A Comparative Computational Study of the Effect of the Preliminary Reduction for the Classical and Multiprocessor Job-Shop Scheduling Problems

Lester Carballo, Nodari Vakhania

Facultad de Ciencias, UAEM,
Av. Universidad 1001, Cuernavaca 62210, Morelos, Mexico

Frank Werner

Fakultät für Mathematik, Otto-von-Guericke-Universität Magdeburg,
PSF 4120, 39016 Magdeburg, Germany

March 12, 2013

Abstract: The multiprocessor job-shop scheduling problem is a generalization of the classical job-shop problem, in which each machine is replaced by a group of parallel machines. In the present model, we consider the most general case when the parallel machines are unrelated. In this setting, the amount of feasible solutions grows drastically even compared with the classical job-shop problem, which itself is one of the most difficultly treatable, in practice, strongly NP-hard problems. In this paper, we exploit a method of the preliminary reduction of the solution space (i.e., before any lower bounds are applied). This reduction has a twofold effect. At the first phase, which is common for both, classical job-shop problem and it multiprocessor generalization, dominance principles are applied. This results in a reduced solution space which is a subspace of the set of the so-called active schedules. The second phase is designed exclusively for multiprocessor environment. The additional dominance rules of phase 2 permit a drastic reduction of the drastically increased solution space. According to these dominance rules, at each stage of the branching in the reduced solution tree, the whole conflict set of operations is partitioned into smaller subsets so that the branching in each subset is carried out independently from the operations of the rest of the subsets. According to an earlier proposed by the authors probabilistic model, this results in an exponential reduction of the whole set of the feasible solutions with the probability close to 1. In this paper, this theoretic estimation is testified by intensive computational experiments. Instances of the multiprocessor job-shop problem have been generated by extending known benchmark instances of the job-shop problem. Despite an exponential increase of the number of feasible solutions in the extended instances, we were able to solve many of these instances optimally just with our reduction algorithm. At the same time, the corresponding job-shop instances have been proven to be computationally hard. Some of them have been only solved (optimally) by sophisticated implicit enumerative (branch-and-bound) algorithms, whereas for other instances no optimal solutions are known. The preliminary reduction of the solution space of our multiprocessor job-shop problem is of an essential importance due to the fact that the elaboration of efficient lower bounds for this problem is complicated. In this paper we address the difficulties connected with this kind of development, proposing, at the same time, a number of possible lower bounds. The lower bounds which are easily calculable seem to be weak, whereas the calculation of stronger lower bounds require essentially more significant computational efforts.

Keywords: Job shop scheduling, Branch-and-bound algorithm, Solution tree, Dominance relations

1 Introduction

The multiprocessor job-shop scheduling problem (MJSP) is a generalization of the classical job-shop problem (JSP) which reflects the operation in many industries. In MJSP each machine in JSP is replaced by a group of parallel machines. In the present model we consider the most general case when the parallel machines are unrelated. In this setting, the amount of feasible solutions grows drastically even compared with the classical job-shop problem, which itself is one of the most difficultly treatable, in practice, strongly NP-hard problems.

In this paper, we exploit a method of the preliminary reduction of the solution space (i.e., before any lower bounds are applied). This reduction has a twofold effect. At the first phase, which is common for both the classical job-shop problem and its multiprocessor generalization, dominance principles are applied. This results in a reduced solution space which is a subspace of the set of the so-called active schedules. As an example, our reduction algorithm has solved optimally a classical 6x6 job-shop problem instance (see [10]) while the earlier reported algorithms solving this instance were using lower bounds for the reduction of the solution space.

The second phase of the reduction algorithm is designed exclusively for the multiprocessor environment. The additional dominance rules of phase 2 permit a drastic reduction of the drastically increased solution space. According to these dominance rules, at each stage of the branching in the reduced solution tree, the whole conflict set of operations is partitioned into smaller subsets so that the branching in each subset is carried out independently from the operations of the rest of the subsets.

According to a probabilistic model earlier proposed by Vakhania and Shchepin [31], the second phase results in an exponential reduction of the whole set of the feasible solutions with a probability close to 1 (to the best of our knowledge, this is the first example of a scheduling problem in which the presence of parallel processors can simplify the solution). In this paper, this theoretic estimation is testified by intensive computational experiments. Instances of the multiprocessor job-shop problem have been generated by extending known benchmark instances of the job-shop problem. Despite an exponential increase of the number of feasible solutions in the extended instances, our reduction algorithm was able to solve many of these instances optimally. At the same time, the corresponding job-shop instances have been proven to be computationally very hard: Some of them have been only solved (optimally) by sophisticated implicit enumerative (branch-and-bound) algorithms, whereas for other instances no optimal solutions are known.

To compare the feasible solution spaces of MJSP and JSP, we note that ν simultaneously available operations can be assigned to μ machines in $(\nu + \mu - 1)!/(\mu - 1)!$ different ways, while the corresponding number for a single machine is $\nu!$. The earlier theoretical estimation from [31] for the number of the schedules, generated by our reduction algorithm for the corresponding MJSP instance was $[(\nu/\mu)!]^{\mu}$. At the same time, our computational experiments have shown that the number of created feasible solutions, in practice, is much less (see Section 4 for more details).

The preliminary reduction of the solution space of our multiprocessor job-shop problem is of an essential importance due to the fact that the elaboration of efficient lower bounds for this problem is complicated. In this paper, we address the difficulties connected with this kind of development, proposing, at the same time, a number of possible lower bounds (to the best of our knowledge, no lower bounds for this problem have been suggested earlier). Those lower bounds which are easily calculable seem to be weak, whereas the calculation of stronger lower bounds require essentially more significant computational efforts.

1.1 Problem Definition

The job-shop scheduling problem in its classical setting deals with m distinct *machines* or *processors* and n distinct *jobs*. Each job is to be performed by some machines in the given order. We call an *operation* the resultant activity of the performance of a job on a machine. Thus, in the job-shop problem we have the *precedence* relations between the operations of the same job in a serial-parallel form. In the version of the job-shop problem studied here, we allow a group of parallel *unrelated* machines instead of a single machine and we do not distinguish jobs, we rather consider arbitrary precedence relations between the operations (instead of serial-parallel relations in a job-shop problem). Here we shall refer to this extension of JSP as the multiprocessor job-shop scheduling problem with unrelated machines (abbreviated MJSP).

As earlier mentioned, JSP is a strongly NP-hard problem with one of the worst practical behavior. At the same time, no approximation algorithm with a guaranteed performance for this problem exists. It is important because it models the actual operation in several industries, complex computer systems and other real-life applications. In many practical circumstances, JSP is still restricted as it allows only one processor for each task group. MJSP meets better the needs in several industries and applications. For example, a computer may have parallel processors each of which might be used by a program task or in a manufacturing plant, a job might be allowed to be processed by any of the available parallel machines. Besides, the precedence relations might be more complicated than serial-parallel type relations. For example, the completion of two or more program tasks (subroutines) might be necessary before some other program task can be processed (as the latter task uses the output of the former tasks); this is a typical situation in parallel and distributed computations.

The MJSP can be formulated as follows. We have a set of *tasks* or *operations*, $\mathcal{O} = \{1, 2, ..., n\}$ and m different processor groups. $\mathcal{M}_k$ is the kth group of parallel *processors* or *machines*, P_{kl} being the lth processor of this group. (A job in a factory, a program task in a computer or a lesson at school are some examples of jobs. A machine in a factory, a processor in a computer, a teacher in a school are some examples of machines.) Each task should be performed by any processor of the given group. d_{iP} is the (uninterrupted) processing time of task i on processor P. O_k is the set of tasks to be performed on the kth group of processors. Each group of parallel processors can be *unrelated*, *uniform* or *identical*. Unlike uniform machines which are characterized by an operation-independent speed function, unrelated machines have no uniform speed characteristic, i.e., the machine speed is operation-dependent; that is, the processing times d_{iP} are independent, arbitrary integer numbers. In the case of identical machines, the processing time of each task on all processors is the same, i.e., all processors have the same speed. Uniform machines are characterized by a speed function (the same for all tasks). Thus, for identical processors, task processing times are processor-independent, whereas for uniform and unrelated machines, these times are processor dependent.

The problem setting imposes the *resource constraints*: For each two tasks i, j assigned to the same processor P, either

$$s_i + d_{iP} \leq s_j \qquad \text{or} \qquad s_j + d_{jP} \leq s_i$$

should hold, where s_i is the starting time of task i; in other words, any processor can handle only one task at a time.

The *precedence constraints* are as follows. For each $i \in \mathcal{O}$, we are given the set of immediate predecessors $pred(i)$ of task i so that i cannot start before all tasks from $pred(i)$ are finished. Task i becomes *ready* when all tasks from $pred(i)$ are finished.

A *schedule (solution)* is a function which assigns to each task a particular processor and a starting

time (on that processor). A *feasible schedule* is a schedule satisfying the above constraints. An *optimal schedule* is a feasible schedule which minimizes the *makespan*, that is, the maximal task completion time.

As it is well known, an optimal schedule of JSP is among the so-called *active schedules*: in an active schedule, no operation can start earlier than it is scheduled without delaying some other operation (for example, see Lageweg et al. [18] for details).

Applying the commonly used notation for scheduling problems, we use $J||C_{\max}$, $JR|prec|C_{\max}$, $JQ|prec|C_{\max}$ and $JP|prec|C_{\max}$, respectively, to denote JSP and the versions of MJSP with unrelated, uniform and identical processors, respectively. If in an instance of MJSP from each group of processors all processors except an arbitrarily selected one is eliminated, then a corresponding instance of JSP is obtained. MJSP can be seen as a so-called resource-constrained project scheduling problem: we associate the kth machine group with the kth resource the amount of which is the number of parallel machines in the group. The requirement of the kth resource of each operation from O_k is 1 and that of any other operation is 0.

As we have already mentioned, JSP and hence MJSP are strongly NP-hard. Though the construction of each feasible schedule takes a polynomial (in the number of operations and machines) time, for finding an optimal schedule we might be forced to enumerate an exponential number of feasible schedules. Since each feasible schedule can be rapidly generated, different priority dispatching rules, insertion algorithms (see e.g. Werner and Winkler [34] for JSP or Sotskov et al. [26] for JSP with setup times) or other heuristics can be used for a rapid generation of some feasible schedule(s). The simplest considerations which reflect priority dispatching rules are not enough to obtain a solution with a good quality. In fact, any of such rules may generate the worst solution that may exist. If the quality of the required solution is important, we need to work with a larger subset of the feasible solution space.

1.2 Some Related Work

One of the earliest published articles mentioning about a generalization of JSP is that of Giffler and Thompson [11]. In that model, identical processors and serial-parallel type precedence relations were introduced, instead of unrelated processors and arbitrary precedence relations in our generalized problem. In [27], solution methods for JSP and other problems related to MJSP are described. An extension of JSP with general multi-purpose machines was studied by Brucker and Schlie [4], Brucker et al. [3] and Vakhania [29], and a lower bound for the special case of this problem when the operation processing time is a constant (i.e. machine-independent) was suggested by Jurisch [16]. Shmoys et al. [25] have proposed a polynomial approximation randomized algorithm for the problem $R|chain|C_{\max}$ which can be applied to the version of our problem with serial-parallel precedence relations $JR|serial|C_{\max}$. Dauzére-Pérés and Paulli [8] have proposed a tabu search algorithm. Vakhania [30] has suggested a version of a beam search algorithm for the generalized problem. Ivens and Lambrecht [15] and Schutten [24] study different extensions of JSP including extensions with setup and transportation times. Vakhania and Shchepin [31] have considered the most general version of MJSP with unrelated machines and have suggested an algorithm that constructs a reduced solution tree for this problem. As earlier mentioned, with a probability of almost 1, the number of generated feasible solutions, as compared to the number of all active feasible schedules, decreases with the number of machines and operations in each group of machines and operations, as follows. If we let ν and μ to be the number of operations and machines in each subset of operations and machines, then with a probability of almost 1, the algorithm generates approximately $(\mu)^{m\nu}$ and $2^{m(\mu-1)}\mu^{m\nu}$ times less feasible schedules than the number of all active feasible schedules of any corresponding instance

of JSP and our generalized problem, respectively.

We conclude our introduction with a brief description of how the paper is organized. In the next section, we give some preliminaries and a general scheme for representing the solutions created by the reduction algorithm. The reduction algorithm is described in Section 3. In Section 4, we define auxiliary scheduling problems. These problems are simpler scheduling problems which, on the one hand, give some useful insight into our general problem, and on the other hand, are useful for the developments of lower bounds for MJSP. In Section 5, using auxiliary scheduling problems from Section 4, we propose some lower bounds and bounds which are not strict lower bounds. Strict bounds may potentially be used in an implicit enumeration algorithm, whereas our non-strict bounds may be used in a heuristic algorithm based on our reduction algorithm. In Section 6, we describe our computational results that reflect the practical behavior of our reduction algorithm for a number of instances obtained from earlier known benchmark instances for JSP. Finally we give some directions for future work in the concluding remarks.

2 Some Basic Notions and Concepts for the Reduction Algorithm

In this section, we introduce the relevant basic notions and concepts.

Our *solution tree* T enumerates the schedules we generate. T is a rooted tree, the root representing a fictitious empty solution. Each internal node of T represents a partial solution, and each of its leaves represents a complete feasible solution.

We call a node h from T a *stage* and denote the (partial and complete) solution of stage h by σ_h.

At each stage h, we branch by the operations from a bunch of concurrent ready operations, the candidates to be scheduled at that stage (called the *branching (quick) set*), and denote it by $\mathcal{C}_h$. By branching in T at stage h, we resolve the resource (machine) conflicts in $\mathcal{C}_h$, each alternative machine in $\mathcal{M}_k$ implies its own conflicts. An alternative solution determined by each operation in $\mathcal{C}_h$ scheduled on a machine from $\mathcal{M}_k$. So one immediate successor h' of h is generated for each $i \in \mathcal{C}_h$; two labels are associated with the arc (h, h'): the task i and the processor from $\mathcal{M}_k$ on which task i is actually scheduled.

Notice that for a complete enumeration, a branching for *each* processor from $\mathcal{M}_k$ is to be generated. However, we avoid such a complete enumeration by selecting a *single* processor from $\mathcal{M}_k$ at stage h, as specified a bit later. In Figure 4 we represent a fragment of T for one of the problem instances tested by our algorithm.

Thus, there will be generated $|\mathcal{C}_h|$ extensions of the current partial schedule σ_h in our tree T. The schedule σ_h can clearly be seen as a (partial) permutation of n tasks. For $i \in \sigma_h$ in that permutation, we use the upper index for specifying the particular processor on which task i is scheduled in σ_h. In particular, $\sigma_h i^P$ is an extension of σ_h with task i scheduled on processor $P \in \mathcal{M}_k$.

The schedule σ_h is identified by the path from the root to the node h in T. Note that the relative order of two tasks $i, j \in \sigma_h$ is relevant only if they are scheduled on the same processor.

As we have mentioned above, while branching by the set $\mathcal{C}_h$, we generate only branchings corresponding to the feasible assignment of each ready operation in $\mathcal{C}_h$ to only one, a specially selected processor from $\mathcal{M}_k$ that we call a *quick* processor: a quick processor is a fastest one for a given operation at a given stage.

A selection of a quick machine takes time, linear to the number of machines in the corresponding group. The branching (quick) set $\mathcal{C}_h$ is formed by ready operations conflicting on a machine, which is quick for at least one of these operations. We are allowed to branch by a quick set, if it is dominant. Intuitively, if we branch by a dominant set, then we are guaranteed that we will not delay any not yet ready operation (not included in the set), competing on the machines of the same group.

The feasible solution set can be further diminished by reducing quick dominant sets in a special way. This reduction is based on an "artificial" relaxation of conflicts between the operations of the conflict sets. At each stage of branching, the branching by some operations is postponed whenever this is possible. Each quick set is partitioned into specially determined subsets, corresponding to different alternative machines. Then, instead of branching by the whole quick set, branchings are performed by the subsets from the partition, on different machines at different levels of the solution tree. So the concurrent jobs from different subsets are processed in parallel.

We represent each feasible solution σ_h by a directed weighted graph G_h. We associate the digraph (task graph) $G_0 = (X, E_0)$ with the root of T (see Figure 3). To each task $i \in O$, there corresponds the unique node $i \in X$. There is one fictitious initial node 0, preceding all nodes, and one fictitious terminal node $n+1$, succeeding all nodes in G_0. E_0 is the arc set consisting of the arcs (i, j), for each task i, directly preceding task j; $(0, i) \in E_0$ if task i has no predecessors and $(j, n+1) \in E_0$ if task j has no successors.

We denote by $w(i, j)$ the weight associated with $(i, j) \in E_0$; initially, we assign to $w(i, j)$ the minimal processing time of task i, later we correct these weights when we assign a task to the particular processor.

Let (h, h') be an edge in T with task j scheduled at iteration h' on processor P. Then we obtain $G_{\sigma_{h'}}$ from G_{σ_h} as follows. We complete the arc set of the latter graph with the arcs of the form (i, j), with the associated weights $w(i, j) = d_{iP}$, for each task i, scheduled earlier on the processor P. We correct the weights of all arcs incident out from node j $(j, o) \in E_0$, as $w(j, o) := d_{jP}$. It is easily seen that the length of a critical path in $G_{h'}$ is the makespan of the (partial or complete) solution $\sigma_{h'} = \sigma_h j^P$ which we denote by $|\sigma_{h'}|$.

Note that the critical path length from node 0 to a node o in G_h is a lower bound on the starting time of operation o in schedule σ_h and in any of its successor schedules. We call it the *earliest starting time* or the *head* of operation o by stage h and denote it by $\text{head}_h(o)$. Likewise, the critical path length from o to the sink node in G_h is a lower bound on the total remained work once operation o is already finished. We denote it by $\text{tail}_h(o)$ and call it the *tail* of operation o at stage h. $R_h(M)$ is the *release time* of machine M at stage h, that is, the completion time of the operation, scheduled last by that stage on M.

3 The Reduction Algorithm

In this section, we describe our reduction algorithm. We need to introduce formally some earlier mentioned notions as well as some new concepts to give a formal description of the algorithm. For self-completeness of the presentation, we also give the basic properties on which the algorithm relies. The proofs of these properties (lemmas and theorems from this section) can be found in [31].

3.1 Phase 1: Constructing the Quick Solution Tree

In this subsection, we describe how the preliminary reduction of the feasible solution space is accomplished in Phase 1 of the reduction algorithm. Recall that Phase 1 is common for both JSP and MJSP.

3.1.1 Dominance Relations for Creating Active Schedules

We first describe the dominance relations which have been traditionally used for a preliminary reduction of the whole feasible solution set for JSP.

Let us denote by $\mathcal{O}_{kh}$ the subset of operations of O_k, not yet scheduled by stage h. Let o be a ready operation from $\mathcal{O}_{kh}$ and p be another operation from $\mathcal{O}_{kh}$, such that $\text{head}_h(o) \leq \text{head}_h(p)$.

We will say that o *conflicts* with p on machine $M \in \mathcal{M}_k$ at stage h, if the overlapping of the execution times of o and p on M is more than one point, i.e., o cannot be completed on M before or at the earliest starting time of p at stage h.

If o does not conflict with p, then o *dominates* p, written $o \ \text{dom}_{Mh} \ p$ (o can be completed on M before or at the earliest moment when p can be started on M).

Thus, if o conflicts with p on M, then by scheduling o on M, we will delay the starting of p on M, otherwise, we will not delay the starting of p.

Let k be a machine group with at least one ready operation in $\mathcal{O}_{kh}$, $M \in \mathcal{M}_k$, and let o be a ready operation in $\mathcal{O}_{kh}$ with the minimal earliest completion time on M. Then the set of operations, consisting of operation o and all the operations of $\mathcal{O}_{kh}$, conflicting with o on M, is called the *conflict set* on M at stage h and is denoted by $\mathcal{C}_h^M$.

Operation o may conflict with operation p on machine M, but may not conflict on another machine Q. We illustrate this by the following simple example. Let $\{1, 2, 3\}$ be ready operations with $\text{head}_h(1) = 5,\ \text{head}_h(2) = 6,\ \text{head}_h(3) = 8,\ d_{1Q} = 2,\ d_{1M} = 6,\ d_{2Q} = 5,\ d_{2M} = 3,\ d_{3Q} = 2,\ d_{3M} = 7$. Assume that $R_h(Q) = R_h(M) = 5$. Observe that $\mathcal{C}_h^Q = \{1, 2\}$ and $\mathcal{C}_h^M = \{1, 2, 3\}$, since $1 \ \text{dom}_{Qh} \ 3$, though not $1 \ \text{dom}_{Mh} \ 3$. However, if, for example, we change d_{1Q} to 4, or we leave d_{1Q} unchanged, but we change the release time of Q to 7, then 1 will no more dominate 3 on machine Q. Thus, for distinct M and Q, the conflict sets $\mathcal{C}_h^M$ and $\mathcal{C}_h^Q$ are not necessarily the same (though both sets will contain all ready operations with the minimal head).

If a conflict set $\mathcal{C}_h^M$ contains at least one non-ready operation p, then we are not allowed to branch by this set. Moreover, by ignoring operation p we may loose the optimal schedule (since p would be delayed on M in any generated schedule). We are allowed to branch by $\mathcal{C}_h^M$ if it contains no non-ready operation; in this case we shall call it a *dominant conflict set* on machine M at iteration h. Observe that in a dominant set $\mathcal{C}_h^M$, the operation o with the minimal early completion time on machine M dominates all operations of $\mathcal{O}_{kh}$ which are not in $\mathcal{C}_h^M$ (if there exist such operations); in this case we will say that $\mathcal{C}_h^M$ is *dominant by operation* o. We call $\sigma_h o^M$ a *dominant extension* of σ_h, if o belongs to a dominant set on M at stage h.

A partial schedule σ_h *dominates* another partial schedule $\sigma_{h'}$, if there exists at least one complete extension σ of σ_h such that $\tau(\sigma) \leq \tau(\sigma')$ for any complete extension σ' of $\sigma_{h'}$. Let $\mathcal{C}_h^M$ be a dominant conflict set and o be its operation with the minimal head. Recall that $\mathcal{C}_h^M$ contains no ready operation $p \in \mathcal{O}_{kh}$ dominated by o. However, $\sigma_h p^M$ is a feasible extension of σ_h. The following lemma shows that this feasible extension can be ignored.

Lemma 1 *$\sigma_h o^M$ dominates $\sigma_h p^M$. Hence, if the branching is carried out by the dominant set*

$\mathcal{C}_h^M$, *then* $\sigma_h p^M$ *can be abandoned at stage* h.

Let us call a machine $Q \in \mathcal{M}_k$ *quick* at stage h for a ready operation $o \in \mathcal{O}_{kh}$ if the earliest completion time of this operation on machine Q is no more than that on any other machine of $\mathcal{M}_k$ at stage h. In the sequel, we will use the following simple method to generate a dominant conflict set at each stage h.

Lemma 2 *If o is a ready operation at stage h with the minimal earliest completion time and Q is the corresponding quick machine, then the conflict set $\mathcal{C}_h^Q$ is dominant.*

3.1.2 Quick Solution Tree

As already mentioned, the dominant conflict sets have been traditionally used in JSP for branching. In an *active schedule*, no operation can be completed earlier than it is scheduled (by altering the processing order of the operations on the machines) without delaying some other operation. It can be easily seen that the schedules, generated by branching at each stage by a dominant conflict set, are active (see [18] for the details).

Thus so far, we have extended the common branching rule for generating active schedules of JSP for MJSP in a straightforward way: it is clear that it suffices to generate all possible dominant extensions of σ_h corresponding to all dominant conflict sets of stage h, i.e., active schedules of MJSP.

Observe that the number of dominant conflict sets might be the same as that of the parallel machines of the corresponding group. We will now show that it is always possible to generate only extensions corresponding to *just one of the dominant conflict sets*, generalizing in this way the concept of an active schedule for MJSP.

The branchings, generated at each stage, restrict the set of feasible solutions which will be created in T, in the following sense. Suppose we carry out the branching only by a conflict set $\mathcal{C}_h^M$ at stage h. Then in any created in T feasible solution, at least one operation of $\mathcal{C}_h^M$ will be scheduled next on machine M. It is easy to see that, if we arbitrarily select a single dominant conflict set for branching, we may loose an optimal schedule. For example, assume that $\mathcal{C}_h^Q = \mathcal{C}_h^M = \{1, 2\}$ are two dominant conflict sets at stage h. Let $\mathrm{head}_h(1) = 4$ and $\mathrm{head}_h(2) = 5$, $d_{1Q} = 2$, $d_{1M} = 6$ and $d_{2Q} = 3$, $d_{2M} = 7$. It is clear that both operations should be scheduled on machine Q, while, if we branch at stage h only by the set $\mathcal{C}_h^M$, then at least one of these operations will be scheduled on machine M and the optimal schedule will be lost.

We select a single dominant set and discard the rest of the dominant sets with a little computational effort. Let us call a conflict set $\mathcal{C}_h^Q$ *quick* at stage h by operation $o \in \mathcal{C}_h^Q$ if Q is a quick machine for o; that is, any conflict set on a machine, which is quick for at least one its operation, forms a quick conflict set.

Lemma 3 *A dominant conflict set according to Lemma 2 is quick.*

Let us call a *quick solution tree* a solution tree, in which the branching at each stage is carried out by a *single* quick dominant conflict set. Similarly, we shall refer to a solution tree, in which the branching at each stage is carried out by *all* dominant conflict sets as an *active solution tree*. We use Tq and Ta to denote quick and active solution trees, respectively. An active solution tree enumerates all active schedules, while a quick solution tree enumerates a subset of active schedules, the so-called *quick schedules*. Observe that for JSP, quick and active solution trees coincide:

Proposition 1 *A quick solution tree for an instance of JSP enumerates all active schedules for this instance.*

Theorem 1 *There is an optimal schedule in Tq for any instance of MJSP.*

As we will see in the example from Section 3.2, different conflict sets $\mathcal{C}_h^M$, $M \in \mathcal{M}_k$, may contain a different number of operations. Intuitively, the lower is the minimal earliest operation completion time in a conflict set, the less is the number of operations in this set. In this sense, the quick dominant set of Lemma 2 contains a potentially small number of operations.

We associate an instance of JSP with an instance of MJSP. In an instance of JSP, *associated* with the instance of MJSP, a single machine $M \in \mathcal{M}_k$ for $k = 1, ..., m$, is selected and the processing time of each operation $o \in O_k$ is set to d_{oM}. Observe that, if no two alternative machines are identical, then there are in total $\prod_{k=1}^{m} |\mathcal{M}_k|$ problem instances of JSP, associated with an instance of MJSP.

Theorem 2 *A quick solution tree Tq, generated for an instance of MJSP, has no more nodes than an active solution tree Ta, generated for any associated instance of JSP.*

3.2 Phase 2: Further Reduction of the Solution Space for MJSP

In this subsection, we describe Phase 2 of the reduction algorithm that drastically reduces the size of the solution space of instances of MJSP.

3.2.1 Partitioning Conflict Sets

In a branch-and-bound solution tree of JSP, we commonly branch by all operations of a conflict set to provide a non-delay starting of each of these operations (since in an optimal schedule, we may need potentially to schedule any of these operations without any delay). If we do not branch by some operation of a conflict set (and branch by the rest of the operations of the conflict set), then that operation cannot be started at its earliest starting time in any subsequently obtained complete feasible schedule, which may cause the loss of the optimal schedule. The situation is less restricted if there are two or more parallel machines in $\mathcal{M}_k$: it might be possible to postpone the branching by some operations of the conflict set to a later stage without delaying the start of these operations.

Let $\mathcal{C}_h^Q$ be a quick dominant set. We wish to partition $\mathcal{C}_h^Q$ into smaller subsets and branch by these smaller subsets instead of branching by the whole conflict set $\mathcal{C}_h^Q$ at stage h. To partition the conflict set, we look for quick machines for the operations in $\mathcal{C}_h^Q$. The number of subsets in the partition is equal to the maximal number of pairs of the form (o, M), $o \in \mathcal{C}_h^Q$, $M \in \mathcal{M}_k$, such that M is a quick machine for o and no o or M is encountered more than once in any two or more pairs. Roughly speaking, we keep a similar number of operations in each subset of the partition to minimize the number of generated schedules. To be more specific, with each operation $o \in \mathcal{C}_h^Q$ a (non-empty) set of its quick machines is associated. We wish to find the maximal subset of $\mathcal{C}_h^Q$ for each operation of which a different quick machine is selected. This selection problem is equivalent to the maximal matching problem in a bipartite graph $G = (V_1, V_2, E)$ with $V_1 = \mathcal{C}_h^Q$, $V_2 = \mathcal{M}_k$, where there is an edge (o, M) in E if and only if M is a quick machine for operation o at iteration h. This problem is solvable in $O(\sqrt{|V_1|}|E|)$ time, see [14]. Suppose $O_h^{\max}$ is a maximal matching, i.e., a maximal subset of $\mathcal{C}_h^Q$, for each operation of which a distinct quick

machine is selected (note that $|O_h^{\max}| > 0$). We call machine M a *co-quick machine* of iteration h if M is selected for some operation of $O_h^{\max}$.

We partition $\mathcal{C}_h^Q$ into $|O_h^{\max}|$ subsets, such that we have exactly one operation from $O_h^{\max}$ in each subset. The total number of operations in each subset is either $[|\mathcal{C}_h^Q|/|O_h^{\max}|]$ or $[|\mathcal{C}_h^Q|/|O_h^{\max}|]+1$; for the latter option $|\mathcal{C}_h^Q|/|O_h^{\max}|$ is not integral, and the subsets with $[|\mathcal{C}_h^Q|/|O_h^{\max}|]+1$ elements are selected arbitrarily. We call the partition of $\mathcal{C}_h^Q$, obtained in this way, a *compressed decomposition* of $\mathcal{C}_h^Q$. Observe that with each subset in the decomposition, a unique co-quick machine is associated. We denote by $\mathcal{C}_h^{Q,M}$ the subset in the decomposition with the associated machine M and call it a *compressed subset* of $\mathcal{C}_h^Q$ at stage h.

3.2.2 Extended Branching and the Compressed Solution Tree

Let $\mathcal{C}_h^{Q,M_i}$, $i = 1, ..., l$, $l = |O_h^{\max}|$, be an enumeration of the elements of a compressed decomposition of the dominant quick set $\mathcal{C}_h^Q$. Instead of branching by the whole set $\mathcal{C}_h^Q$ at stage h, we branch by subsets of its compressed decomposition on consecutive levels of the solution tree T. We call such a branching, extended at different levels of the tree, an *extended branching* by the set $\mathcal{C}_h^Q$ and the decomposition $\mathcal{C}_h^{Q,M_i}$, and we denote it by $\mathcal{B}(h, Q)$ (we omit the term $\mathcal{C}_h^Q$ for notational simplicity). The corresponding subtree will be referred to as the $\mathcal{B}(h, Q)$-*subtree* (the root of this subtree is the node h). $\mathcal{B}(h, Q)$ originates in total $\prod_{i=1}^{l} |\mathcal{C}_h^{Q,M_i}|$ different paths (leaves of the $\mathcal{B}(h, Q)$-*subtree*). It allocates operations of $\mathcal{C}_h^{Q,M_i}$ on machine M_i at the ith level of the $\mathcal{B}(h, Q)$-*subtree* extending all generated nodes of level $i - 1$ (extending the root of the $\mathcal{B}(h, Q)$-*subtree* initially). So there are $\prod_{l=1}^{i} |\mathcal{C}_h^{Q,M_l}|$ nodes at the ith level of the $\mathcal{B}(h, Q)$-*subtree*. While enumerating our schedules, the $\mathcal{B}(h, Q)$-*subtree* is not generated in a continuous manner. $\mathcal{B}(h, Q)$ consists of different *continuous parts*, i.e., continuously generated subtrees of the $\mathcal{B}(h, Q)$-subtree. The first continuous part of $\mathcal{B}(h, Q)$ starts at stage h and generates $\sum_{i=1}^{l} |\mathcal{C}_h^{Q,M_i}|$ nodes at l consecutive levels, extending at each level the leftmost node, generated at the previous level. Each two consecutive continuous parts are "connected" by a backtracking as we describe a bit later.

Let us call a tree node *interior* if its immediate successor(s) and the immediate predecessor also belong to this tree. The nodes which are interior for the subtree associated with a continuous part of $\mathcal{B}(h, Q)$ are called *intermediate nodes* for $\mathcal{B}(h, Q)$, and the rest of the nodes are called *non-intermediate*. A non-intermediate node h' of $\mathcal{B}(h, Q)$ which is a leaf of the $\mathcal{B}(h, Q)$-subtree is called a *renewal node* (by the convention, the root of T is also a renewal node); otherwise h' is called an *interruption node*. If h' is an interruption node, then upon backtracking to h' the extended branching $\mathcal{B}(h, Q)$ is resumed from this node (i.e., another continuous part of $\mathcal{B}(h, Q)$ is generated from h'). If h' is a renewal node, then either there occurs no backtracking to h' (in this case, h' is either the root of T, or the leftmost non-intermediate node of $\mathcal{B}(h, Q)$, or it is a leaf of T), or upon backtracking to h' a new extended branching starts from this node; in the latter case, the corresponding quick dominant set and the decomposition are determined.

We call a solution tree, in which at each renewal stage an extended branching by a quick dominant set is initiated and at each interruption stage the corresponding extended branching is resumed the *compressed solution tree* and denote it by Tc. The schedules, enumerated in Tc, are called *compressed*. If C_h^Q is a quick dominant set determined at a renewal stage, then instead of generating a subtree, consisting of $|\mathcal{C}_h^Q|$ direct successors of node h, a bigger subtree with

$\sum_{i=1}^{l} \prod_{j=1}^{i} |\mathcal{C}_h^{Q,M_j}|$ nodes is created in Tc at once. The following example gives an intuition on the relative sizes of Tc and Tq.

Assume that $\mathcal{C}_h^Q$ is a quick dominant set and $M_1, ..., M_l \in \mathcal{M}_k,\ M_1 = Q$, is an enumeration of all respective co-quick machines. The compressed subsets $\mathcal{C}_h^{Q,M_i}$, $i = 1, ..., l$, partition the conflict set $\mathcal{C}_h^Q$. We generate in total $|\mathcal{C}_h^Q|!$ nodes in Tq, and $(\prod_{i=1}^{l} |\mathcal{C}_h^{Q,M_i}|!)$ nodes in Tc for the operations of $\mathcal{C}_h^Q$. It is not difficult to see that the latter magnitude is drastically smaller than the former one, even for small values of l. For example, let $\mathcal{C}_h^Q = \{1, 2, 3\}$ be a quick dominant set with $d_{1Q} = 2,\ d_{1P} = 4,\ d_{1R} = 7,\ d_{2Q} = 5,\ d_{2P} = 3,\ d_{3P} = 6,\ d_{3Q} = 7,\ d_{3P} = 6,\ d_{3R} = 4$. For the sake of simplicity, assume that all operations and machines are released simultaneously. Q, P and R are co-quick machines and $\mathcal{C}_h^{Q,Q} = \{1\},\ \mathcal{C}_h^{Q,P} = \{2\}$ and $\mathcal{C}_h^{Q,R} = \{3\}$. In Figure 1(a) and 1(b), the corresponding fragments of Tq and Tc are illustrated.

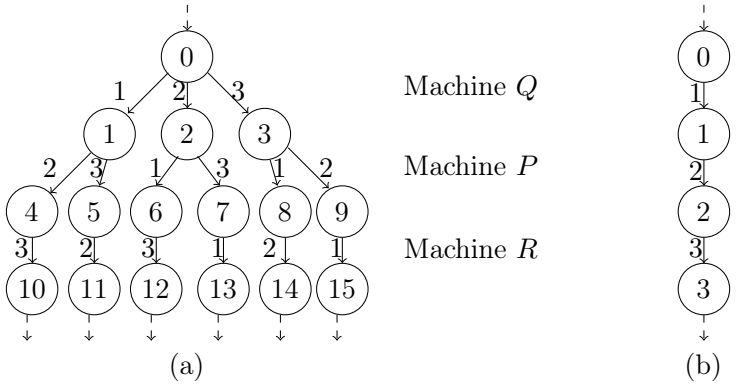


Figure 1: Corresponding fragments of (a) Tq and (b) Tc

Theorem 3 *There is a compressed optimal schedule in Tc.*

3.2.3 Formal Description

Below, in the formal description of our reduction algorithm, under a non-intermediate node we mean a non-intermediate node of the subtree of a continuous part of some extended branching. For each renewal stage (different from a leaf of Tc), the algorithm looks for a quick dominant set (Lemmas 2 and 3), constructs its compressed decomposition (Section 3.2) and starts the extended branching by this decomposition, i.e., it generates its first (leftmost) continuous part. Each time a new complete solution is generated, the backtracking to the leftmost non-intermediate node of the highest level (less than n) is performed (the latter node will be referred to as the *retreat* node). Whenever we backtrack, we either resume an earlier interrupted extended branching (the retreat node is an interruption node) or we initiate a new extended branching (the retreat node is a renewal node). succ(o) denotes the set of immediate successors of operation o.

1 COMPRESSION-ALGORITHM()

1: $h := 0$; *{initial settings}*
2: $R := \text{succ}(0)$; *{set the current set of the ready operations}*
3: ***while*** $R \neq \emptyset$ ***or*** *there exists a retreat node* ***do***
4: ***if*** $R \neq \emptyset$ ***then***
5: ***if*** *h is a renewal node* ***then***
6: *{form a quick dominant set and its compressed decomposition}*
7: $o :=$ *a ready operation with the minimal completion time at stage* h;
8: $Q :=$ *a quick machine for* o;
9: $\mathcal{C}_h^Q :=$ *the conflict set (include in it operation* o *and all ready operations, which conflict with* o *on* Q*)*;
10: $\mathcal{C}_h^{Q,M_i} :=$ *a compressed decomposition,* $i = 1, ..., l$, *of* $\mathcal{C}_h^Q$;
11: $\mathcal{B}(h, Q)$; *{call to generate the first continuous part of the extended branching}*
12: $h' :=$ *the retreat node;*
13: $SCHEDULE(h, h')$;
14: $h := h'$;
15: ***else***
16: *{*h *is an interruption node of some extended branching, say* $\mathcal{B}$*}*
17: $\mathcal{B}(h, Q)$; *{carry out the next continuous part of the extended branching* $\mathcal{B}$ *from node* h*}*
18: $h' :=$ *the retreat node;*
19: $SCHEDULE(h, h')$;
20: $h := h'$;
21: ***end if***
22: ***else***
23: $\sigma_h :=$ *is the current complete solution;{i.e., a complete solution is obtained}*
24: $\sigma_{opt} :=$ *is the best complete solution at this moment;*
25: ***if*** *there exists a retreat node* ***then***
26: $h' :=$ *the retreat node;*
27: $h :=$ *father*(h');
28: $SCHEDULE(h, h')$;
29: $h := h'$;
30: ***end if***
31: ***end if***
32: ***end while***
33: ***return*** σ_{opt};

2 SCHEDULE(h, h') *schedule each operation, associated with an edge on the path from node* h *to node* h'

1: ***for all*** *edge of the path from node* h *to node* h' *in* $T_{h'}$ ***do***
2: $o :=$ *operation associated with this edge;*
3: $M :=$ *machine associated with this edge;*
4: $R := R \cup \text{succ}(o)$; *{add to the current set of the ready operations the ones from* $\text{succ}(o)$*}*
5: $R := R - o$; *{delete operation* o *from the set* R*}*
6: ***for all*** $p \in \sigma_h$, *already scheduled on machine* M ***do***
7: $G_{h'} := G_h \cup (p, o)$;
8: $w(p, o) := d_{pM}$;
9: ***end for***
10: ***for all*** $p \in \text{succ}(o)$ ***do***
11: $w(o, p) := d_{oM}$;
12: ***end for***
13: ***end for***

Observation 1 *Let* ν *and* μ, *respectively, be an upper limit on the number of operations and machines, respectively in each group. The cost spent at each node of* Tc *is determined by Step 1c and is of the order* $O(\nu\sqrt{\nu}\mu)$.

4 Auxiliary Scheduling Problems

In this section, we define the auxiliary scheduling problems which are simpler to solve than MJSP and which have a close useful relationship with MJSP. In particular, these problems may be used for obtaining lower bounds for MJSP, as we show in the next section.

Remind that in a branch-and-bound scheme, if a lower bound $L(\sigma_h)$ of the partial solution σ_h is more than or equal to the makespan $|\sigma|$ of some already generated complete solution σ (a current upper bound), then all extensions of σ_h can be abandoned. Clearly, $L(\sigma_h)$ cannot be more than the makespan of the best potential extension of σ_h (otherwise we could loose this extension). At the same time, we try to make it as close as possible to this value: then the chances are larger that $L(\sigma_h) \geq |\sigma|$). Let $\sigma_h \in T$ and $o \in \mathcal{C}_h$ for an instance of MJSP. We would like to obtain a lower bound for an extension of σ_h with operation o scheduled on machine $Q \in \mathcal{M}_k$, $\sigma_h o^Q \in T$. A trivial lower bound is

$$L_T(\sigma_h o^Q) = |\sigma_h| + \mathrm{tail}_h(o),$$

where $|\sigma_h|$ is the makespan of σ_h, i.e., the critical path length in G_h. Note that the remained work determined by $\mathrm{tail}_h(o)$ reflects all the original precedence constraints and all the resource constraints have been resolved so far by stage h. So this bound ignores all yet unresolved potential conflicts, i.e., the processing times of yet unscheduled tasks.

Though it is easy and fast to obtain L_T, it is clear that we cannot get a good estimation of the desired optimal makespan by the complete ignorance of the potential contribution of the unscheduled tasks. A stronger lower bound would take into account a possible contribution of the latter tasks (this would obviously need additional computational efforts). Clearly, we cannot know in advance how yet unresolved conflicts will be resolved in an optimal schedule. But we can make some assumptions about this ("simulating" in advance some "future" resource constraints). However, we should be careful since we are not allowed to violate the condition $L(\sigma_h) \leq |\sigma'|$, σ' being an arbitrary complete extension of σ_h. Roughly speaking, we would like to have a lower estimation on how the future resource conflicts will be resolved; this will involve some optimal scheduling on parallel machines.

Now we derive an auxiliary multiprocessor scheduling problem which we use for our lower bounds. For JSP, the most commonly used is a one-machine relaxation (for example, see [1], [2], [6], [18], [21]): all resource constraints are relaxed (ignored) except the ones of one particular (not yet completely scheduled) machine, and the resulting one-machine problem with heads and tails, $1|r_i, q_i|C_{max}$ is then solved. A *bottleneck* machine is one which results in the maximal makespan among all yet unscheduled machines (intuitively, a bottleneck machine gives a maximal expected contribution in the makespan of extensions of σ_h). This approach can be generalized as follows. Basically, we relax the resource constraints on all machines except the ones from some (bottleneck) set of the machines $\mathcal{M}_k$.

To be specific, let at iteration h, $|\mathcal{O}_{kh}| \geq 2$, where $\mathcal{O}_{kh}$ is the subset of O_k consisting of the tasks not yet scheduled by stage h; i.e., we have yet unresolved resource constraints associated with the machines of $\mathcal{M}_k$. An operation $i \in \mathcal{O}_{kh}$ is characterized by its early starting (release) time $\mathrm{head}_h(i)$ and tail $\mathrm{tail}_h(i)$; that is, i cannot be started earlier than at time $\mathrm{head}_h(i)$, and once it is completed, it will take at least $\mathrm{tail}_h(i)$ time for all successors of i to be finished. Operation i can be scheduled on any of the machines of $\mathcal{M}_k$ and has a processing time d_{iP} on machine $P \in \mathcal{M}_k$. Each machine $P \in \mathcal{M}_k$ has its release time $R_h(P)$.

Observe that the operation tails and release times are derived from G_h (this ignores all resource constraints unresolved by stage h). Besides, the tails require no machine time, i.e., no time on any of the machines of $\mathcal{M}_k$. We are looking for an optimal (i.e., minimizing the makespan with tails) ordering of the operations of $\mathcal{O}_{kh}$ on the machines from $\mathcal{M}_k$ under the above stated conditions. Thus, for each stage h for the partial solution σ_h, the auxiliary problem of scheduling tasks with release times and tails on a group of parallel machines $\mathcal{M}_k$ with the objective to minimize the makespan has been obtained. We denote this auxiliary problem by $\mathcal{A}_{kh}$ and the respective optimal makespan by $|\mathcal{A}_{kh}|$.

Let μ_h be the set of indexes of all machine groups such that for each $k \in \mu_h$, $|\mathcal{O}_{kh}| \geq 2$. It is clear

that $|\mathcal{A}_{kh}|$, for any $k \in \mu_h$, is a lower bound for node h. We may find all $|\mu_k| \leq m$ lower bounds for node h and take the maximum thus finding a bottleneck machine group. Thus, instead of dealing with the problem $1|r_i, q_i|C_{max}$ in the case of JSP, now we deal with the problem $R|r_i, q_i|C_{max}$. Both problems are NP-hard, though there exist exponential algorithms with a good practical behavior for the first above problem, have been commonly used in one-machine relaxation based branch-and-bound algorithms for JSP (see, for example [21], [5] and [6]). Unfortunately, there are no known algorithms with a good practical performance for the problem $P|r_i, q_i|C_{max}$ (the version with identical machines) and so also for the problems $R|r_i, q_i|C_{max}$ and $Q|r_i, q_i|C_{max}$. In the following section, we suggest several ways to obtain lower bounds for these problems.

5 Lower Bounds

In this section, we describe some lower bounds for our problem and also some bounds which are not strict lower bounds. Some of these bounds are quite simple to obtain, basically, using a heuristic approach. Other bounds are more difficult to obtain, though, still in polynomial time (the solution of a linear programming is required for some of them).

Straightforward bounds. Carlier and Pinson [7] have suggested a lower bound for the problem $JP|prec|C_{max}$. They proposed an $O(n \log n + nm \log m)$ algorithm for the non-sequential version of the problem $P|r_i, q_i, prmt|C_{max}$ which is a tight lower estimation of the optimal makespan for the problem $P|r_i, q_i, prmt|C_{max}$. At the expense of weakening the bound, the solution of the above problem can be used as a lower bound for the version with unrelated machines as we describe below.

Let $d_o^{\min}$ be the minimal processing time of operation $o \in \mathcal{O}_k$, i.e.,

$$d_o^{\min} = \min\{d_{oM}, M \in \mathcal{M}_k\}.$$

We replace the unrelated machine group $\mathcal{M}_k$ by the identical machine group $\mathcal{M}_k'$, defined as follows: the number of machines in both groups is the same, and for each $o \in O_k$ and $M \in \mathcal{M}_k'$, $d_{oM} = d_o^{\min}$. It is clear that an optimal solution of the obtained instance of the problem $P|r_i, q_i, prmt|C_{max}$ with $\mathcal{M}_k'$ is no more than that of the corresponding instance of the problem $R|r_i, q_i, pmtn|C_{max}$ with $\mathcal{M}_k$. Hence, the former solution can be used for the calculation of a lower bound for the original problem. Obviously, the bound obtained in this way would be weak if the difference between the above two solutions is significant. It might be possible to find a better "approximation" with an identical machine group of the unrelated machine group $\mathcal{M}_k$, i.e., to increase d_{oM}, $o \in \mathcal{O}_k$, $M \in \mathcal{M}_k'$ (this could be a subject of further research).

For uniform machines, we can obtain a stronger lower bound by using the algorithm of Federgruen and Groenevelt [9] for the problem $Qm|r_i, q_i, pmtn|C_{max}$ with a time complexity of $O(tn^3)$ (here t is the number of machines with distinct speeds).

As to $JR|prmt|C_{max}$, the technique based on linear programming of Lawler and Labetoulle [19] yields a polynomial-time algorithm for $Rm|r_i, q_i, pmtn|C_{max}$. This is clearly a lower estimation of the optimal makespan for the problem $Rm|r_i, q_i|C_{max}$ which, in turn, provides a lower bound for the problem $JR|prmt|C_{max}$.

Alternative lower bounds. Now we describe alternative methods to obtain lower bounds. For the versions with identical and uniform machines our lower bounds are obtained in almost linear (in $|\mathcal{O}_{kh}|$ and $|\mathcal{M}_k|$) time. For the version with unrelated machines, we apply linear programming. We obtain a lower estimation, which is not a strict lower bound for the same version again in almost linear time. This bound can be used in approximation algorithms such as a beam search.

For simplifying the notations, let $a_i = \text{head}_h(i)$ and $q_i = \text{tail}_h(i)$ for $i \in \mathcal{O}_{kh}$, where $k \in \mu_h$. Let, further, d_i^S be the processing time of i in S (d_i^S may vary from schedule to schedule depending on the particular machine, to which i is assigned), t_i^S ($c_i^S = t_i^S + d_i^S$, respectively), be the starting (finishing, respectively) time of operation i in schedule S. We call $c_i^S + q_i$ the *full* completion time of operation i.

First we apply the "Greatest Tail Heuristic" (GTH) to the operations of $\mathcal{O}_{kh}$: iteratively, among all ready operations, we determine one with a longest tail and schedule it on a machine on which the minimal completion time of this operation is reached. We refer to such a machine as a *quick* one. Because of the space limitation, we omit a formal description of this heuristic.

The time complexity of this algorithm is $O(\mu n \log n)$, where $\mu = |\mathcal{M}_k|$. In the following, S denotes a greatest tail schedule obtained by the algorithm GTH for the operations of $\mathcal{O}_{kh}$. S, in general, consists of a number of *blocks*. Intuitively, a block is a maximal independent part in a schedule. More precisely, B is a maximal consecutive part in S (that is, a maximal sequence of the successively scheduled jobs on the adjacent machines), such that for each two successively scheduled tasks i and j, task j starts no later than task i finishes. Let $r \in \mathcal{O}_{kh}$ be the latest scheduled in S operation such that $c_r^S + q_r$ is equal to $|S|$ (clearly, there exists at least one such operation in S). If $t_r = a_r$, S is optimal (as task r is scheduled on its quick machine) and $|S| = |\mathcal{A}_{kh}|$ is the optimal makespan. If $t_r > a_r$, then r might potentially be completed earlier by rescheduling some operation(s), scheduled before r, after r. Next we will see how this works.

Let us call an operation $l \in S$, scheduled before r with $q_l < q_r$, an *emerging* operation in S, if l belongs to the same block as r. The set of operations scheduled in S between the latest scheduled emerging operation and operation r is called the *kernel*. Thus, any kernel operation has a tail, no less than q_r. We increase "artificially" the readiness time of some emerging operation l by setting $a_l := a_r$ and apply again algorithm GREATEST_TAIL. Then we will get a new greatest tail schedule, S_l, in which l is rescheduled after all operations of the kernel. We call the above rescheduling of task l its *application*. Once we apply l, we liberate space for the kernel operations (in particular, for operation r). These operations will be rescheduled earlier in the new obtained greatest tail schedule S_l. Hence, the makespan in S_l might be decreased (in comparison with that in S). Let us call the maximal magnitude, by which a kernel operation can be rescheduled earlier in this way, the *shifting value* of that operation. Note that it makes no sense to apply any non-emerging operation. For further details, we refer the reader to [32] and [33].

It can be proved that the shifting value of any kernel operation, including r, is strictly less than the maximal operation processing time d_{max}. Then the successive application of no more than $|\mathcal{M}_k|$ emerging operations is sufficient to construct a greatest tail schedule, say S', in which the first $|\mathcal{M}_k|$ kernel operations (all kernel operations if their number is less than $|\mathcal{M}_k|$) are antedated by the newly arisen gaps. Let C be the sequence of the kernel operations in S' (observe that C starts with the kernel operations in S', antedated by the newly arisen gaps). In S', an operation $i \in C$ either starts at time a_i or it starts right at the moment of completing another operation of C. Hence,

$$\min\{t_i^{S'} \mid i \in C\} = \min\{a_i \mid i \in C\}$$

is the minimal possible starting time for C.

Let C^S be the sequence in which the kernel operations were scheduled in S. Observe that, although C^S might be different from C, all the emerging operations, applied in S', have been initially scheduled before C^S in S. In S, the sequence C^S is started with a delay that is determined by the finishing times of the μ' emerging operations directly preceding the kernel operations in S. Suppose that, respecting this delay of C^S in S, the sequence C itself is optimal (i.e., it minimizes the maximal completion time of the kernel operations, subject to the release times of the $\mathcal{M}_k$

machines). Then, from the definition of C^S and r and the earlier made observation,

$$|S| - d_{max} = c_r^S + q_r^S - d_{max}$$

is a lower bound on the optimal schedule makespan. Note that its calculation takes $O(\mu n \log n)$ time. This bound, in general, is not a strict lower bound for the problem $JP|prec|C_{max}$ (as the sequence C^S is not optimal), though it can be successfully applied as a thorough estimation in approximate algorithms such as beam search (see for example [30]).

The above bound can be easily transferred to a strict lower bound for the versions with identical and uniform machines. In principle, we need to find a good lower estimation for an optimal sequence of the kernel operations. This task can be solved in almost linear time for both identical and uniform machines, while for unrelated machines we will apply (also polynomial) linear programming. We obtain a good lower estimation for the problem $Q|r_i|C_{max}$ (which itself is NP-hard) by solving its preemptive version $Q|r_i, pmtn|C_{max}$ in $O(n \ \log n + mn)$ time (see [23] and [17]). If we ignore the operation release times (this, in general, is possible since the optimal makespan without the release times is no more than that with the readiness times), we can apply an $O(n + m \ \log m)$ algorithm for the problem $Q|pmtn|C_{max}$ by Gonzalez and Sahni [12]. Similarly, we obtain a good lower estimation for the problem $R|r_i|C_{max}$ by solving its preemptive version by linear programming (see [19]).

The above estimations provide us with the earliest possible finishing time, c^*, of the kernel operations. Let $q = \min\{q_i,\ i \in C^S\}$ and d be the maximal processing time among all emerging operations in S. Then $L_1(C^S) = c^* + q - d$ is clearly a lower bound on the makespan of $\mathcal{A}_{kh}$. This bound can be further strengthened. Earlier we saw how the sequence C is obtained after the application of no more than $|\mathcal{M}_k|$ emerging operations (which were the latest scheduled ones in S). Denote this set of emerging operations by E and the set of all emerging operations in S by $\mathcal{E}$. In general, the emerging operations from $\mathcal{E} \setminus E$ can be applied instead of some emerging operations of E (note that the emerging operations from the latter set precede those from E in S). Indeed, if the emerging operations $l_1, ..., l_p$ are all released by time t, they will be successively scheduled in S till the moment when the earliest non-emerging operation gets ready. Thus we may have a choice, which emerging operations to apply. By choosing emerging operations from E, we guarantee that the sequence C will start without any delay; at the same time, the emerging operations of E, applied in S', having a "long enough" tail may obviously affect the resulted makespan (i.e., the maximal *full* job completion time.

This consideration makes it clear that, by taking into the account the actual tails and processing times of the rescheduled emerging operations, the earlier bound might be further improved. A simple solution might be as follows. Assume that on each machine from $\mathcal{M}_k$, an operation of C is scheduled (otherwise, as it is easily seen, there is no need in this additional estimation). At least one emerging operation should be rescheduled after C; hence, any $E' \subset \mathcal{E}$ will be fully completed no earlier than at time $L_2(E') = c' + d' + q'$, where c' is the minimal finishing time of the operations of C scheduled last on one of the machines of $\mathcal{M}_k$, $d' = \min\{d_{iP},\ i \in \mathcal{E},\ P \in \mathcal{M}_k\}$ and $q' = \min\{q_i,\ i \in \mathcal{E}\}$. Thus,

$$L_{kh} = \max\{L_1(C), L_2(E)\}$$

is a lower bound for $\mathcal{A}_{kh}$.

6 Computational Experiments

We have implemented our reduction algorithm in C++, using the IDE Qt Creator v.2.4.1 on a computer equipped with 8 GB of RAM, AMD Phenom (tm) II X6 1100T x6 Processor, and

operating system Ununtu 12.04 to 64-bit.

The implemented software has three different threads. The main thread takes care of the graphical part representing all data including the problem data, all the necessary intermediate results from the reduction algorithm available for the user and the interaction with the user (Figure 2 illustrates some of this data). Figures 3 and 4 illustrate the graphical representation used in the main thread of task graphs and solution trees. The second thread executes the reduction algorithm. The third auxiliary thread serves as an intermediate thread between the main thread and the second thread. In particular, whenever an intermediate result from the second thread is available, the third auxiliary thread informs the main thread about such an event. Then the main thread updates the corresponding graphical data.

The user may have two different modes for the interaction with the software. The debugger mode provides a step by step update from the reduction algorithm (concerning basic constructions such as dominant and conflict groups, compressed decomposition, scheduling the operations, the determination of the critical path, the current state of the solution tree, etc.). Alternatively, the user may only ask for updates on the total number of the so far generated feasible solutions and when a new complete solution, better than the earlier best one is created. The access to this data is provided by the third auxiliary thread, and it is optional.

We have based on a number of instances of JSP. In particular, the 3 classical instances with 6 jobs on 6 machines (mt06), 20 jobs on 5 machines (mt20) and 10 jobs on 10 machines (mt10) were taken from [10]. The other five JSP instances were taken from [20] with 10 jobs on 5 machines (la01-la05), and five more instances were taken from the same reference with 15 jobs on 5 machines (la06-la10). The last two instances are from [1] with 10 jobs on 10 machines (abz5 and abz6).

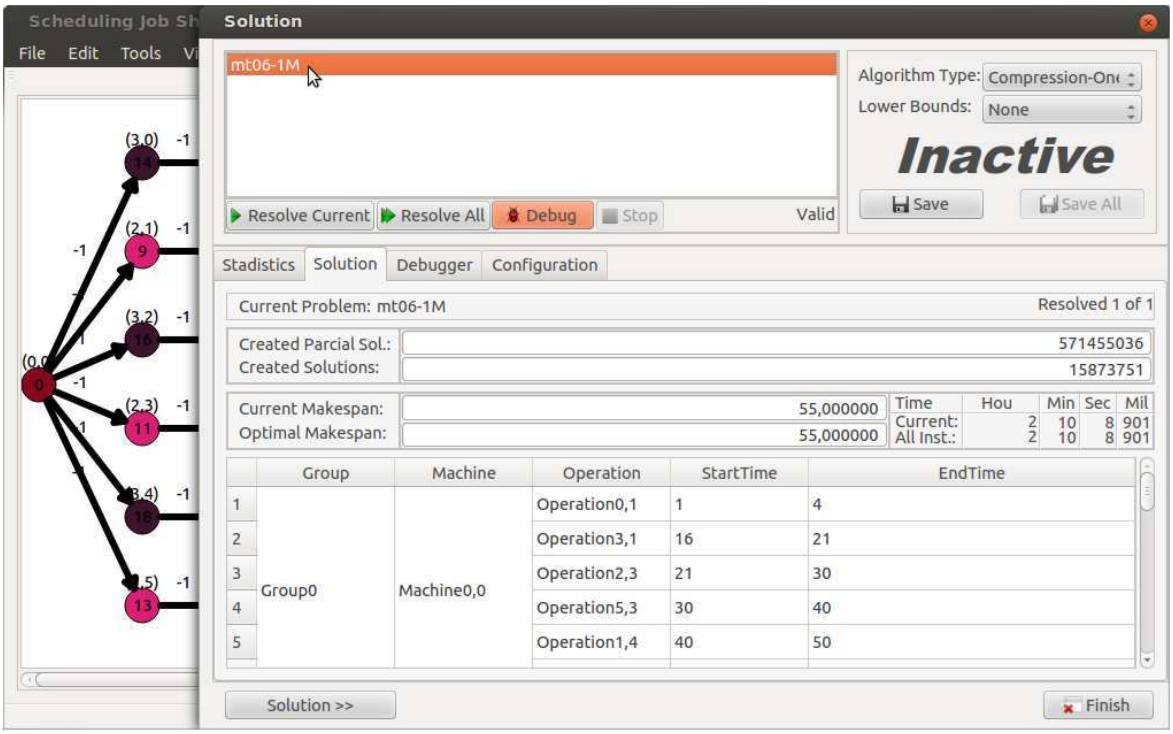
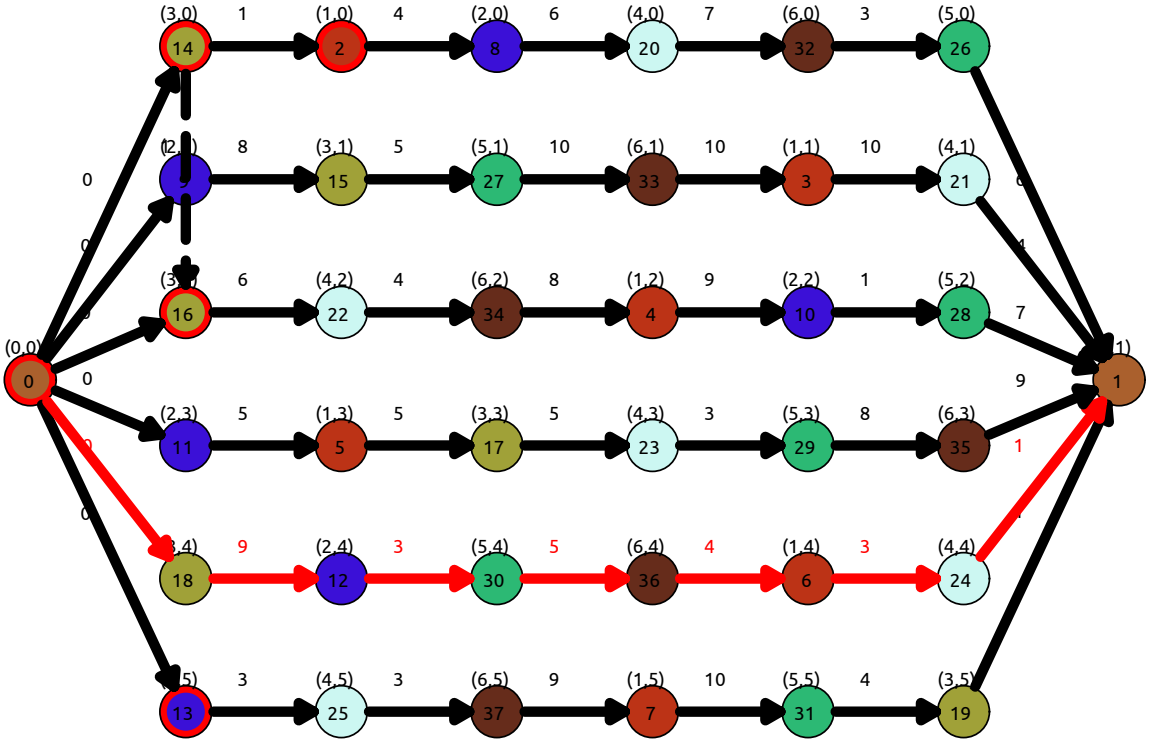


Figure 2: Graphical user interface

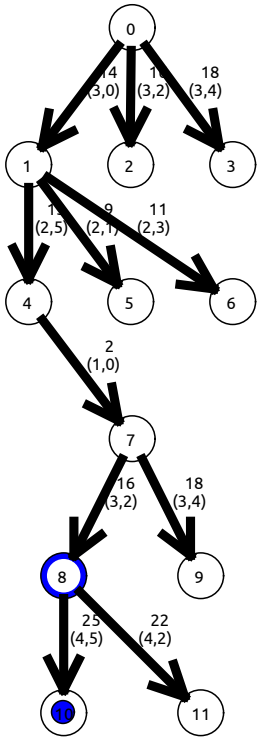


Precedence arcs - solid; Disjunctive arcs - dashed; Critical path is in red; Each color refers to a group of parallel machines; Operations marked in red are already scheduled.

Figure 3: The task graph for mt06

In all these JSP instances (which are also available in the OR-Library), every job has exactly one operation on each of the machines. Based on them, we have generated instances of MJSP. Each machine in the corresponding instance of JSP was replaced by a group of parallel machines. The number of parallel machines in every group is the same integer number. In each generated instance of MJSP, the original operation processing times are assigned to every first machine of each group. The operation processing times on the rest of the parallel machines of each group were generated randomly from the interval $[10, 100]$. Observe that our MJSP instance with only one machine in every group is the same as the corresponding JSP instance, and that a feasible solution to a MJSP instance with i machines in each group is also a feasible solution to an extended instance with $i + 1$ machines in each group.

We have used available regular PC-s with a restricted memory and CPU speed for testing our instances. Because of the limitation on the processor time, we have imposed a limit of 10^8 on the total number of the created solutions. Most of the instances were solved in less than 2 hours, a few relatively large instances with a small number of parallel processors per group were not solved optimally within our limit.



The stages h and h' from the description of the algorithm are marked in external and internal blue color.

Figure 4: A fragment of the compressed solution tree for mt06

As it can be seen from Table 1, we have tested MJSP instances corresponding to mt06 with up to 4 machines per group. In particular, our reduction algorithm was able to solve mt06 (with one machine per group) optimally (in less than 2 hours). The MJSP instances derived from la01-la04 were solved for 2, 3, 4 and 5 machines per group (the number of the created solutions for an instance with one machine per group has surpassed our limit and has required more than 48 hours). Our algorithm has solved optimally MJSP instances created from abz5 and abz6 for 3, 4, 5 and 6 machines per group. The MJSP instances derived from mt10 and mt20, respectively, were solved for 4, 5 and 6 machines, and 8, 9 and 10 machines per group, respectively.

As a practical proof of the earlier theoretic estimations from [31], for all the created MJSP instances, we have observed an exponential asymptotic decrease of the number of generated feasible solutions with an increase of the number of parallel machines per group. Figure 5 illustrates this important observation (notice that the scaling on the number of the created complete solutions is logarithmic in this figure). Moreover, our practical results have overcome our theoretical estimations. Recall that according to the theoretical estimation from [31], the number of the compressed schedules generated for an instance of MJSP was $\Psi = \prod_{k=1}^{m}[(|O_k|/|\mathcal{M}_k|)!]^{|\mathcal{M}_k|}$. As we illustrate in Table 1, the actual number of the generated compressed schedules turned out to be essentially less than the above magnitude, for all of the instances with large number of feasible solutions.

p	n	g	Ψ	μ	m	ν	C_{max}
mt06	6	6	15399682740640811135241	1	6	15873751	55
			244140625	2	12	1865676	51
			117649	3	18	202269	47
			729	4	24	158562	47
mt10	10	10	282475249	4	40	59163287	292
			282475249	5	50	20497	253
			59049	6	60	341	211
mt20	20	5	16807	8	40	56539895	135
			16807	9	45	15854040	125
			16807	10	50	1349739	98
			16807	11	55	102846	90
abz5	10	10	95367431640625	3	30	13746407	461
			282475249	4	40	262156	365
			282475249	5	50	6919	336
			59049	6	60	1135	291
abz6	10	10	95367431640625	3	30	6559490	451
			282475249	4	40	30670	366
			282475249	5	50	4884	355
			59049	6	60	282	321
la01	10	5	194839193667601	2	10	510468511	345
			9765625	3	15	17961150	234
			16807	4	20	77304	205
			16807	5	25	7249	192
la02	10	5	194839193667601	2	10	618596793	349
			9765625	3	15	194138	214
			16807	4	20	5208	192
			16807	5	25	65	167
la03	10	5	194839193667601	2	10	904394870	304
			9765625	3	15	3826177	192
			16807	4	20	42362	176
			16807	5	25	13215	164
la04	10	5	194839193667601	2	10	244483378	305
			9765625	3	15	1015814	221
			16807	4	20	6673	179
			16807	5	25	389	174
la05	10	5	9765625	3	15	2821538	225
			16807	4	20	82560	185
			16807	5	25	107956	176
			243	6	30	29931	145
la06	15	5	9765625	5	25	14591953	172
			16807	6	30	145889	151
			16807	7	35	681	151
			16807	8	40	206	151
la07	15	5	9765625	5	25	7525885	172
			16807	6	30	312414	168
			16807	7	35	47662	156
			16807	8	40	56469	156
la08	15	5	9765625	5	25	175925311	198
			16807	6	30	18492889	169
			16807	7	35	670404	169
			16807	8	40	538148	141
la09	15	5	9765625	5	25	4039015	203
			16807	6	30	706916	186
			16807	7	35	90341	186
			16807	8	40	3732	186
la10	15	5	9765625	5	25	108813452	196
			16807	6	30	2230028	175
			16807	7	35	43971	143
			16807	8	40	36021	143

p: Problem instance; μ: Number of machines per group; n: Number of jobs; m: Total number of machines; g: Number of groups of parallel machines; ν : Number of complete solutions; C_{max}: makespan value; Ψ Theoretical estimation from [31].

Table 1: Experimental results

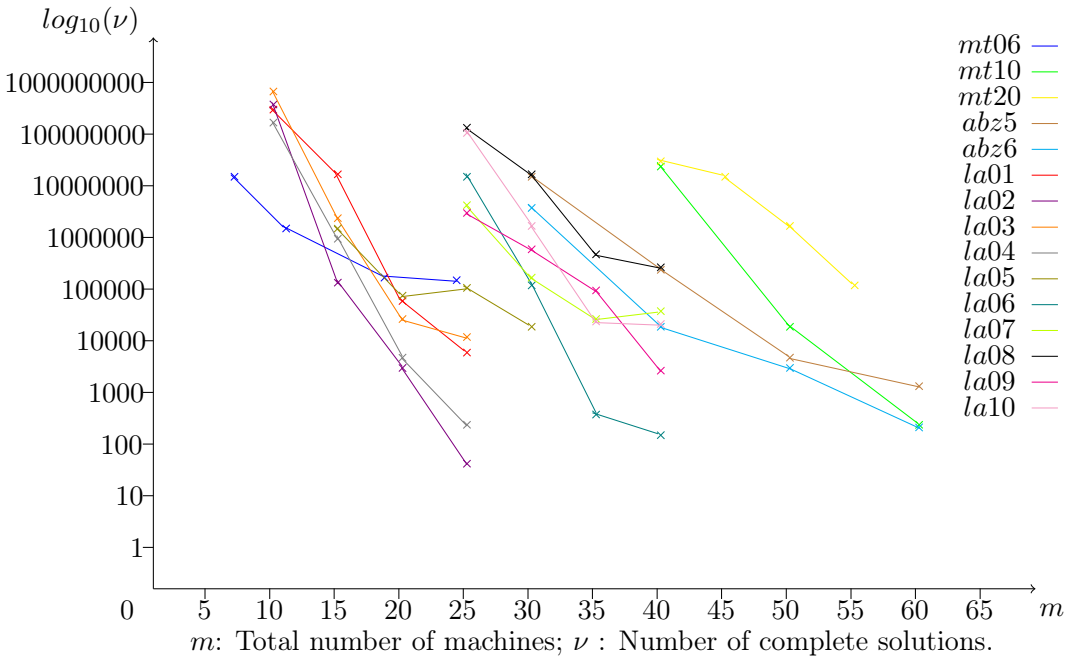


Figure 5: Experimental data

7 Concluding Remarks

We see the following basic directions for the further relevant research. First, the proposed lower bounds might be tested independently and compared with each other, both in efficiency and in the required computational efforts. This may yield an implicit enumeration algorithm for MJSP, more efficient than our reduction algorithm.

At the same time, a further theoretical improvement of some of the proposed bounds might be possible. In this connection, we have to look for alternative, more efficient algorithms for the problems $R|r_i, q_i|C_{max}$, $Q|r_i, q_i|C_{max}$ (or their preemptive versions), or even for the problem $R||C_{max}$, $Q||C_{max}$ (or their preemptive versions).

Our reduction algorithm and the proposed bounds may serve as a basis for sophisticated heuristic algorithms. In particular, the reduced solution tree constructed by our algorithm may well serve for a filtered beam search algorithm, using our non-strict bounds.

The presented framework can be aggregated by an additional graph-completion mechanism for taking into account transportation and setup times.

References

[1] Adams, J., Balas, E. and Zawack, D., 1988. The shifting bottleneck procedure for job shop scheduling. *Management Science*, 34, 391 – 401.

[2] Blazewicz, J., Cellary, W., Slowinski, R. and Weglarz, J. 1986. Scheduling under resource constraints - Deterministic models. *Annals of Operations Research*, 7, 1986.

[3] Brucker, P., Jurisch, B. and Krämer, A., 1997. Complexity of scheduling problems with multi-purpose machines. *Annals of Operations Research*, 70, 57 – 73.

[4] Brucker, B. and Schlie, R., 1990. Job shop scheduling with multi-purpose machines. *Computing*, 45, 369–375.

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

[6] Carlier, J. and Pinson, E., 1989. An algorithm for solving job shop problem. *Management Science*, 35, 164–176.

[7] Carlier, J. and Pinson, E., 1998. Jackson's pseudo preemptive schedule for the $Pm/r_i, q_i/C_{max}$ problem. *Annals of Operations Research*, 83, 41–58.

[8] Dauzére-Pérés, S. and Paulli, J., 1997. An integrated approach for modeling and solving the general multiprocessor job shop scheduling problem with tabu search. *Annals of Operations Research*, 70, 281–306.

[9] Federgruen, A. and Groenevelt, H., 1986. Preemptive scheduling of uniform machines by ordinary network flow techniques. *Management Science*, 32, 341–349.

[10] Fisher, H. and Thompson, G.L., 1963. Probabilistic learning combinations of local job-shop scheduling rules, in Muth, J.F. and Thompson, G.L. (eds.): *Industrial Scheduling*, 225 – 251.

[11] Giffler, B. and Thompson, G.L., 1960. Algorithm for solving production scheduling problems. *Operations Research*, 8, 487 – 503.

[12] Gonzalez, T. and Sahni, S., 1978. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25, 92–101.

[13] Herroelen, W.S. and Demeulemeester, E.L., 1997. Recent advances in branch-and-bound procedures for resource-constrained project scheduling problems. *Scheduling Theory and its Applications P. Chrétienne et al. (eds.), John Wiley & Sons*, 25, 259–276.

[14] Hopcroft, J.E. and Karp, R.M., 1973. A $n^{5/2}$ algorithm for maximum matching in bipartite graphs. *SIAM J. Computing*, 2, 225–231.

[15] Ivens, P. and Lambrecht, M., 1996. Extending the shifting bottleneck procedure to real-life applications. *European J. of Operational Research*, 90, 252–268.

[16] Jurisch, B., 1995. Lower bounds for job-shop scheduling problem on multi-purpose machines. *Discrete Applied Mathematics*, 58, 145–156.

[17] Labetoulle, J., Lawler, E.L., Lenstra J.K. and Rinnooy Kan, A.H.G., 1984. Preemptive scheduling of uniform machines subject to release dates. *Pulleyblank*, 25, 245–261.

[18] Lageweg, B.J., Lenstra, J.K. and Rinnooy Kan, A.H.G., 1977. Job shop scheduling by implicit enumeration. *Management Science*, 24, 441–450.

[19] Lawler, E.L. and Labetoulle, J., 1978. On preemptive scheduling of unrelated parallel processors by linear programming. *J. of the ACM*, 25, 612–619.

[20] Lawrence, S., 1984. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (Supplement), *Graduate School of Industrial Administration*, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

[21] McMahon, G.B. and Florian, M., 1975. On scheduling with ready times and due dates to minimize maximum lateness. *Operations Research*, 23, 475 – 482.

[22] Ow, P.S. and Morton, T.E., 1988. Filtered beam search in scheduling. *International J. of Production Research*, 26, 35–62.

[23] Sahni, S. and Cho, Y., 1979. Nearly on-line scheduling of a uniform processor system with release times *SIAM J. Comput.*, 8, 275 – 285.

[24] Schutten, J.M.J., 1998. Practical job shop scheduling. *Annals of Operations Research*, 83, 161 – 177.

[25] Shmoys, D.B., Stein, C. and Wein, J., 1994. Improved approximation algorithms for shop scheduling problems. *SIAM J. on Computing*, 23, 617 – 632.

[26] Sotskov, Y.N., Tautenhahn, T. and Werner, F., 1999. On the application of insertion techniques for job shop problems with setup times. *RAIRO Rech. Oper.*, 33, 209 – 245.

[27] Tanaev, V.S., Sotskov, Y.N. and Strusevich, V.A., 1994. Scheduling Theory: Multi-Stage Systems *Springer*, 1 –420.

[28] Vakhania, N., 1991. Algorithms for solving generalized job shop scheduling problems with the use of reduced solution trees. *Ph.D. Thesis, Computing Center, Russian Academy of Sciences, Moscow, Russian*, 1991.

[29] Vakhania, N., 1995. Assignment of jobs to parallel computers of different throughput. *Automation and Remote Control*, 56, 280 – 286.

[30] Vakhania, N., 2000. Global and local search for scheduling job shop with parallel machines. *Lecture Notes in Artificial Intelligence (IBERAMIA-SBIA 2000)*, 1952, 63 – 75.

[31] Vakhania, N. and Shchepin, E., 2002. Concurrent operations can be parallelized in scheduling multiprocessor job shop. *Journal of Scheduling*, 5, 227 – 245.

[32] Vakhania, N., 2002. Scheduling equal-length jobs with delivery times on identical processors. *Int. J. Computer Math.*, 82.

[33] Vakhania, N., 2003. A better algorithm for sequencing with release and delivery times on identical processors. *Journal of Algorithms*, 48, 273–293.

[34] Werner; F. and Winkler, A., 1995. Insertion techniques for the heuristic solution of the job shop problem. *Discrete Applied Mathematics*, 58, 191 – 211.