unity vector, for $i = 1, \ldots, m$. Note that $l_i^-(\lambda)$ is undefined for any λ with $\lambda_i = 0$.

The second step is to simplify the above expressions. First, recall that any problem (S_λ) may have multiple optimal solutions; let $X^{(\lambda)}$ be the set containing them. Second, for any h $(h = 1, \ldots, m)$, a sufficiently small step size $\Delta_h^+ > 0$ exists to ensure that some $x(h)^+ \in X^{(\lambda)}$ is also optimal for problem $(S_{\lambda+\epsilon e^{(h)}})$ for any ϵ with $0 \leqslant \epsilon \leqslant \Delta_h^+$; this can easily be proven in a constructive way. Hence, such an $x(h)^+$ must be an optimal solution to problem (S_λ) with as few as possible jobs assigned to M_h. To get such an $x(h)^+$, each job J_j with $\lambda_h p_{hj}$ minimal and with $\lambda_i p_{ij} = \lambda_h p_{hj}$ for some $M_i \neq M_h$ is not assigned to M_h; all other ties are settled arbitrarily. Let $C_h^+(\lambda)$ be the completion time of M_h for such an $x(h)^+$; let $\mathcal{J}_h^+(\lambda)$ denote the set of jobs on M_h for such an $x(h)^+$. Similarly, for any h $(h = 1, \ldots, m)$, a sufficiently small step size $\Delta_h^- > 0$ exists to ensure that some $x(h)^- \in X^{(\lambda)}$ is also optimal for problem $(S_{\lambda-\epsilon e^{(h)}})$ for any ϵ with $0 \leqslant \epsilon \leqslant \Delta_h^-$. Such an $x(h)^-$ must be an optimal solution to problem (S_λ) with as many as possible jobs assigned to M_h. To get such an $x(h)^-$, each J_j with $\lambda_h p_{hj}$ minimal is assigned to M_h; all other ties are settled arbitrarily. Let $C_h^-(\lambda)$ be the completion time of M_h for such an $x(h)^-$; let $\mathcal{J}_h^-(\lambda)$ denote the set of jobs on M_h for such an $x(h)^-$.

For a specific $x(h)^+$, we have then that $C_i(\lambda + \epsilon e^{(h)}) = C_i(\lambda)$ for each $i (i = 1, \ldots, m)$ for $0 \leqslant \epsilon \leqslant \Delta_h^+$, and that $C_h(\lambda) = C_h^+(\lambda)$.

Hence, for $0 \leqslant \epsilon \leqslant \Delta_h^+$, we have

$$S(\lambda + \epsilon e^{(h)}) = \frac{\epsilon C_h^+(\lambda) + \sum_{i=1}^m \lambda_i C_i(\lambda)}{\epsilon + \sum_{i=1}^m \lambda_i}$$

$$= \frac{\epsilon C_h^+(\lambda) + (\sum_{i=1}^m \lambda_i C_i(\lambda)/\sum_{i=1}^m \lambda_i)\sum_{i=1}^m \lambda_i}{\epsilon + \sum_{i=1}^m \lambda_i}$$

$$= \frac{\epsilon C_h^+(\lambda) + S(\lambda)\sum_{i=1}^m \lambda_i}{\epsilon + \sum_{i=1}^m \lambda_i}.$$

This gives that

$$S(\lambda + \epsilon e^{(h)}) - S(\lambda) = \epsilon[C_h^+(\lambda) - S(\lambda)] \Big/ \left(\epsilon + \sum_{i=1}^m \lambda_i\right).$$

Using this, we obtain for the primitive directional derivative that

$$l_h^+(\lambda) = [C_h^+(\lambda) - S(\lambda)] \Big/ \sum_{i=1}^m \lambda_i.$$

In a similar fashion, we get that

$$l_h^-(\lambda) = [S(\lambda) - C_h^-(\lambda)] \Big/ \sum_{i=1}^m \lambda_i.$$

If $C_h^+(\lambda) > S(\lambda)$, then machine M_h is *overloaded*. For a sufficiently small $\epsilon > 0$, ensuring that some optimal solution for problem (S_λ) remains optimal for problem $(S_{\lambda+\epsilon e^{(h)}})$, we have then that $S(\lambda + \epsilon e^{(h)}) > S(\lambda)$. In other words, increasing λ_h is an ascent direction: we will obtain an improved surrogate objective value by moving along this direction. If $C_h^-(\lambda) < S(\lambda)$, then machine M_h is called *underloaded*: decreasing λ_h is an ascent direction. Later on, we show how to compute an appropriate step size to move by along an ascent direction.

Now we show that the shape of S between the points of non-differentiability is immaterial. Suppose λ is not a point of non-differentiability. For $h = 1, \ldots, m$, let $\Delta_h^- > 0$ be the step size for decreasing λ_h to reach the nearest point of non-differentiability; such a Δ_h^- always exists, because any λ with $\lambda_h = 0$ is a point of non-differentiability. For $h = 1, \ldots, m$, let $\lambda(h)^- = (\lambda_1, \ldots, \lambda_h - \Delta_h^-, \ldots, \lambda_m)$. Using the derivation of the primitive directional derivatives, we have

$$S(\lambda(h)^-) - S(\lambda) = \Delta_h^-[S(\lambda) - C_h(\lambda)] \Big/ \left(-\Delta_h^- + \sum_{i=1}^m \lambda_i\right).$$

Since $S(\lambda)$ is a convex combination of the machine completion times $C_1(\lambda), \ldots, C_m(\lambda)$, we have $S(\lambda(h)^-) \geqslant S(\lambda)$ for at least one h. Hence, for problem (D), we can restrict ourselves to the optimization over all $\lambda \geqslant 0$ corresponding to points of non-differentiability; the form of S between these points does not matter. The function S can therefore be treated as if it were the lower envelope of a finite set of linear functions, implying that a local optimum for problem (D) is also a global optimum. This means that condition (5) is also sufficient for optimality.

If we find an ascent direction, then we travel along this direction to the nearest point where the associated primitive directional derivative changes. The required step size is easily determined. Suppose $l_h^+(\lambda) > 0$: M_h is overloaded. Increasing λ_h makes M_h less attractive to schedule jobs on. Eventually, we reach the first point where some J_j currently scheduled on M_h can equally well be scheduled on some other machine M_g; moving on beyond this point, we enforce the removal of J_j from M_h. The step size Δ to reach this point is the smallest positive value for which $(\lambda_h + \Delta)p_{hj} = \lambda_g p_{gj}$ for some J_j on M_h and some M_g $(M_g \neq M_h)$; hence, it is computed as

$$\Delta = -\lambda_h + \min_{1 \leqslant i \leqslant m, i \neq h, J_j \in \mathcal{J}_h^+(\lambda)} \lambda_i p_{ij}/p_{hj}.$$

Accordingly, we get $\bar{\lambda} = (\lambda_1, \ldots, \lambda_h + \Delta, \ldots, \lambda_m)$; the increment to the objective value is $S(\bar{\lambda}) - S(\lambda) = \Delta[C_h^+(\lambda) - S(\lambda)]/(\Delta + \Sigma_{i=1}^m \lambda_i) > 0$. Furthermore, we move J_j from M_h to M_g, and examine whether increasing $\bar{\lambda}_h$ is also an ascent direction.

Now, suppose $l_h^-(\lambda) > 0$: M_h is underloaded. Decreasing λ_h makes M_h more attractive. Eventually, we reach the first point where some J_j on M_g $(M_g \neq M_h)$ can equally well be scheduled on M_h; moving on beyond this point will force J_j to go to M_h. The required step size Δ is the smallest positive value for which $(\lambda_h - \Delta)p_{hj} = \lambda_g p_{gj}$, for some J_j scheduled on some M_g; it is computed as

$$\Delta = \lambda_h - \min_{1 \leqslant i \leqslant m, i \neq h, J_j \in \mathcal{J} - \mathcal{J}_h^-(\lambda)} \lambda_i p_{ij}/p_{hj}.$$

Accordingly, we get $\bar{\lambda} = (\lambda_1, \ldots, \lambda_h - \Delta, \ldots, \lambda_m)$, and the increment to the objective value is $S(\bar{\lambda}) - S(\lambda) = \Delta[S(\lambda) - C_h^-(\lambda)]/(-\Delta + \Sigma_{i=1}^m \lambda_i) \geq 0$; we move J_j to M_h, and examine whether decreasing $\bar{\lambda}_h$ is also an ascent direction.

If the ascent direction method is started in some $\lambda > 0$, then the ascent direction can never reach a boundary point where $\lambda_i = 0$ for some i $(i = 1, \ldots, m)$. Also, we must have

$\lambda^* > 0$, since there exists an ascent direction, which may be non-primitive, for any vector with multipliers equal to zero. Termination of the ascent direction method happens therefore at some $\bar{\lambda}$ where all primitive directional derivatives exist. At such a $\bar{\lambda}$, we have

$$l_i^+(\bar{\lambda}) \leqslant 0, \quad \text{and} \quad l_i^-(\bar{\lambda}) \leqslant 0, \qquad \text{for} \quad i = 1, \ldots, m,$$

or equivalently,

$$\sum_{J_j \in \mathcal{J}_i^+(\bar{\lambda})} p_{ij} \leqslant S(\bar{\lambda}) \leqslant \sum_{J_j \in \mathcal{J}_i^-(\bar{\lambda})} p_{ij}, \qquad \text{for} \quad i = 1, \ldots, m.$$

We call these the *termination conditions*. Comparing them with condition (5), we see that they are necessary but not sufficient for optimality; hence, termination may occur having $\bar{\lambda} \neq \lambda^*$, i.e., before finding the optimal vector of surrogate multipliers.

The identification of the ascent direction and the computation of the step size can be implemented in different ways. We have freedom concerning the choice of the initial vector, and, for each iteration, the choice of the ascent direction. The vector $\bar{\lambda}$ depends on these choices. Since $\lambda_i^* > 0$ for each i ($i = 1, \ldots, m$), we best start with a positive vector. Moreover, since the surrogate multipliers are normalized values, we can fix one multiplier a priori without running the risk of missing the optimum. The choice of the ascent direction affects the upper bounds that we get as by products: for machine load balancing, it may be better to choose the direction of steepest ascent. Nonetheless, we give below a stepwise description of a rudimentary version, stripped from most of such considerations.

Ascent Direction Algorithm for Problem (D)

Step 0. For $h = 1, \ldots, m$, set $\lambda_h \leftarrow 1$. Solve problem (S_λ), settling ties arbitrarily. Determine $S(\lambda)$.

Step 1. For $h = 1, \ldots, m$, do the following:

(a) While $C_h^+(\lambda) > S(\lambda)$, compute

$$\Delta = -\lambda_h + \min_{1 \leqslant i \leqslant m, i \neq h, J_j \in \mathcal{J}_h^+(\lambda)} \lambda_i p_{ij}/p_{hj},$$

set $\lambda_j \leftarrow \lambda_j + \Delta$, and update $S(\lambda)$ and $C_h^+(\lambda)$.

(b) While $C_h^-(\lambda) < S(\lambda)$, compute

$$\Delta = \lambda_h - \min_{1 \leqslant i \leqslant m, i \neq h, J_j \in \mathcal{J} - \mathcal{J}_h^-(\lambda)} \lambda_i p_{ij}/p_{hj},$$

set $\lambda_j \leftarrow \lambda_j - \Delta$, and update $S(\lambda)$ and $C_h^-(\lambda)$.

Step 2. Stop if no ascent direction was identified; otherwise, go to Step 1.

Let us reconsider our example and the solution of (S_λ) with $\lambda = (1, 1, 1)$. Machine M_1 is overloaded. The step size to remove some job from M_1 is $\Delta = {}^1/_6$; increasing λ_1 by ${}^1/_6$ allows us to move J_8 to M_2. We get $\lambda = ({}^7/_6, 1, 1)$, a schedule with makespan 27, and $S(\lambda) = 19.1$ (see Figure 3; the dotted line indicates the virtual capacity of the machines).

Machine M_1 remains overloaded; we increase λ_1 to ${}^5/_4$, and subsequently move J_7 to M_3. We get $\lambda = ({}^5/_4, 1, 1)$, a schedule with makespan 20, and $S(\lambda) = 19.3$ (see Figure 4). Since all processing times are integral, the optimal

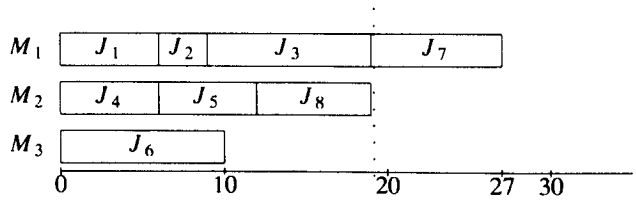

Figure 3. Gantt chart for $\lambda = ({}^7/_6, 1, 1)$.

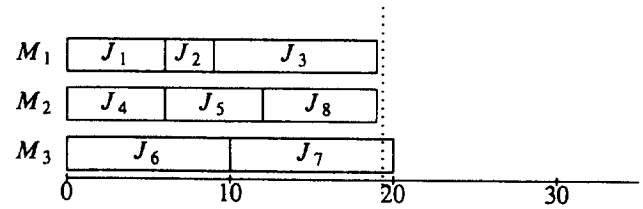

Figure 4. Gantt chart for $\lambda = ({}^5/_4, 1, 1)$.

makespan is integral as well. Hence, we have found an optimal primal solution. However, an ascent direction still exists: M_2 is underloaded. If we decrease λ_2 by ${}^1/_{11}$, then J_6 goes to M_2. We obtain $\lambda = ({}^5/_4, {}^{10}/_{11}, 1)$ and $S(\lambda) = {}^{1944}/_{139}$. At this point, no primitive ascent direction exists anymore; the ascent direction method is terminated at $\bar{\lambda} = ({}^5/_4, {}^{10}/_{11}, 1)$.

The vector $\bar{\lambda}$ is not optimal for the surrogate dual problem, i.e., $\bar{\lambda} \neq \lambda^*$. For λ^*, we have that $v(\bar{P}) = S(\lambda^*)$ (Observation 5), and that the complementary slackness conditions hold. These conditions can be shown to imply that for each j ($j = 1, \ldots, n$) we have that

$$x_{ij} > 0 \Rightarrow \lambda_i^* p_{ij} = \min_{1 \leqslant k \leqslant m} \lambda_k^* p_{kj}.$$

Considering Figure 4, we are allowed to split only J_6 over M_3 and M_2 and J_7 over M_3 and M_1, if we want a solution that satisfies the complementary slackness relations. However, a solution with the desired value can then never be obtained.

We now discuss Potts' 2-approximation algorithm and its relation to the ascent direction algorithm. For an arbitrary number of machines, the first phase of Potts' algorithm is to solve the linear programming relaxation $(\bar{P})$. The solution of $(\bar{P})$ shows at least $n - m + 1$ jobs each assigned to exactly one machine, and at most $m - 1$ jobs split over two or more machines. The jobs assigned to exactly one machine are retained as a partial schedule. The split jobs are assigned so as to minimize the makespan, given the partial schedule. Since $v(\bar{P}) \leqslant v(P)$, the length of the partial schedule is no more than $v(P)$. The scheduling of the split jobs proceeds by complete enumeration; this adds at most $v(P)$ to the length of the partial schedule. Hence, the resulting schedule has a makespan at most twice the optimal makespan. Since $(\bar{P})$ is solvable in polynomial time and complete enumeration for at most $m - 1$ split jobs requires $O(m^m)$ time, the procedure is polynomial for fixed m.

Consider now an optimal solution of problem $(S_{\bar{\lambda}}^-)$. Since $\bar{\lambda}$ is the vector upon termination of the ascent direction

method, we have

$$\sum_{J_j \in \mathcal{J}_i^+(\bar{\lambda})} p_{ij} \leqslant S(\bar{\lambda}) \leqslant \sum_{J_j \in \mathcal{J}_i^-(\bar{\lambda})} p_{ij}, \quad \text{for} \quad i = 1, \ldots, m.$$

Exploiting these termination conditions, we point out a 2-approximation algorithm that proceeds entirely in the spirit of Potts' 2-approximation algorithm. In the first phase, we assign each $J_j \in \mathcal{J}_i^+(\bar{\lambda})$ to M_i, thus obtaining a partial schedule with length no more than $v(P)$. The remaining jobs, contained in the set $\mathcal{J} - \cup_{i=1}^{m} \mathcal{J}_i^+(\bar{\lambda})$, have ties concerning their minimal dual processing times. In the second phase, we assign these jobs by complete enumeration so as to minimize the makespan, given the partial schedule; this adds at most $v(P)$ to the length of the partial schedule. Hence, the resulting schedule has makespan no more than $2\ v(P)$.

In fact, we get also a schedule with worst-case ratio 2 with fewer jobs to assign in the second phase as follows. Let $\mathcal{J}_i^0(\bar{\lambda}) \subseteq \mathcal{J}_i^-(\bar{\lambda}) - \mathcal{J}_i^+(\bar{\lambda})$ $(i = 1, \ldots, m)$ be mutually disjoint subsets of jobs such that

$$\sum_{J_j \in \mathcal{J}_i^+(\bar{\lambda}) \cup \mathcal{J}_i^0(\bar{\lambda})} p_{ij} \leqslant S(\bar{\lambda}), \text{ for each } i = 1, \ldots, m;$$

hence, $\mathcal{J}_i^0(\bar{\lambda})$ contains only jobs with ties concerning their minimal dual processing time. In the first phase, we assign each $J_j \in \mathcal{J}_i^+(\bar{\lambda}) \cup \mathcal{J}_i^0(\bar{\lambda})$ to M_i; in the second phase, we assign the remaining jobs. The sets $\mathcal{J}_i^0(\bar{\lambda})$ should be chosen so as to minimize the number of jobs left for the second phase. In general, we cannot bound the number of jobs to be assigned in the second phase by a polynomial in m. However, if $\bar{\lambda} = \lambda^*$, then this procedure is exactly Potts' 2-approximation algorithm; since the complementary slackness relations holds for $\bar{\lambda} = \lambda^*$, we can choose the sets $\mathcal{J}_i^0(\bar{\lambda})$ in such a way that no more than $m - 1$ jobs remain for the second phase.

For the special case $m = 2$, we have $v(\bar{P}) = S(\bar{\lambda})$. Since the surrogate multipliers represent normalized values, only $m - 1$ multipliers need to be involved to find λ^*. For the case $m = 2$, only one multiplier is involved; the termination conditions at $\bar{\lambda}$ are then sufficient for optimality. For $m = 2$, problem $(\bar{P})$ is solvable in $O(n)$ time (Gonzalez, Lawler, and Sahni[13]). Moreover, there is at most one split job. Considering the ascent direction algorithm, we observe that the best solution value generated by Potts' 2-approximation algorithm concurs with the best upper bound found when solving problem (D) by use of the ascent direction procedure.

2. Duality-Based Heuristic Search

The principle of Potts' 2-approximation algorithm and specifically the termination conditions of the ascent direction algorithm give rise to the idea that a near-optimal solution for the surrogate dual problem induces a near-optimal solution for the primal problem. In this respect, we need a scheme that generates a series of promising surrogate multipliers. The example suggests that the ascent direction method, perhaps with some minor adjustments, is such a scheme. The ascent direction method, however, is too restrictive for our purpose. Computational experiments show that is usually terminated after only a small number of iterations. We need a scheme that allows us to browse quickly through many near-optimal solutions for problem (D). The approximate algorithm differs therefore from the ascent direction method on two counts.

First, the machine with the largest overload is always selected for multiplier adjustment. From a primal point of view, this is an obvious choice: one of the jobs on this machine must be removed in order to reduce the machine completion time that induces the current makespan. Second, we make the step size larger than necessary to enforce such a removal: this avoids early termination. Specifically, we move to the *second* point where the primitive directional derivative changes. Let machine M_h be the machine with the largest overload in the solution of problem (S_λ); hence, we have $C_{\max}(\lambda) \leqslant C_h^+(\lambda)$. Then we compute

$$\Delta = -\lambda_h + \underset{1 \leqslant i \leqslant m,\ J_j \in \mathcal{J}_h^+(\lambda)}{\min 2} \lambda_i p_{ij}/p_{hj},$$

where min2 denotes the second minimum of these values. If we put $\bar{\lambda} = (\lambda_1, \ldots, \lambda_h + \Delta, \ldots, \lambda_m)$, then we enforce the move of some J_k from M_h to some M_g, and that another job on M_h can equally well be scheduled on some other machine. Nonetheless, this second job is kept on M_h. The next step is to compute the new makespan $C_{\max}(\bar{\lambda})$, and the machine with the largest overload; this machine is determined by computing $\max_{1 \leqslant i \leqslant m} C_i^+(\bar{\lambda})$. We have no guarantee that the rescheduling of J_k induces an improved schedule: we can have either $C_{\max}(\bar{\lambda}) \leqslant C_{\max}(\lambda)$ or $C_{\max}(\bar{\lambda}) > C_{\max}(\lambda)$. The latter occurs if $C_g(\bar{\lambda}) = C_g(\lambda) + p_{gk} > C_{\max}(\lambda)$. Hence, the approximation algorithm is equipped with a mechanism that accepts deteriorations of the makespan. We repeat this process for the machine with the largest load, and store the best solution on the way. We put an upper bound on the number of iterations, since this procedure does not have any convergence properties. Below we give a stepwise description of the algorithm; *maxiter* is a prespecified maximum number of iterations and *UB* is the currently best solution value.

Approximation Algorithm

Step 0. Put $\lambda \leftarrow (1, \ldots, 1)$, $t \leftarrow 1$. Solve (S_λ), settling ties arbitrarily. Let $UB \leftarrow C_{\max}(\lambda)$, and store the schedule.

Step 1. Determine M_h with the largest overload: $C_h^+(\lambda) \geqslant C_i^+(\lambda)$ for each i $(i = 1, \ldots, m)$. Compute Δ, and identify a job J_k and a machine M_g such that $\lambda_g p_{gk}/p_{hk} = \min_{1 \leqslant i \leqslant m,\ i \neq h,\ J_j \in \mathcal{J}_h^+(\lambda)} \lambda_i p_{ij}/p_{hj}$. Put $t \leftarrow t + 1$.

Step 2. Put $\lambda \leftarrow (\lambda_1, \ldots, \lambda_h + \Delta, \ldots, \lambda_m)$, $C_h(\lambda) \leftarrow C_h(\lambda) - p_{hk}$, $C_g(\lambda) \leftarrow C_g(\lambda) + p_{gk}$. If $C_{\max}(\lambda) < UB$, then $UB \leftarrow C_{\max}(\lambda)$, and store the schedule. If $t < maxiter$, then go to Step 1; if not, then stop.

We call the approximation algorithm described above the *duality-based approximation algorithm*, and the particular strategy employed as *duality-based heuristic search*. For the example, the approximation algorithm goes through the same steps as described in Section 1.

Many heuristic search strategies are applicable to the parallel machine scheduling problem. Most of them have in

common that they adjust the current schedule somewhat to try to improve on its value. Let σ be some arbitrary schedule and let σ_{jk} be the schedule obtained from σ by swapping J_j and J_k ($j \neq k$). We define the so-called *single pairwise interchange neighborhood* for σ as the set N_σ containing the schedules σ_{jk} for all $j = 1, \ldots, n-1$, $k = j+1, \ldots, n$. Suppose M_h is such that $C_h(\sigma) = C_{\max}(\sigma)$, where $C_h(\sigma)$ and $C_{\max}(\sigma)$ denote the completion time of M_h and the maximum machine completion time in σ, respectively. Let (J_j, J_k) be a pair of jobs such that J_j is scheduled on M_h and J_k on some other machine M_g ($g \neq h$) for which we have

$$C_g + p_{gj} - p_{gk} < C_h, \text{ and } C_h - p_{hj} + p_{hk} < C_h.$$

If we interchange J_j and J_k, that is, we put J_j on M_g and J_k on M_h, then we reduce the makespan. In other words, we have identified a schedule $\sigma_{jk} \in N_\sigma$ with $C_{\max}(\sigma_{jk}) < C_{\max}(\sigma)$. This process is repeated and terminates when no further improvement is found. *Iterative local improvement procedures* are based upon these concepts. The main danger is to get trapped in a poor local optimum. We may circumvent this pitfall by using multiple schedules as starting points, which may lead us to multiple locally optimal solutions. Hopefully, one of them is a good approximate solution.

More sophisticated techniques to avoid early entrapment have been developed, among which *simulated annealing* and *tabu search* take prominent places. Simulated annealing (see e.g. Van Laarhoven and Aarts[27]) leaves the possibility open to travel from one local optimum to another. This is achieved by accepting deteriorations of the objective value with a probability that is a decreasing function of the running time. Tabu search (Glover,[12] de Werra and Hertz[7]) is much similar to simulated annealing, but provides a deterministic mechanism to accept deteriorations. The willingness to accept deteriorations unconditionally distinguishes the duality-based search technique from simulated annealing, tabu search, and general iterative local improvement schemes.

Anticipating on the implementation and the evaluation of the duality-based approximation algorithm in Section 4, however, we will consider two versions of the algorithm. On the one hand, we evaluate the duality-based algorithm on its own; on the other hand, we evaluate the algorithm in conjunction with the iterative local improvement procedure we described. We only submitted the best solution to the improvement procedure. The duality-based algorithm in conjunction with the iterative local improvement procedure produces very good results. Apparently, the duality-based approximation algorithm finds an attractive initial solution for the iterative local improvement procedure.

3. The Branch-and-Bound Algorithm

The first step in the branch-and-bound algorithm is to run the ascent direction method to approximate the optimal solution of problem (D). Upon termination, we have the vector $\bar{\lambda} = (\bar{\lambda}_1, \ldots, \bar{\lambda}_m)$ of surrogate multipliers. On the way, we store the best primal solution. We also use the duality-based approximation algorithm and the constructive heuristics presented by De and Morton,[6] Ibarra and Kim,[20] and Davis and Jaffe[5] to find approximate solutions for problem (P). The implementation of these algorithms is described in Section 4. The vector $\bar{\lambda}$ plays an important role in the truncation of the search tree.

3.1. Initial Reductions

The size of an instance may be reduced by a simple reduction test, which is common in linear programming. It can be conducted for any vector of surrogate multipliers, but success is most likely for λ^* and vectors close to it.

Theorem 1. *If for a given vector of multipliers* $\lambda = (\lambda_1, \ldots, \lambda_m)$, *we have some* J_k *and* M_h *for which*

$$\left(\lambda_h p_{hk} - \min_{1 \leq i \leq m} \lambda_i p_{ik}\right) \Big/ \sum_{i=1}^{m} \lambda_i > UB - S(\lambda) - 1,$$

where UB is a given upper bound on v(P), *then* $x_{hk} = 0$ *in any schedule with* $C_{\max} < UB$, *if such a schedule exists.*

Proof. Suppose there is a schedule with makespan less than UB, and yet with J_k scheduled on M_h. Solving the surrogate relaxation problem (S_λ) under the additional constraint $x_{hk} = 1$ gives the lower bound LB with

$$\begin{aligned} LB &= \left(\lambda_h p_{hk} + \sum_{j=1, j \neq k}^{n} \min_{1 \leq i \leq m} \lambda_i p_{ij}\right) \Big/ \sum_{i=1}^{m} \lambda_i \\ &= \left[\left(\lambda_h p_{hk} - \min_{1 \leq i \leq m} \lambda_i p_{ik}\right) + \sum_{j=1}^{n} \min_{1 \leq i \leq m} \lambda_i p_{ij}\right] \Big/ \\ &\quad \sum_{i=1}^{m} \lambda_i > UB - 1, \end{aligned}$$

which is a contradiction. ■

3.2. The Search Tree

A node at level k of the search tree corresponds to a partial schedule with a specific assignment of $J_1, \ldots, J_k$. Each node at level k ($k = 1, \ldots, n-1$) has at most m descendant nodes: one node for the assignment of job J_{k+1} to each machine M_i, for $i = 1, \ldots, m$. The jobs and the machines will be reindexed in compliance with the branching rule we propose in the next subsection. The algorithm we use is of the "depth-first" type. We employ an *active node* search: at each level, we consider only one node to branch from, thereby adding some job to the partial schedule. The nodes are branched from in order of increasing indices of the associated machines. We backtrack if we reach the bottom of the tree or if we can discard the active node.

3.3. Branching Rule

The dual processing times $\bar{\lambda}_i p_{ij}$ ($i = 1, \ldots, m$, $j = 1, \ldots, n$) also serve to structure the search tree. We define $\gamma_j = \min 2_{1 \leq i \leq m} \bar{\lambda}_i p_{ij} - \min_{1 \leq i \leq m} \bar{\lambda}_i p_{ij}$, where $\min 2$ denotes the second minimum. In view of Theorem 3, a large value γ_j suggests that there exists an optimal solution with J_j

scheduled on the machine with minimum dual processing time for it; we call this machine the *favorite* machine for J_j. We like to structure the search tree in such a way that we first explore the configurations with jobs with large γ_j assigned to their favorite machines. This is achieved by reindexing the jobs in order of nonincreasing values γ_j and by reindexing the machines at each level k ($k = 1, \ldots, n-1$) in order of nondecreasing values $\bar{\lambda}_i p_{i,k+1}$ ($i = 1, \ldots, m$). We note that the first complete schedule encountered in the tree is an optimal solution for the surrogate problem ($S_{\bar{\lambda}}$).

Such a structure of the search tree has two advantages. First, for the optimal solution and good approximate solutions of the primal problem, most jobs are expected to have been assigned to their favorite machines. Second, if we find an improved upper bound, then most of the additional variable reductions are associated with the nodes of the still unexplored part of the search tree.

3.4. Discarding Nodes

Here, we describe in detail the various rules to discard nodes. Computational experiments show, surprisingly enough, that even a quick ascent direction method is not worthwhile to be run in each node of the tree. We use therefore the vector $\bar{\lambda} = (\bar{\lambda}_1, \ldots, \bar{\lambda}_m)$ throughout the search tree. The reduction test and the following rules depend on $\bar{\lambda}$. The vector λ^* may therefore be more effective; it may be worthwhile to use a linear programming algorithm in the root of the tree to obtain λ^*. On the other hand, if $\bar{\lambda}$ is close to λ^*, then the additional effect will be negligible. Suppose the values z_{ij} ($i = 1, \ldots, m$, $j = 1, \ldots, k$) record the current partial schedule at level k of the tree. That is, $z_{ij} = 1$ if J_j has been assigned to M_i, and $z_{ij} = 0$ otherwise. Let $S(\bar{\lambda}, k)$ denote the optimal solution of problem ($S_{\bar{\lambda}}$) subject to $x_{ij} = z_{ij}$ for $i = 1, \ldots, m$, $j = 1, \ldots, k$. Then we have

$$S(\bar{\lambda}, k) = S(\bar{\lambda}) + \sum_{j=1}^{k} \left(\sum_{i=1}^{m} \left(\bar{\lambda}_i p_{ij} - \min_{1 \leqslant i \leqslant m} \bar{\lambda}_i p_{ij} \right) z_{ij} \right) \Bigg/ \sum_{i=1}^{m} \bar{\lambda}_i.$$

Note that $S(\bar{\lambda}, k) \geqslant S(\bar{\lambda})$. A node at level k that assigns J_k to machine M_h can be discarded if

$$\left(\bar{\lambda}_h p_{hk} - \min_{1 \leqslant i \leqslant m} \bar{\lambda}_i p_{ik} \right) \Bigg/ \sum_{i=1}^{m} \bar{\lambda}_i > UB - S(\bar{\lambda}, k-1) - 1. \tag{F1}$$

This test requires constant time per node. In addition, the node can be discarded if

$$\sum_{j=1}^{k-1} p_{hj} z_{hj} + p_{hk} > UB - 1. \tag{F2}$$

The third test tries to establish whether the current partial schedule is dominated by another partial schedule for the same k jobs. Suppose we have some job J_l ($1 \leqslant l \leqslant k-1$) that is currently scheduled on M_i for which

$$p_{il} > p_{ik} \quad \text{and} \quad p_{hl} < p_{hk}. \tag{F3}$$

Interchanging J_l and J_k reduces the load of both M_i and M_h. The current partial schedule can then be discarded, since there is at least one optimal schedule with no such pair of jobs.

Conditions similar to (F2) apply to each job J_j ($j = k+1, \ldots, n$). In case there is a job J_l ($k+1 \leqslant l \leqslant n$) for which

$$\sum_{j=1}^{k} p_{ij} z_{ij} + p_{il} > UB - 1, \quad \text{for each } M_i,\ i = 1, \ldots, m, \tag{F4}$$

we discard the node, too. Similarly, if the condition (F4) applies to some J_l ($k+1 \leqslant l \leqslant n$) for all machines M_i ($i = 1, \ldots, m$) but one, we can assign J_l to this machine. Subsequently, we can possibly carry out additional assignments; these, in turn, enhance the likelihood that the node is closed on account of (F1), (F2), (F3), or (F4).

In addition, we try to identify a machine M_h ($1 \leqslant h \leqslant m$) for which

$$\sum_{j=1}^{l} p_{hj} z_{hj} + p_{hl} > UB - 1, \quad \text{for each } J_l,\ l = k+1, \ldots, n.$$

In this case, M_h is ignored for the assignment of any remaining job. Therefore, we discard the node if

$$\left[\sum_{j=1}^{k} \sum_{i=1, i \neq h}^{m} \bar{\lambda}_i p_{ij} z_{ij} + \sum_{j=k+1}^{n} \min_{1 \leqslant i \leqslant m, i \neq h} \bar{\lambda}_i p_{ij} \right] \Bigg/ \sum_{i=1, i \neq h}^{m} \bar{\lambda}_i > UB - 1.$$

4. Computational Experiments

Both algorithms have been coded in the computer language C; the experiments were conducted on a Compaq-386/20 Personal Computer.

The algorithms were tested on a broad range of instances with n and m varying from 20 to 200 and from 2 to 20, respectively, giving rise to 80 combinations altogether. The processing times were generated from the uniform distribution [10, 100]. For each combination of n and m we considered 10 instances.

4.1. The Branch-and-Bound Algorithm

For the branch-and-bound algorithm we put an upper bound of 100,000 nodes; computation for any instance was discontinued at this limit. In Table II, we present for each combination the number of unsolved problems. An empty cell indicates that the branch-and-bound algorithm was not run; considering adjacent cells or initial computations, we expected that most of the instances would remain unsolved. Table III shows the average number of nodes explored. The average for a particular combination of n and m is computed by aggregating the number of nodes for

each of the instances and dividing the sum by 10, the total number of instances for each combination. Unresolved instances contribute therefore 10,000 nodes each to the average number of nodes. Table IV presents the average computation time for the branch-and-bound algorithm, including the running time for the heuristics and the duality-based approximation algorithm. The time spent on unsolved instances is included, too. The average computation time for a particular combination is computed in a similar fashion as the average number of nodes.

From a practical point of view, the instances with a few machines are easy. The effort required to solve a problem seems to increase more with the number of machines than with the number of jobs. Surprising exceptions are the instances with $m \geqslant 12$ and $n \leqslant 40$. Note that the 100,000-node limit for the branch-and-bound algorithm is arbitrary: it induces distinct *time* limits across the instances. For example, instances with $m = 20$ and $n = 50$ and 60 require about 10,000 nodes on the average; however, they require about 5 minutes of running time. Nonetheless, one can easily form some idea about the instances that are within reach of, say, one minute of computation time.

Significant deviations from the averages occur. For the combination $n = 30$ and $m = 15$, for example, a single instance accounts for the remarkably large number of nodes and large running time. It is also conceivable that the performance of the algorithm is enhanced by fine-tuning the algorithm to particular instances. For large values of n and m, for example, it may be worthwhile to use the ascent direction method in each node of the tree after all. Even then, however, such instances are not solvable within reasonable time limits.

Table II. Number of Unsolved Problems Out of 10 for Each Cell

	m									
n	2	3	4	5	6	8	10	12	15	20
20	0	0	0	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0	0	0	0
40	0	0	0	0	0	1	2	0	0	0
50	0	0	0	0	1	5	—	—	—	0
60	0	0	0	0	1	—	—	—	—	1
80	0	0	0	2	—	—	—	—	—	—
100	0	0	0	3	—	—	—	—	—	—
200	0	1	0	—	—	—	—	—	—	—

4.2. The Duality-Based Approximation Algorithm

Implementing the duality-based approximation algorithm, we have put *maxiter* = nm. Note that cycling may occur. This happens, for instance, if J_j can be scheduled on both M_1 and M_2. If J_j is scheduled on M_1, then M_1 has the largest overload; if J_j is scheduled on M_2, then M_2 has the largest overload. In such a situation, J_j would oscillate

Table III. Average Number of Nodes; in Case of Unsolved Problems, It Is a Lower Bound on the Average Number of Nodes

	m									
n	2	3	4	5	6	8	10	12	15	20
20	16	46	68	203	180	75	33	37	11	0
30	31	90	340	752	434	1,440	784	145	4,784	64
40	37	170	615	4,488	10,149	6,786	23,936	3,800	192	342
50	59	171	1,188	6,133	26,022	98,202	—	—	—	5,848
60	68	358	1,127	12,715	37,942	—	—	—	—	20,669
80	85	1,232	3,386	37,110	—	—	—	—	—	—
100	132	2,503	5,198	58,116	—	—	—	—	—	—
200	330	22,245	14,274	—	—	—	—	—	—	—

Table IV. Average Computation Time in Seconds; the Time Spent on Unsolved Instances Is Included, Too

	m									
	2	3	4	5	6	8	10	12	15	20
20	1	1	1	1	1	1	1	1	1	1
30	1	1	1	2	2	9	6	2	43	2
40	1	1	2	12	39	39	214	63	3	10
50	1	1	3	16	57	285	—	—	—	204
60	1	1	3	33	105	—	—	—	—	373
80	1	3	8	96	—	—	—	—	—	—
100	1	6	12	87	—	—	—	—	—	—
200	3	40	52	—	—	—	—	—	—	—

between M_1 and M_2. The procedure is discontinued upon detection of this phenomenon.

The duality-based approximation algorithm was compared with the constructive heuristics of De and Morton,[6] Ibarra and Kim,[20] Davis and Jaffe,[5] and with our version of Potts' 2-approximation algorithm [Potts[24] (see Section 1)]; the latter is easy to embed in the branch-and-bound algorithm. We have evaluated neither Potts' original version, nor the 2-approximation algorithm presented by Lenstra, Shmoys, and Tardos,[22] nor the two-phase heuristics presented by Hariri and Potts.[18] All these algorithms proceed in the same spirit; none is expected to outperform the others significantly in practice. In the remainder, when referring to Potts' 2-approximation algorithm, we are actually referring to our version of it. Recall that the versions are identical for $m = 2$. The constructive heuristics display a very erroneous behavior. For instance, the De and Morton heuristic, taking the best result from 10 underlying heuristics, produces solutions with an average deviation from the best solution of 27%. We have therefore treated the constructive heuristics as a single algorithm by considering only the best schedule.

In Table V, we present the average relative deviation for the best schedule generated by the constructive heuristics from the optimal solution, or if this is not available, from the best known solution. In the latter case, brackets have been placed around the figures. Table VI shows the same information for the duality-based approximation algorithm.

As a whole, the duality-based approximation algorithm performs much better than the constructive heuristics, which behave poorly. This certainly applies to instances with a larger number of machines. The performance of the constructive heuristics is easily improved by submitting them to an iterative local improvement scheme. Therefore, the schedules generated by the constructive heuristics should merely be seen as initial solutions that serve as input for some iterative local improvement procedure.

Each schedule generated by the constructive heuristics was therefore subsequently submitted to the iterative local improvement procedure we described in Section 2. In contrast, only the best schedule generated by the duality-based approximation algorithm was submitted to the improvement procedure.

In Tables VII, VIII and IX, we present the results for the constructive heuristics, Potts' 2-approximation algorithm, and the duality-based approximation algorithm after local improvement, respectively. The sign "*" behind an entry in these tables indicates that the corresponding algorithm has the best average performance for the associated instances. Table VII exhibits that the iterative local improvement technique is effective for the constructive heuristics in case of few machines or jobs. However, its effectiveness deteriorates with an increasing number of machines. Only two machines at a time are involved in the job interchanges. For large m, it is more difficult to find an attractive local neighborhood, even in case of multiple start solutions.

Table V. Average Relative Deviation for the Constructive Heuristics

	m									
n	2	3	4	5	6	8	10	12	15	20
20	2.9	8.0	6.8	12.8	19.0	20.3	7.6	22.6	8.9	5.4
30	2.2	6.3	8.2	18.0	18.8	25.5	22.5	23.1	14.6	13.0
40	3.0	6.5	10.8	14.6	13.3	(27.5)	(27.4)	25.6	28.6	19.0
50	2.0	7.2	12.0	12.1	(19.3)	(23.4)	(17.2)	(18.4)	(14.7)	32.8
60	1.4	6.0	10.3	10.5	(15.9)	(14.6)	(17.2)	(15.4)	(16.9)	(42.5)
80	1.8	4.6	8.4	(11.1)	(13.4)	(11.4)	(16.9)	(17.5)	(24.7)	(20.5)
100	2.8	3.3	7.2	(9.7)	(11.1)	(15.0)	(19.8)	(18.3)	(19.8)	(21.2)
200	0.7	(2.4)	3.9	(4.2)	(5.0)	(8.2)	(15.5)	(17.0)	(15.5)	(24.5)

Table VI. Average Relative Deviation for the Duality-Based Approximation Algorithm

	m									
n	2	3	4	5	6	8	10	12	15	20
20	4.2	5.4	6.6	10.5	11.3	16.4	15.2	14.6	7.4	3.0
30	1.5	4.9	6.0	9.8	8.9	14.7	16.7	21.0	14.0	15.2
40	1.9	4.2	3.5	9.0	8.3	(10.0)	(19.0)	14.4	13.5	19.3
50	1.6	3.3	4.9	7.4	(6.5)	(8.3)	(4.1)	(1.9)	(2.5)	18.1
60	1.2	1.1	4.1	5.5	(5.0)	(4.0)	(1.8)	(1.0)	(1.8)	(24.3)
80	1.4	2.3	2.9	(3.5)	(2.4)	(1.9)	(2.1)	(2.8)	(2.7)	(4.5)
100	2.3	2.3	2.4	(3.6)	(1.8)	(2.2)	(1.9)	(1.9)	(0.8)	(1.4)
200	0.4	(1.5)	1.1	(1.1)	(1.2)	(1.8)	(3.3)	(1.3)	(3.3)	(0.6)

Generally, the running time, which seems to be increasing with n, is modest: instances up to $n = 100$ require only one or two seconds; approximately 10 seconds of computation time are required for instances with $n = 200$. Because the job interchanges affect only two machines at a time, the number of machines hardly seems to play a role in the computation time.

Potts' 2-approximation algorithm was embedded in a branch-and-bound algorithm that differs on two points from the branch-and-bound algorithm described in Section 3. First, we omitted the dominance rule (F3); second, we initially put $UB = \infty$. The condition (F3) is useful for finding an optimal solution, but might eliminate good approximate solutions. That is why Potts' 2-approximation algorithm sometimes took more time than the optimization algorithm. Occasionally, more than $m - 1$ jobs remained for the second phase. It is surprising that the final solution was rarely improved by the local improvement procedure, although it was applied to all jobs. The computational effort for the algorithm was modest and seemed to increase more with the number of machines than with the number of jobs. For instances up to $m = 12$, it was one or two seconds; for instances with $m = 15$ and $m = 20$, it was about 15 to 20 seconds. Instances with $n = 20$ and $m = 20$

Table VII. Average Relative Deviation for the Constructive Heuristics after Iterative Local Improvement

	m									
n	2	3	4	5	6	8	10	12	15	20
20	0.0*	1.0*	3.0*	7.1	8.3*	10.4	4.9*	11.1	1.9*	5.1
30	0.1*	1.2*	3.1*	4.4*	7.6	13.3*	15.0	17.4*	11.0	9.2
40	0.2*	1.3	1.7*	3.8*	7.9	(13.6)	(15.4)	17.8	22.2	14.7*
50	0.3*	1.1*	2.7	5.2	(7.8)	(11.6)	(5.0)	(6.9)	(7.0)	20.4
60	0.2*	1.0	3.1	3.8	(5.9)	(4.8)	(2.5)	(7.4)	(10.6)	(32.8)
80	0.1*	0.9	2.3	(2.8)	(1.9)	(1.9)	(4.7)	(6.7)	(12.5)	(12.4)
100	2.5	0.7	1.9	(2.9)	(1.7)	(2.1)	(6.1)	(6.1)	(6.1)	(10.6)
200	0.2	(0.6)	1.1	(0.8)	(1.3)	(1.8)	(3.5)	(4.1)	(3.5)	(8.7)

Table VIII. Average Relative Deviation for Potts' 2-Approximation Algorithm after Iterative Local Improvement

	m									
n	2	3	4	5	6	8	10	12	15	20
20	1.7	3.0	5.4	9.1	11.0	8.9*	10.8	14.2	4.8	—
30	0.2	2.7	4.6	7.9	6.9	13.4	12.7*	22.0	10.4*	3.7*
40	0.4	2.1	2.8	5.3	8.7	(13.6)	(17.6)	19.5	20.6	15.1
50	0.4	1.6	3.3	5.6	(7.6)	(11.8)	(8.6)	(6.6)	(6.2)	21.6
60	0.3	2.6	2.8	5.1	(6.7)	(1.9)	(3.1)	(10.1)	(4.2)	(37.0)
80	0.1*	1.9	2.2	(5.7)	(4.0)	(3.4)	(9.5)	(7.2)	(10.3)	(12.4)
100	2.3*	1.7	1.9	(4.4)	(3.1)	(2.6)	(3.9)	(3.9)	(9.3)	(10.6)
200	0.1*	(0.8)	1.0	(1.7)	(2.4)	(3.9)	(5.0)	(6.3)	(5.0)	(14.7)

Table IX. Average Relative Deviation for the Duality-Based Approximation Algorithm after Iterative Local Improvement

	m									
n	2	3	4	5	6	8	10	12	15	20
20	1.7	1.0*	5.4	5.4*	9.8	14.5	14.0	10.8*	7.4	3.0*
30	0.2	2.6	3.9	5.2	6.7*	14.2	15.0	19.7	11.3	14.8
40	0.4	1.0*	2.8	5.2	4.4*	(9.3)*	(15.2)*	13.9*	12.0*	16.4
50	0.4	1.5	2.3*	4.1*	(4.2)*	(7.2)*	(1.3)*	(0.0)*	(1.2)*	17.0*
60	0.3	0.5*	2.0*	2.7	(3.5)*	(1.0)*	(0.8)*	(0.8)*	(0.9)*	(22.8)*
80	0.1*	0.7*	1.0*	(1.9)*	(1.8)*	(0.6)*	(0.6)*	(2.2)*	(1.6)*	(4.5)*
100	2.3*	0.6*	1.1*	(2.2)*	(0.8)*	(0.9)*	(0.7)*	(0.7)*	(0.3)*	(0.7)*
200	0.1*	(0.5)*	0.7*	(0.3)*	(0.1)*	(0.8)*	(1.1)*	(0.4)*	(1.2)*	(0.0)*

Table X. Number of Times (Out of 10) That the Duality-Based Approximation Algorithm Performed at Least as Well as the Other Approximation Algorithms

	m									
n	2	3	4	5	6	8	10	12	15	20
20	1	7	5	7	4	4	3	6	7	9
30	7	4	4	4	7	6	4	7	4	4
40	5	6	3	3	8	9	4	7	9	5
50	7	4	5	7	7	6	6	10	8	9
60	4	8	7	6	8	4	5	9	8	9
80	2	5	7	6	9	6	8	6	9	7
100	9	4	5	6	5	6	8	8	9	9
200	6	7	6	7	9	7	8	8	8	10

were not run; for these instances, Potts' algorithm requires explicit enumeration of almost the entire state space.

As can be seen from the number of "*" signs in Table IX, the duality-based approximation algorithm has the best performance on the average. Note that the entries for $m = 2$ are identical for the duality-based algorithm and Potts' algorithm. In spite of their close relation, the duality-based approximation algorithm performs considerably better than Potts' algorithm.

Table X presents the number of times (out of 10) that the duality-based approximation algorithm produced the best or equally best solution. The algorithm performs remarkably well if m and n are large; apparently, these instances are beyond the reach of the iterative local improvement procedure and Potts' 2-approximation algorithm. In a sense, the duality-based approximation algorithm and the branch-and-bound algorithm are supplementary: the latter is effective for instances for which the former performs not so well as the other approximation algorithms. The running time is about a factor of two more than the running time of the constructive heuristics and Potts' approximation algorithm, but it is comparable or less in the extreme combinations with $n = 200$ or $m = 20$.

5. Conclusions

The $R||C_{max}$ problem is a practical scheduling problem for which we have proposed an optimization algorithm and an approximation algorithm. The optimization algorithm, of the branch-and-bound type, solves large instances to optimality within reasonable time limits. The approximation algorithm is based upon a simple and intuitively appealing idea for local search: heuristic duality-based search in conjunction with iterative local improvement. For instances that are beyond the reach of an optimization algorithm, it produces very good results.

Acknowledgment

The author is grateful to Jan Karel Lenstra and the referees for their helpful comments on earlier drafts of this paper.

References

1. E. Balas and N. Christofides, 1981. A Restricted Lagrangean Approach to the Traveling Salesman Problem, *Mathematical Programming 21*, 19–46.
2. M. Berrada and K.E. Stecke, 1986. A Branch-and-Bound Approach for Machine Load Balancing in Flexible Manufacturing Systems, *Management Science 32*, 1316–1335.
3. O. Bilde and S. Krarup, 1977. Scharp Lower Bounds and Efficient Algorithms for the Simple Plant Location Problem, *Annals of Discrete Mathematics 1*, 79–88.
4. N. Christofides, 1970. The Shortest Hamiltonian Chain of a Graph, *SIAM Journal of Applied Mathematics 19*, 689–696.
5. E. Davis and J.M. Jaffe, 1981. Algorithms for Scheduling Tasks on Unrelated Parallel Processors, *Journal of the Association of Computing Machinery 28*, 721–736.
6. P. De and T.E. Morton, 1980. Scheduling to Minimize Makespan on Unequal Parallel Processors, *Decision Sciences 11*, 586–603.
7. D. de Werra and A. Hertz, 1989. Tabu Search Techniques: A Tutorial and An Application to Neural Networks, *OR Spektrum 11*, 131–141.
8. D. Erlenkotter, 1978. A Dual-Based Procedure for Uncapacitated Facility Location, *Operations Research 26*, 992–1009.
9. M.L. Fisher, 1981. The Lagrangian Relaxation Method for Solving Integer Programming Problems, *Management Science 27*, 1–18.
10. M.L. Fisher, R. Jaikumar and L.N. Van Wassenhove, 1986. A Multiplier Adjustment Method for the Generalized Assignment Problem, *Management Science 32*, 1098–1103.
11. M.L. Fisher and P. Kedia, 1990. Optimal Solution of Set Covering/Partitioning Problems using Dual Heuristics, *Management Science 32*, 1098–1103.
12. F. Glover, 1989. Tabu Search—Part I. *ORSA Journal on Computing 1*, 190–206.
13. T. Gonzalez, E.L. Lawler, and S. Sahni, 1990. Optimal Preemptive Scheduling of Two Unrelated Processors, *ORSA Journal on Computing 2*, 219–224.
14. R.L. Graham, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan, 1979. Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey, *Annals of Discrete Mathematics 5*, 287–326.
15. H.J. Greenberg, W.P. Pierskalla, 1970. Surrogate Mathematical Programming, *Operations Research 18*, 924–939.
16. M. Guignard and M.B. Rosenwein, 1989. An Application-Oriented Guide for Designing Lagrangean Dual Ascent Algorithms, *European Journal of Operational Research 43*, 197–205.
17. M. Guignard and K. Spielberg, 1979. A Direct Dual Method of the Mixed Plant Location Problem with Some Side Constraints, *Mathematical Programming 17*, 198–228.
18. A.M.A. Hariri and C.N. Potts, 1991. Heuristics for Scheduling Unrelated Parallel Machines, *Computers and Operations Research 18*, 313–321.
19. E. Horowitz and S. Sahni, 1976. Exact and Approximate Algorithms for Scheduling Nonidentical Processors, *Journal of the Association of Computing Machinery 23*, 317–327.
20. O.H. Ibarra and C.G. Kim, 1977. On Heuristic Algorithms for Scheduling Independent Tasks on Nonidentical Processors, *Journal of the Association of Computing Machinery 24*, 280–289.
21. M.H. Karwan and R.L. Rardin, 1979. Some Relationships between Lagrangian and Surrogate Duality in Integer Programming, *Mathematical Programming 17*, 320–334.
22. J.K. Lenstra, D.B. Shmoys and E. Tardos, 1990. Approximation Algorithms for Scheduling Unrelated Parallel Machines, *Mathematical Programming 46*, 259–271.

23. C.H. PAPADIMITRIOU and K. STEIGLITZ, 1982. *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ.
24. C.N. POTTS, 1985. Analysis of a Linear Programming Heuristic for Scheduling Unrelated Parallel Machines, *Discrete Applied Mathematics 10*, 155–164.
25. H.I. STERN, 1976. Minimizing Makespan for Independent Jobs on Nonidentical Machines—An Optimal Procedure, Working Paper 2/75, Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer-Sheva.
26. S.L. VAN DE VELDE, 1991. Machine Scheduling and Lagrangian Relaxation, Ph.D. Thesis, CWI, Amsterdam.
27. P.J.M. VAN LAARHOVEN and E.H.L. AARTS, 1987. *Simulated Annealing: Theory and Applications*, Reidel, Dordrecht.