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 \( \mathcal{APX} \)-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 \( J = \{ J_1, \ldots, J_n \} \). 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. Barrada 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 \( \mathcal{APX} \)-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 \( \rho \) times the optimal length is referred to as a \( \rho \)-approximation algorithm; such an approximation algorithm is said to have performance guarantee \( \rho \). 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.
A polynomial approximation scheme is a family of algorithms that contains for any \( \rho > 1 \) a \( \rho \)-approximation algorithm with a running time that is bounded by a polynomial in the problem size; this running time may depend on \( \rho \). A family of algorithms is called a fully polynomial approximation scheme if it contains for any \( \rho > 1 \) a \( \rho \)-approximation algorithm for which the running time is bounded by a polynomial in the problem size as well as in \( 1/(\rho - 1) \).

Potting 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, Potting improves the worst-case ratio to \((1 + \sqrt{2})/2\). 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 Potting’s algorithm as the basis, Lenstra, Shmoys, and Tardos 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 \( \mathcal{NP} = \mathcal{NPH} \), no polynomial \( \rho \)-approximation algorithm exists for any \( \rho < \sqrt{2} \).

Two papers consider the \( R || C_{\text{max}} \) from an empirical point of view. De and Morton present several hybrid list scheduling algorithms and perform a large-scale computational testing. Hariri and Potting propose several two-phase heuristics that proceed in the spirit of Potting’s 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 Potting’s 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 \( R || C_{\text{max}} \) 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 \( R || C_{\text{max}} \) 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 \( R || C_{\text{max}} \) 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 (Potting). 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 \( R || C_{\text{max}} \), obtained from a 0-1 linear program 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 \( j \) 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 \( C_{\text{max}} \), is then the length of the schedule.

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

\[ C_{\text{max}} \]  

subject to

\[ \sum_{j=1}^{n} p_{ij} x_{ij} \leq C_{\text{max}}, \quad \text{for } i = 1, \ldots, m, \]  

\[ \sum_{i=1}^{m} x_{ij} = 1, \quad \text{for } j = 1, \ldots, n, \]  

\[ x_{ij} \in \{0, 1\}, \quad \text{for } i = 1, \ldots, m, j = 1, \ldots, n. \]

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 \( \mathcal{NPH} \)-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 \( x_{ij} \geq 0 \) \( (i = 1, \ldots, m, j = 1, \ldots, n) \) 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[13] 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 \( \lambda = (\lambda_1, \ldots, \lambda_m) \geq 0 \) with \( \lambda_i > 0 \) for at least one \( i (i = 1, \ldots, m) \), we replace the conditions (1) with
\[
\sum_{i=1}^{m} \lambda_i \sum_{j=1}^{n} p_{ij} x_{ij} \leq \sum_{i=1}^{m} \lambda_i c_{\max}, \tag{1a}
\]
or, equivalently,
\[
c_{\max} \geq \sum_{i=1}^{m} \sum_{j=1}^{n} \lambda_i p_{ij} x_{ij} / \sum_{i=1}^{m} \lambda_i. \tag{1b}
\]
The surrogate relaxation problem, referred to as problem \((S_\lambda)\), is then to determine \( S(\lambda) \), which is the minimum of
\[
\sum_{i=1}^{m} \sum_{j=1}^{n} \lambda_i p_{ij} x_{ij} / \sum_{i=1}^{m} \lambda_i \tag{S_\lambda}
\]
subject to
\[
\sum_{i=1}^{m} x_{ij} = 1, \quad \text{for} \ j = 1, \ldots, n, \tag{2}
\]
\[
x_{ij} \in \{0, 1\}, \quad \text{for} \ i = 1, \ldots, m, \ j = 1, \ldots, n. \tag{3}
\]
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 \( u(\cdot) \) denote the optimal solution value of problem (\( S_\lambda \)).

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

**Observation 2.** Problem \((S_\lambda)\) is solvable in \( O(nm) \) time by assigning each job \( J_j \) to a machine \( M_k \) for which \( \lambda_k p_{kj} = \min_{1 \leq k \leq m} \lambda_k p_{kj} \). Ties may be settled arbitrarily.

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

**Observation 3.** Any solution to \((S_\lambda)\) is also a feasible solution to the primal problem (P), for any vector \( \lambda = (\lambda_1, \ldots, \lambda_m) \geq 0 \) 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 \((S_\lambda)\), let \( C(\lambda) \) denote the completion time of \( M_i \). The approximate solution value is then \( C_{\max}(\lambda) = \max_{1 \leq i \leq m} C(\lambda) \). The way we settle ties when solving problem \((S_\lambda)\) affects \( C_{\max}(\lambda) \).

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

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 \((S_\lambda)\) 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 \( S(\lambda) = 18/3 \); 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
\[
u(D) = \max \{ S(\lambda) \mid \lambda > 0 \}. \tag{D}
\]

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

**Table I.** Processing Time Matrix

<table>
<thead>
<tr>
<th></th>
<th>( J_1 )</th>
<th>( J_2 )</th>
<th>( J_3 )</th>
<th>( J_4 )</th>
<th>( J_5 )</th>
<th>( J_6 )</th>
<th>( J_7 )</th>
<th>( J_8 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( M_1 )</td>
<td>6</td>
<td>3</td>
<td>10</td>
<td>12</td>
<td>11</td>
<td>14</td>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>( M_2 )</td>
<td>10</td>
<td>( \infty )</td>
<td>15</td>
<td>6</td>
<td>6</td>
<td>11</td>
<td>14</td>
<td>7</td>
</tr>
<tr>
<td>( M_3 )</td>
<td>11</td>
<td>9</td>
<td>14</td>
<td>( \infty )</td>
<td>10</td>
<td>10</td>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 1.** Gantt chart for \( \lambda = (1, 1, 1) \); the dotted line indicates the lower bound \( S(\lambda) \).
Observation 5. Since \((S_d)\) 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 \(\lambda^*\) can be found in polynomial time by solving problem \((\bar{P})\). Nonetheless, we develop a so-called ascent direction algorithm to approximate \(\lambda^*\). 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\(^{[12]}\), the traveling salesman problem (Balas and Christofides\(^{[1]}\); Christofides\(^{[3]}\)), 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 Veldoo\(^{[26]}\).

The shape of the function \(S: \lambda \rightarrow S(\lambda)\) 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 \((S_d)\), 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 \(K \leq m^n\). Problem (D) is then equivalent to the following problem: maximize

\[
\begin{align*}
\max_{z} & \quad z \\
\text{subject to} & \quad A x^k \leq \lambda, & & k = 1, \ldots, K, \\
& \lambda \geq 0 \quad \text{and} \quad \sum_{i=1}^{m} \lambda_i > 0,
\end{align*}
\]

where \(A\) represents the processing time matrix. Hence, the function \(S: \lambda \rightarrow S(\lambda)\) is the lower envelope of a finite set of functions of the type \(\lambda \rightarrow \lambda A x^k / \sum_{i=1}^{m} \lambda_i\); it implies that \(S\) is continuous in \(\lambda\) and everywhere differentiable except at all \(\lambda\) where \((S_d)\) has multiple optimal solutions. We call such \(\lambda\) the points of non-differentiability. We have depicted in Figure 2 the shape of \(S\) for \(m = 2\) and \(K = 3\) with \(\lambda_1\) fixed. The form of the function \(S\) is unusual; this is because of the term \(\sum_{i=1}^{m} \lambda_i\) appearing in the denominator. If the term \(\sum_{i=1}^{m} \lambda_i\) were absent, then the function \(S\) would be the lower envelope of a finite set of linear functions (cf. Fisher\(^{[3]}\)).

The directional derivative of the function \(S(\lambda)\) is defined as

\[
S_u(\lambda) = \left. \frac{S(\lambda + \epsilon u) - S(\lambda)}{\epsilon} \right|_{\epsilon \to 0}, \tag{4}
\]

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

\[
S_u(\lambda) \leq 0, \quad \forall \, u \in \mathbb{R}^m; \tag{5}
\]

later on, we show that condition (5) is also sufficient for optimality. If \(S_u(\lambda) > 0\) for some \(u \in \mathbb{R}^m\), then \(\bar{u}\) is called an ascent direction of \(L\) at \(\lambda\); we get an improved lower bound by moving some scalar step size \(\Delta\) 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 \(\lambda\).

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 \(\lambda\) for any primitive vector with the \(i\)th component positive; hence, \(l^+_i(\lambda)\) is the derivative for increasing \(\lambda_i\). For \(i = 1, \ldots, m\), let \(l^-_i(\lambda)\) be the directional derivative at \(\lambda\) for any primitive vector with the \(i\)th component negative; hence, \(l^-_i(\lambda)\) is the derivative for decreasing \(\lambda_i\). The first step is to rewrite definition (4) for the primitive directional derivatives; accordingly, we get

\[
l^+_i(\lambda) = \lim_{\epsilon \to 0} \frac{S(\lambda + \epsilon e_i) - S(\lambda)}{\epsilon}, \quad \text{and}
\]

\[
l^-_i(\lambda) = \lim_{\epsilon \to 0} \frac{S(\lambda - \epsilon e_i) - S(\lambda)}{\epsilon},
\]

where \(e_i\) is the \(i\)th unity vector, for \(i = 1, \ldots, m\). Note that \(l^-_i(\lambda)\) is undefined for any \(\lambda\) with \(\lambda_i = 0\).
The second step is to simplify the above expressions. First, recall that any problem \( (S_i) \) may have multiple optimal solutions; let \( X^{(a)} \) be the set containing them. Second, for any \( h (h = 1, \ldots, m) \), a sufficiently small step size \( \Delta_i^h > 0 \) exists to ensure that some \( x(h)^+ \in X^{(a)} \) is also optimal for problem \( (S_{i = 1, \ldots, m}) \) for any \( \epsilon \) with \( 0 < \epsilon < \Delta_i^h \); this can easily be proven in a constructive way. Hence, such an \( x(h)^+ \) must be an optimal solution to problem \( (S_i) \) with as few as possible jobs assigned to \( M_i \). To get such an \( x(h)^+ \), each job \( j \) with \( \lambda_j p_{j} \) minimal and with \( \lambda_j p_{j} = \lambda_j p_{j} \) for some \( M_i \neq M_j \), is not assigned to \( M_i \); all other ties are settled arbitrarily. Let \( C_i^+ (\lambda) \) be the completion time of \( M_i \) for such an \( x(h)^+ \); let \( J_i^+(\lambda) \) denote the set of jobs on \( M_i \) for such an \( x(h)^+ \). Similarly, for any \( h (h = 1, \ldots, m) \), a sufficiently small step size \( \Delta_i^h > 0 \) exists to ensure that some \( x(h)^- \in X^{(a)} \) is also optimal for problem \( (S_{i = 1, \ldots, m}) \) for any \( \epsilon \) with \( 0 < \epsilon < \Delta_i^h \). Such an \( x(h)^- \) must be an optimal solution to problem \( (S_i) \) with as many as possible jobs assigned to \( M_i \). To get such an \( x(h)^- \), each job \( j \) with \( \lambda_j p_{j} \) minimal is assigned to \( M_j \); all other ties are settled arbitrarily. Let \( C_i^- (\lambda) \) be the completion time of \( M_i \) for such an \( x(h)^- \); let \( J_i^-(\lambda) \) denote the set of jobs on \( M_i \) for such an \( x(h)^- \).

For a specific \( x(h)^+ \), we have then that \( C_i(\lambda + \epsilon x(h)^+) = C_i(\lambda) \) for each \( i (i = 1, \ldots, m) \) for \( 0 < \epsilon < \Delta_i^h \), and that \( C_i(\lambda) = C_i^+(\lambda) \).

Hence, for \( 0 < \epsilon < \Delta_i^h \), we have

\[
S(\lambda + \epsilon x(h)^+) = \frac{\epsilon C_i^+(\lambda) + \sum_{j=1}^{m} \lambda_j C_j(\lambda)}{\epsilon + \sum_{i=1}^{m} \lambda_i} \leq \frac{\epsilon C_i^+(\lambda) + \sum_{j=1}^{m} \lambda_j C_j(\lambda)}{\epsilon + \sum_{i=1}^{m} \lambda_i} = \frac{\epsilon C_i^-(\lambda) \sum_{j=1}^{m} \lambda_j}{\epsilon + \sum_{i=1}^{m} \lambda_i}.
\]

This gives that

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

Using this, we obtain for the primitive directional derivative that

\[
l_i^+(\lambda) = \frac{[C_i^+(\lambda) - S(\lambda)]}{\sum_{i=1}^{m} \lambda_i}.
\]

In a similar fashion, we get that

\[
l_i^-(\lambda) = \frac{[S(\lambda) - C_i^-(\lambda)]}{\sum_{i=1}^{m} \lambda_i}.
\]

If \( C_i^+(\lambda) > S(\lambda) \), then machine \( M_i \) is overloaded. For a sufficiently small \( \epsilon > 0 \), ensuring that some optimal solution for problem \( (S_i) \) remains optimal for problem \( (S_{i = 1, \ldots, m}) \), we have then that \( (S(\lambda + \epsilon x(h)^+)) > S(\lambda) \). In other words, increasing \( \lambda_h \) is an ascent direction: we will obtain an improved surrogate objective value by moving along this direction. If \( C_i^+(\lambda) < S(\lambda) \), then machine \( M_i \) is called underloaded: decreasing \( \lambda_h \) 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 \( \lambda \) is not a point of non-differentiability. For \( h = 1, \ldots, m \), let \( \Delta_h^- > 0 \) be the step size for decreasing \( \lambda_h \) to reach the nearest point of non-differentiability; such a \( \Delta_h^- \) always exists, because any \( \lambda \) 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

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

Since \( S(\lambda) \) is a convex combination of the machine completion times \( C_i(\lambda), \ldots, C_m(\lambda) \), we have \( S(\lambda(h)^-) \geq S(\lambda) \) for at least one \( h \). Hence, for problem \( (D) \), we can restrict ourselves to the optimization over all \( \lambda \geq 0 \) 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 to where the associated primitive directional derivative changes. The required step size is easily determined. Suppose \( \Delta_h^-(\lambda) > 0 \) for some \( M_i \) is overloaded. Increasing \( \lambda_h \) makes \( M_i \) less attractive to schedule jobs on. Eventually, we reach the first point where some \( J_i \) currently scheduled on \( M_i \) can equally well be scheduled on some other machine \( M_j \); moving on beyond this point, we enforce the removal of \( J_i \) from \( M_i \). The step size \( \Delta \) to reach this point is the smallest positive value for which \( \lambda_h + \Delta p_{j} = \lambda_g p_{g_j} \) for some \( i \) on \( M_i \) and some \( M_j \). Hence, it is computed as

\[
\Delta = -\lambda_h + \min_{i \in J_i, j \in J_i, \epsilon \in [-\Delta, 0]} \lambda_j p_{j} / \lambda_i.
\]

Accordingly, we get \( \lambda = (\lambda_1, \ldots, \lambda_h + \Delta, \ldots, \lambda_m) \); the increment to the objective value is \( S(\lambda) - S(\lambda) = \Delta [C_i^+(\lambda) - S(\lambda)] / \left( \lambda_h + \sum_{i=1}^{m} \lambda_i \right) > 0 \). Furthermore, we move \( J_i \) from \( M_i \) to \( M_j \), and examine whether increasing \( \lambda_h \) is also an ascent direction.

Now, suppose \( \Delta_h^-(\lambda) > 0 \); \( M_i \) is underloaded. Decreasing \( \lambda_h \) makes \( M_i \) more attractive. Eventually, we reach the first point where some \( J_i \) on \( M_i \) can equally well be scheduled on \( M_i \); moving on beyond this point, we force \( J_i \) to go to \( M_j \). The required step size \( \Delta \) is the smallest positive value for which \( \lambda_h - \Delta p_{j} = \lambda_g p_{g_j} \) for some \( J_i \) on \( M_i \); it is computed as

\[
\Delta = \lambda_h - \min_{i \in J_i, j \in J_i, \epsilon \in [-\Delta, 0]} \lambda_j p_{j} / \lambda_i.
\]

Accordingly, we get \( \lambda = (\lambda_1, \ldots, \lambda_h - \Delta, \ldots, \lambda_m) \), and the increment to the objective value is \( S(\lambda) - S(\lambda) = \Delta [S(\lambda) - C_i^+(\lambda)] / \left( \lambda_h - \sum_{i=1}^{m} \lambda_i \right) > 0 \); we move \( J_i \) to \( M_i \), and examine whether decreasing \( \lambda_h \) 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 \( \bar{\lambda} \), we have
\[
\begin{align*}
I_{1i}^+ (\bar{\lambda}) &< 0, \quad \text{and} \quad I_{1i}^- (\bar{\lambda}) < 0, \quad \text{for} \quad i = 1, \ldots, m, \\
\sum_{I_{1i}^+ (\bar{\lambda})} p_{ij} &\leq S(\bar{\lambda}) \leq \sum_{I_{1i}^- (\bar{\lambda})} p_{ij}, \quad \text{for} \quad i = 1, \ldots, m.
\end{align*}
\]
or equivalently,
\[
\sum_{I_{1i}^+ (\bar{\lambda})} p_{ij} \leq S(\bar{\lambda}) \leq \sum_{I_{1i}^- (\bar{\lambda})} p_{ij}, \quad \text{for} \quad i = 1, \ldots, m.
\]
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 (Q)

Step 0. For \( k = 1, \ldots, m \), set \( \lambda_k \leftarrow 1 \). Solve problem (\( S_\lambda \)), settling ties arbitrarily. Determine \( S(\lambda) \).

Step 1. For \( k = 1, \ldots, m \), do the following:

(a) While \( C_k^+ (\lambda) > S(\lambda) \), compute
\[
\Delta = -\lambda_k + \min_{1 \leq i \leq m, j \in J_k} \frac{\Delta_i}{\lambda_j} p_{ij}/p_{jh},
\]
set \( \lambda_k \leftarrow \lambda_k + \Delta \), and update \( S(\lambda) \) and \( C_k^+ (\lambda) \).

(b) While \( C_k^- (\lambda) < S(\lambda) \), compute
\[
\Delta = \lambda_k - \min_{1 \leq i \leq m, j \in J_k} \frac{\Delta_i}{\lambda_j} p_{ij}/p_{jh},
\]
set \( \lambda_k \leftarrow \lambda_k - \Delta \), and update \( S(\lambda) \) and \( C_k^- (\lambda) \).

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

Let us reconsider our example and the solution of (\( S_\lambda \)) with \( \lambda = (1, 1, 1) \). Machine \( M_1 \) is overloaded. The step size to remove some job from \( M_1 \) is \( \Delta = 1/\gamma_1 \); increasing \( \lambda_1 \) by \( \gamma_1 \) allows us to move \( J_6 \) to \( M_2 \). We get \( \lambda = (1, 1, 1) \), 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 \( \lambda_1 \) to \( \gamma_4 \), and subsequently move \( J_7 \) to \( M_3 \). We get \( \lambda = (1, 1, 1) \), a schedule with makespan 20, and \( S(\lambda) = 19.3 \) (see Figure 4).

Since all processing times are integral, the optimal 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 \( \lambda_2 \) by \( \gamma_1 \), then \( J_6 \) goes to \( M_2 \). We obtain \( \lambda = (1, 1, 1) \) and \( S(\lambda) = 19.3 \). At this point, no primitive ascent direction exists anymore; the ascent direction method is terminated at \( \lambda = (1, 1, 1) \).

The vector \( \lambda \) is not optimal for the surrogate dual problem, i.e., \( \lambda \neq \lambda^* \). For \( \lambda^* \), we have that \( \nu(\hat{\lambda}) = S(\lambda^*) \) (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
\[
x_{ij} > 0 \Rightarrow \lambda^*_i p_{ij} = \min_{1 \leq i \leq m, j} \lambda^*_i p_{ij}.
\]

Considering Figure 4, we are allowed to split only \( J_6 \) over \( M_3 \) and \( M_2 \) and \( J_7 \) over \( M_3 \) and \( M_4 \), 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 (\( \hat{P} \)). The algorithm stops if some \( n - m + 1 \) jobs are 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 \( \nu(\hat{P}) < \nu(P) \), the length of the partial schedule is no more than \( \nu(P) \). The scheduling of the split jobs proceeds by complete enumeration; this adds at most \( \nu(P) \) to the length of the partial schedule. Hence, the resulting schedule has a makespan at most twice the optimal makespan. Since \( \hat{P} \) is solvable in polynomial time and complete enumeration for at most \( m - 1 \) split jobs requires \( O(m^2) \) time, the procedure is polynomial for fixed \( m \).

Consider now an optimal solution of problem (\( S^- \)). Since \( \bar{\lambda} \) is the vector upon termination of the ascent direction...
method, we have
\[ \sum_{j_i \in \mathcal{F}_i(\lambda)} p_{j_i} \leq S(\lambda) \leq \sum_{j_i \in \mathcal{F}_i(\lambda)} p_{j_i}, \quad \text{for } i = 1, \ldots, m. \]

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 \( j_i \in \mathcal{F}_i(\lambda) \) to \( M_i \), thus obtaining a partial schedule with length no more than \( \nu(P) \). The remaining jobs, contained in the set \( \mathcal{J} = \bigcup_{i=1}^{m} \mathcal{F}_i(\lambda) \), 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 \( \nu(P) \) to the length of the partial schedule. Hence, the resulting schedule has makespan no more than \( 2 \nu(P) \).

In fact, we get also a schedule with worst-case ratio 2 with fewer jobs to assign in the second phase as follows. Let \( \mathcal{F}_o(\lambda) \subseteq \mathcal{F}_i(\lambda) \cup \mathcal{F}_o(\lambda) \) be mutually disjoint subsets of jobs such that
\[ \sum_{j_i \in \mathcal{F}_o(\lambda)} p_{j_i} \leq S(\lambda), \quad \text{for each } i = 1, \ldots, m; \]
hence, \( \mathcal{F}_o(\lambda) \) contains only jobs with ties concerning their minimal dual processing time. In the first phase, we assign each \( j_i \in \mathcal{F}_o(\lambda) \cup \mathcal{F}_o(\lambda) \) to \( M_i \); in the second phase, we assign the remaining jobs. The sets \( \mathcal{F}_o(\lambda) \) 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 \( \lambda = \lambda^* \), then this procedure is exactly Potts’ 2-approximation algorithm; since the complementary slackness relations holds for \( \lambda = \lambda^* \), we can choose the sets \( \mathcal{F}_o(\lambda) \) in such a way that no more than \( m - 1 \) jobs remain for the second phase.

For the special case \( m = 2 \), we have \( \nu(P) = S(\lambda) \). Since the surrogate multipliers represent normalized values, only \( m - 1 \) multipliers need to be involved to find \( \lambda^* \). For the case \( m = 2 \), only one multiplier is involved; the termination conditions at \( \lambda = \lambda^* \) are then sufficient for optimality. For \( m = 2 \), problem \( \tilde{P} \) is solvable in \( O(n) \) time (Gonzalez, Lawler, and Sahni[32]). 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 \( M_i \) be the machine with the largest overload in the solution of problem (S); hence, we have \( C_{\max}(\lambda) = C_i(\lambda) \). Then we compute
\[ \Delta = -\lambda_i + \min_{1 \leq h \leq m, j_i \in \mathcal{F}_i(\lambda)} \lambda_j p_{j_h} / p_{j_i}, \]

where \( \min 2 \) denotes the second minimum of these values. If we put \( \lambda = (\lambda_1, \ldots, \lambda_i + \Delta, \ldots, \lambda_m) \), then we enforce the move of some \( j_h \) from \( M_i \) to some \( M_j \), and that another job on \( M_j \) can equally well be scheduled on some other machine. Nonetheless, this second job is kept on \( M_i \). The next step is to compute the new makespan \( C_{\max}(\lambda) \), and the machine with the largest overload; this machine is determined by computing \( \max_{1 \leq i \leq m} C_i(\lambda) \). We have no guarantee that the rescheduling of \( j_h \) induces an improved schedule: we can have either \( C_{\max}(\lambda) \leq C_{\max}(\lambda) \) or \( C_{\max}(\lambda) > C_{\max}(\lambda) \). The latter occurs if \( C_i(\lambda) = C_i(\lambda) + p_{j_h} > C_{\max}(\lambda) \). 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 \( \lambda = (1, \ldots, 1) \), \( t = 1 \). Solve (S), settling ties arbitrarily. Let \( UB \leftarrow C_{\max}(\lambda) \), and store the schedule.

**Step 1.** Determine \( M_b \) with the largest overload: \( C_i(\lambda) = C_i(\lambda) \) for each \( i = 1, \ldots, m \). Compute \( \Delta \), and identify a job \( j_1 \) and a machine \( M_b \) such that \( \lambda_j p_{j_k} / p_{j_b} = \min_{1 \leq i \leq m, 1 \leq h \leq m, j_i \in \mathcal{F}_i(\lambda)} \lambda_j p_{j_h} / p_{j_b} \). Put \( t = t + 1 \).

**Step 2.** Put \( \lambda = (\lambda_1, \ldots, \lambda_i + \Delta, \ldots, \lambda_m) \), \( C(\lambda) \leftarrow C(\lambda) - p_{j_b} \), \( C(\lambda) \leftarrow C(\lambda) + p_{j_k} \). If \( C_{\max}(\lambda) < UB \), then \( UB \leftarrow C_{\max}(\lambda) \), and store the schedule. If \( t < maxiter \), 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 \( \sigma \) be some arbitrary schedule and let \( \sigma_k \) be the schedule obtained from \( \sigma \) by swapping \( J_j \) and \( J_k \) \((j \neq k)\). We define the so-called *single pairwise interchange neighborhood* for \( \sigma \) as the set \( N_\sigma \) containing the schedules \( \sigma_k \) 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 \( \sigma \), 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

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

If we interchange \( J_j \) and \( J_k \), that is, we put \( J_j \) on \( M_h \) and \( J_k \) on \( M_h \), then we reduce the makespan. In other words, we have identified a schedule \( \sigma_k \in N_\sigma \) with \( C_{\max}(\sigma_k) < C_{\max}(\sigma) \). 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\(^{[7]}\)) 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.

### 3.1. 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 \( \lambda^* \) and vectors close to it.

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

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

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 \( J_i \) scheduled on \( M_h \). Solving the surrogate relaxation problem \( (S_\lambda) \) under the additional constraint \( x_{hk} = 1 \) gives the lower bound \( LB \) with

\[
LB = \left( \lambda_h p_{hk} + \sum_{j=1, j \neq k}^{n} \min_{1 \leq i \leq m} \lambda_i p_{ij} \right) \sum_{i=1}^{m} \lambda_i \cdot \left( \lambda_h p_{hk} - \min_{1 \leq i \leq m} \lambda_i p_{ik} \right) + \sum_{j=1}^{n} \min_{1 \leq i \leq m} \lambda_i p_{ij}
\sum_{i=1}^{m} \lambda_i > UB - 1,
\]

which is a contradiction. \( \blacksquare \)

### 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 \( \gamma_j = \min_{2 \leq i \leq m} \bar{\lambda}_i p_{ij} - \min_{1 \leq i \leq m} \bar{\lambda}_i p_{ij} \), where \( \min 2 \) denotes the second minimum. In view of Theorem 3, a large value \( \gamma_j \) 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 \( J_i \).

We like to structure the search tree in such a way that we first explore the configurations with jobs with large \( \gamma_j \) assigned to their favorite machines. This is achieved by reindexing the jobs in order of nonincreasing values \( \gamma_j \) and by reindexing the machines at each level \( k (k = 1, \ldots, n-1) \) in order of nondecreasing values \( \bar{\lambda}_i p_{i,k} (i = 1, \ldots, m) \). We note that the first complete schedule encountered in the tree is an optimal solution for the surrogate problem (S2).

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 \( \bar{\lambda} \) may therefore be more effective; it may be worthwhile to use a linear programming algorithm in the root of the tree to obtain \( \bar{\lambda} \). On the other hand, if \( \bar{\lambda} \) is close to \( \bar{\lambda} \), 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 (S2) subject to \( x_{ij} = z_{ij} \) for \( i = 1, \ldots, m, \) \( j = 1, \ldots, k \). Then we have

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

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

\[
(\bar{\lambda}_h p_{hk} - \min_{1 \leq i \leq m} \bar{\lambda}_i p_{ik})/\sum_{i=1}^{m} \bar{\lambda}_i > UB - S(\bar{\lambda}, k-1) - 1.
\]

(F1)

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

\[
\sum_{j=1}^{k-1} p_{hj} z_{hj} + p_{hk} > UB - 1.
\]

(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 \( J_l (1 \leq l \leq k-1) \) that is currently scheduled on \( M_i \) for which

\[
p_{li} > p_{ik} \quad \text{and} \quad p_{hl} < p_{hk}.
\]

(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 \( J_l (k + 1 \leq l \leq n) \) for which

\[
\sum_{j=1}^{k} p_{lj} z_{lj} + p_{hl} > UB - 1, \quad \text{for each } M_i, \ i = 1, \ldots, m,
\]

(F4)

we discard the node, too. Similarly, if the condition (F4) applies to some \( J_l (k + 1 \leq l \leq n) \) 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 \( M_h (1 \leq h \leq m) \) for which

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

(F5)

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

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

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 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 \( m \geq 12 \) and \( n \leq 40 \). 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.

### 4.2. The Duality-Based Approximation Algorithm

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

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

<table>
<thead>
<tr>
<th>( m )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>30</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>50</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>60</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>80</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
<td>0</td>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>200</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

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

<table>
<thead>
<tr>
<th>( m )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>30</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>9</td>
<td>6</td>
<td>2</td>
<td>43</td>
<td>2</td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>12</td>
<td>39</td>
<td>39</td>
<td>214</td>
<td>63</td>
<td>3</td>
<td>10</td>
</tr>
<tr>
<td>50</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>16</td>
<td>57</td>
<td>285</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>204</td>
</tr>
<tr>
<td>60</td>
<td>1</td>
<td>1</td>
<td>3</td>
<td>33</td>
<td>105</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>373</td>
</tr>
<tr>
<td>80</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>96</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>6</td>
<td>12</td>
<td>87</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>200</td>
<td>3</td>
<td>40</td>
<td>52</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
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,[8] 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**

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

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

<table>
<thead>
<tr>
<th>$n$</th>
<th>$m$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td></td>
<td>4.2</td>
<td>5.4</td>
<td>6.6</td>
<td>10.5</td>
<td>11.3</td>
<td>16.4</td>
<td>15.2</td>
<td>14.6</td>
<td>7.4</td>
<td>3.0</td>
</tr>
<tr>
<td>30</td>
<td></td>
<td>1.5</td>
<td>4.9</td>
<td>6.0</td>
<td>9.8</td>
<td>8.9</td>
<td>14.7</td>
<td>16.7</td>
<td>21.0</td>
<td>14.0</td>
<td>15.2</td>
</tr>
<tr>
<td>40</td>
<td></td>
<td>1.9</td>
<td>4.2</td>
<td>3.5</td>
<td>9.0</td>
<td>8.3</td>
<td>(10.0)</td>
<td>(19.0)</td>
<td>14.4</td>
<td>13.5</td>
<td>19.3</td>
</tr>
<tr>
<td>50</td>
<td></td>
<td>1.6</td>
<td>3.3</td>
<td>4.9</td>
<td>7.4</td>
<td>(6.5)</td>
<td>(8.3)</td>
<td>(4.1)</td>
<td>(1.9)</td>
<td>(2.5)</td>
<td>18.1</td>
</tr>
<tr>
<td>60</td>
<td></td>
<td>1.2</td>
<td>1.1</td>
<td>4.1</td>
<td>5.5</td>
<td>(5.0)</td>
<td>(4.0)</td>
<td>(1.8)</td>
<td>(1.0)</td>
<td>(1.8)</td>
<td>(24.3)</td>
</tr>
<tr>
<td>80</td>
<td></td>
<td>1.4</td>
<td>2.3</td>
<td>2.9</td>
<td>(3.5)</td>
<td>(2.4)</td>
<td>(1.9)</td>
<td>(2.1)</td>
<td>(2.8)</td>
<td>(2.7)</td>
<td>(4.5)</td>
</tr>
<tr>
<td>100</td>
<td></td>
<td>2.3</td>
<td>2.3</td>
<td>2.4</td>
<td>(3.6)</td>
<td>(1.8)</td>
<td>(2.2)</td>
<td>(1.9)</td>
<td>(1.9)</td>
<td>(0.8)</td>
<td>(1.4)</td>
</tr>
<tr>
<td>200</td>
<td></td>
<td>0.4</td>
<td>(1.5)</td>
<td>1.1</td>
<td>(1.1)</td>
<td>(1.2)</td>
<td>(1.8)</td>
<td>(3.3)</td>
<td>(1.3)</td>
<td>(3.3)</td>
<td>(0.6)</td>
</tr>
</tbody>
</table>
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>
<thead>
<tr>
<th>$n$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>15</th>
<th>20</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>0.0*</td>
<td>1.0*</td>
<td>3.0*</td>
<td>7.1</td>
<td>8.3*</td>
<td>10.4</td>
<td>4.9*</td>
<td>11.1</td>
<td>19*</td>
<td>5.1</td>
</tr>
<tr>
<td>30</td>
<td>0.1*</td>
<td>1.2*</td>
<td>3.1*</td>
<td>4.4*</td>
<td>7.6</td>
<td>13.3*</td>
<td>15.0</td>
<td>17.4*</td>
<td>11.0</td>
<td>9.2</td>
</tr>
<tr>
<td>40</td>
<td>0.2*</td>
<td>1.3</td>
<td>1.7*</td>
<td>3.6*</td>
<td>7.9</td>
<td>(13.6)</td>
<td>(15.4)</td>
<td>17.8</td>
<td>22.2</td>
<td>14.7*</td>
</tr>
<tr>
<td>50</td>
<td>0.3*</td>
<td>1.1*</td>
<td>2.7</td>
<td>5.2</td>
<td>(7.8)</td>
<td>(11.6)</td>
<td>(5.0)</td>
<td>(6.9)</td>
<td>(7.0)</td>
<td>20.4</td>
</tr>
<tr>
<td>60</td>
<td>0.2*</td>
<td>1.0</td>
<td>3.1</td>
<td>3.8</td>
<td>(5.9)</td>
<td>(4.8)</td>
<td>(2.5)</td>
<td>(7.4)</td>
<td>(10.6)</td>
<td>(32.8)</td>
</tr>
<tr>
<td>80</td>
<td>0.1*</td>
<td>0.9</td>
<td>2.3</td>
<td>(2.8)</td>
<td>(1.9)</td>
<td>(1.9)</td>
<td>(4.7)</td>
<td>(6.7)</td>
<td>(12.3)</td>
<td>(12.4)</td>
</tr>
<tr>
<td>100</td>
<td>2.5</td>
<td>0.7</td>
<td>1.9</td>
<td>(2.9)</td>
<td>(1.7)</td>
<td>(2.1)</td>
<td>(6.1)</td>
<td>(6.1)</td>
<td>(6.1)</td>
<td>(10.6)</td>
</tr>
<tr>
<td>200</td>
<td>0.2</td>
<td>(0.6)</td>
<td>1.1</td>
<td>(0.8)</td>
<td>(1.3)</td>
<td>(1.8)</td>
<td>(3.5)</td>
<td>(4.1)</td>
<td>(3.5)</td>
<td>(8.7)</td>
</tr>
</tbody>
</table>

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

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

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

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

Table IX. Average Relative Deviation for the Duality-Based Approximation Algorithm after Iterative Local Improvement
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 $R||C_{\text{MAX}}$ 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


