Search on the enumeration tree in the multiprocessor job-shop problem

Lester Carballo [*] **Alexander A. Lazarev,** [**]
Nodari Vakhania [***] **Frank Werner** [****]

[*] *Facultad de Ciencias, UAEM, Av. Universidad 1001, Cuernavaca 62210, Morelos, Mexico, (e-mail: lestcape@gmail.com)*
[**] *Institute of Control Sciences of the Russian Academy of Sciences, Profsoyuznaya st. 65, 117997 Moscow, Russia, (e-mail: jobmath@mail.ru)*
[***] *Facultad de Ciencias, UAEM, Av. Universidad 1001, Cuernavaca 62210, Morelos, Mexico, (Tel: +52 777 329 70 20; e-mail: nodari@uaem.mx).*
[****] *Fakultät für Mathematik, Otto-von-Guericke-Universität, Magdeburg, 39106 Germany (Tel: +49 391 6712025; email: frank.werner@mathematik.uni-magdeburg.de)*

Abstract: We present an approach based on a two-stage filtration of the set of feasible solutions for the multiprocessor job-shop scheduling problem. On the first stage we use extensive dominance relations, whereas on the second stage we use lower bounds. We show that several lower bounds can efficiently be obtained and implemented.

Keywords: job shop scheduling, branch-and-bound, algorithm, solution tree, dominance relations

1. INTRODUCTION

The job-shop scheduling problem (JSP) 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 the 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 in a job-shop problem). We shall refer to this extension of JSP as the multiprocessor job-shop scheduling problem with unrelated machines (MJSP for short)

JSP is a strongly NP-hard problem with one of the worst practical behavior. 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.

Although the feasible solution space of MJSP grows essentially compared to JSP, it has turned out that it is possible to reduce its solution space to a subspace containing an optimal solution, drastically smaller than even the solution space of a corresponding instance of JSP. Moreover, as we show, several lower bounds for MJSP might be efficiently obtained. In this way, we further reduce the solution space. 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 (see Vakhania and Shchepin (2002)).

We may formulate the MJSP as follows. We have the 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., a machine speed is operation-dependent; that is, 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 with 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 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 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. (1977) for the 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 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 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 (1995) for JSP or Sotskov et al. (1999) for JSP with setup tiomes) 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 such a rule 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. One of the earliest published articles mentioning about a generalization of JSP is that of Giffler and Thompson (1960). 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 Tanaev et al. (1994) solution methods for JSP and other problems related with MJSP are described. An extension of JSP with general multi-purpose machines was studied by Brucker and Schlie (1966), Brucker et al. (1997), Vakhania (1995), and a lower bound for the special case of this problem when an operation processing time is a constant (i.e. machine-independent) was suggested by Jurisch (1995). Shmoys et al. (1994) have proposed a polynomial approximation randomized algorithm for 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 (1997) have proposed a tabu search algorithm. Vakhania (2000) has suggested a version of a beam search algorithm for the generalized problem. Ivens and Lambrecht (1996), Schutten (1998) study different extensions of JSP including extensions with setup and transportation times. Vakhania and Shchepin (2002) 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. Surprisingly, with a probability of almost 1, the number of the 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.

In this paper, we strengthen the above reduction method with a number of lower bounds, giving the rise to different exact (branch-and-bound) and approximation algorithms. To the best of our knowledge, no lower bounds for the problem have been suggested earlier. It turned out that a simple $O(n \log n)$ time heuristic for the derived auxiliary multiprocessor scheduling problem serves well our bounds. We also give some progress in our preliminary computational experiments.

In the next section, we give some preliminaries and a general scheme for representing the created solutions. In Section 3, we define auxiliary scheduling problems used to obtain our bounds, presented in Section 4. Section 5 contains a brief description of the ongoing computational experiments.

2. BASIC FRAMEWORK

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 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 $\mathcal{C}_h$ (we refer the reader to Vakhania and Shchepin (2002) for the further details regarding this and some following notions in this section). By branching in T on 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 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 shall avoid such a complete enumeration by selecting a *single* processor from $\mathcal{M}_k$ at stage h, as specified a bit later.

Thus, there will be generated $|\mathcal{C}_h|$ extensions of the current partial schedule σ_h in our tree T. 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 $P \in \mathcal{M}_k$. Note that σ_h is identified by the path from the root to node h in T, and 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 already mentioned, 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. On each stage of branching, the branching by some operations is postponed whenever this is possible. Each quick set is partitioned into the 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 on 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 $G_0 = (X, E_0)$ with the root of T. 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 its successor schedule. We call it the *early starting time* or the *head* of operation o by stage h and denote 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 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. AUXILIARY SCHEDULING PROBLEMS

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 more are the chances 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

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

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 $\text{tail}_h(o)$ 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 auxiliary multiprocessor scheduling problem which we use for our lower bounds. For JSP, most commonly is used a one-machine relaxation (for example, see Adams et al. (1988), Blazewicz et al. (1986), Carlier and Pinson (1976), Lageweg et al. (1977), McMahon and Florian (1975)): all resource constraints are relaxed (ignored) except the ones of a one particular (not yet completely scheduled) machine, and the resulted one-machine problem with heads and tails, $1|r_i, q_i|C_{max}$ is then solved. A *bottleneck* machine is a one which results 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 $\text{head}_h(i)$ and tail $\text{tail}_h(i)$; that is, i cannot be started earlier than at time $\text{head}_h(i)$, and once it is completed, it will take at least $\text{tail}_h(i)$ 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 unresolved by stage h resource constraints). Besides, the tails require no machine time, i.e., 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 $1|r_i, q_i|C_{max}$ in case of JSP, now we deal with 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 McMahon and Florian (1975), Carlier (1982) and Carlier and Pinson (1976)). Unfortunately, there are no known algorithms with good practical performance for $P|r_i, q_i|C_{max}$ (the version with identical machines) and so also for 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.

4. LOWER BOUNDS

Straightforward bounds. Carlier and Pinson (1998) have suggested a lower bound for $JP|prec|C_{max}$. They proposed an $O(n \log n + nm \log m)$ algorithm for the non-sequential version of problem $P|r_i, q_i, prmt|C_{max}$ which is a tight lower estimation of the optimal makespan for 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 $\mathcal{M}_k$ with 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 problem $P|r_i, q_i, prmt|C_{max}$ with $\mathcal{M}_k^{'}$ is no more than that of the corresponding instance of 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 the subject of a further research).

For uniform machines, we can obtain a stronger lower bound by using the algorithm of Federgruen and Groenevelt (1986) 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 (1978) 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 problem $Rm|r_i, q_i|C_{max}$ which, in turn, provides a lower bound for 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 an 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 to such a machine as a *quick*. 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 $c_r^S + q_r$ 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 potentially might 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 kernel. We call the above rescheduling of task l its *application*. Once we apply l, we liberate space for 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 in this way a kernel operation can be rescheduled earlier, 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 Vakhania (2002) and Vakhania (2003).

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 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 completion of 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 applied in S' emerging operations 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 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 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 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 Vakhania (2000)).

The above bound can be easily transferred to a strict lower bound for the versions with identical and uniform machines. In principal, we need to find a good lower estimation for an optimal sequence of 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 estimations 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 Sahni and Cho (1979) and Labetoulle et al. (1984)). 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 $Q|pmtn|C_{max}$ by Gonzalez and Sahni (1978). Similarly, we obtain a good lower estimation for problem $R|r_i|C_{max}$ by solving its preemptive version by linear programming (see Lawler and Labetoulle (1978)).

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, emerging operations from $\mathcal{E} \setminus E$ can be applied instead of some emerging operations of E (note that emerging operations from the latter set precede those from E in S). Indeed, if 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 rescheduled after C emerging operations of E having "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 $\mathcal{A}_{kh}$.

5. ON SOME COMPUTATIONAL EXPERIMENTS

We have implemented our basic reduction algorithm in C + +, using the IDE Qt Creator v.2.2.1 on a computer equipped with 8 GB of RAM, AMD Phenom (tm) II X6 1100T x6 Processor, and operating system Ununtu 11.10 to 64-bit.

The generated code was tested for about 100 randomly generated problem instances with moderate sizes (up to 20 operations on 5 groups of parallel machines). We have used a topological order of the nodes in the dependency graph G_h to avoid the creation of cycles during the computation of the critical path (the makespan of each created solution). The reduction of the solution space was between 15% and 25 % compared to the number of active solutions for these small instances without using lower bounds. We expect a more essential reduction for larger instances when lower bounds will also be incorporated into the generated code.

REFERENCES

J. Adams, E. Balas and D. Zawack. The Shifting Bottleneck Procedure for Job Shop Scheduling. *Management Science* 34:391-401, 1988.

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

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

P. Brucker and R. Schlie. Job shop scheduling with multipurpose machines. *Computing*, 45:369-375, 1990.

J. Carlier. The one-machine sequencing problem. *European J. of Operational Research*, 11:42-47, 1982.

J. Carlier and E. Pinson. An Algorithm for Solving Job Shop Problem. *Management Science*, 35:164-176, 1989.

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

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

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

T. Gonzalez and S. Sahni. Preemptive scheduling of uniform processor systems. *Journal of the ACM*, 25: 92-101, 1978.

W.S. Herroelen and E.L. Demeulemeester. 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, 1997.

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

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

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

B.J. Lageweg, J.K. Lenstra and A.H.G. Rinnooy Kan. Job Shop Scheduling by Implicit Enumeration. *Management Science*, 24:441-450, 1977.

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

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

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

B. Giffer and G.L. Thompson Algorithm for Solving Production Scheduling Problems. *Operations Research*, 8:487-503, 1960.

P.S. Ow and T.E. Morton. Filtered beam search in scheduling. *Int. J. Prod. Research*, 26:35-62, 1988.

S. Sahni and Y. Cho. Filtered beam search in scheduling. *Nearly on-line scheduling of a uniform processor system with release times*, 8:275-285, 1979.

J.M.J. Schutten. Practical job shop scheduling. *Annals of Operations Research*, 83:161-177, 1998.

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

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

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

N. Vakhania. 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.

N. Vakhania. Assignment of jobs to parallel computers of different throughput. *Automation and Remote Control*, 56:280-286, 1995.

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

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

N. Vakhania. Scheduling equal-length jobs with delivery times on identical processors. *Int. J. Comp. Math.*, 82: 2002.

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

F. Werner and A. Winkler. Insertion techniques for the heuristic solution of the job shop problem. *Discrete Applied Mathematics*, 58:191 - 211.