ROMA TRE
UNIVERSITÀ DEGLI STUDI

Dipartimento di Informatica e Automazione
Via della Vasca Navale, 79
00146 Roma, Italy

A job shop problem with one additional resource type

Alessandro Agnetis[1], Marta Flamini[2,3], Gaia Nicosia[3], Andrea Pacifici[4]

RT-DIA-163-2010

Febbraio 2010

(1) Università degli Studi di Siena,
via Roma, 56
53100 Siena, Italy

(2) Università degli studi Roma Tre
Via della Vasca Navale, 79
00146 Roma, Italy

(3) Data Management s.p.a.
Largo Lido Duranti, 1
00128 Roma, Italy

(4) Università di Tor Vergata
Via delPolitecnico, 1
00133, Roma, Italy.

This research was partially supported by the PRIN grant of the Italian Ministry of Education, 2007ZMZK5T.

ABSTRACT

We consider a job shop scheduling problem with n jobs and the constraint that at most $p < n$ jobs can be processed simultaneously. This model arises in several manufacturing processes, where each operation has to be assisted by one human operator and there are p (versatile) operators. The problem is binary NP-hard even with $n = 3$ and $p = 2$. When the number of jobs is fixed, we give a pseudopolynomial dynamic programming algorithm and a fully polynomial time approximation scheme (FPTAS). We also propose an enumeration scheme based on a generalized disjunctive graph, and a dynamic programming-based heuristic algorithm. The results of an extensive computational study for the case with $n = 3$ and $p = 2$ are presented.

Keywords: Job shop, scheduling with resource constraints, disjunctive graph, dynamic programming.

1 Introduction

In this paper we study a resource-constrained job shop scheduling problem motivated by certain manufacturing processes, in which part of the production process is carried out by human operators sharing the same set of tools. In a resource-constrained scheduling problem the execution of the jobs requires, in addition to machines, the use of scarce extra resources.

Formally, the problem we address can be defined as follows. We are given a set $\mathcal{J} = \{J_1, J_2, \ldots, J_n\}$ of n *jobs* that require, for their processing, a set $\mathcal{M} = \{M_1, M_2, \ldots, M_m\}$ of m *machines* and a set of p *operators*. Each job J_i is a sequence of ν_i operations. The j-th operation of job J_i has to be processed by a specific machine for p_j^i time units (where p_j^i is integer). As usual in job shop problems, a feasible schedule is such that (i) at any time each machine can process at most one operation; (ii) the operations of the same job are totally ordered; (iii) no preemption is allowed. Additionally, in our problem (iv) at any time, each operation has to be assisted by an operator and each operator can assist at most one operation. The problem consists in finding a feasible schedule having minimum makespan. If there are n jobs and p operators, we denote the problem as $JSO(n, p)$ (Job Shop with Operators). Observe that a feasible schedule for $JSO(n, p)$ is also feasible for the standard job shop problem, and satisfies the restriction that at most p machines work simultaneously. Therefore, significant cases are those in which $p < \min\{n, m\}$, otherwise our problem is a standard job shop problem. To rule out trivial cases, we consider $p > 1$ and hence $m \geq 3$ and $n \geq 3$. From the above observation, we point out that a minimal relevant case for $JSO(n, p)$ is with $m = 3$, $n = 3$, and $p = 2$. In fact, when $m = 3$, $n = 3$ and $p = 3$, the problem is equivalent to $J3|n = 3|C_{\max}$, that is known to be NP-hard [24].

Many scheduling problems with resource constraints have been addressed in the literature [8, 20, 21]. Using the classification scheme proposed by Blazewicz *et al.* [5], $J|res\ 1p1|C_{\max}$ denotes the job shop problem in which there is *one* additional resource type, at most p units of the resource are available at any time, and each operation may require *up to one* unit of the resource. $JSO(n, p)$ is a special case of $J|res\ 1p1|C_{\max}$, in which each operation requires *exactly* one unit of the resource for its processing.

Another type of resource-constrained environment has been considered by Sethi *et al.* and Wang *et al.* [23, 25]. In a flow shop, each job requires a pallet (additional resource) throughout its processing, and there is a limited number p of pallets. The main difference between such problem and ours in a flow shop setting is that in [23, 25] the same pallet is allocated to a job from the start of the first operation through the completion of the last operation, while in our problem distinct operations of the same job can be assisted by different operators. In [25] the problem is shown to be NP-Hard in the strong sense if $p \geq 3$ and $m \geq 2$ or if $p \geq 2$ and $m \geq 3$, while it is polynomially solvable if $p = 2$ and $m = 2$. In [23] the authors focus primarily on the case $m = 2$ and elaborate on the impact of the number of pallets on the makespan.

Scheduling problems in which part of the processing, usually setup operations, has to be tended by one operator (sometimes called *server*) have been also studied. Flow shop scheduling is considered for instance in [4, 7, 12], while an open shop case is studied in [16].

Chen and Lee [11] address a scheduling problem where single task jobs have several processing alternatives, each requiring a given set of machines. Our problem can be formulated in terms of Chen and Lee's model, by defining p auxiliary machines

$M_{m+1}, M_{m+2}, \ldots, M_{m+p}$. Each operation, besides the "original" machine M_i, requires one of the p auxiliary machines (processing alternatives). However, the techniques presented in [11] cannot be applied to our problem since no precedence constraints among the tasks are considered.

The paper is organized as follows. First, we briefly report on the NP-hardness of two minimally relevant special cases of $JSO(n, p)$, both with $m = 3$ and $p = 2$ (Section 2). We then present a dynamic programming procedure for solving $JSO(n, p)$ (Section 3). This algorithm runs in pseudopolynomial time when the number of jobs is fixed. Also for the case of a fixed number of jobs, a fully polynomial time approximation scheme (FPTAS) is presented in Section 4. An implicit enumeration scheme based on a generalization of the well known disjunctive graph introduced by Roy and Sussmann [22] for the classical Job Shop Scheduling is presented in Section 5. In Section 6, a heuristic approach is proposed. Finally, the results of an extensive computational study comparing the performance of the two exact algorithms and the heuristic are discussed (Section 7).

2 Hard cases

In this section, we briefly report on the complexity of two minimally relevant special cases of $JSO(n, p)$, both with $m = 3$ and $p = 2$, namely:

- $JSO(3, 2)$ with $m = 3$.
- $JSO(n, 2)$ with single-task jobs and $m = 3$.

In [1] it is proved that the makespan minimization problem of scheduling tasks subject to chain precedence constraints on two parallel machines ($P2|chains|C_{\max}$) is binary NP-hard, even when the number of chains is three. Thus, the following theorem holds.

Theorem 1 *Problem $JSO(3, 2)$ is binary NP-hard even with $m = 3$.*

Proof. An instance of $P2|chains|C_{\max}$ with three chains can be viewed as an instance of $JSO(3, 2)$ in which the three jobs correspond to the three chains, all the operations of job J_i require machine M_i $(i = 1, 2, 3)$, and the role of the parallel machines is played by the two operators. □

We now discuss the complexity of $JSO(n, p)$ in the case of single-task jobs. This can also be viewed as a *parallel dedicated machines* problem in which there are m parallel machines, and each job consists of a single operation, which has to be performed on a *given* machine. Kellerer and Strusevich [20, 21] address a class of parallel machine scheduling problems where additional resources *may* be required for processing a task. In particular, they show that the problem with only one additional resource, in which each job requires at most one unit of it at any time, denoted as $PD3|res\ 111|C_{\max}$, is computationally hard [20], while $JSO(n, 1)$ is trivial. However, as shown below, the problem $JSO(n, p)$ in the single task case is NP-hard for $p \geq 2$.

Theorem 2 *$JSO(n, p)$ is NP-hard even when $p = 2$, $m = 3$, and $\nu_i = 1$ for all $i \in \{1, \ldots, n\}$.*

Proof. Consider the following instance I of Partition (see [15]):

Given $\tilde{n}$ items $\{a_1, a_2, \ldots, a_{\tilde{n}}\}$, let $W = \sum_{i=1}^{\tilde{n}} a_i$, is there a subset $S \subseteq \{1, \ldots, \tilde{n}\}$ such that $\sum_{i \in S} a_i = \frac{W}{2}$?

We define the corresponding instance I' of $JSO(n, p)$ with $p = 2$, $\nu_i = 1$, and $m = 3$. Define the job set as $\{J_1, J_2, \ldots, J_{\tilde{n}}, J_{\tilde{n}+1}, J_{\tilde{n}+2}\}$, where each job J_i is a single task. The first $\tilde{n}$ jobs $J_1, \ldots, J_{\tilde{n}}$ must be processed by machine M_3 and require processing times $a_1, \ldots, a_{\tilde{n}}$, respectively and are called *short* jobs. The last two jobs, $J_{\tilde{n}+1}$ and $J_{\tilde{n}+2}$, must be processed by machine M_1 and M_2 respectively, and they both require a processing time B, where B is "large enough", e.g. $B > \frac{W}{2}$. They are called *long* jobs. Then I' is a yes-instance if it has a solution with makespan less than or equal $B + \frac{W}{2}$.

It is easy to see that, in any optimal schedule, each operator must necessarily assist the processing of one *long* job and of a subset of *short* jobs. Moreover, the two subsets of *short* jobs cannot be simultaneously processed since they all require machine M_3. Hence, I' has a solution value with $C_{\max} \leq B + \frac{W}{2}$ if and only if the PARTITION instance I is a yes-instance. □

3 Dynamic programming

In this section we present a dynamic programming procedure that runs in pseudopolynomial time when the number of jobs is fixed. The procedure associates to an instance of $JSO(n, p)$ a shortest path instance on an acyclic digraph $G = (N, A)$, where the set of nodes represents the *states* of the dynamic program and the arcs correspond to possible transitions.

For the sake of simplicity, we first describe, in Section 3.1, the procedure for the case $n = 3$ and $p = 2$ (which is a minimally NP-hard case). In Section 3.2, we show how to extend the dynamic program to the general case.

3.1 $JSO(3, 2)$

Let us denote by $A = \{A_1, \ldots, A_{\nu_A}\}$, $B = \{B_1, \ldots, B_{\nu_B}\}$, and $C = \{C_1, \ldots, C_{\nu_C}\}$ the three jobs and their operations. Moreover, let p_i^A, p_j^B, and p_k^C be the operations processing times. Since $p = 2$, we may consider two different types of states in the dynamic program.

Type 1. A state in which operations up to A_i, B_j, and C_k have been completed for some $0 \leq i \leq \nu_A$, $0 \leq j \leq \nu_B$, $0 \leq k \leq \nu_C$. (A zero index indicates that the corresponding job has not started yet.)

Type 2. A state in which two operations of two different jobs have been completed, while a third job's operation is being processed.

A state can be represented by a quintuple (i, j, k, X, r), where

- i, j, and k indicate operations of jobs A, B, and C respectively (completed or in execution), $0 \leq i \leq \nu_A$, $0 \leq j \leq \nu_B$, $0 \leq k \leq \nu_C$
- $X \in \{A, B, C, -\}$ indicates either that the state is of type 1 ($X = -$), or that the state is of type 2 and an operation of job X is in execution ($X = A, B, C$)

- If the state is of type 1, $r = 0$. If the state is of type 2, r is the remaining processing time of the current operation of job X.

We can build the digraph G by generating nodes starting from the initial (type 1) state $s = (0, 0, 0, -, 0)$. In the following, we denote with w_{uv} the weight of arc (u, v).

A node $u = (i, j, k, -, 0)$ of type 1 has at most six successors v_1, v_2, and v_3 corresponding to the three possible choices of starting only one job operation and v_4, v_5, and v_6, corresponding to all possible ways of choosing two operations out of three for simultaneous execution. Note that, due to machine incompatibilities, it may actually be profitable to keep one operator idle, see Figure 1.

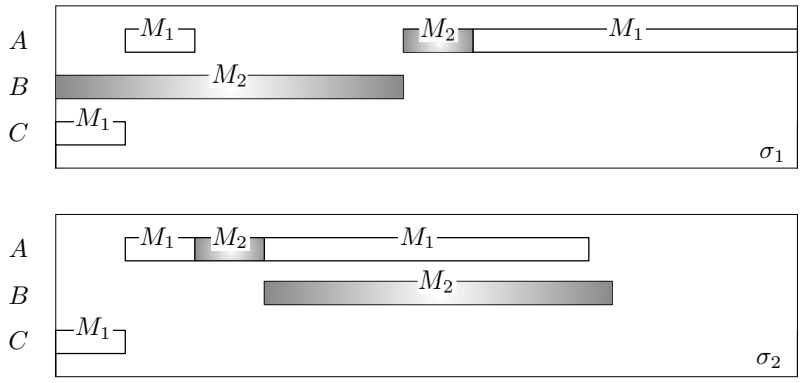

Figure 1: Keeping an operator idle may be beneficial.

If only job A is processed (and $i < \nu_A$), the successor of u is $v_1 = (i+1, j, k, -, 0)$ and $w_{uv_1} = p^A_{i+1}$. Similarly, $v_2 = (i, j+1, k, -, 0)$ with $w_{uv_2} = p^B_{j+1}$, and $v_3 = (i, j, k+1, -, 0)$ with $w_{uv_3} = p^C_{k+1}$. If both jobs A and B are processed, the successor of u is $v_4 = (i+1, j+1, k, X, r)$, where $r = |p^A_{i+1} - p^B_{j+1}|$ and $w_{uv_4} = \min\{p^A_{i+1}, p^B_{j+1}\}$. If $p^A_{i+1} = p^B_{j+1}$, then the two operations will complete simultaneously and hence $X = -$, else $X = A$ (respectively, B) if $p^A_{i+1} > p^B_{j+1}$ (respectively, $p^A_{i+1} < p^B_{j+1}$). Similar definitions hold for $v_5 = (i+1, j, k+1, X, r)$ and $v_6 = (i, j+1, k+1, X, r)$, corresponding to the execution of job pairs A, C and B, C respectively (with $w_{uv_5} = \min\{p^A_{i+1}, p^C_{k+1}\}$ and $w_{uv_6} = \min\{p^B_{j+1}, p^C_{k+1}\}$). Clearly, successors v_4, v_5 and v_6 exist if no machine incompatibility occurs.

A node $u = (i, j, k, X, r)$ of type 2 has at most three successors v_1, v_2, and v_3. Assume that $X = C$ (the cases with $X = A$ or $X = B$ are symmetrical). If an operation of job A is started in parallel with C, the successor v_1 is $(i+1, j, k, X', r')$, where $r' = |p^A_{i+1} - r|$ and $w_{uv_1} = \min\{p^A_{i+1}, r\}$. If $p^A_{i+1} > r$, the operation from job C completes before the operation from job A, and hence $X' = A$, whereas if $p^A_{i+1} < r$, then $X' = C$. If $p^A_{i+1} = r$, then $X' = -$ and v_1 is of type 1. Similar definitions hold for $v_2 = (i, j+1, k, X', r')$, corresponding to the execution of job B. Also here, successor v_1 (or v_2) does not exist if A_{i+1} (or B_{j+1}) requires the same machine of C_k. The third possible successor of u corresponds to the completion of the current operation of job C. In this case $v_3 = (i, j, k, -, 0)$ is of type 1 and $w_{uv_3} = r$.

In order to better understand how the graph of dynamic programming algorithm is constructed, a simple example is presented.

Example 3 *Let $\nu_A = 2$, $\nu_B = 3$ and $\nu_C = 2$ the processing times and the machine required by each operation are reported in Table 1.*

Jobs	op_1	op_2	op_3
A	2, M_1	1, M_3	
B	4, M_2	4, M_1	2, M_2
C	4, M_1	5, M_2	

Table 1: Data for example 3.

In Figure 2 the digraph G is partially depicted. In particular, all successors of the initial node s (of type 1) and all successors of a node of type 2, namely $(1, 0, 0, B, 2)$, are depicted. Note that node s does not have the successor corresponding to the simultaneous execution of operations A_1 and C_1 due to machine incompatibility.

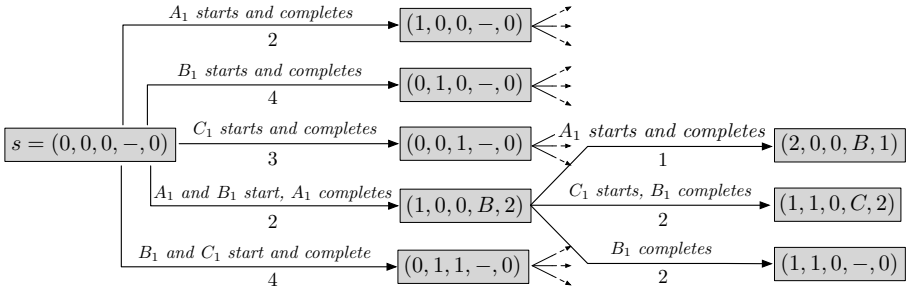

Figure 2: Illustration of Example 3.

Let $\nu_{\max} = \max\{\nu_A, \nu_B, \nu_C\}$, T be the largest processing time of an operation, and $t = (\nu_A, \nu_B, \nu_C, -, 0)$ be the *final* state.

Theorem 4 *$JSO(3, 2)$ can be solved in $O(\nu_{\max}^3 T)$ time*

Proof. It is easy to prove that any (s, t)-path on G corresponds to a feasible schedule for $JSO(3, 2)$ and therefore a shortest path corresponds to a minimum makespan schedule for $JSO(3, 2)$. The number of possible states of type 1 is $(\nu_A + 1)(\nu_B + 1)(\nu_C + 1)$. The number of possible states of type 2 is $O(\nu_{\max}^3 T)$. Since at most six arcs emanate from each node and checking for possible machine incompatibilities can be done in constant time, the shortest path can be found in $O(\nu_{\max}^3 T)$ time. □

3.2 $JSO(n, p)$

The algorithm described above can be extended to the general case yielding a pseudopolynomial algorithm when the number of jobs is fixed.

We have p types of state, where a state of type ℓ corresponds to a scenario in which the operations of $\ell - 1$ jobs are being processed, while a subset of the operations of the remaining $n - \ell + 1$ jobs have been completed ($\ell = 1, 2, \ldots, p$). The successors of a state correspond to all possible ways of starting the processing of operations of jobs currently not in execution (note that this includes the case in which no operation is started, i.e., one or more operations currently in process are completed). Thus, it is easy to show that a state of type $\ell + 1$ has at most $\sum_{i=1}^{p-l} \binom{n}{i} = O(2^p)$ successors.

Extending the notation of the previous section, we may represent a state by a $(n + 2(p-1))$-tuple $(i_1, i_2, \ldots, i_n, X_1, r_1, \ldots, X_{p-1}, r_{p-1})$, where

- i_j, for $j = 1, 2, \ldots n$, indicates the operation of job J_j currently in execution or just completed, $0 \leq i_j \leq \nu_j$
- indices $X_1 \leq X_2 \leq \ldots \leq X_{p-1}$ identify the subset of (at most) $p-1$ jobs currently in execution. If $X_{\ell-1} = 0$ and $X_\ell > 0$, for some $1 < \ell \leq p-1$ then the state corresponds to the case where exactly $p - \ell - 1$ jobs are in process. In particular, when $X_{p-1} = 0$ we are in a state of type 1.
- r_j is the remaining processing time of the current operation of job X_j. If $X_j = 0$ then $r_j = 0$ (and vice versa).

Again, the initial and final states are $s = (0, \ldots, 0)$ and $t = (\nu_1, \ldots, \nu_n, 0 \ldots, 0)$. A shortest (s,t)-path on G corresponds to a minimum makespan schedule for $JSO(n,p)$.

Assume that the number of jobs is fixed and equal to a constant q. In this case, the above described procedure runs in pseudopolynomial time. In fact, if we let $\nu_{\max} = \max\{\nu_i : \ i = 1, 2, \ldots n\}$ and T be the largest processing time, then the number of possible states is $O(\nu_{\max}^q \cdot T^p \cdot q^p)$, since $0 \leq i_j \leq \nu_{\max}$ for each $j = 1, 2 \ldots q$, $0 \leq r_i \leq T$ for $i = 1, 2 \ldots p-1$, and X_i can assume at most $q+1$ values, one for each job. Moreover observe that no more than $2^p \leq 2^q = O(1)$ arcs emanate from each node and checking for possible machine incompatibilities can be done in $O(p)$. Therefore, the dynamic programming procedure runs in $O(\nu_{\max}^q \cdot T^p \cdot p)$ time, which is pseudopolynomial since $p \leq q$.

4 FPTAS

In this section we briefly illustrate how the dynamic programming approach presented in the previous section can be exploited to devise a fully polynomial time approximation scheme (FPTAS) for $JSO(3,2)$. The idea, as usual, is to trade solution quality for efficiency.

We follow the same line of thought of the classical approach for the knapsack problem by Ibarra and Kim [19]. Roughly speaking, the algorithm scales down the processing times so that they are polynomially bounded in the input size and use dynamic programming on the new instance. By scaling with respect to the desired approximation ratio ϵ, we are able to (i) determine a solution with makespan at most $(1+\epsilon)$ times the optimal solution value with (ii) a running time that is polynomial with respect to both the number of operations and $1/\epsilon$.

Let I be an instance of $JSO(3,2)$. As in Section 3, we denote by $A = \{A_1, \ldots, A_{\nu_A}\}$, $B = \{B_1, \ldots, B_{\nu_B}\}$, and $C = \{C_1, \ldots, C_{\nu_C}\}$ the three jobs and their operations, and by p_i^A, p_j^B, and p_k^C the operations processing times. The optimal makespan for I is denoted by z^*. Given I and a positive number $K > 1$, the following algorithm, denoted as A_K, returns a feasible schedule for I:

1. Consider an instance $\hat{I}$ with processing times $\hat{p}_i^X = \lfloor p_i^X / K \rfloor$, $X = A, B, C$. Run the dynamic program on $\hat{I}$ and get an optimal schedule $\hat{\sigma}$ for $\hat{I}$ of value $\hat{z}$.
2. Replace each operation in $\hat{\sigma}$ with an operation K times longer, obtaining a schedule σ' of makespan $z' = K\hat{z}$

3. Renumber all operations in nondecreasing order of completion times in σ', and for each h from 1 to $\bar{\nu} = \nu_A + \nu_B + \nu_C$, do the following:

   Consider the schedules $\sigma'(1)$ and $\sigma'(2)$ of operations assigned to the two operators 1 and 2 (note that, given σ', there are many $\sigma'(1)$ and $\sigma'(2)$ compatible with σ'). Let t_h be the completion time of operation h assisted by an operator ℓ, ($\ell = 1, 2$). Insert an idle interval of length K starting at t_h on $\sigma'(\ell)$, and also insert an idle interval of length K on the other operator's schedule at the beginning of the operation currently in execution (possibly, starting) at t_h. Note that, by doing so, no job or machine precedence is violated.

   Let σ'' be the schedule thus obtained and z'' its value. Note that $z'' \leq z' + \bar{\nu}K$.

4. Replace each operation in σ'' with an operation of the original length at the same starting time—there is certainly enough room for it—obtaining a schedule (in general not semi-active) σ of value $z = z''$.

Algorithm A_K returns a feasible schedule for I of value z. We want to choose K so that the approximation ratio z/z^* is not greater than $1 + \epsilon$, for any $\epsilon > 0$. Again, T is the largest processing time of an operation.

Theorem 5 *If $K = \epsilon T/\bar{\nu}$, the algorithm A_K has approximation ratio not greater than $1 + \epsilon$ and runs in time $O(\bar{\nu}^4/\epsilon)$.*

Proof. Since σ' is optimal for an instance with processing times $K\lfloor p_i^X/K \rfloor \leq p_i^X$ ($X = A, B, C$) it follows that $z' \leq z^*$. On the other hand, $z^* \leq z \leq z' + \bar{\nu}K$. Hence,

$$\frac{z}{z^*} \leq \frac{z' + \bar{\nu}K}{z^*} \leq \frac{z^* + \bar{\nu}K}{z^*} \leq 1 + \frac{\bar{\nu}K}{T} = 1 + \epsilon.$$

The complexity is dominated by the dynamic program, which requires, if $\hat{T}$ is the largest processing time in $\hat{I}$, $O(\bar{\nu}^3\hat{T}) = O(\bar{\nu}^3 T/K) = O(\bar{\nu}^4/\epsilon)$. □

We conclude by observing that, as for the dynamic programming algorithm, the fully polynomial time approximation scheme can be easily extended to the case with (fixed) q jobs, yielding an algorithm of complexity $O(\bar{\nu}^{q+1}/\epsilon)$, with $\bar{\nu} = \sum_{i=1}^{q} \nu_i$.

5 Branch and bound

In this section we describe a branch and bound procedure for finding an optimal solution of $JSO(n, p)$ for the general case. We first introduce a *generalized disjunctive graph*, denoted as GDG and defined as follows: $GDG = (N, A \cup D_M \cup D_O)$, where N is the set of all jobs operations plus an initial s and a final t nodes; A is the set of *conjunctive* arcs corresponding to the precedence relationships between the processing operations of each job plus arcs (s, i), for all i *initial* operations of the jobs; and (i, t) for all i *final* operations of the jobs. D_M is the set of *machine-disjunctive* arc-pairs connecting two operations which belong to two different jobs, that are to be processed on the same machine. These are the classical disjunctive arcs modeling the precedence decision between the two operations.

D_O is the set of *operator-disjunctive* arc-pairs connecting two operations which belong to two different jobs, that are to be processed on different machines. These arcs model the following two alternatives: the two operations are assisted by the same or different operators. If the operations have to be assisted by the same operator, one of the two arcs will be selected. Otherwise we do not to select any of the two arcs, i.e., the two operations are assisted by two different operators (and an overlap in the processing of the two operations is allowed). Finally, each arc outgoing a node $i \in N$ has a weight equal to the processing time of the operation corresponding to node i. Arcs outgoing node s have zero weight.

A *complete selection* is a particular choice for all machine-disjunctive and operator-disjunctive arc-pairs. More precisely, if the pair $\{(u,v),(v,u)\}$ is a machine-disjunctive arc-pair, we choose one between the two alternatives. If the arc-pair is an operator-disjunctive arc-pair, we either select (u,v), (v,u), or none of them. In a *partial* selection, the choice is restricted only to a proper subset of arc-pairs. As a consequence, any (complete or partial) selection S is a vector in which there is an entry for each pair of operations not belonging to the same job. For the entry corresponding to the pair of operations $\{u,v\}$ three (if $\{u,v\} \in D_M$) or four (if $\{u,v\} \in D_O$) values are allowed, namely: (i) unset; $(ii)(u,v)$; (iii) (v,u); (iv) $\emptyset$. The latter value corresponds to establishing that no arc connects the two nodes, allowing an overlap in the processing of the two operations, and it can be set only for operator-disjunctive pairs. Given a selection S, $A(S)$ denotes the set of arcs corresponding to those pairs of operations for which the vector entry is of type (ii) or (iii). We let $B(S)$ be set of operation pairs for which the vector entry is $\emptyset$.

In the following we illustrate, through a small example, the above conjunctive/disjunctive formulation.

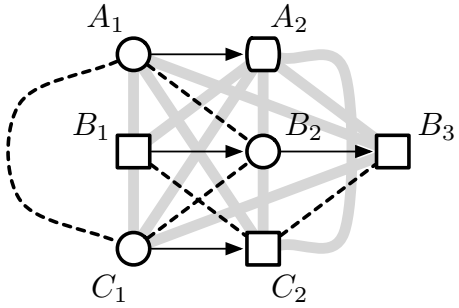

Figure 3: Illustration of Example 6

Example 6 *Consider the data of Example 3. In Figure 5, the corresponding graph GDG is depicted. Operations to be processed on different machines are drawn using different shapes. All machine-disjunctive pairs (dotted arcs) and operator-disjunctive pairs (gray bold arcs) are displayed. Figure 5 exemplifies a possible complete selection: All machine-disjunctive arcs are oriented and operator-disjunctive arcs are either oriented or set to value $\emptyset$ (gray bold arcs). Gray or white colored nodes indicate a possible assignment of the corresponding operations to two different operators. Note that all the operator-disjunctive*

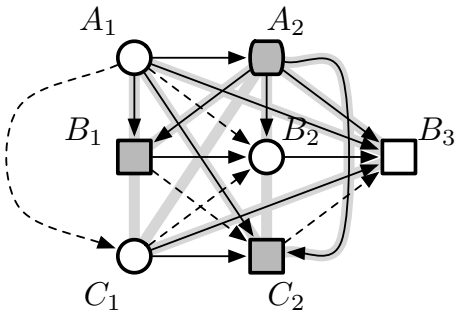

Figure 4: Illustration of Example 6.

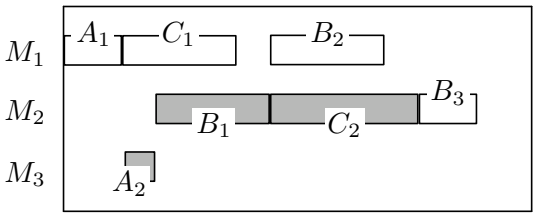

Figure 5: Illustration of Example 6.

arcs set to $\emptyset$ have their extremes associated to two different operators. Finally, Figure 5 illustrates the semi-active schedule corresponding to the selection of Figure 5.

As for the classical job shop, the duration of a minimum makespan schedule corresponding to a feasible complete selection S equals the length of the longest path in $GDG(S) = (N, A \cup A(S))$. Hence, we are looking for a selection S^* corresponding to a minimum makespan schedule.

5.1 Enumeration tree

In our enumeration scheme, following the ideas developed for the classical disjunctive graph, each node h is associated to a partial selection S^h. The root node corresponds to an empty selection. Branching at node h corresponds to assigning a value to an unset component $\{u, v\}$ in S^h. We distinguish two cases, depending on whether the pair $\{u, v\}$ is (i) a machine-disjunctive pair or (ii) an operator disjunctive pair.

Machine-disjunctive pair. As for the classical job-shop, the branching is binary and the selections corresponding to the children of node h are obtained from S^h by setting the $\{u, v\}$ component to (u, v) or (v, u).

Operator disjunctive pair. In this case, since the alternatives are three, the selections corresponding to the three children of node h are obtained from S^h by setting the $\{u, v\}$ component to (u, v), (v, u) or $\emptyset$. If S^h is a partial selection, the length of the longest path in $GDG(S^h)$ is a lower bound on the makespan of all the feasible schedules satisfying the choices imposed by S^h.

5.2 Fathoming rules

A node h of the enumeration tree, corresponding to a selection S^h, is fathomed for one of the following reasons.

Bounding. The lower bound at node h is greater or equal to the incumbent solution value.

Precedence Inconsistency. S^h is infeasible because a directed cycle is detected in $GDG(S^h)$.

Operator Infeasibility. S^h is infeasible because the current selection requires more than p operators.

Precedence Redundancy. This is a domination rule. Selection S^h implies a precedence (u, v), and in one predecessor i of h in the enumeration tree the selection S^i has the $\{u, v\}$ component set to $\emptyset$. Consider the example of Figure 6. Selection S^h is obtained from S^i by setting the $\{u, w\}$ component to (u, w), which in turn implies that u precedes v. In this case, node h is fathomed since the same subproblem (in which u precedes w and w precedes v) is considered in one of the successors of node j.

Completeness. S^h is (or implies) a complete selection.

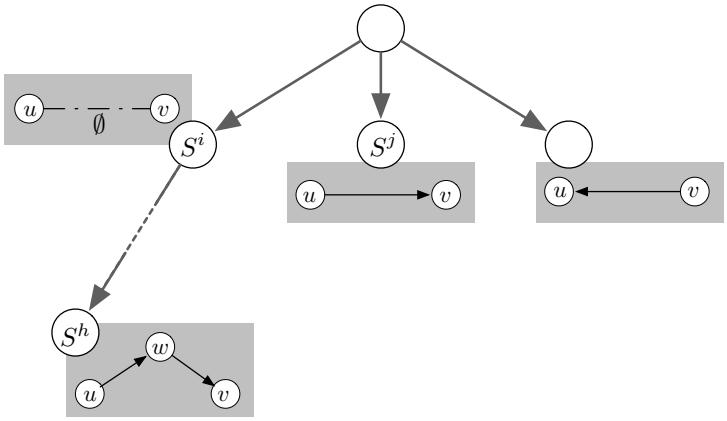

Figure 6: S^h is fathomed due to precedence redundancy.

Precedence Inconsistency and Precedence Redundancy can be easily detected by computing the transitive closure of $GDG(S^h)$ which immediately highlights (i) cycles in $GDG(S^h)$, (ii) *implied* precedences, and (iii) precedence redundancies. Regarding the computational complexity, we observe that the transitive closure of GDG, corresponding to the root node of the enumeration tree, can be constructed in $O((n\nu_{\max})^2)$ time, since the only precedences are those between operations of the same job. Each node S^h in the enumeration tree inherits the transitive closure of its parent S^k. If node S^h differs from its parent S^k for precedence (u, v), then the transitive closure of $GDG(S^h)$ can be obtained from $GDG(S^k)$ by adding (u, v) and all *implied* precedences (x, y), where x is a predecessor of u and y is a successor of v in $GDG(S^k)$. This upgrade can be done in $O(\nu_{\max}^2)$ time and allows to simultaneously check for the existence of 2-cycles in the transitive closure (which reveals a cycle in $GDG(S^h)$, and hence a precedence inconsistency) and precedence redundancies (when a component $\{x, y\}$, already set to $\emptyset$ in the selection S^h, exists and (u, v) implies (x, y) or (y, x)).

In order to detect operator infeasibilities, for each selection S^h we consider an undirected graph G_h having node set N and arc set $B(S^h)$. If a clique $K(G_h)$ of size greater than p in G_h exists, then node S^h is fathomed. Note that this does not necessarily imply that selection S^h is infeasible. In fact, it may happen that a feasible schedule exists in which the operations in $K(G_h)$ are not all performed simultaneously. However, in this case, at least two of such operations will not overlap and hence there is another node of the enumeration tree representing this scenario. Since finding a clique of size $p+1$ can be very costly, to speed up the procedure one can use a lower bound on the size of the maximum clique of G_h, for instance the one proposed in [27]. In particular, if S^h is a complete selection and $GDG(S^h)$ is acyclic, the operator infeasibilities may be detected in polynomial time by computing the width (i.e., the maximum number of incomparable nodes) of $GDG(S^h)$. Note that we are not allowed to do this if S^h is a partial selection, since the width of $GDG(S^h)$ may be greater than p, however some pair of unset relationships could successively reduce this value.

6 A heuristic algorithm

In this section we describe a heuristic algorithm for the special case in which only two operators exist. For the sake of simplicity, we limit our description to the case of three jobs, however it is straightforward to extend this approach to any number of jobs.

6.1 Block heuristic

Let A, B and C be the three jobs, where A_i, B_i, and C_i, denote the i-th operation of the three jobs. The basic idea of the procedure consists in computing a schedule as a sequence of *blocks*, each defined as a subset of consecutive operations of two jobs. The blocks correspond to the arcs of a state graph G in which a node corresponds to a triple (A_i, B_j, C_k), meaning that operations up to A_i, B_j, and C_k have been completed for some $0 \leq i \leq \nu_A$, $0 \leq j \leq \nu_B$, $0 \leq k \leq \nu_C$. (A null index indicates that the job has not started yet.) Nodes $s = (A_0, B_0, C_0)$ and $t = (A_{\nu_A}, B_{\nu_B}, C_{\nu_C})$ are the *initial* and *final* nodes of G respectively.

There is an arc (u, u') from $u = (A_i, B_j, C_k)$ to $u' = (A_{i'}, B_{j'}, C_{k'})$ if $0 \leq i \leq i' \leq \nu_A$, $0 \leq j \leq j' \leq \nu_B$, $0 \leq k \leq k' \leq \nu_C$ and either $i = i'$, $j = j'$, or $k = k'$. Any arc corresponds to a feasible state transition in which exactly two jobs (assisted by the two operators) are processed. For instance, if $k = k'$, arc (u, u') corresponds to a block including operations $A[i, i'] = \langle A_i, A_{i+1}, \ldots, A_{i'} \rangle$ of job A and $B[j, j'] = \langle B_j, B_{j+1}, \ldots, B_{j'} \rangle$ of job B, while job C is idle. $A[i, i']$ and $B[j, j']$ are called *stripes*. For instance, in Figure 11 the third and last block has two stripes, namely $A[3, 5]$ and $B[2, 3]$. The weight of an arc is the minimum makespan of the corresponding block, that can be easily computed by any (polynomial) algorithm for the job shop problem with two jobs.

The standard approach for the job shop problem with two jobs consists in computing a shortest path on a rectangular area with obstacles, representing machine incompatibilities between operations [3]. In particular, consider a two-axes plane, where the two axes correspond to the two jobs. On each axis, the operations of the respective job are represented by segments, the lengths being proportional to durations. Drawing parallel lines to the two axes from the endpoints of each segment results in a *grid* in the plane. If the i-th

operation of the first job and the j-th operation of the second require the same machine, they cannot be done in parallel, and the corresponding rectangle (i, j) is *forbidden*. Any feasible schedule of the two jobs can be represented on the grid by a path consisting of horizontal, vertical and diagonal (45°) segments. Diagonal segments correspond to the parallel execution of the two jobs, while a horizontal (vertical) segment represents the advancement of the first (second) job only. Operation pairs requiring the same machine cannot be processed in parallel and therefore it is not feasible to take a diagonal path through forbidden rectangles. The shortest path on the grid corresponds to a minimum makespan schedule. Brucker [6] has shown that it can be found in $O(r \log r)$, where r is the number of forbidden rectangles.

Back to the state graph G, it is easy to see that any (s, t)-path on G corresponds to a feasible schedule for $JSO(3, 2)$. However, even the shortest (s, t)-path does not correspond, in general, to a semi-active schedule. In fact, as long as we retain the block structure, an operator cannot start the execution of an operation outside the block, until all the operations of the block have been completed (see Figure 11). In other words, once one of the operators has completed the execution of its stripe, it is not allowed to switch to another operation until the other operator has completed its own stripe. Such a switch is permitted only simultaneously for the two operators. Hence, a post-processing phase has been designed to make the schedule semi-active, described in the next section.

The time complexity of the algorithm can be computed as follows. Recalling that $\nu_{\max} = \max\{\nu_A, \nu_B, \nu_C\}$, the number of nodes in G is $O(\nu_{\max}^3)$. The successors of each node $u = (A_i, B_j, C_k)$ are $O(\nu_{\max}^2)$, namely one for each pair of stripes from two different jobs among $A[i+1, i']$, $B[j+1, j']$, $C[k+1, k']$, $i' = i+1, \ldots, \nu_A$, $j' = j+1, \ldots, \nu_B$ or $k' = k+1, \ldots, \nu_C$. Thus, the number of arcs is $O(\nu_{\max}^5)$ which is also the cost of computing the shortest path. Arc weights may be computed offline by solving an instance of two-jobs job shop for each block. Using Brucker's algorithm, each such instance can be solved in $O(\nu_{\max}^2 \log \nu_{\max})$ (note that the number of forbidden rectangles is at most $O(\nu_{\max}^2)$). Since there are $O(\nu_{\max}^4)$ blocks, the overall computational cost of the block heuristic is therefore $O(\nu_{\max}^6 \log \nu_{\max})$.

As we mentioned above, this heuristic approach can be extended to the case in which there are n jobs and two operators. For n jobs, the time complexity of the heuristic becomes $O(n^4 \nu_{\max}^6 \log \nu_{\max})$.

6.2 Post-processing

In this section we describe a procedure that attempts to reduce the makespan of the scheduled produced by the above heuristic. The basic idea is to make the schedule semi-active by eliminating the unnecessary idle times induced by the block structure. To this purpose, we (*i*) first build two macro-jobs from the stripes of each block and (*ii*) compute a new overall schedule taking into account the precedences between operations.

In (*i*), we build two macro-jobs Γ_1 and Γ_2 by iteratively appending one stripe of each block to one macro-job, and the other stripe to the other macro-job. The two macro-jobs will correspond to the two operators' duties. The iterative assignment of stripes to Γ_1 and Γ_2 can be performed according to various reasonable criteria, related e.g. to balancing the workload between the operators (see Section 7).

Phase (*ii*) computes an optimal schedule of the two macro-jobs. Note that by construction, the operations in each of the two macro-jobs Γ_1 and Γ_2 satisfy the precedence

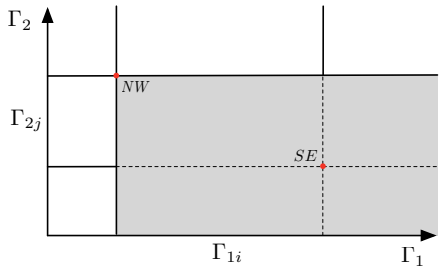

Figure 7: Forbidden areas in the modified two jobs job-shop algorithm.

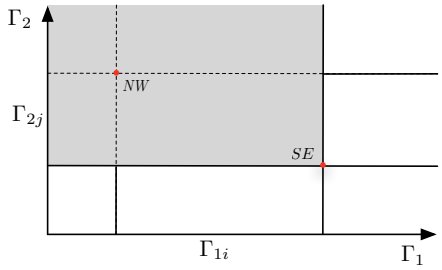

Figure 8: Forbidden areas in the modified two jobs job-shop algorithm.

constraints among operations of the same original job (A, B or C). However, since two stripes of the same job may have been assigned to different macro-jobs, we need to take into account these precedence constraints in any feasible schedule of Γ_1 and Γ_2. Hence, precedence constraints between consecutive operations of the same original job may result in precedence constraints between operations *across* the two macro-jobs. As a consequence, a modified version of the standard algorithm for the two-jobs job shop problem must be used, to account for precedences between operations belonging to different macro-jobs.

This can be attained by extending the notion of forbidden rectangle on the grid. As depicted in Figure 7, if the i-th operation of macro-job Γ_1, Γ_{1i}, precedes the j-th operation Γ_{2j} of macro-job Γ_2, the interior of the whole region lying north-west of point SE in Figure 7 must not be traversed. Analogously, if Γ_{2j} precedes Γ_{1i}, the forbidden region lies strictly south-east of point NW, see Figure 8. In conclusion, the optimal scheduling of the two macro-jobs can be computed as the shortest path on a grid which includes, besides the usual forbidden rectangles due to machine incompatibility, also all forbidden regions due to precedence constraints across the two macro-jobs.

Figure **??** sketches the whole heuristic approach. In particular, Figure 11 presents a first schedule generated by the heuristic algorithm. Figure 10 shows a corresponding schedule for the two operators, that is a schedule of the operations of the generated macro-jobs. Finally, in Figure **??**, the schedule resulting from the post processing procedure is depicted.

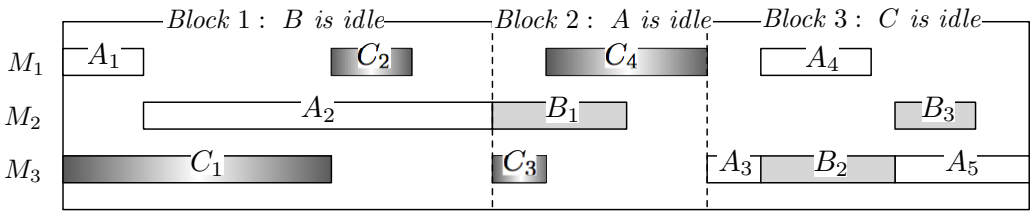

Figure 9: Sketch of the heuristic approach.

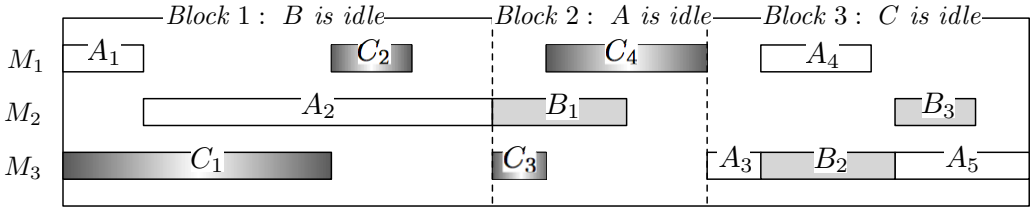

Figure 10: Sketch of the heuristic approach.

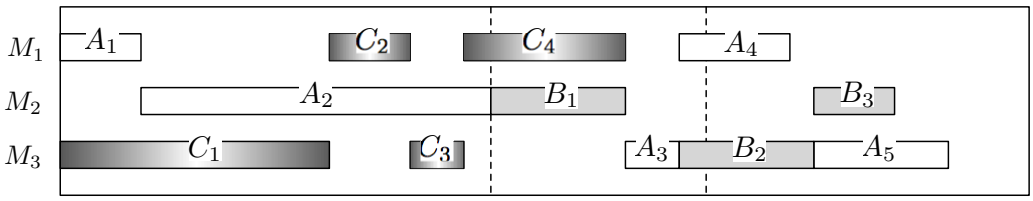

Figure 11: Sketch of the heuristic approach.

7 Computational Results

Hereafter we present the results of computational experiments on randomly generated instances of $JSO(3,2)$. We compare the computation times and the solution values obtained by the following algorithms (i) branch and bound, (ii) dynamic program, (iii) heuristic approach (with and without postprocessing). All the instances were also tested on Cplex 9.1 using a standard time indexed ILP formulation. All the experiments were run on a PC with a 3 GHz clock, 2 GB RAM and coded in C++. In the dynamic program and in the heuristic approach, all shortest paths are computed using the A^* algorithm [18], which has proved to be particularly effective on two-jobs job shop problems [2]. In the post processing phase of the heuristic algorithm we implemented two different stripes assignment criteria for obtaining the two macro jobs Γ_1 and Γ_2 and the best resulting solution is returned. The two criteria are (i) random assignment; (ii) the stripe with the largest total processing time is appended to the macro job having minimum workload.

In the following, we give some details concerning the implementation of the branch and bound.

Lower bounds. At each node h of the enumeration tree, two lower bounds on the makespan of the associated subproblem are computed. Namely, (i) the longest (s,t)-path on $GDG(S^h)$ (LB_1) and (ii) a lower bound (LB_2) derived from that of Carlier and Pinson for the classical job shop [9, 10] where, at each node, rather than solving the auxiliary instance of $1|r_j|L_{\max}$ to optimality, we use the lower bound provided by Jackson's preemptive schedule. Note that a (trivial) lower bound at the root node is also $\frac{1}{2}\sum_i p_i$.

Max clique computation. In order to detect operator infeasibility, since in these experiments $p = 2$, it is sufficient to check for the existence of triangles in $G_h = (N, B(S^h))$, which can be done in $O(|N|^3)$ time.

Branching scheme. At each node h of the enumeration tree, potential branching variables are in fact unset components of the corresponding partial selection S^h. The implementation of our algorithm chooses (randomly) among unset machine-disjunctive pairs first, and then among operator-disjunctive pairs.

Upper bound. Only for the nodes where all the machine-disjunctive pairs have been set, the algorithm computes an upper bound by building a feasible solution of $JSO(3,2)$, starting from a feasible solution of the corresponding standard job shop instance. The procedure, starting from the beginning of the schedule, determines the first interval (if it exists) in which $p' > 2$ operations are simultaneously executed. If this is the case, a suitable set of operations is greedily shifted forward to remove operator infeasibilities. More precisely, when the first infeasibility is detected, we shift forward an operation, say i, together with all the successors of i. We choose the operation i, among the p' overlapping operations, whose shift causes the minimum increase in the makespan. This procedure is repeated until all infeasibilities are removed.

We performed two sets of experiments, namely on *small* instances (up to 10 operations per job on the average, processing times up to 100) and *large* instances (up to 30 operations per job on the average, processing times up to 200).

7.1 Small instances

In Table 2, we report the characteristics of each class of randomly generated instances, based on number of machines (m), average number of operations per job ($\bar{\nu}_{\max}$), and range of processing times (p_i). For each class, 10 instances are randomly generated, drawing processing times from a uniform distribution in the specified range. Despite the small size of these instances, Cplex is not able to solve most of them (except for those in class 1) within a time limit of 3 hours.

class	m	$\bar{\nu}_{\max}$	p_i	class	m	$\bar{\nu}_{\max}$	p_i	class	m	$\bar{\nu}_{\max}$	p_i
1	3	5	$[1, 10]$	**10**	3	7	$[1, 10]$	**19**	3	10	$[1, 10]$
2	3	5	$[1, 50]$	**11**	3	7	$[1, 50]$	**20**	3	10	$[1, 50]$
3	3	5	$[1, 100]$	**12**	3	7	$[1, 100]$	**21**	3	10	$[1, 100]$
4	5	5	$[1, 10]$	**13**	5	7	$[1, 10]$	**22**	5	10	$[1, 10]$
5	5	5	$[1, 50]$	**14**	5	7	$[1, 50]$	**23**	5	10	$[1, 50]$
6	5	5	$[1, 100]$	**15**	5	7	$[1, 100]$	**24**	5	10	$[1, 100]$
7	7	5	$[1, 10]$	**16**	7	7	$[1, 10]$	**25**	7	10	$[1, 10]$
8	7	5	$[1, 50]$	**17**	7	7	$[1, 50]$	**26**	7	10	$[1, 50]$
9	7	5	$[1, 100]$	**18**	7	7	$[1, 100]$	**27**	7	10	$[1, 100]$

Table 2: Instances classes.

The experimental results are summarized in Tables 3 and 4. In particular, Table 3 reports the average solution CPU times (in seconds) for, from column 2 to 6, the branch and bound with LB_1 (B&B), branch and bound using both lower bounds (B&B+), dynamic programming (DP), block heuristic (Heur), and block heuristic with post processing (Heur+). The symbol * in the second column means that at least one instance is not solved to optimality within 3 hours of computation (classes 13–27) and the reported average values are computed considering this time limit. Table 4 reports the average optimal solution values (column 2) and the average gaps (percentage) between the optimal solution and the solution found by the two versions of the branch and bound within 3 hours (columns 3 and 4), block heuristic (column 5), and block heuristic with post processing (column 6).

A few comments are in order. It is clear from Table 3 that the dynamic programming procedure outperforms the branch and bound. In fact, computation times are dramatically smaller than those of the branch and bound (which in turn are smaller than Cplex computation times). This is due to the fact that the time complexity of the dynamic program for $JSO(3, 2)$ is $O(\nu_{\max}^3 T^2)$, i.e., computation time is still manageable even for a reasonable growth in the number of operations and/or the processing times. Note that using the lower bound LB_2 provided by Jackson's preemptive schedule becomes beneficial, in terms of efficiency, starting from class 10 (with few exceptions). On the average, the improvement is roughly 3.7%, over all the 27 classes, and 31.7%, only over the classes where B&B+ performs better than B&B. Note also that computation times of the heuristic algorithms are definitely not as sensitive to increasing instance size as their exact counterparts do. This behavior is even more apparent in larger instances (see the results Section 7.2). In addition, as one may expect, the efficiency of the dynamic program is affected by processing times values and by the number of machines (i.e., computation times increase with those quantities). On the contrary, the performances of branch and

Class	B&B	B&B+	DP	Heur.	Heur+
1	2.6	2.9	0.0	0.0	0.0
2	13.2	15.6	0.0	0.0	0.1
3	25.0	24.4	0.0	0.0	0.0
4	6.2	7.5	0.0	0.0	0.1
5	71.1	81.7	0.0	0.1	0.1
6	65.7	86.9	0.1	0.0	0.1
7	2.8	1.5	0.0	0.0	0.0
8	75.8	79.4	0.0	0.0	0.0
9	111.1	139.8	0.2	0.0	0.0
10	4137.7	667.7	0.0	0.0	0.1
11	2692.4	2461.5	0.1	0.0	0.2
12	2826.3	318.2	0.3	0.0	0.2
13	9328.5*	4328*	0.2	0.2	0.3
14	8229.4*	7379.8*	0.7	0.0	0.5
15	9527.0*	9148.3*	1.0	0.1	0.6
16	10800*	10800*	0.3	0.1	0.5
17	9776.2*	10800*	3.0	0.0	0.9
18	8819.6*	9916.8*	6.0	0.0	0.9
19	10800*	9786.2*	0.5	0.1	1.0
20	10800*	10800*	1.5	0.1	1.3
21	10800*	10800*	5.1	0.3	1.5
22	10800*	10800*	1.0	0.3	2.5
23	10800*	10800*	8.0	0.5	4.6
24	10800*	10800*	31.4	0.4	6.5
25	10800*	10800*	2.6	0.7	4.7
26	10800*	10800*	19.7	0.9	11.0
27	10800*	10800*	64.6	0.9	10.6

Table 3: Efficiency results: CPU times (seconds) for solution algorithms.

Class	Optimal val.	B&B	B&B+	Heur.	Heur+
1	42.7	0.0	0.0	15.3	10.5
2	201.1	0.0	0.0	14.0	8.9
3	389	0.0	0.0	14.9	12.4
4	39.4	0.0	0.0	14.5	10.4
5	184.3	0.0	0.0	18.6	12.5
6	348.6	0.0	0.0	17.3	15.2
7	31.0	0.0	0.0	16.2	12.4
8	177.1	0.0	0.0	15.5	9.5
9	391.4	0.0	0.0	11.7	9.2
10	57.9	0.0	0.0	18.0	13.2
11	266.2	0.0	0.0	14.8	11.6
12	556.4	0.0	0.0	12.9	9.8
13	53.9	0.0	0.0	15.7	10.5
14	253.5	1.0	1.0	15.7	10.3
15	495.1	2.0	1.7	15.3	12.6
16	45.0	4.8	3.7	13.5	8.4
17	250.4	7.3	7.2	15.6	13.3
18	516.1	4.3	4.3	11.8	9.1
19	65.2	17.7	7.2	15.6	13.1
20	353.0	11.8	8.1	17.6	13.3
21	768.3	18.0	13.0	15.4	11.8
22	60.6	14.2	11.9	13.2	9.2
23	363.0	17.0	15.1	16.2	11.9
24	698.2	16.0	14.2	12.6	9.2
25	70.4	14.9	12.04	14.5	9.6
26	363.8	18.7	17.05	17.4	10.8
27	744.8	14.9	15.5	15.3	11.2

Table 4: Effectiveness results: solution values and optimality gaps.

bound and heuristic algorithms are completely unrelated to jobs durations and are only slightly sensitive to the number of machines.

Regarding the effectiveness (see Table 4) of the algorithms, we note that the branch and bound is able to solve most of the instances with up to 20 operations and 7 machines to optimality (classes 1–18). For larger size instances (classes 19–27), both versions of the branch and bound reach the time limit (three hours), attaining solutions which are around 15% and 12% above the optimal value. Observe that the heuristic algorithm always obtains feasible solutions of the same quality in less than 1 second. The optimality gap is further reduced to around 11% by the post processing (Heur+). This requires up to 10 seconds of computation for instances in classes 19–27. In terms of effectiveness, the lower bound LB_2 is always useful with the exception of class 27. In this case B&B performs better than B&B+ due to the effect of both time limit and larger computation time for B&B+. On the average, the improvement is 23.5%, over all the 27 classes and 24.1%, only over the classes where B&B+ performs better than B&B.

In the enumeration scheme, when looking at the ratio between the number of subproblems in which $LB_1 < LB_2$ and the total number of subproblems, no clear trend can be inferred depending on the instance classes. Furthermore, in each class, there are always instances with highly divergent ratios.

7.2 Large Instances

Since the dynamic programming and heuristic algorithms are very efficient on Classes 1–27, we designed a second set of experiments in order to compare the performance of the two algorithms on instances of larger size. The characteristics of these experiments are summarized in Table 5. Again, for each class, 10 instances are randomly generated.

Class	m	$\bar{\nu}_{\max}$	p_i	Class	m	$\bar{\nu}_{\max}$	p_i	Class	m	$\bar{\nu}_{\max}$	p_i
L1	3	20	$[1, 50]$	**L4**	3	25	$[1, 50]$	**L7**	3	30	$[1, 50]$
L2	3	20	$[1, 100]$	**L5**	3	25	$[1, 100]$	**L8**	3	30	$[1, 100]$
L3	3	20	$[1, 200]$	**L6**	3	25	$[1, 200]$	**L9**	3	30	$[1, 200]$

Table 5: Large instances classes.

The experimental results are summarized in Table 6. In particular, columns 2–4 show the average solution CPU times (in seconds) for the dynamic programming, block heuristic, and block heuristic with post processing, while in columns 5 and 6 the average gaps (percentage) between the optimal solution and the solution found by the block heuristic and block heuristic with post processing are reported.

For an increasing number of operations, the heuristic computation times grow less than those of the dynamic program. However, the post processing procedure requires a considerable effort. On the average, the heuristic with post processing (Heur+) is more than 5 times faster than the dynamic program and, without post processing, roughly 50 times faster. We observe that the behavior of the dynamic program largely varies even within instances of the same class. This trend is particularly apparent in this second set of instances, where computation times are larger and therefore differences are more evident.

As above, the symbol * in the second column of Table 6 indicates that at least one instance is not solved to optimality by the dynamic program after 3 hours of computation.

Class	DP	Heur	Heur+	Gap Heur.	Gap Heur+
L1	163.6	9	86.7	20.2	14.6
L2	904.6	9.7	92.8	14.1	11.2
L3	894.1	9.5	94.9	19.1	12.9
L4	1331.3	36	363.0	15.0	12.1
L5	1466.5*	36	360.8	17.1	12.5
L6	2253*	35.3	376.4	14.1	11.3
L7	4821.3*	81.3	387.3	21.2	12.5
L8	4930.1*	103.6	1029.0	14.6	11.30
L9	3301.6*	106.1	1055.6	13.7	10.9

Table 6: Efficiency and effectiveness results: CPU times (seconds) and optimality gap on larger instances.

More precisely, in classes L5–L9, the algorithm reached the time limit in at most 2 out of the 10 instances for each class. The reported average values in column 2 are computed considering the time limit when reached, therefore they are lower than the real computation times. Moreover, the percentage gap reported in the last two columns is computed only over the instances solved to optimality within the time limit.

Concerning effectiveness, the heuristic approach still provides solutions of reasonable quality for large instances. The average gaps for the algorithms with and without post processing are around 12% and 16% respectively. Compared with the 11% and 15% average values for small instance classes, this shows the robustness of the heuristic approach for an increasing size of the instances.

8 Conclusions

In this paper we addressed a new class of resource-constrained job shop scheduling problems, where at most $p < n$ jobs can be processed simultaneously. We studied its complexity and proposed two exact algorithms, a polynomial time approximation scheme and two versions of a heuristic algorithm. Extensive computational results show that the dynamic programming algorithm is the winner for instances with up to 90 operations in the case $n = 3$, $p = 2$. While the efficiency of the proposed branch and bound algorithms is very far from that of the dynamic program, the heuristic algorithms prove to be competitive, providing solutions of good quality in a significantly smaller computation time.

The heuristic approach can be extended to the case in which there are n jobs (and two operators) but its complexity makes its use burdensome. On the other hand, there are reasons to believe that the procedure becomes more effective as n grows. In fact, in the post processing phase, the chances that conflicts between operations of the same job assigned to two different macro-jobs occur, become smaller for larger n values. We believe that this can be material for further research.

On these grounds, possible directions for future investigations include (*i*) the study of better lower bounds to be used in the branch and bound; (*ii*) additional computational studies on instances with larger values of n and p; (*iii*) refinements of the block heuristic in which the nodes of the graph are generated more efficiently.

References

[1] Agnetis A., M. Flamini, G. Nicosia, A. Pacifici. Scheduling three chains on two parallel machines, *European Journal of Operational Research* 202(3), 669–674, 2010.

[2] Agnetis, A., G. Oriolo. The Machine Duplication Problem in a Job Shop with Two Jobs, *International Transactions on Operational Research*, 2(1), 45–60,1995.

[3] Akers S.B. A graphical approach to production scheduling problems. *Operations Research*, 4, 244-255, 1956.

[4] Baki, M.F. and R.G. Vickson. One-operator, two-machine open shop and flow shop problems with setup times for machines and weighted number of tardy jobs objective, *Optimization Methods and Software*, 19(2), 165–178, 2004.

[5] Blazewicz J., W. Cellary, R. Slowinski, J. Weglarz. Scheduling under resource constraints–deterministic models. *Annals of Operations Research*, 7, 1–359, 1986.

[6] Brucker P. An efficient algorithm for the job-shop problem with two jobs. *Computing* 40, 4, 353–359, 1988.

[7] Brucker P., S. Knust, G. Wang. Complexity results for flow-shop problems with a single server, *European Journal of Operational Research*, 165, 398–407, 2005.

[8] Brucker P., A. Krämer. Polynomial algorithms for resource-constrained and multiprocessor task scheduling problems. *European Journal of Operational Research*, 90, 214–226, 1996.

[9] Carlier J. The one-machine sequencing problem, *European Journal of Operational Research*, 11, 42–47, 1982.

[10] Carlier J. and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job shop problem, *Annals of Operations Research*, 26, 269–287, 1990.

[11] Chen J. and C.-Y. Lee. General multiprocessor task scheduling, *Naval Research Logistics*, 46, 57–74, 1999.

[12] Cheng T. C. E., G. Wang, C. Sriskandarajah. One-operator two-machine flowshop scheduling with setup and dismounting times, *Computers & Operations Research*, 26, 715–730, 1999.

[13] Du J., J.Y.T. Leung, G.H. Young. Scheduling chain-structured tasks to minimize makespan and mean flow time. *Information and Computation*, 92(2), 219–236, 1991.

[14] Floyd R.W. Algorithm 97: Shortest Path. *Communication A.C.M.*, 5, 6, 219, 1963.

[15] Garey M.R. and D.S. Johnson. "Computers and intractability", Freeman, NY, 1979.

[16] Glass C.A., Y.M. Shafransky, V.A. Strusevich. Scheduling for parallel dedicated machines with a single server, *Naval Research Logistics*, 47, 304–328, 2000.

[17] Graham R.L., E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics* 4, 287–326, 1979.

[18] Hart P.E., N.J. Nilsson, B. Raphael., A formal basis for he heuristic determination of minimum cost paths, IEEE Trans. Syst. and Cybern. SSC-4 (1968) 100–108.

[19] Ibarra O.H., C. Kim. Fast approximation algorithms for the knapsack and sum of subset problems. *Journal of ACM*, 22, 463–468, 1975.

[20] Kellerer H., V.A. Strusevich. Scheduling parallel dedicated machines under a single non-shared resource. *European Journal of Operations Research*, 174, 345–364, 2003.

[21] Kellerer H., V.A. Strusevich. Scheduling problems for parallel dedicated machines under multiple resource constraints. *Discrete Applied Mathematics*, 133, 45–68, 2004.

[22] Roy B., B. Sussmann. Les problemes d'ordonnancement avec constraintes disjonctives, *SEMA, Note D.S.*, No. 9, Paris, 1964

[23] Sethi S., C. Sriskandarajah, S. van de Velde, M. Y. Wang, and H. Hoogeveen. Minimizing makespan in a pallet-constrained flowshop. *Journal of Scheduling*, 2(3), 115–133, 1999.

[24] Sotskov Y.N. and N.V. Shakhlevich. NP-hardness of shop-scheduling problems with three jobs. *Discrete Applied Mathematics*, 59(3), 237–266, 1995.

[25] Wang M.Y., S.P. Sethi, C. Sriskandarajah and S.L. van de Velde. Minimizing makespan in a flowshop with pallet requirements: computational complexity. *INFOR (Information Systems and Operational Research)*, 35, 277–285, 1997.

[26] Warshall A. A Theorem on Boolean Matrices *Journal A.C.M.*, 9, 1, 11–12, 1963.

[27] Wei V. K. A lower bound on the stability number of a simple graph, Technical Memorandum No. 81-11217-9, Bell Laboratories, 1981.