Journal of Combinatorial Optimization (2020) 40:876–900
https://doi.org/10.1007/s10878-020-00634-9

A characterization of optimal multiprocessor schedules and new dominance rules

Rico Walter[1] · **Alexander Lawrinenko**[1]

Published online: 12 August 2020

Abstract
The paper on hand approaches the classical makespan minimization problem on identical parallel machines from a rather theoretical point of view. Using an approach similar to the idea behind inverse optimization, we identify a general structural pattern of optimal multiprocessor schedules. We also show how to derive new dominance rules from the characteristics of optimal solutions. Results of our computational study attest to the efficacy of the new rules. They are particularly useful in limiting the search space when each machine processes only a few jobs on average.

Keywords Scheduling · Identical parallel machines · Makespan · Solution structure · Dominance rules

1 Introduction

The present paper is concerned with the multiprocessor scheduling problem. Given a set $M = \{M_1, \ldots, M_m\}$ of $m \geq 2$ identical parallel machines and a set $J = \{J_1, \ldots, J_n\}$ of $n > m$ independent jobs with positive processing times $p_1, p_2, \ldots, p_n$, the objective is to assign the jobs to the machines so that the latest machine completion time (also called *makespan*) $C_{max} = \max\{C_1, \ldots, C_m\}$—with C_i being the sum of processing times of all jobs assigned to M_i—is minimized. Preemption is not allowed. Using the three-field notation of Graham et al. (1979) this problem is abbreviated as $P||C_{max}$. In the literature, $P||C_{max}$ is also known as the makespan minimization problem on identical parallel machines.

✉ Rico Walter
rico.walter@uni-jena.de
https://www.mansci.uni-jena.de

Alexander Lawrinenko
a.lawrinenko86@gmail.com

[1] Chair for Management Science, Friedrich Schiller University Jena, Carl-Zeiß-Straße 3, 07743 Jena, Germany

The $\mathcal{NP}$-hard problem $P||C_{max}$ (see Garey and Johnson 1979) represents one of the very basic and fundamental problems in scheduling theory. It has received and still receives a lot of attention from both the academic world and practitioners. The large body of literature, that has evolved over the years, contains papers on approximation algorithms, (meta-)heuristics, exact solution procedures, and lower bounding techniques.

From the numerous publications on (meta-)heuristic algorithms within the last two decades, we selected the following few ones to outline the broad range of near-optimal solution approaches. Alvim and Ribeiro (2004) exploited the "dual" relation between $P||C_{max}$ and the bin packing problem (BPP). They proposed a hybrid improvement heuristic that consists of construction, redistribution, and improvement phases. In the latter phase, tabu search is applied. Frangioni et al. (2004) proposed new neighborhood operators for local search algorithms that perform multiple exchanges of jobs among machines. Dell'Amico et al. (2008) presented an effective meta-heuristic algorithm based on the scatter search paradigm. Kashan and Karimi (2009) presented a discrete particle swarm optimization algorithm and a hybrid version, that makes use of an efficient local search algorithm to further improve on the makespan. Paletta and Vocaturo (2011) developed a composite heuristic. In the construction phase, families of partial solutions are combined until a feasible solution is generated. The construction phase is followed by an improvement phase. Local search techniques are used to improve on the initial solution. Davidović et al. (2012) applied a bee colony optimization approach. In the same year, Chen et al. (2012) proposed a dynamic harmony search algorithm and a hybrid version, that additionally performs a variable neighborhood search based local search. Among the most recently published meta-heuristic algorithms are the grouping evolutionary strategy of Kashan et al. (2018) and an improved cuckoo search of Laha and Gupta (2018). Only recently, Della Croce et al. (2019) and Della Croce and Scatamacchia (2018) have revisited the famous longest processing time (LPT) rule of Graham (1969).

A few approaches towards the exact solution of $P||C_{max}$ have also been published. Dell'Amico and Martello (1995) implemented a depth-first branch-and-bound algorithm. They also derived tight lower bounds from the relationship between $P||C_{max}$ and BPP. Mokotoff (2004) designed a cutting plane algorithm. Dell'Amico et al. (2008) proposed a specialized binary search and a branch-and-price scheme. Haouari and Jemmali (2008) suggested a new symmetry-breaking branching scheme and lifting procedures to tighten lower bounds. Lenté et al. (2013) derived a new exponential-time algorithm from their extension of the *Sort and Search* method. Mnich and Wiese (2015) presented the first fixed-parameter algorithm for $P||C_{max}$. Recently, Mrad and Souayah (2018) proposed an arc-flow formulation.

Despite the large number of publications, only little is known about the structure of optimal solutions. This might be due to the fact that $P||C_{max}$ itself has very little structure compared to other $\mathcal{NP}$-hard optimization problems. To the best of our knowledge, only Dell'Amico and Martello (1995) addressed this issue casually by developing upper and lower bounds on the number of jobs per machine. These bounds are then used to derive lower bounds on the optimal makespan.

To close this gap, we aim at identifying general characteristics of optimal multiprocessor schedules. Using an approach that is to some extent related to the concept of

inverse optimization, we show that a schedule has to have a specific characteristic in order to be (uniquely) optimal. This allows us to restrict the solution space effectively during the search for an optimal schedule. In one of our earlier papers (Walter et al. 2017), we have already applied this approach successfully to the exact solution of the "dual" problem $P||C_{min}$, i.e., the problem of maximizing the minimum machine completion time. As the said paper overtook the present one during the review process, it does not contain any proofs of the underlying mathematical theory. However, we think that it is important to provide the formal results as well. This makes it easier for future researchers to transfer them to other combinatorial optimization problems such as the bin packing problem. The mathematical groundwork, therefore, constitutes the main contribution of this paper.

We identify and prove a general characteristic of optimal multiprocessor schedules and translate it into new dominance rules. Although this paper focuses on theoretical foundations, we implemented these rules in order to determine their benefit in a computational study. Used within a rather simple depth-first search, we obtained promising results: The new rules are quite effective in eliminating dominated (partial) solutions when each machine processes only a few jobs on average (i.e., $2 < n/m < 4$). Those instances are known to be typically more difficult to solve than large-sized instances with multiple jobs per machine (cf. the computational results published in Dell'Amico and Martello (1995), Dell'Amico et al. (2008) and Haouari and Jemmali (2008)). With increasing n/m, bounding arguments often become tighter (cf., e.g., Haouari et al. 2006) and this helps to verify optimality more quickly.

Our paper is divided into a theoretical part (Sects. 2, 3) and a practical part (Sects. 4, 5). The theoretical part represents the main contribution. Here, we undertake a thorough investigation of the solution space and identify a general characteristic of optimal multiprocessor schedules. We then translate our findings into new dominance rules and discuss prerequisites for their application within a tree search. In the second part, we describe the elements of the implemented branch-and-bound algorithm and address the efficient implementation of the new dominance rules. We then analyze the results of our experimental study and assess the benefit of the new rules. Finally, Sect. 6 concludes the paper and describes future research directions.

2 A theoretical study of the solution space

In this section we provide a profound theoretical study of the underlying solution space. Using an approach similar to the idea behind inverse optimization (cf., e.g., Ahuja and Orlin 2001), we aim at the identification of a general characteristic of optimal multiprocessor schedules. In the remainder of the paper we presuppose the jobs to be labeled so that $p_1 \geq p_2 \geq \cdots \geq p_n$.

2.1 A symmetry-breaking solution representation

We represent a schedule S as a sting of length n where each component can take the values $\{1, 2, \ldots, m\}$ with $S(j) = i$ meaning that job j is assigned to (or processed

by) machine i. Noticing that there always exists an optimal solution with $S(1) = 1$, $S(2) \in \{1, 2\}$, and so on, we restrict the values of $S(j)$ as follows:

1. $S(1) = 1$
2. $S(j) \in \left\{1, \ldots, \min\{m, 1 + \max_{1 \leq k \leq j-1} S(k)\}\right\}$ for all $j = 2, \ldots, n$.

With these restrictions we eliminate symmetric solutions that result from a simple renumbering of the machines. We, therefore, call the remaining solutions *non-permuted schedules*. It is readily verified that the number of non-permuted schedules is approximately equal to $m^n/m!$. Throughout this paper we use the aforementioned symmetry-breaking solution representation. For brevity, we often omit the adjunct "non-permuted" when we speak of schedules.

2.2 Methodological approach: the concept of potential optimality and the path conditions

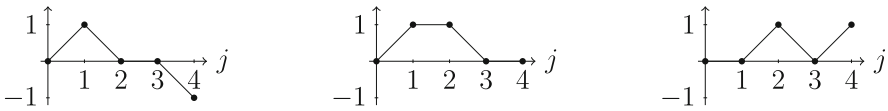
Our study originates from the question whether there exists a general pattern that characterizes non-optimal solutions no matter what selection of processing times is given. The existence of such a characteristic would allow us to limit the solution space to those schedules that do not have this characteristic and therefore have the potential to become (uniquely) optimal. We call them *potentially (unique) optimal schedules*.

Our main goal is to identify such a characteristic and to find a preferably small set of schedules that contains at least one optimal solution for any feasible input data. If this would succeed, then it suffices to search this set for an optimal schedule. To achieve this goal we apply an approach that is related to the concept of inverse optimization (see Ahuja and Orlin 2001; Heuberger 2004). In inverse optimization one aims at determining unknown exact values of (some) adjustable parameters—such as processing times—within given boundaries so that a pre-specified solution becomes optimal. Until now, this concept has been applied to scheduling problems only by very few researchers (e.g., Brucker and Shakhlevich 2009, 2011; Koulamas 2005). Slightly deviating from the basic idea of inverse optimization, we consider arbitrary schedules and ask whether we can select n feasible processing times so that the given schedule becomes uniquely makespan-optimal. If no such set of processing times exists, we can eliminate this schedule from the solution space.

Before we start with the characterization of potentially optimal solutions, we introduce a new way of illustrating schedules. Usually, Gantt charts are used to display which machine performs which job and what is the start and end time of processing. However, as our methodological approach mainly builds on the number of jobs on each machine rather than on processing times, we propose to illustrate a schedule S by $\binom{m}{2}$ *paths* $P_S^{(i_1,i_2)}$ $(1 \leq i_1 < i_2 \leq m)$—one for each pair of machines. Simply put, the (i_1, i_2)-path $P_S^{(i_1,i_2)}$ is a string of length $n+1$ where the j-th entry $(j = 1, \ldots, n)$ represents the difference between the number of jobs on machine i_1 and i_2 after the j longest jobs have been assigned according to schedule S. We set $P_S^{(i_1,i_2)}(0) = 0$ for each path to represent initially empty machines. Example 2.1 shows how we depict paths.

Example 2.1 Let $n = 4$, $m = 3$ and consider the non-permuted schedule $S = (1, 2, 3, 2)$. The three corresponding paths are $P_S^{(1,2)} = (0, 1, 0, 0, -1)$, $P_S^{(1,3)} = (0, 1, 1, 0, 0)$, and $P_S^{(2,3)} = (0, 0, 1, 0, 1)$. They are illustrated below.

$P_S^{(1,2)}(j)$ $P_S^{(1,3)}(j)$ $P_S^{(2,3)}(j)$



The difference $P_S^{(i_1,i_2)}(j) - P_S^{(i_1,i_2)}(j-1)$ between any two successive entries can take only one of the three values 1, -1, or 0 for each path. A difference equal to 1 means that job j is assigned to machine i_1 (illustrated by an upward line), a difference equal to -1 means that job j is assigned to machine i_2 (illustrated by a downward line), and if the difference equals 0 this means that j is assigned neither to machine i_1 nor to i_2 but to one of the other $m - 2$ machines (illustrated by a horizontal line).

As will become apparent in the next two subsections (cf. Theorems 2.3 and 2.5), schedules in the set $\mathcal{S}(n, m)$ $(n > m \geq 2)$ are central to the concept of potentially unique optimal $P||C_{max}$-solutions. We define this set as follows:

$$\begin{aligned}\mathcal{S}(n,m) = \Big\{ S \ &: \text{for each pair } (i_1, i_2) \text{ where } 1 \leq i_1 < i_2 \leq m \\ &\text{either "} P_S^{(i_1,i_2)}(j) < 0 \text{ for at least one } j \in \{3, \ldots, n\}" \\ &\text{or "} P_S^{(i_1,i_2)}(j) = 1 \text{ for } j = j_1, \ldots, j_2 \ (0 < j_1 \leq j_2 < n) \text{ and} \\ &P_S^{(i_1,i_2)}(j) = 0 \text{ for } j = 0, \ldots, j_1 - 1, j_2 + 1, \ldots, n" \Big\}. \end{aligned} \tag{1}$$

The set $\mathcal{S}(n, m)$ contains all schedules S that feature two characteristics: (i) each machine processes at least one job and (ii) each path has at least one negative entry when the total number of jobs on the two corresponding machines is greater than 2. We say that schedules in $\mathcal{S}(n, m)$ satisfy the *path conditions* or, equivalently, each of the $\binom{m}{2}$ paths satisfies the *path condition*. Returning to Example 2.1, schedule $S = (1, 2, 3, 2)$ is obviously not in $\mathcal{S}(4, 3)$ as the $(2, 3)$-path does not satisfy the path condition, whereas the other two paths satisfy the path condition.

Example 2.2 The elements of $\mathcal{S}(n, 2)$ $(n = 3, 4, 5)$ are:

$$\begin{aligned}&\mathcal{S}(3,2) = \{(1,2,2)\};\ \mathcal{S}(4,2) = \{(1,2,2,2), (1,2,2,1)\};\\ &\mathcal{S}(5,2) = \{(1,2,2,2,2), (1,2,2,2,1), (1,2,2,1,2),\\ &\qquad (1,2,2,1,1), (1,2,1,2,2), (1,1,2,2,2)\}.\end{aligned}$$

The elements of $\mathcal{S}(n, 3)$ $(n = 4, 5, 6)$ are:

$$\mathcal{S}(4, 3) = \{(1, 2, 3, 3)\};\ \mathcal{S}(5, 3) = \{(1, 2, 3, 3, 3), (1, 2, 3, 3, 2)\};$$
$$\mathcal{S}(6, 3) = \{(1, 2, 3, 3, 3, 3), (1, 2, 3, 3, 3, 2),$$
$$(1, 2, 3, 3, 2, 3), (1, 2, 3, 3, 2, 2), (1, 2, 3, 3, 2, 1),$$
$$(1, 2, 3, 2, 3, 3), (1, 2, 2, 3, 3, 3)\}.$$

Table 1 provides the share of non-permuted schedules that belong to $\mathcal{S}(n, m)$ (in %) for $m \leq 7$ and $n \leq 20$.

The shares are quite small as can be seen from Table 1. In particular, when $m \in \{5, 6, 7\}$ the number of schedules in $\mathcal{S}(n, m)$ is smaller by some orders of magnitude than the number of non-permuted schedules. However, recalling that the total number of non-permuted schedules is approximately equal to $m^n/m!$, $\mathcal{S}(n, m)$ may contain a great number of schedules despite small shares.

2.3 Potentially optimal schedules on two machines

We start with the case of two identical parallel machines and prove the following theorem.

Theorem 2.3 *Let S be a schedule that is not in $\mathcal{S}(n, 2)$. Then, S is not a potentially unique makespan-optimal schedule.*

Proof Consider a schedule $S \notin \mathcal{S}(n, 2)$ and let $J_1(S) = \{a_1, \ldots, a_r\}$ and $J_2(S) = \{b_1, \ldots, b_s\}$ denote the set of jobs (to be more accurate: their indices) that are assigned to machine 1 and 2, respectively. Without loss of generality, we assume that $a_1 < a_2 < \cdots < a_r$ and $b_1 < \cdots < b_s$. Since $S \notin \mathcal{S}(n, 2)$, the number of jobs on machine 1 must be at least as large as the number of jobs on machine 2, i.e., $r \geq s$ (with $r + s = n$). Moreover, for each $k \in \{1, \ldots, s\}$, the processing time of the k-th longest job on machine 1 is at least as large as the processing time of the k-th longest job on machine 2, i.e., $p_{a_k} \geq p_{b_k}$ which is equivalent to $a_k < b_k$. Thus, machine 1 runs at least as long as machine 2. The completion time of the last job on machine 1 gives the makespan of schedule S, i.e., $C_{max}(S) = \sum_{k=1}^{r} p_{a_k}$.

In order to prove that S is not a potentially unique makespan-optimal solution, we construct a schedule $\bar{S}$ that is not "longer" than S, i.e., $C_{max}(\bar{S}) \leq C_{max}(S)$, no matter what problem instance is given. We distinguish two cases depending on the number of jobs on machine 2 in S.

1. $s < 2$.
   Let $\bar{S}$ be the schedule that is obtained when job a_2 is shifted from machine 1 to machine 2 in S. Obviously, this cannot increase the makespan.
2. $s \geq 2$.
   Now, let $\bar{S}$ be the schedule that is obtained when the jobs a_s and b_s are swapped in S, i.e., a_s is processed on machine 2 and b_s on machine 1 in $\bar{S}$. Then, the makespan

Table 1 Shares of non-permuted schedules that belong to $\mathcal{S}(n, m)$ (in %)

		n																	
		3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
m	2	25.00	25.00	37.50	37.50	45.31	45.31	50.78	50.78	54.88	54.88	58.11	58.11	60.72	60.72	62.91	62.91	64.76	64.76
	3		7.14	4.88	5.74	7.12	9.23	11.31	13.11	14.97	16.72	18.27	19.81	21.26	22.58	23.87	25.09	26.22	27.32
	4			1.96	1.07	0.98	0.97	1.19	1.56	2.02	2.61	3.17	3.79	4.40	5.05	5.67	6.30	6.91	7.55
	5				0.50	0.23	0.18	0.15	0.15	0.18	0.22	0.29	0.39	0.51	0.65	0.81	0.98	1.16	1.35
	6					0.11	0.05	0.03	0.02	0.02	0.02	0.02	0.03	0.04	0.05	0.07	0.09	0.12	0.15
	7						0.02	0.01	0.006	0.004	0.003	0.003	0.003	0.003	0.004	0.004	nda	nda	nda

of schedule $\bar{S}$ is

$$C_{max}(\bar{S}) = \max\left\{\sum_{k=1}^{s-1} p_{a_k} + p_{b_s} + \sum_{k=s+1}^{r} p_{a_k}, \sum_{k=1}^{s-1} p_{b_k} + p_{a_s}\right\}.$$

In case $C_{max}(\bar{S}) = \sum_{k=1}^{s-1} p_{a_k} + p_{b_s} + \sum_{k=s+1}^{r} p_{a_k}$, we can conclude

$$\sum_{k=1}^{s-1} p_{a_k} + p_{b_s} + \sum_{k=s+1}^{r} p_{a_k} \leq \sum_{k=1}^{r} p_{a_k} = C_{max}(S).$$

In the other case, i.e., $C_{max}(\bar{S}) = \sum_{k=1}^{s-1} p_{b_k} + p_{a_s}$, we can conclude

$$\sum_{k=1}^{s-1} p_{b_k} + p_{a_s} \leq \sum_{k=1}^{s-1} p_{a_k} + p_{a_s} = \sum_{k=1}^{s} p_{a_k} \leq \sum_{k=1}^{r} p_{a_k} = C_{max}(S).$$

Thus, we have $C_{max}(\bar{S}) \leq C_{max}(S)$.

This completes the proof of the theorem as in either case a schedule $\bar{S}$ exists that is not longer than S. □

We remark that the schedule $\bar{S}$ itself is not required to be an element of $\mathcal{S}(n, 2)$. However, it is readily verified that we can convert any schedule $S \notin \mathcal{S}(n, 2)$ into a schedule $S' \neq S$ that belongs to $\mathcal{S}(n, 2)$ by an iterative application of the shifting operation (as in Case 1 of the proof of Theorem 2.3) and/or swapping operation (as in Case 2). We call the entire process *path conversion (on two machines)*. Clearly, the path conversion does not increase the makespan. Example 2.4 illustrates the procedure.

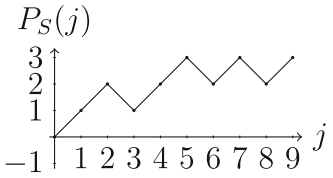
Example 2.4 We consider $m = 2$ machines, $n = 9$ jobs with processing times (20, 18, 15, 12, 10, 10, 8, 5, 2), and the initial schedule $S = (1, 1, 2, 1, 1, 2, 1, 2, 1)$. The path conversion takes two steps: First, S is converted into $\bar{S} = (1, 1, 2, 2, 1, 2, 1, 1, 1)$ and then $\bar{S}$ is converted into the potentially unique makespan-optimal schedule $S' = (1, 1, 2, 2, 2, 1, 1, 1, 1)$ (Tables 2, 3, 4).

In view of Sect. 2.4 it is useful to record two important properties of the swapping operation (recall that swaps imply $s \geq 2$ jobs on machine 2):

- The $s - 1$ longest jobs on machine 1 are not affected by any swap.
- Let $\bar{S}$ denote the schedule that is obtained when one swap is performed on a schedule $S \notin \mathcal{S}(n, 2)$. Then, the entries in the path $P_{\bar{S}}$ can be computed as follows:

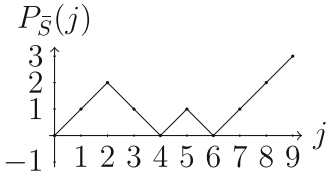
$$P_{\bar{S}}(j) = \begin{cases} P_S(j), & \text{if } j = 1, \ldots, a_s - 1, \\ P_S(j) - 2, & \text{if } j = a_s, \ldots, b_s - 1, \\ P_S(j), & \text{if } j = b_s, \ldots, n. \end{cases}$$

Table 2 Schedule S



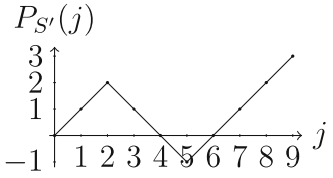
$J_1(S) = \{1, 2, \mathbf{4}, 5, 7, 9\}$	$J_2(S) = \{3, 6, \mathbf{8}\}$
$C_1(S) = 70$	$C_2(S) = 30$

Table 3 Schedule $\bar{S}$



$J_1(\bar{S}) = \{1, 2, \mathbf{5}, 7, 8, 9\}$	$J_2(\bar{S}) = \{3, 4, \mathbf{6}\}$
$C_1(\bar{S}) = 63$	$C_2(\bar{S}) = 37$

Table 4 Schedule S'



$J_1(S') = \{1, 2, 6, 7, 8, 9\}$	$J_2(S') = \{3, 4, 5\}$
$C_1(S') = 63$	$C_2(S') = 37$

Thus, the path conversion takes exactly $(P_S(2s-1)+1)/2$ swaps. The first negative entry in the path of the resulting schedule S' occurs at position $2s-1$.

To sum up, we can say that $\mathcal{S}(n, 2)$ contains at least one optimal solution for every selection of n feasible processing times. Consequently, when searching for an optimal solution, it is not necessary to consider schedules that are not in $\mathcal{S}(n, 2)$, i.e., schedules that do not satisfy the path condition. In a preliminary experimental study we found for every schedule S in $\mathcal{S}(n, 2)$ ($n \leq 25$) a selection of n processing times so that S is the unique optimal solution. This implies that further reductions in the solution space appear to be only realizable when processing times are explicitly taken into account.

2.4 Potentially optimal schedules on three or more machines

Using our findings from the previous subsection we now address potentially optimal schedules on more than two identical parallel machines.

Theorem 2.5 *Let S be a schedule that is not in $\mathcal{S}(n,m)$ $(m \geq 3)$. Then, S is not a potentially unique makespan-optimal schedule. Moreover, any schedule $S \notin \mathcal{S}(n,m)$ can be converted into a schedule that belongs to $\mathcal{S}(n,m)$ by a successive application of the path conversion.*

We will prove this theorem with the help of the following two Lemmata 2.6 and 2.7.

Lemma 2.6 *Let $1 \leq i_1 < i_2 < i_3 \leq m$ and S be a schedule where $P_S^{(i_1,i_2)}$ satisfies the path condition but $P_S^{(i_1,i_3)}$ does not satisfy it. Then, the (i_1,i_2)-path still satisfies the path condition after application of the path conversion to $P_S^{(i_1,i_3)}$.*

Proof We consider the application of the path conversion to $P_S^{(i_1,i_3)}$. Each single shift and swap also affects the entries in the (i_1,i_2)-path. We start with the case of a shift. Assume that job k is shifted from machine i_1 to i_3. Then, the entries in the (i_1,i_2)-path change as follows:

$$P_{\bar{S}}^{(i_1,i_2)}(j) = \begin{cases} P_S^{(i_1,i_2)}(j), & \text{if } \ j = 1,\ldots,k-1, \\ P_S^{(i_1,i_2)}(j) - 1, & \text{if } \ j = k,\ldots,n. \end{cases}$$

Now consider the case of a swap. Assume that job k on machine i_1 is swapped with job l on machine i_3. Recall from Sect. 2.3 that $k < l$. This leads to the following entries in the (i_1,i_2)-path:

$$P_{\bar{S}}^{(i_1,i_2)}(j) = \begin{cases} P_S^{(i_1,i_2)}(j), & \text{if } \ j = 1,\ldots,k-1, \\ P_S^{(i_1,i_2)}(j) - 1, & \text{if } \ j = k,\ldots,l-1, \\ P_S^{(i_1,i_2)}(j), & \text{if } \ j = l,\ldots,n \end{cases}$$

with S and $\bar{S}$ denoting the schedule before and after the current swap is performed, respectively.

As can be seen from the two formulas, after each single step of the (i_1,i_3)-path conversion we have $P_{\bar{S}}^{(i_1,i_2)}(j) \leq P_S^{(i_1,i_2)}(j)$ for all positions j. Hence, it is impossible that the (i_1,i_2)-path does not contain a negative entry anymore after the conversion of the (i_1,i_3)-path is completed. □

Lemma 2.7 *Let $1 \leq i_1 < i_2 < i_3 \leq m$ and S be a schedule where $P_S^{(i_1,i_2)}$ and $P_S^{(i_1,i_3)}$ satisfy the path condition but $P_S^{(i_2,i_3)}$ does not satisfy it. Then, the (i_1,i_2)-path and the (i_1,i_3)-path still satisfy the path condition after application of the path conversion to $P_S^{(i_2,i_3)}$.*

Proof Assume that the number of jobs on the three machines i_1, i_2, and i_3 are r, s, and q in schedule S, respectively. We let b_k $(k = 1, \ldots, s)$ denote the job with the k-th smallest index on machine i_2 and c_q denote the job with the largest index on machine i_3. We use b_k and c_q also as the index of the corresponding job.

We distinguish two cases for q.

1. $q = 1$.
   Due to the assumptions on the three considered paths, $q = 1$ implies $r = 1$ and $s > 1$. Then, according to Sect. 2.3, the conversion of the (i_2, i_3)-path takes at most two steps (one shift and at most one subsequent swap). After completing the conversion of the (i_2, i_3)-path, the number of jobs on machine i_2 is still greater than or equal to one and the number of jobs on machine i_3 equals two. The number of jobs on i_1 remains unchanged. Thus, the (i_1, i_2)-path and the (i_1, i_3)-path still satisfy the path condition.
2. $q > 1$.
   In this case we have $s \geq q$ since $P_S^{(i_2,i_3)}(j) \geq 0$ for all $j = 1, \ldots, n$. Let j_1 and j_2 denote the position of the first negative entry in $P_S^{(i_1,i_2)}$ and $P_S^{(i_1,i_3)}$, respectively. As $P_S^{(i_2,i_3)}(j) \geq 0$ for all $j = 1, \ldots, n$ we have $P_S^{(i_1,i_2)}(j) \leq P_S^{(i_1,i_3)}(j)$ for all $j = 1, \ldots, n$ and, thus, $j_1 < j_2$. Moreover, we have $j_1 \leq b_q$ as $P_S^{(i_1,i_3)}$ is supposed to satisfy the path condition.
   Now we consider the path conversion of $P_S^{(i_2,i_3)}$. First, recall from Sect. 2.3 that this conversion does not affect the $q - 1$ longest jobs $b_1, \ldots, b_{q-1}$ on machine i_2. Furthermore, note that the first step of the conversion consists in swapping job b_q on i_2 with job c_q on i_3, i.e., no previous shift is performed which means that the number of jobs on each machine remains unchanged. We distinguish two subcases depending on the relation between j_1 and b_q.

   (a) $j_1 < b_q$.
   Note that $j_1 < b_q$ is equivalent to $j_1 \leq b_{q-1}$. Since $b_1, \ldots, b_{q-1}$ remain the $q - 1$ longest jobs on i_2 in the resulting schedule $\bar{S}$, we have $P_{\bar{S}}^{(i_1,i_2)}(j) = P_S^{(i_1,i_2)}(j)$ for $j \leq b_{q-1}$ and, in particular, $P_{\bar{S}}^{(i_1,i_2)}(j_1) = -1$ which means that the (i_1, i_2)-path still satisfies the path condition.

   (b) $j_1 = b_q$.
   First, note that this subcase implies $j_2 = c_q$. Hence, there are exactly $q - 1$ jobs on machine i_1 whose index is not greater than $c_q - 1$ in schedule S. Since the path conversion of $P_S^{(i_2,i_3)}$ leaves the jobs $b_1, \ldots, b_{q-1}$ unchanged but swaps at least the jobs b_q and c_q, there are at least q jobs on machine i_2 whose index is not greater than c_q in the resulting schedule $\bar{S}$. Thus, we have $P_{\bar{S}}^{(i_1,i_2)}(c_q) \leq P_S^{(i_1,i_3)}(c_q) = -1$ which means that the (i_1, i_2)-path still satisfies the path condition.

   It remains to show that the (i_1, i_3)-path of the resulting schedule $\bar{S}$ also still satisfies the path condition. This is readily done because (i) jobs on machine i_1 were not affected by the conversion of the (i_2, i_3)-path and (ii) some of the "downward

lines" in the resulting (i_1, i_3)-path occur earlier than in the initial (i_1, i_3)-path. Hence, in either subcase we have $P_{\bar{S}}^{(i_1,i_3)}(j) \leq P_S^{(i_1,i_3)}(j)$ for all $j = 1, \ldots, n$. □

Proof of Theorem 2.5 The proof of the first part of the theorem is straightforward. Since S is not in $\mathcal{S}(n, m)$, there exists at least one path that does not satisfy the path condition. Let the (i_1, i_2)-path be such a path. Application of the path conversion to $P_S^{(i_1,i_2)}$ neither increases the maximum completion time of the two machines i_1 and i_2 (cf. Sect. 2.3) nor involves any jobs on the other $m - 2$ machines. Thus, the makespan of the resulting schedule cannot be greater than the makespan of S. This proves that S cannot be a potentially unique makespan-optimal schedule.

We prove the second part of the theorem with the help of the two Lemmata 2.6 and 2.7. First, we consider the paths $(1, 2), (1, 3), \ldots, (1, m)$ one by one. If any of these does not satisfy the path condition, we apply the path conversion. According to Lemma 2.6, each of the $m - 1$ paths $(1, i)$ $(i = 2, \ldots, m)$ is then satisfying the path condition. However, a renumbering of the machines may now be required in order to restore the representation as a non-permuted schedule. In the second round, we consider the paths $(2, 3), (2, 4), \ldots, (2, m)$ one by one. If any of these does not satisfy the path condition, we apply the path conversion. According to Lemmata 2.6 and 2.7, each of the $2m - 3$ paths $(1, i)$ $(i = 2, \ldots, m)$ and $(2, i)$ $(i = 3, \ldots, m)$ is then satisfying the path condition. Again, a renumbering of the machines may be required. We repeat this iterative process until we finally arrive at the $(m-1, m)$-path. This shows that we can convert any schedule $S \notin \mathcal{S}(n, m)$ into a schedule that belongs to $\mathcal{S}(n, m)$ by a successive application of the path conversion. □

To sum up the results of Sects. 2.3 and 2.4, we can say that for every $m \geq 2$ and $n > m$ the set $\mathcal{S}(n, m)$ always contains at least one optimal solution no matter what processing times are given. Hence, when searching for an optimal solution to a given sequence of processing times it is not necessary to consider schedules that are not in $\mathcal{S}(n, m)$. Those schedules can be excluded from the solution space since there exists always at least one optimal solution that satisfies the path conditions. In view of the fact that $P||C_{max}$ has only very little problem-inherent structure, we did not quite expect such a universal result. However, we shall also remark that we can select processing times in such a way that not every optimal solution satisfies the path conditions. An obvious example is the case of identical processing times where $p_1 = \ldots = p_n$. Then, any schedule with either $\lceil n/m \rceil$ or $\lfloor n/m \rfloor$ jobs on each of the m machines is makespan-optimal. However, not every such schedule is in $\mathcal{S}(n, m)$, e.g., the schedule $S = (1, 2, \ldots, m, 1, 2, \ldots, m, \ldots)$.

As before, we conducted a small experimental study for $m = 3$ and $n \leq 12$. Again, we were able to find processing times for every $S \in \mathcal{S}(n, m)$ so that S is the unique optimal solution. Although we do not have a mathematical proof yet, we strongly conjecture that processing times exists for each $S \in \mathcal{S}(n, m)$ so that S is the unique optimal solution. This would imply that $\mathcal{S}(n, m)$ cannot be further reduced without explicitly taking into account the processing times. However, a watertight proof remains as a challenging task for future research. Keeping in mind that the concept of potential optimality does not require knowledge about the actual processing times, we feel that there might be some room to tighten our universal results when

specific classes of processing times (e.g., depending on the ratio of the longest to the smallest processing time) are considered.

3 New dominance rules derived from the path conditions

From our study of the solution space (see Sect. 2) we have learned that there exists always an optimal solution that satisfies the path conditions. We will now use this result to derive and formulate new *existential property-based* dominance rules for $P||C_{max}$ (cf. Jouglet and Carlier 2011, for an overview on different formulations and types of dominance rules in combinatorial optimization). These rules will then be integrated into an exact solution procedure to guide the search towards schedules in $\mathcal{S}(n, m)$ (see Sect. 4).

We now describe the rationale behind the new rules. Given a partial solution we want to decide whether or not it is possible to complete this solution in such a way that the path conditions are satisfied. Basically, this can be done by counting for each machine separately the minimum number of jobs that still have to be assigned until the path conditions are satisfied. Obviously, the counting strongly depends on which jobs have already been assigned and which jobs still have to be assigned, i.e., assumptions on the order in which the jobs are selected for assignment are required. To derive effective rules, that preferably allow for an early decision whether or not the path conditions can be satisfied, we assume the jobs to be successively assigned in order of non-increasing processing times. This is a common job selection principle—not only in a job-oriented branching scheme but also in construction heuristics such as the well-known LPT-rule (cf. Graham 1969). However, it is important to keep in mind that neither the validity of the theoretical results derived in Sect. 2 nor their translation into dominance rules presupposes this specific job selection principle. At the end of this section we will sketch how to derive special dominance rules for other job selection principles or a machine-oriented branching scheme.

3.1 Counting the minimum number of required jobs

In order to count the minimum number of required jobs we identify all machine-pairs that do currently not satisfy the path condition and determine the *minimum number of required jobs* on these machines. Let us consider a partial solution $\tilde{S}_k$ in which the $k < n$ longest jobs have already been assigned. For each pair of machines (i_1, i_2) with $1 \leq i_1 < i_2 \leq m$ we introduce a dummy variable $\delta_{\tilde{S}_k}^{(i_1,i_2)} \in \{0, 1\}$ that indicates whether or not the corresponding path condition is *currently* fulfilled ($\delta_{\tilde{S}_k}^{(i_1,i_2)} = 1$ if the answer is yes, $\delta_{\tilde{S}_k}^{(i_1,i_2)} = 0$ if the answer is no). In line with the definition of the set $\mathcal{S}(n, m)$ in (1), a pair (i_1, i_2) *currently* satisfies the path condition if either the corresponding partial path has already at least one negative entry or each of the two machines processes exactly one of the first k jobs. At this point it is important to note that in the former case, the path condition remains satisfied no matter how the remaining $n - k$ jobs are assigned, whereas in the latter case, the (i_1, i_2)-path might

not satisfy the path condition after the assignment of the remaining jobs (e.g., when i_1 receives another job but i_2 does not).

Obviously, only the pairs (i_1, i_2) with $\delta_{\tilde{S}_k}^{(i_1,i_2)} = 0$ have to be considered in the calculation of the minimum number of required jobs. Depending on the current number of jobs on machine i_1, we distinguish two cases:

1. i_1 processes at most one of the first k jobs.
   In this case it is sufficient to assign one job to i_2 in order to satisfy the path condition.
2. i_1 processes at least two of the first k jobs.
   In this case at least $P_{\tilde{S}_k}^{(i_1,i_2)}(k) + 1$ jobs still have to be assigned to i_2 in order to obtain a negative entry in the (i_1, i_2)-path. Recall from Sect. 2.2 that $P_{\tilde{S}_k}^{(i_1,i_2)}(k)$ gives the difference between the current number of jobs on i_1 and i_2 in the partial schedule $\tilde{S}_k$.

As each pair (i_1, i_2) has to satisfy the path condition,

$$v_{i_2}(k) = \max_{\substack{i_1=1,\ldots,i_2-1 \\ \delta_{\tilde{S}_k}^{(i_1,i_2)}=0}} \left\{ P_{\tilde{S}_k}^{(i_1,i_2)}(k) \right\} + \alpha_{i_2}(k) \tag{2}$$

gives the minimum number of jobs that still have to be assigned to machine i_2. If $\delta_{\tilde{S}_k}^{(i_1,i_2)} = 1$ for all $i_1 = 1, \ldots, i_2 - 1$, we set $v_{i_2}(k) = 0$. The additional $\alpha_{i_2}(k)$-term in Eq. (2) corresponds to the aforementioned case differentiation. More precisely, if Case 1 holds for all machines $i_1 = 1, \ldots, i_2 - 1$, then $\alpha_{i_2}(k) = 0$. Otherwise, if at least one of the machines $1, \ldots, i_2 - 1$ processes more than one job (cf. Case 2), then $\alpha_{i_2}(k) = 1$.

It is readily verified that by assigning the next $v_m(k)$ jobs to machine m, the following $v_{m-1}(k)$ jobs to machine $m - 1$ and so on until machine 2 finally receives its $v_2(k)$ required jobs, all $\binom{m}{2}$ path conditions are satisfied, i.e., $\sum_{i_2=2}^{m} v_{i_2}(k)$ is the minimum number of required jobs. If this number exceeds the number of remaining jobs, i.e.,

$$\sum_{i_2=2}^{m} v_{i_2}(k) > n - k \tag{3}$$

the current partial solution cannot be completed in such a way that the resulting schedule belongs to $\mathcal{S}(n, m)$.

We remark that this first dominance rule is a very basic one. It does not require any explicit information on the (current) objective function value. In what follows, we derive two makespan-specific dominance rules from the results of Sect. 2. Afterwards, Example 3.1 illustrates the benefit of each of our new rules.

3.2 Increasing the minimum number of required jobs

Given the $v_i(k)$-values for $i = 2, \ldots, m$ as determined in Sect. 3.1, we now present a procedure that checks whether some of these values can be increased by one. Requiring an upper bound U on the optimal makespan, the procedure determines the minimum number m' of machines that have to process at least two jobs in order that the makespan of the corresponding schedule does not exceed U. To determine m', we successively consider the ratios

$$q_i = \frac{P_\Sigma - iU}{m - i} \quad (i = 0, 1, \ldots, m-1) \tag{4}$$

where $P_\Sigma = \sum_{j=1}^{n} p_j$. Starting with $i = 0$, q_0 represents the average machine completion time. If $q_0 > p_1$, at least one machine has to process more than one job. Assuming that the completion time of this machine equals U, q_1 represents the minimum average load of the remaining $m - 1$ machines. If $q_1 > p_1$, one of these $m - 1$ machines also has to process at least two jobs. We continue this process with considering q_2 and so on. The process stops as soon as $q_i \leq p_1$ and we obtain $m' = i$.

Instead of using the aforementioned iterative procedure, we can determine m' also analytically. It is readily verified that $m' = \left\lceil \frac{P_\Sigma - mp_1}{U - p_1} \right\rceil$ provided that $P_\Sigma > mp_1$ and $U > p_1$. If $P_\Sigma < mp_1$ or $U = p_1$, we set $m' := 0$.

To decide whether some of the $v_i(k)$-values can be increased, we take a look at those machines to which currently at most one job is assigned. Let $i'(k)$ denote the smallest index of all machines to which currently at least two jobs are assigned. If no such machine exists, we can increase $v_i(k)$ by one for $i = m - m' + 1, \ldots, m$. In the other case, i.e., $i'(k) \leq m$, each machine $i > i'(k)$ also has to process at least two jobs in order to satisfy the path conditions. These are $m - i'(k) + 1$ machines (including machine $i'(k)$). If $m' > m - i'(k) + 1$, then the machines $m - m' + 1, \ldots, i'(k) - 1$ also have to process at least two jobs which means that we can increase $v_i(k)$ by one for $i = m - m' + 1, \ldots, i'(k) - 1$.

3.3 Incorporating the processing times

After having determined all $v_i(k)$-values, we now also take the processing times into account. Our intention is to decide whether it is possible to assign the required number of jobs $\sum_{i=1}^{m} v_i(k)$ to the machines in such a way that no machine runs longer than $U - 1$. As this problem is $\mathcal{NP}$-hard in the strong sense (proof by reduction from 3-PARTITION, cf. Garey and Johnson 1979), we solve a relaxed version instead. The relaxation concerns the restriction that each job has to be assigned exactly once, i.e., we now allow jobs to be assigned more than once.

Assume that $1 \leq r \leq m$ machines still require at least one job and let $I = \{i_1, i_2, \ldots, i_r\}$ be the corresponding set of machines, i.e., $v_i(k) > 0$ for all $i \in I$. We then determine for each $i \in I$ the longest job j_i that can be assigned to machine i in combination with the $v_i(k) - 1$ shortest jobs so that i finishes not later than $U - 1$.

More formally,

$$j_i = \min \left\{ j \in \{k+1, \ldots, n\} \mid C_i^k + p_j + \sum_{l=0}^{v_i(k)-2} p_{n-l} \leq U - 1 \right\} \tag{5}$$

where C_i^k is the current completion time of machine i after the first k jobs have already been assigned to the machines. Note that an assignment of a job $j \in \{k+1, \ldots, j_i - 1\}$ to machine i cannot improve on U and will therefore not lead to a new incumbent solution that satisfies the path conditions. The same holds true for the case that $C_i^k + \sum_{l=0}^{v_i(k)-2} p_{n-l}$ is already exceeding $U - 1$.

Let π denote a permutation of the machines in I that sorts the corresponding jobs j_i ($i \in I$) in non-increasing order of their indices. Obviously, in case that $n - j_{\pi(1)} + 1 < v_{\pi(1)}(k)$, the current solution cannot be completed in such a way that both the path conditions are satisfied and the makespan is less than U. In the other case, i.e., $n - j_{\pi(1)} + 1 \geq v_{\pi(1)}(k)$, we go on and check whether $n - j_{\pi(2)} + 1$ is smaller than $v_{\pi(1)}(k) + v_{\pi(2)}(k)$. If this is the case, the partial solution can be fathomed using the same argument as before. Otherwise, we repeat this iterative process and consider the next machines according to π one by one, i.e., we check for $n - j_{\pi(3)} + 1 < \sum_{b=1}^{3} v_{\pi(b)}(k)$, $n - j_{\pi(4)} + 1 < \sum_{b=1}^{4} v_{\pi(b)}(k)$ and so on. In case that one of the inequalities $n - j_{\pi(r')} + 1 < \sum_{b=1}^{r'} v_{\pi(b)}(k)$ ($r' = 1, \ldots, r$) is fulfilled, the current partial solution cannot lead to a new incumbent solution that satisfies the path conditions.

Example 3.1 We consider $m = 5$ machines and $n = 11$ jobs with processing times (187, 162, 140, 127, 119, 108, 101, 71, 62, 50, 25). Application of the well-known LPT-rule yields an upper bound value of $U = 237$ on the optimal makespan. Given the partial schedule $\tilde{S} = (1, 2, 3, 4, 5, 4)$, i.e., the longest $k = 6$ jobs have already been assigned, Table 5 provides the entries of all paths at position 6. The superscript * indicates that the corresponding path does currently not satisfy the path condition.

According to Sect. 3.1, we readily obtain $v_i(6) = 0$ for $i = 1, \ldots, 4$ and $v_5(6) = 1 + \alpha_5(6) = 1 + 1 = 2$. However, as $n - k = 5 > 2 = \sum_{i=1}^{5} v_i(6)$, we cannot fathom the current partial solution. Next, we try to increase the $v_i(6)$-values by application of the procedure as described in Sect. 3.2. We get $m' = \lceil \frac{1152 - 5 \cdot 187}{237 - 187} \rceil = 5$ and $i'(6) = 4$ and can increase the $v_i(6)$-values by one for $i = 1, 2, 3$. Nevertheless, $\tilde{S}$ cannot be fathomed as the minimum number of required jobs is still not greater than the number of unassigned jobs ($\sum_{i=1}^{5} v_i(6) = 5 \not> n - k = 5$). Finally, we take the processing times of the five unassigned jobs into account as suggested in Sect. 3.3. We have $I = \{1, 2, 3, 5\}$, $j_1 = 11$, and $j_2 = j_3 = j_5 = 8$. As $n - j_5 + 1 = 4 < 5 = \sum_{i \in I} v_i(6)$ (at iteration number 4 of the above-mentioned procedure), $\tilde{S}$ cannot be completed in such a way that both the resulting makespan is less than U and the path conditions are satisfied, i.e., we can fathom $\tilde{S}$ after all.

Table 5 Entries of all 10 paths at position $k = 6$

$P_{\tilde{S}}^{(i_1,i_2)}(6)$	2	3	4	5
1	0	0	−1	0
2	–	0	−1	0
3	–	–	−1	0
4	–	–	–	$1^\star$

3.4 Outlook

In this finishing subsection of the theoretical part we briefly show that the translation of the new structural characteristics into dominance rules is not restricted to a specific job selection rule. By relaxing the assumption that jobs are selected in non-increasing order of their processing times, the results of Sect. 2 can still be applied to evaluate partial solutions with respect to the satisfiability of the path conditions. We clarify this by means of two examples.

Example 3.2 Let $n = 8$, $m = 2$ and consider the partial solution $S = (1, x, 1, 2, 2, x, 2, 1)$, i.e., job 2 and 6 still have to be assigned. In order to satisfy the path condition, at least one of the remaining two jobs has to be assigned to machine 2.

Example 3.3 Let $n = 8$, $m = 2$ and consider the partial solution $S = (1, x, 1, 2, 1, x, x, 1)$, i.e., job 2, 6, and 7 still have to be assigned. This time, the path condition can only be satisfied when all three remaining jobs are assigned to machine 2.

The previous two examples reveal that there is a lot of potential in translating the theoretical results of Sect. 2 into methods that evaluate partial solutions and restrict the remaining job assignments when other job selection rules are applied within a job-oriented branching scheme or even when a machine-oriented branching strategy is used. The formulation of general rules for different branching schemes appears to be a challenging but valuable task for future research.

4 A simple branch-and-bound algorithm

We implemented a basic branch-and-bound algorithm in order to determine the effectiveness of the new (path-related) dominance rules in a computational study. Our procedure performs a depth-first search similar to the one in Dell'Amico and Martello (1995). At each level of the branching-tree, the job with the longest processing time amongst all unassigned jobs is chosen. More specifically, at level k, the current node generates at most m son-nodes by assigning job k to those machines M_i that fulfill $C_i^{k-1} + p_k < U^*$. The corresponding machines are selected according to increasing current completion times C_i^{k-1}. The makespan of the currently best known solution is denoted by U^*.

Note that selecting the job with the longest remaining processing time at each level of the tree is necessary for the application of the dominance rules derived in Sects. 3.1–

3.3, whereas the depth-first nature of our search is not a prerequisite. One can also implement a breadth-first or minimum lower bound strategy instead.

To avoid complete enumerations we also implemented a few lower and upper bounding procedures from the literature (see Sect. 4.1). Details on the application of the new dominance rules are provided in Sect. 4.2. However, it is beyond the scope of this paper to design a state-of-the-art algorithm for $P||C_{max}$. This would require the implementation of even more sophisticated branching and bounding techniques than the ones described here.

4.1 Implemented lower and upper bounding procedures

To guide the search and to assess the quality of partial solutions, we implemented some lower and upper bound arguments. Concerning lower bounds, we apply two procedures of Dell'Amico and Martello (1995). The first one, $L_{TV} = \max\left\{\lceil\sum_{j=1}^{n} p_j/m\rceil,\ p_1,\ p_m + p_{m+1}\right\}$, is an immediate bound obtained from simple relaxations of $P||C_{max}$. The second one, $L_{DM} = \max\{C + 1 : \exists p \leq C/2$ for which $B_\alpha(C, p) > m$ or $B_\beta(C, p) > m\}$, exploits the coherence between $P||C_{max}$ and the bin packing problem (BPP). In its core, L_{DM} consists of two sophisticated lower bounds $B_\alpha(C, p)$ and $B_\beta(C, p)$ for BPP. For any further details we refer to Dell'Amico and Martello (1995).

To enhance lower bounds we implemented a lifting procedure of Haouari et al. (2006). Roughly speaking, this procedure determines lower bounds for specific partial instances that are also valid for the entire instance. We let $\tilde{L}$ denote the lifted version of a bound L. Finally, we implemented a procedure of Haouari and Jemmali (2008). This procedure tries to tighten a lower bound L by solving a specific subset-sum-problem (SSP). It checks whether a subset of J exists so that the corresponding processing times sum up exactly to a known lower bound value L. If no such subset exists, then the smallest realizable sum of processing times greater than L constitutes an improved lower bound. We denote the tightened bound by L_{SSP}.

Concerning upper bounds, we implemented three procedures: the well-known LPT-rule (cf. Graham 1969), the Multifit-algorithm (cf. Coffman et al. 1978), and a multi-start local search improvement heuristic (cf. Haouari et al. 2006). The latter procedure iteratively solves specific $P2||C_{max}$-instances. We denote the three corresponding upper bounds by U_{LPT}, U_{MF}, and U_{LS}, respectively. For further details we refer to the literature.

To obtain global bounds we applied the above-mentioned bounding procedures at the root node in the following order. At first, we compute L_{TV} and U_{LPT}. In case $L_{TV} = U_{LPT}$, an optimal solution is obtained. Otherwise, we determine L_{DM}. If $L_{DM} < U_{LPT}$, we compute U_{MF} and if $L_{DM} < U_{MF}$, we additionally determine U_{LS}. If there is still a gap between L_{DM} and U_{LS}, the lifted bound $\tilde{L}_{DM}$ is computed. L_{SSP} is only determined in case that $\tilde{L}_{DM} < U_{LS}$. To obtain a local bound and to save up computation time, we only compute L_{DM} at each branched node of the search tree.

4.2 Dominance rules

If we cannot fathom a current partial solution after application of the bounding procedures, we check whether the path conditions are already satisfied. If they are not yet satisfied, we make use of our new path-related dominance rules. To allow for an efficient application, it is advisable to store not only the current position of each path but also the information whether or not the path (currently) satisfies the path condition. Using simple data structures such as two-dimensional arrays, an update of the relevant information consumes $\mathcal{O}(m^2)$ time at each generated node.

Provided that all these information is available, it takes $\mathcal{O}(m)$ time to determine the minimum number of required jobs (cf. Sect. 3.1). The same asymptotic run time is required for the attempt to increase the number of required jobs according to Sect. 3.2. Finally, incorporating the processing times as explained in Sect. 3.3 can be realized in $\mathcal{O}(mn)$ time. If none of the new rules confirms that the current solution can be fathomed, we branch the corresponding node.

Having in mind that there might be some optimal solutions that do not satisfy the path conditions (cf. end of Sect. 2.4), it does not seem to be useful to apply the new dominance rules at deep levels of the branching tree. Indeed, our preliminary tests indicated that their benefit decreases when they are applied to almost complete solutions. We do therefore not apply them when the number of remaining jobs is smaller than $\lceil 0.3n \rceil$.

5 Computational study

This section reports on the results of our computational study and we discuss the benefits of the new (path-related) dominance rules. To appropriately assess their benefit we implemented the branch-and-bound algorithm of Sect. 4 and a variant thereof. We label them BB_{Paths} and $\text{BB}_{\text{NoPaths}}$, respectively. Both algorithms are identical except that our new dominance rules are only applied in BB_{Paths} but not in $\text{BB}_{\text{NoPaths}}$.

5.1 Setup of the tests

Following the existing literature we considered different combinations of m and n and different distributions of processing times to generate our test instances. Specifically, we chose $m \in \{3, 5, 10, 15, 20\}$ and $n = \lceil km \rceil$ with $k \in \{2, 2.25, 2.5, 2.75, 3, 3.5, 4, 5\}$. Processing times are randomly drawn from five different distributions (see Table 6) as proposed in Dell'Amico and Martello (1995).

For each parameter setting $(Class, m, n)$, we successively generated instances until five of them fulfilled the property of not being solved to proven optimality already at the root node by application of the global bounds. In other words, we tested our two branch-and-bound algorithms only on those instances which require branching in order to find an optimal solution or to verify optimality. We also recorded the total number of instances (column "Inst" in Table 8) that had to be generated. Thus, "Inst" serves as an indicator for the difficulty of finding optimal solutions or verifying

Table 6 Distribution of processing times

Class	Distribution
1	Discrete uniform distribution in [1, 100]
2	Discrete uniform distribution in [20, 100]
3	Discrete uniform distribution in [50, 100]
4	Cut-off normal distribution with $\mu = 100$ and $\sigma = 20$
5	Cut-off normal distribution with $\mu = 100$ and $\sigma = 50$

Table 7 Performance criteria

Criterion	Description
Nodes	Average number of generated branch-and-bound nodes
Time	Average computation time in seconds
US	Number of unsolved instances (no optimal solution found or verified)

optimality by means of upper and lower bounds at the root node. Since the likelihood of being solved at the root node rapidly increases with increasing ratios of n to m (cf., e.g., Haouari et al. 2006, and column "Inst" in Table 8), we concentrate on those cases where $n/m \leq 5$. To avoid trivial instances we omitted the settings $(3, m, 2m)$ for all m and $(Class, 3, 6)$ for all five classes. Hence, our data set contains a total of 955 instances. We applied both BB_{Paths} and $\text{BB}_{\text{NoPaths}}$ to each of them.

Table 7 lists our three main performance criteria. To allow for a fair and meaningful comparison, the "Nodes"-criterion considers only those instances that have been solved by both algorithms within a prespecified time limit, whereas "Time" averages over all instances. Additionally, we recorded how often one algorithm returned a better solution than the other one. In case that no optimal solution has been found or optimality could not have been verified, we also determined the average and maximum relative deviation between the returned objective function value and the global lower bound.

We have implemented our algorithms in Java language (version 7.2). The computational tests were performed on a personal computer with an Intel Core i7-2600 processor (3.4 GHz), 8 GB RAM, and Windows 7 Professional SP1 (64 bit). The maximal computation time was set to 600 s per instance for each of our two algorithms. BB_{Paths} and $\text{BB}_{\text{NoPaths}}$ were run as single processes/threads.

5.2 Experimental results on the effectiveness of the new rules

Table 8 contains the results of our experiments on the effectiveness of the new dominance rules. For reasons of comprehensibility, we abstain from providing the results for each individual setting of the 5×39 parameter combinations $(Class, m, n)$. Instead, we average the results over the 25 (20) instances per (m, n)-pair and provide the

influence of the processing time classes in a compact way in a separate table (see Table 10).

Starting with the "US"-criterion, i.e., the number of unsolved instances, it can be seen that both algorithms show the same performance for the majority of the investigated parameter combinations. However, there are seven (m, n)-pairs [(15, 30), (15, 34), (15, 38), (15, 42), (20, 40), (20, 45), and (20, 50)] where BB_{Paths} finds significantly more optimal solutions than $BB_{NoPaths}$—132 compared to 90, i.e., 42 optimal solutions more. Note that all these (m, n)-pairs satisfy $2 \leq n/m < 3$. In total, 371 out of the 955 instances remained unsolved after application of $BB_{NoPaths}$, whereas only 329 instances remained unsolved when BB_{Paths} was applied. In case of unsolved instances, relative deviations from the global lower bound are fairly small (0.55% on average and a maximum of 4.19%). Solving the small-sized instances with $m \leq 5$ machines did not pose a problem to our algorithms. None of the corresponding 370 instances remained unsolved. In contrast, almost all of the large-sized instances with $m \geq 15$ machines and $n \geq 3m$ jobs remained unsolved within the time limit. However, it is also worth noting that the solution returned by BB_{Paths} is at least as good as the $BB_{NoPaths}$-solution for each of the 955 instances. In particular, BB_{Paths} is superior to $BB_{NoPaths}$ in verifying optimality.

The superior performance of BB_{Paths} over $BB_{NoPaths}$ becomes even more obvious when we take a look at the two other criteria "Time" and "Nodes". BB_{Paths} does not only find more optimal solutions, the new dominance rules also help to identify optimality more quickly (overall average of 215 s vs. 242 s) and to considerably reduce the number of generated branch-and-bound nodes (overall average of about 6.2 millions vs. 9.4 millions). While BB_{Paths} usually generates far less nodes than $BB_{NoPaths}$, distinctly shorter computation times can only be realized when $m \geq 10$. For smaller values of m, average computation times of the two variants are almost identical. However, for very few (m, n)-pairs [e.g., (5, 20) and (5, 25)], $BB_{NoPaths}$ is even slightly faster than BB_{Paths} despite generating more nodes. Thus, the additional time required for application of the new rules could not always be compensated for by smaller search trees.

Table 9 summarizes the results depending on the ratio of n to m. The results reveal that the new dominance rules are particularly effective in limiting the search space when n/m ranges between 2 and 3. When solving instances of the two smallest investigated ratios, only about 15.8% of the average computation time is required and only about 6.5% of the decision nodes are generated. For larger ratios, the effect diminishes as now more and more solutions exist that satisfy the path conditions. In particular, it becomes more difficult for the new rules to prune partial solutions at early levels of the decision tree and, thus, to restrict the search space effectively since the number of possible ways to satisfy the path conditions increases with increasing n/m. However, the entries in the "Inst"-column of Table 8 immediately reveal that the larger the ratio of n to m the more often instances can already be solved at the root node without requiring any branching effort at all. Almost all of the generated instances (249,816 out of 251,948, i.e., 99.15%) belong to the group of instances with $n/m \in [4, 5]$. We, therefore, did not consider any larger ratios in our tests.

Table 10 summarizes the results depending on the processing time classes. As can be seen the new rules achieve the greatest improvements in terms of "US" (up to

Table 8 Detailed performance depending on (m, n)

				$BB_{NoPaths}$			BB_{Paths}		
m	n	TO	Inst	US	Time	Nodes	US	Time	Nodes
3	7	25	169	0	0.001	5	0	0.001	4
	8	25	397	0	0.002	7	0	0.002	6
	9	25	35	0	0.003	8	0	0.003	8
	10	25	42	0	0.004	26	0	0.004	22
	11	25	48	0	0.005	40	0	0.005	36
	12	25	61	0	0.006	72	0	0.006	56
	15	25	199	0	0.006	1815	0	0.009	998
Σ/Avg		175	951	0	0.004	282	0	0.004	161
5	10	20	205	0	0.003	15	0	0.003	6
	12	25	108	0	0.004	33	0	0.004	12
	13	25	37	0	0.008	42	0	0.006	22
	14	25	30	0	0.009	141	0	0.009	81
	15	25	28	0	0.010	111	0	0.011	106
	18	25	32	0	0.055	78,163	0	0.049	45,321
	20	25	78	0	0.093	100,012	0	0.131	75,334
	25	25	522	0	81.583	80,018,661	0	87.750	63,089,740
Σ/Avg		195	1040	0	10.483	10,281,689	0	11.277	8,103,926
10	20	20	90	0	0.317	299,941	0	0.006	41
	23	25	43	0	3.792	2,258,140	0	0.025	7577
	25	25	32	0	2.509	1,070,870	0	0.068	22,106
	28	25	31	0	1.502	995,093	0	1.192	552,355
	30	25	31	2	97.317	50,176,331	2	94.891	41,950,814
	35	25	54	24	576.184	2,360,385	24	576.255	2,319,624
	40	25	207	25	600.000	–	25	600.000	–
	50	25	24,165	25	600.000	–	25	600.000	–
Σ/Avg		195	24,653	76	241.225	10,676,617	76	240.056	8,249,917
15	30	20	70	1	36.180	4,349,088	0	0.033	3011
	34	25	62	9	254.074	31,464,084	0	1.721	2,660,472
	38	25	51	8	254.272	22,145,666	1	53.465	7,070,422
	42	25	46	18	454.719	47,178,837	15	414.321	45,565,493
	45	25	40	22	550.786	188,700,054	22	550.313	180,059,094
	53	25	91	25	600.000	–	25	600.000	–
	60	25	967	25	600.000	–	25	600.000	–
	75	25	162,082	25	600.000	–	25	600.000	–
Σ/Avg		195	163,409	133	428.563	29,982,038	113	361.519	16,483,188

Table 8 continued

				$BB_{NoPaths}$			BB_{Paths}		
m	n	TO	Inst	US	Time	Nodes	US	Time	Nodes
20	40	20	78	6	189.901	6,429,464	1	32.369	288,517
	45	25	53	14	365.783	21,893,449	3	99.542	1,563,022
	50	25	46	19	466.834	6,277,367	13	364.109	5,732,183
	55	25	40	23	552.038	25,032	23	552.045	25,032
	60	25	38	25	600.000	–	25	600.000	–
	70	25	105	25	600.000	–	25	600.000	–
	80	25	437	25	600.000	–	25	600.000	–
	100	25	61,098	25	600.000	–	25	600.000	–
Σ/Avg		195	61,895	162	504.689	11,168,324	140	441.230	1,687,141
Overall		955	251,948	371	241.956	9,422,838	329	215.232	6,232,307

Table 9 Overall performance depending on n/m

		$BB_{NoPaths}$			BB_{Paths}		
n/m	TO	US	Time	Nodes	US	Time	Nodes
[2, 2.5)	205	30	98.143	5,596,297	4	15.515	365,989
[2.5, 3)	225	68	192.433	5,070,677	52	153.913	3,108,045
[3, 4)	275	123	274.942	11,345,229	123	274.685	9,924,362
[4, 5]	250	150	368.169	20,030,140	150	368.790	15,791,532

Table 10 Overall performance depending on the processing time classes

		$BB_{NoPaths}$			BB_{Paths}		
Class	TO	US	Time	Nodes	US	Time	Nodes
1	195	62	197.243	9,361,678	61	196.955	7,572,801
2	195	71	229.670	9,731,305	68	216.847	4,120,017
3	175	79	278.742	15,770,803	59	214.671	13,905,970
4	195	84	268.983	7,443,097	68	215.887	3,333,063
5	195	75	238.925	5,924,761	73	231.753	3,472,163

26% less unsolved instances) and "Time" (savings of up to 23%) for the processing time classes 3 and 4. These two classes have in common that the processing times of the jobs do not vary widely among each other, i.e., the range of values is rather small. In particular, the ratio p_1/p_n of the longest to the shortest processing time is small. Smaller ratios seem to be beneficial for the dominance rule of Sect. 3.3. The greatest improvements in terms of "Nodes" are realized when processing times are drawn according to Class 2 and 4 (savings of up to 58%).

6 Conclusions

The present paper addressed the fundamental makespan minimization problem on identical parallel machines from a theoretical point of view. Using an approach similar to the idea behind inverse optimization, we identified and proved general characteristics of optimal schedules. These new structural insights were then translated into dominance rules to restrict the solution space during the search for an optimal schedule. Although focusing on the theoretical foundation and mathematical groundwork, we implemented the new dominance rules into a depth-first branch-and-bound algorithm in order to determine their effectiveness. In our computational study the new rules proved to be very useful. Depending on the ratio of n to m they did not only help to find more optimal solutions but also to identify them more quickly.

Based on the output of our experiments we believe that it is worthwhile to pursue and develop the concept of potential optimality. Firstly, there might be some room to tighten our results either by considering specific classes of problem instances or by taking the job processing times explicitly into account. Although this appears to be a technically challenging task, it might not only result in a further restriction of the set of potentially optimal solutions but also allow for tighter v_i-values. As our new dominance rules largely depend on the v_i-values and the v_i-values themselves depend on the theoretical results on structural patterns of optimal schedules, we can even expect tighter versions of our dominance rules. Secondly, it is useful to develop our first ideas on deriving dominance rules for other job selection rules than LPT. This way, the new structural insights can also be used in other branching schemes than the one implemented here. Thirdly, it would be interesting to integrate our findings into other exact solution approaches, such as column generation or dynamic programming, or to define efficient neighborhoods for local search procedures based on the path conditions. Last but not least, it appears promising to tackle similarly structured optimization problems by our methodological approach.

Acknowledgements Open Access funding provided by Projekt DEAL. The authors are grateful to Christoph Haas for proposing the underlying idea for the dominance rule described in Sect. 3.3.

References

Ahuja RK, Orlin JB (2001) Inverse optimization. Oper Res 49:771–783

Alvim ACF, Ribeiro CC (2004) A hybrid bin-packing heuristic to multiprocessor scheduling. Lect Notes Comput Sci 3059:1–13

Brucker P, Shakhlevich NV (2009) Inverse scheduling with maximum lateness objective. J Sched 12:475–488

Brucker P, Shakhlevich NV (2011) Inverse scheduling: two-machine flow-shop problem. J Sched 14:239–256
Chen J, Pan Q-K, Wang L, Li J-Q (2012) A hybrid dynamic harmony search algorithm for identical parallel machines scheduling. Eng Optim 44:209–224
Coffman EG, Garey MR, Johnson DS (1978) An application of bin-packing to multiprocessor scheduling. SIAM J Comput 7:1–17
Davidović T, Šelmić M, Teodorović D, Ramljak D (2012) Bee colony optimization for scheduling independent tasks to identical processors. J Heuristics 18:549–569
Della Croce F, Scatamacchia R (2018) The longest processing time rule for identical parallel machines revisited. J Sched. https://doi.org/10.1007/s10951-018-0597-6
Della Croce F, Scatamacchia R, T'kindt V (2019) A tight linear time $\frac{13}{12}$-approximation algorithm for the $P2||C_{max}$ problem. J Comb Optim 38:608–617
Dell'Amico M, Martello S (1995) Optimal scheduling of tasks on identical parallel processors. ORSA J Comput 7:191–200
Dell'Amico M, Iori M, Martello S, Monaci M (2008) Heuristic and exact algorithms for the identical parallel machine scheduling problem. INFORMS J Comput 20:333–344
Frangioni A, Necciari E, Scutellà MG (2004) A multi-exchange neighborhood for minimum makespan parallel machine scheduling problems. J Comb Optim 8:195–220
Garey MR, Johnson DS (1979) Computers and intractability: a guide to the theory of NP-completeness. W. H. Freeman, San Francisco
Graham RL (1969) Bounds on multiprocessing timing anomalies. SIAM J Appl Math 17:416–429
Graham RL, Lawler EL, Lenstra JK, Rinnooy Kan AHG (1979) Optimization and approximation in deterministic sequencing and scheduling: a survey. Ann Discret Math 5:287–326
Haouari M, Jemmali M (2008) Tight bounds for the identical parallel machine-scheduling problem: part II. Int Trans Oper Res 15:19–34
Haouari M, Gharbi A, Jemmali M (2006) Tight bounds for the identical parallel machine-scheduling problem. Int Trans Oper Res 13:529–548
Heuberger C (2004) Inverse combinatorial optimization: a survey on problems, methods, and results. J Comb Optim 8:329–361
Jouglet A, Carlier J (2011) Dominance rules in combinatorial optimization problems. Eur J Oper Res 212:433–444
Kashan AH, Karimi B (2009) A discrete particle swarm optimization algorithm for scheduling parallel machines. Comput Ind Eng 56:216–223
Kashan AH, Keshmiry M, Dahooie JH, Abbasi-Pooya A (2018) A simple yet effective grouping evolutionary strategy (GES) algorithm for scheduling parallel machines. Neural Comput Appl 30:1925–1938
Koulamas C (2005) Inverse scheduling with controllable job parameters. Int J Services Oper Manag 1:35–43
Laha D, Gupta JND (2018) An improved cuckoo search algorithm for scheduling jobs on identical parallel machines. Comput Ind Eng 126:348–360
Lenté Ch, Liedloff M, Soukhal A, T'kindt V (2013) On an extension of the sort & search method with application to scheduling theory. Theoret Comput Sci 511:13–22
Mnich M, Wiese A (2015) Scheduling and fixed-parameter tractability. Math Program Ser B 154:533–562
Mokotoff E (2004) An exact algorithm for the identical parallel machine scheduling problem. Eur J Oper Res 152:758–769
Mrad M, Souayah N (2018) An arc-flow model for the makespan minimization problem on identical parallel machines. IEEE Access 6:5300–5307
Paletta G, Vocaturo F (2011) A composite algorithm for multiprocessor scheduling. J Heuristics 17:281–301
Walter R, Wirth M, Lawrinenko A (2017) Improved approaches to the exact solution of the machine covering problem. J Sched 20:147–164