A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem

Paul Martin * and David B. Shmoys **

School of Operations Research and Industrial Engineering,
Cornell University, Ithaca, NY 14853.

Abstract. From a computational point of view, the job-shop scheduling problem is one of the most notoriously intractable $\mathcal{NP}$-hard optimization problems. In spite of a great deal of substantive research, there are instances of even quite modest size for which it is beyond our current understanding to solve to optimality. We propose several new lower bounding procedures for this problem, and show how to incorporate them into a branch-and-bound procedure. Unlike almost all of the work done on this problem in the past thirty years, our enumerative procedure is not based on the disjunctive graph formulation, but is rather a time-oriented branching scheme. We show that our approach can solve most of the standard benchmark instances, and obtains the best known lower bounds on each.

1 Introduction

In the *job-shop scheduling problem* we are given a set of n jobs, $\mathcal{J}$, a set of m machines, $\mathcal{M}$, and a set of operations, $\mathcal{O}$. Each job consists of a chain of operations, let $\mathcal{O}_j$ be the chain of operations required by job j, whereas $\mathcal{O}^i$ is the set of operations that must be processed on machine i. Each operation k requires a specified amount of processing, p_k, on a specified machine $M(k)$. A schedule is an assignment of starting times to the operations such that each operation runs without interruption, no operation starts before the completion time of its predecessor in its job and such that each machine processes at most one operation at any moment in time. The goal is to find the schedule which minimizes the *makespan*, the time at which the last operation completes.

In this paper we focus on the special case in which each job has exactly one operation on each machine. Therefore, $|\mathcal{O}_j| = m$ for each $j \in \mathcal{J}$. The main ideas of this paper are easily extended to the general case. The *feasibility problem* is, given a time T, to determine whether there exists a schedule that completes by time T. Throughout this paper we consider the feasibility problem and use a bisection search to reduce the optimization problem to the feasibility problem.

* Research supported in part by the NEC Research Institute and in part by NSF grant CCR-9307391.

** Research supported in part by NSF grant CCR-9307391 and DMS-9505155 and ONR grant N00014-96-1-0050O.

The job-shop scheduling problem is $\mathcal{NP}$-hard [12] and has proven to be very difficult even for relatively small instances. An instance with 10 jobs and 10 machines posed in a 1963 book by Muth & Thompson [10] remained unsolved until Carlier & Pinson [7] finally solved it in 1986, and there is a 20 job and 10 machine instance of Lawrence [15] that remains unsolved despite a great amount of effort that has been devoted to improving optimization codes for this problem.

The most effective optimization techniques to date have been branch-and-bound algorithms based on the disjunctive graph model for the job-shop scheduling problem. The decision variables in this formulation indicate the order in which operations are processed on each machine. Once these variables are set, the time at which each operation starts processing can be easily computed. Algorithms of this type have been proposed by Carlier & Pinson [7], Brucker, Jurisch & Sievers [6] and Applegate & Cook [2].

Although solving job-shop scheduling problems to optimality is difficult, recently there has been progress developing heuristics that find good schedules. Adams, Balas & Zawack [1] proposed the *shifting bottleneck procedure*, which uses a primitive form of iterated local search to produce substantially better schedules than were previously computed. Currently, all of the best known algorithms use iterated local search. The current champions are a *taboo search* algorithm by Nowicki & Smutnicki [16] and a technique called *reiterated guided local search* by Balas & Vazacopoulos [4]. Vaessens, Aarts & Lenstra [21] provide an excellent survey of recent developments in this area.

There has been less progress in finding good lower bounds for the job-shop scheduling problem. The *job bound* is the maximum total processing required for the operations on a single job. Similarly, the maximum amount of processing on any one machine gives the *machine bound*. We can improve upon the machine bound by considering the *one-machine subproblem* obtained by relaxing the capacity constraint on all machines except one. This problem is the one-machine scheduling problem with heads and tails for each operations: $1 \mid r_j \mid L_{\max}$ in the notation of [13]. Here each operation $k \in \mathcal{O}^i$ has a head r_k, equal to the total processing requirement of all operations which precede k in its job, a tail q_k, the analogous sum for operations which follow k, and a body p_k, the amount of processing required by k. Each operation k may not begin processing before r_k and must be processed for p_k time units; if C_k denotes the time at which k completes processing, the *length* of the schedule is $\max_{k \in \mathcal{O}^i}\{C_k + q_k\}$. The *one-machine bound* of Bratley, Florian, & Robillard [5] is the the maximum over all machines $i \in \mathcal{M}$ of the bound for the one-machine subproblem for i. This subproblem is $\mathcal{NP}$-hard, although in practice it is not difficult to solve. The *preemptive one-machine subproblem* is the relaxation of the one-machine subproblem obtained by relaxing the constraint that each operation must be processed without interruption. Jackson [14] gives an $O(n \log n)$ algorithm for computing the optimal schedule for this subproblem. Carlier & Pinson [8] give an algorithm that strengthens the preemptive one-machine bound but maintains polynomial computability. Their algorithm updates the heads and tails of operations to account for the fact that we are looking for a non-preemptive schedule.

In Section 3 we show that their updates can be seen as adjusting each operation's head (or tail) to the earliest time that the operation can start such that it processes continuously and such that there is a preemptive schedule for the remaining operations on that machine.

Fisher, Lageweg, Lenstra & Rinnooy Kan [11] examine surrogate duality while Balas [3] and Applegate & Cook [2] attempt to find bounds by examining the polyhedral structure of the problem. Neither approach has proven to be practical so far. The strongest computational results of Applegate & Cook [2] use only combinatorial bounds in their branch-and-bound algorithm.

In this paper we present new lower bounding techniques based on a time-oriented approach to job-shop scheduling. Recently there has been significant success applying time-oriented approaches to other scheduling problems [19] [22].

The first approach we propose is based upon the *packing formulation*, a time-indexed integer programming formulation of the job-shop scheduling problem. Here we attempt to pack schedules for each job so that they do not overlap on any machine. The bound we obtain is based on the *fractional packing formulation* which is the linear relaxation of the packing formulation. This approach was motivated by the packing approximation algorithm of Plotkin, Shmoys & Tardos [17] along with an application to scheduling given by Stein [20]. This bound gives stronger bounds than those obtained by other linear programming relaxations but appears to be too time consuming.

To improve on the packing bound we attempt to restrict the time in which operations are allowed to start processing. We call this interval a *processing window*. The Carlier-Pinson head and tail modification algorithm can be used in an iterated fashion to reduce each operation's processing window. In addition to reducing processing windows for use with the fractional packing formulation, the iterated Carlier-Pinson can itself be a bound since if the feasible processing window is empty then we have solved the feasibility problem: no feasible schedule of length T exists. Iterated Carlier-Pinson gives results that are approximately the same as those obtained from the fractional packing formulation and it obtains these bounds in only a fraction of the time. Combining the two techniques provides only a slightly improved bound.

In order to reduce the size of the processing windows we introduce a technique called shaving. Here we suppose that an operation starts at one end of its processing window and then try to determine whether we can prove that there is no feasible schedule satisfying this assumption. If we can prove this, then we can conclude that the operation cannot start at that end of its processing window and we can therefore reduce its window. This technique depends on our ability to determine that there is no schedule after restricting the processing window of one operation. When we use the preemptive one-machine relaxation to determine that there is no schedule we call it *one-machine shaving*. We prove that this is equivalent to iterated Carlier-Pinson. When we use the non-preemptive one-machine relaxation we call it *exact one-machine shaving* and we show that this is only a slightly better bound then iterated Carlier-Pinson but significantly more costly to compute. If we use iterated Carlier-Pinson to show that there is no schedule we call it *C-P shaving*. This bound is significantly better than iterated Carlier-Pinson but requires more time to compute. Finally if we use C-P shaving to show that there is no schedule, then we call it *double shaving*. This bound is excellent but takes a substantial amount of time to compute, much too long to be practical in most branch-and-bound schemes.

Traditional job-shop branch-and-bound schemes are not particularly well-suited to a time-oriented lower bound. We give two new schemes designed for use with a processing window approach. The first is a simplistic approach which starts at time zero and either assigns an available operation to start immediately or delays it. This approach works well only if through the use of one of the shaving techniques we have been able to greatly reduce the processing windows. The second technique works by examining a machine over a time period where we know that there is very little idle time. We can then branch on which operation goes first in this time period. This technique is very effective on a wide range of problems. It is particularly effective when the one-machine bound is tight, a situation in which many other algorithms seem to perform poorly.

Using these techniques we were able to improve upon the best known lower bound for several of the standard benchmark problems, and in some cases we establish optimality. We have tested all of our algorithms on the standard benchmark problems. We have used all the problems for which Vaessens, Aarts & Lenstra [21] give recent local search results as well as three larger problems from Adams, Balas & Zawack [1]. All computations were performed on a 90MHz Pentium computer. Table 1 gives the bounds that we were able to compute along with the amount of time (in seconds) that it took to compute them.

One important advantage of our time-oriented approach is that we can trivially extend it to incorporate additional constraints, such as release dates and deadlines for jobs, as well as internal release dates and deadlines for operations. In fact, one would expect that adding these constraints would improve the performance of our algorithms.

2 A Packing Formulation

In spite of a great deal of effort, the disjunctive integer programming formulation of the job-shop problem appears to be of little assistance in solving instances of even moderate size; furthermore, its natural linear programming relaxation has been shown to give very poor lower bounds for the problem. We propose a new linear programming formulation which gives better bounds.

A *job-schedule* is an assignment of starting times to the operations of one job such that no operation starts before its predecessor completes and such that all operations of that job complete by time T. The job-shop feasibility problem can then be restated: does there exist a set of job-schedules, one for each job, such that no two job-schedules require the same machine at the same time? This gives rise to the following *packing formulation*, where we attempt to pack job-schedules so as to minimize the maximum number of jobs concurrently assigned to some machine.

For each job $j \in \mathcal{J}$, let $\mathcal{F}_j$ denote the set of all job-schedules, and let $x_{j\sigma}$ be a 0-1 variable that indicates if j is scheduled according to job-schedule $\sigma \in \mathcal{F}_j$. Let $\gamma(\sigma, i, t)$ indicate whether job-schedule σ requires machine i at time t and let λ be the maximum number of jobs concurrently on any machine. Then the integer programming formulation is to minimize λ subject to

$$\sum_{\sigma \in \mathcal{F}_j} x_{j\sigma} = 1, \quad \text{for all } j \in \mathcal{J}, \tag{1}$$

$$\sum_{j \in \mathcal{J}} \sum_{\sigma \in \mathcal{F}_j} \gamma(\sigma, i, t) x_{j\sigma} \leq \lambda, \quad \text{for all } i \in \mathcal{M}, t = 1, \ldots, T, \tag{2}$$

$$x_{j\sigma} \in \{0, 1\}, \text{for all } j \in \mathcal{J}, \sigma \in \mathcal{F}_j. \tag{3}$$

There is a feasible job-shop schedule which completes by T if and only if the optimal value of the above integer programming problem is 1.

A natural linear programming relaxation can be obtained by replacing (3) by $x_{j\sigma} \geq 0$, for all $j \in \mathcal{J}$, $\sigma \in \mathcal{F}_j$; let λ^* denote its optimal value. If we show that $\lambda^* > 1$, then we have proved that there is no schedule with makespan at most T: $T+1$ is a valid lower bound for this instance.

One natural way to show that $\lambda^* > 1$ is to exhibit a feasible dual solution of value greater than 1. We will give the dual linear program in a compact form, which is easy to derive. The dual variables are $y_{i,t}$, $i \in \mathcal{M}$, $t = 1, \ldots, T$, where each can be viewed as the cost of scheduling on machine i at time t. Thus, the cost of $\sigma \in \mathcal{F}_j$ is $C_\sigma(y) = \sum_{i \in \mathcal{M}} \sum_{t=1}^{T} \gamma(\sigma, i, t) y_{i,t}$. If we let $\mathcal{S} = \{y \geq 0 : \sum_{i \in \mathcal{M}} \sum_{t=1}^{T} y_{i,t} = 1\}$, then the dual is $\max_{y \in \mathcal{S}} \sum_{j \in \mathcal{J}} \min_{\sigma \in \mathcal{F}_j} C_\sigma(y)$.

Solving the packing relaxation The packing formulation and its linear relaxation contain an exponential number of variables; fortunately, fractional packing problems can be approximately solved by algorithms that are insensitive to the number of variables. Plotkin, Shmoys, & Tardos [17] provide a framework for obtaining solutions within a specified accuracy for such problems.

We view the relaxation as the fractional packing problem where we are attempting to pack job-schedules subject to a restriction on the amount of work on each machine at each time unit. Let P^j be the set consisting of all convex combinations of the job-schedules for job j and let $P = P^1 \times \ldots \times P^n$. Then our relaxation can be rewritten as $\min\{\lambda \mid x \in P, x \text{ satisfies (2)}\}$. To apply the algorithm of [17], we need to be able to efficiently evaluate the dual objective for a given y, and to efficiently optimize over P.

To evaluate the dual, we need to find the minimum-cost job-schedule for each job $j \in \mathcal{J}$. This can be done by a simple dynamic programming algorithm. We look at each operation $k \in \mathcal{O}_j$ in processing order and for each possible completion time t we calculate $c_k(t)$, the minimum cost of k and all operations preceding k in $\mathcal{O}_j$, such that k completes by t. Then if l is the last operation of job j, $c_l(T)$ gives the minimum cost job-schedule for j with respect to y. We can then use this dynamic program to find an optimal solution in P.

The fractional packing algorithm works as follows. Start with some $x \in P$ and an error parameter $\epsilon > 0$. Calculate λ for x; if $\lambda \leq 1$, then we are done.

Otherwise, compute $y_{i,t}$ as an exponential function of the load on machine i at time t, scaled appropriately. Then apply the dynamic programming algorithm to evaluate y and to find the point $\tilde{x} \in P$ of minimum cost with respect to y. If the dual value exceeds 1 or is within a $(1+\epsilon)$ factor of the primal value then stop. Update x by taking a convex combination of x and $\tilde{x}$ and start over. If the dual value is greater than 1 when we terminate, then we can conclude that $T+1$ is a valid lower bound. Otherwise we can not determine a lower bound. The running time of the algorithm is proportional to T and ϵ^{-2}.

Computational results The times that we report in Table 1 include only the time that our packing algorithm uses to show that the bound is valid: it does not include any time to search for the correct value of T. We cut off our algorithm after 20,000 iterations so there is no guarantee that the bounds we have given are the best possible.

For a given T, we are trying to determine whether $\lambda^* > 1$. When λ^* is very close to 1, this can be difficult; the number of iterations required to determine whether $\lambda^* > 1$ increases dramatically as λ^* approaches 1.

The bounds found by the packing algorithm are significantly better than other bounds in the literature based on LP-relaxations, such as the cutting plane approach of Applegate & Cook [2]. The algorithm however is still to slow to be of practical significance.

3 Processing Windows

When considering an instance of the feasibility problem with target makespan T, we can view each operation as constrained to be started within a given time interval. For example, consider an operation $k \in \mathcal{O}^i$, and let r_k and q_k denote, respectively, its head and tail in the corresponding one-machine instance of $1|r_j|L_{\max}$; that is, r_k is the total processing requirement of operations preceding k in job j, and q_k is the analogous sum for operations following k in j. Then the job bound implies that the starting time of operation k must occur in the interval $[r_k, T - q_k - p_k]$. In the packing formulation, we consider only job-schedules in which each operation starts in this window. We can strengthen this formulation by computing restricted *processing windows*; that is, for each operation k, specify a smaller interval $[u_k, v_k]$ within which the operation must begin processing.

In this section, we describe several approaches to compute restricted processing windows, by relying on bounds stronger than the job bound. In particular, we rely on lower bounds given by the one-machine subproblem. In describing our algorithms, we focus only on increasing the start of the processing window, since updating the end of the processing window can be done analogously, due to the symmetry of the problem.

One-machine shaving We shall first consider the one-machine relaxation in order to compute improved processing windows.

Focus on machine i and suppose that we are given a proposed processing window $[u_k, v_k]$ for each operation $k \in \mathcal{O}^i$: we wish to decide if machine i can be scheduled so that each operation starts within its designated window. It is equivalent to decide if there is a feasible schedule of the one-machine subproblem in which k has head u_k, body p_k, and tail $T - p_k - v_k$. Hence, we can rephrase the problem of computing a reduced processing window to that of adjusting heads and tails, which was an idea first introduced by Carlier & Pinson [7].

We can exploit the fact that the problem $1|r_j, pmtn|L_{\max}$ can be solved in polynomial time, and in fact, admits a nice min-max theorem. For any $S \subseteq \mathcal{O}^i$, let $p(S) = \sum_{k \in S} p_k$, $r(S) = \min_{k \in S} r_k$, and $q(S) = \min_{k \in S} q_k$. Since the first job in S can start no earlier than $r(S)$ and the tail of the last job of S to complete processing is at least $q(S)$, the length of the schedule is at least $r(S) + p(S) + q(S)$. Remarkably, Carlier & Pinson [7] have shown that minimum length of a preemptive schedule is equal to $\max_{S \subseteq \mathcal{O}^i}\{r(S) + p(S) + q(S)\}$.

Consider an operation $k \in \mathcal{O}^i$ with current processing window $[u_k, v_k]$; we shall maintain the invariant that if there is feasible schedule for the job-shop instance (of length T), then there is a schedule in which each operation starts within its current window. We shall compute $\bar{q}_k$, the maximum tail for operation k for which there still is a feasible preemptive schedule for the one-machine subproblem (where the other data are unchanged). Operation k is processed without interruption in any job-shop schedule, therefore its window may be updated to $[T - \bar{q}_k - p_k, v_k]$.

While one could merely apply a bisection search to find $\bar{q}_k$, for each $k \in \mathcal{O}^i$, we can instead apply the following more efficient procedure, One-Machine Shave, which is based on the min-max theorem above. For each ordered pair of operations $(b, c) \in \mathcal{O}^i$, first compute $P_{b,c} = \sum_{k \in S} p_k$, where $S = \{k \in \mathcal{O}^i \mid r_k \geq r_b \text{ and } q_k \geq q_c\}$, provided $S \neq \emptyset$. If $r_b + P_{b,c} + q_c > T$, then there does not exists a feasible job-shop schedule of length T. For each operation $k \in \mathcal{O}^i$ and each ordered pair (b, c) such that (i) $q_k < q_c$, (ii) $r_k \geq r_b$, and (iii) $r_b + P_{b,c} + p_k + q_c > T$, update $u_k \leftarrow \max\{u_k, r_b + P_{b,c}\}$

Lemma 1. One-Machine Shave *correctly updates the processing window of each operation on the machine in $O(n^3)$ time.*

Once we have updated the starting point of each interval, we must also update the ending point. We can then repeat these updates until no further changes can be made for this machine.

Carlier-Pinson updates Carlier & Pinson [8] gave the first procedure to update heads and tails for the one-machine problem. An *ascendent set* (k, S) consists of an operation k and a set of operations S, $k \notin S$ such that:

$$r(S) + p(S) + p_k + q(S) > T, \tag{4}$$

$$r_k + p(S) + p_k + q(S) > T. \tag{5}$$

Ascendent sets are important because condition (4) implies that in any feasible schedule, k is processed first or last among $S \cup \{k\}$, and condition (5) implies that k cannot be processed first. Therefore, if there is a feasible schedule, k must be processed last. Once we find an ascendent set (k, S), we can update

$u_k \leftarrow \max\{r_k, c(S)\}$, where $c(S) = \max\{r(T) + p(T) | T \subseteq S\}$ is a lower bound on the time at which the operations in S can complete processing.

Carlier and Pinson [8] give an $O(n^2)$ algorithm based on Jackson's algorithm to solve $1|r_j, pmtn|L_{\max}$ that finds ascendent sets and updates all of the heads or all of the tails for a one-machine problem. We can use their algorithm directly on the one-machine relaxation to improve the processing windows. Recently, Carlier & Pinson [9] have given an $O(n \log n)$ implementation of this algorithm.

Once again, we need to apply this procedure alternately to heads and tails until no further updates are possible. Surprisingly, the two algorithms, One-Machine Shave and Carlier-Pinson Updates yield the same result.

Theorem 2. *For any one-machine subproblem, applying* Carlier-Pinson Updates *until no more updates can be found is equivalent to applying* One-Machine Shave *until no updates are found.*

Another $O(n^2)$ algorithm for performing Carlier-Pinson updates We shall give another algorithm to compute Carlier-Pinson updates, by combining ideas from the previous two algorithms, which we will see in §4 is faster when solving a sequence of closely related problems. We can perform all updates for the ascendent sets in $O(n^2)$ time by repeatedly applying an $O(n)$ algorithm that performs updates for all ascendent sets (l, S) with a fixed value of $q(S)$.

Suppose that we wish to find all ascendent sets (l, S) in which $q(S) = q_k$, $k \in S$. We first construct a nested list of candidates for the set S. Set $S^0 = \{l | r_l \geq r_k, q_l \geq q_k\}$. Find the operation $l \in \mathcal{O}^i - S^0$ with maximum head length r_l. If $q_l \geq q_k$, then create a new set, $S^1 = S^0 \cup \{l\}$. Continue in this manner, iteratively choosing the operation with the next largest head, and creating a new set whenever its tail is at least q_k. Let $S^0 \subset \cdots \subset S^b$ be the sets constructed. The construction ensures that $r(S^0) \geq r(S^1) \geq \cdots \geq r(S^b)$. Finally, as we construct each S^i, also compute $c(S^i)$, $i = 0, \ldots, b$.

Let L denote the set of operations $l \in \mathcal{O}^i$ for which $q_l < q_k$. Let l be the operation in L for which p_l is maximum. If (l, S^b) does not satisfy condition (4), then each (l', S^b), $l' \in L$, also does not satisfy it, by the choice of l. So S^b can be eliminated as a candidate set. If (l, S^b) does not satisfy condition (5), then each (l, S^i), $i = 0, \ldots, b$ also does not satisfy it, and so l can be removed from L. If both ascendent set conditions are satisfied, then (l, S^b) is an ascendent set. We can then increase the head of l to $c(S^b)$. We can remove l from L, since S^b has the latest completion time of all candidate sets. We can continue this process, removing either one operation or one set until one list is empty. We have now found all updates for ascendent sets (l, S) with $q(S) = q_k$.

Once again, we must repeatedly apply this algorithm until no further updates are made.

Iterated one-machine window reduction When we update the processing window $[u_k, v_k]$ for an operation $k \in \mathcal{O}_j$, this can refine the windows of other operations of job j. For example, if we increase the start of k's interval from u_k to $\tilde{u}_k$, then its successor in j cannot start until $\tilde{u}_k + p_k$. Hence, updating windows on machine i can cause further updates on all other machines. This approach was used by Applegate & Cook [2], Brucker, Sievers, & Jurisch [6], and by Carlier & Pinson [8] as well, although this is apparent only from their computational results.

We shall refer to the procedure of iteratively updating the processing windows for one machine and propagating the implications of these updates through each job, until no further updates can be made, as an *iterated one-machine windows reduction algorithm.*

Exact one-machine shaving We can also base the update of the processing window for an operation $k \in \mathcal{O}^i$ on finding the maximum tail $\bar{q}_k$ such that there exists a feasible non-preemptive schedule for this one-machine subproblem. Once again, after computing $\bar{q}_k$, we can update the start of k's interval to $\max\{u_k, T - p_k - \bar{q}_k\}$. To compute $\bar{q}_k$ requires solving several instances of $1|r_j|L_{\max}$; while this is an "easy" $\mathcal{NP}$-hard problem, it is still fairly time-consuming to compute $\bar{q}_k$. This technique is interesting because it gives the maximum window reduction obtainable by considering only the one-machine relaxation. This technique does only slightly better than the techniques based on the preemptive model and requires much more time to compute.

Computational results By simply reducing some windows until they are empty, iterated Carlier-Pinson provided lower bounds that are approximately the same as those found by the packing algorithm and it obtained those bounds in less than 1/4 of a second for each problem tested. This time includes the time for a search to determine the correct value of T. So this algorithm is several orders of magnitude faster than the packing algorithm.

Exact one-machine shaving gave better bounds for several of the smaller problem that we examined but it gave virtually no improvement on the larger problems and is significantly slower. The running times include a bisection search between a lower bound provided by iterated Carlier Pinson and an upper bound provided by Applegate & Cook's implementation of the shifting bottleneck procedure. The time to compute the initial bounds is not included in the running times. The exact one-machine shaving results were obtained using the one-machine scheduling code of Applegate & Cook and no effort was made to optimize performance for our application.

4 Shaving

We shall show how to strengthen the ideas from One-Machine Shave to further reduce the size of the processing windows. For some operation $k \in \mathcal{O}$ and its operation window $[u_k, v_k]$ we attempt to reduce the window by showing that the operation could not have started at one end of the interval. If we assume that the operation starts at one end of the interval and show that this implies that there is no feasible schedule, then we can conclude that the operation cannot start then. Thus we can shave a portion of the window by applying this idea. Carlier & Pinson [9] have independently proposed this idea; a preliminary version of our results for this was announced in [18].

We shall describe our algorithm only in terms of shaving the start of the windows; of course these ideas apply to the end as well. Let the window for operation k be $[u_k, v_k]$ and fix w_k such that $u_k \leq w_k < v_k$. Consider the problem where $[u_v, v_k]$ is replaced by $[u_k, w_k]$ and all other windows are unchanged. If we can show that there is no feasible schedule for the modified data then if there is a feasible schedule for the original instance, k must start in the interval $[w_k+1, v_k]$.

This approach relies on the fact that we are able to show quickly that no feasible schedule exists in which the given operation is constrained to start within a small window. The algorithms described in §3 are ideally suited for this. In particular, in our procedure C-P Shave, we use Iterated Carlier-Pinson to determine the maximum value w_k such that $[u_k, w_k]$ is infeasible. Once again, updates for one operation's window may lead to further updates for other operations. Hence we repeatedly apply this until no further updates are possible.

The algorithm C-P Shave can also be viewed as an algorithm to verify that there exists a schedule consistent with a particular window, and hence we can apply this idea recursively. However, if we apply more than one level of recursion then the algorithm becomes impractical, so we consider only Double Shave, which simply calls C-P Shave.

Implementation issues We wish to find the maximum amount to shave off a given window. Since frequently we cannot shave off anything, we initially attempt to shave just one unit, and repeatedly double this trial length until the shaving procedure fails. We then perform a normal bisection search to find the optimal amount to shave.

We can specify exact starting times for some operations while ensuring that feasibility is maintained. For example, if $k \in \mathcal{O}^i$ is the first operation of a job and for each other $l \in \mathcal{O}^i$, $u_l \geq u_k + p_k$ then we can schedule k to start at u_k. We preprocess the instance to find all of these operations.

Each time we attempt to shave a portion of a window we compute a bound for an instance that differs only slightly from the original one. If we have already computed a bound for the original instance, it may be possible to exploit this fact. An advantage of our new $O(n^2)$ algorithm implementation to compute iterated Carlier-Pinson updates is that after updating one head we can compute the implied updates for all of the tails in essentially linear time.

Computational results For each algorithm we report a lower bound T and two running times: the first is the time to show that $T-1$ is infeasible, whereas the second also includes the bisection search for T. For C-P Shave, we use Iterated Carlier-Pinson as our initial lower bound and an upper bound provided by Applegate & Cook's implementation of the shifting bottleneck procedure. For Double Shave, we use C-P Shave as our initial lower bound and the best known upper bound, primarily due to Balas & Vazacopoulos [4].

C-P Shave provides a significant improvement over Iterated Carlier Pinson. The computation time for C-P Shave ranged from 3 seconds for the fastest problem to 160 seconds for the slowest; the more difficult problems are generally in the one to two minute range.

Double Shave is an excellent bound: in all but three of the instances it is tight. The three problems where Double Shave is not tight are currently unsolved. However we report better bounds in §5 using a branch-and-bound procedure. Double Shave is quite slow: running times range from 10 minutes for some of the easier problems to several days for the hardest. In addition to finding tight bounds for some problems, Double Shave was able to reduce the processing windows to such an extent that it is straightforward to construct a schedule.

It is worth noting the sensitivity to scaling of the various algorithms. The running time of the packing algorithm is proportional to T whereas the running time of iterated Carlier-Pinson is independent of T. The running times of the various Shave algorithms are all proportional to $\log T$, since they involve bisection searches to find the amount to shave.

5 Time-Oriented Branching

Almost all branch-and-bound algorithms for the job shop scheduling problem have been based on the disjunctive graph formulation. We give two branching rules that more directly exploit our processing windows formulation.

Branching on the next available operation We first consider a simple strategy: focus on the first available operation that has not yet been assigned a starting time: this operation either starts immediately or it is delayed. This gives us two branches. If an operation is delayed, we know that it must start at the completion time of some operation on the same machine (since we can restrict our attention to active schedules): in this case we can increase the start of the processing window to the earliest time that an unscheduled operation on that machine can finish.

While the above branching scheme is technically correct, it is not as efficient as possible. Since we do not restrict our search to active schedules, we can find the same schedule in both branches. When we update the head of an operation, we do not require that it immediately follow another operation on the same machine. Hence, it is possible that we delay an operation $k \in \mathcal{O}^i$, but still do not schedule another operation on machine i before we schedule k. The schedule obtained might also fail to be active in that there might be idle time immediately prior to some operation that could be used to process another operation which is currently being processed later.

To solve the first problem, when we delay an operation $k \in \mathcal{O}^i$ we mark it delayed and then require that it be scheduled to start at the completion of some other operation $l \in \mathcal{O}^i$. (To ensure this, we further delay k until it is true.) The second problem requires a bit more care, but we can nonetheless modify this approach to ensure that we generate only active schedules.

Branching on tight set and nearly tight sets A *tight set* is a set $S \subset \mathcal{O}^i$ such that

$$r(S) + P(S) + q(S) = T. \qquad (6)$$

An α-*tight set* is a set $S \subseteq \mathcal{O}^i$ such that

$$r(S) + P(S) + q(S) \geq T - \alpha. \tag{7}$$

A tight set $S \subset \mathcal{O}^i$ is significant because we know that machine i must process S without interruption from $r(S)$ until $T - q(S)$. Thus we first attempt to compute the order in which the operations of S are processed. To do this, we start at $r(S)$ and branch depending on which operation starts at time $r(S)$.

Unfortunately, for many instances, tight sets are hard to find, and so we relax the concept and consider α-tight sets. For any α-tight set S, we know that some operation starts in the interval $[r(S), r(S) + \alpha]$. If α is sufficiently small, each permutation of operations of the α-tight set will result in a set of disjoint processing windows. Thus, a natural extension of the branch-and-bound algorithm for tight sets can be used for α-tight sets.

We branch depending on which operation in S is processed first. If operation k is first, we can reduce the size of its window to be at most α and update the window size of the other operations to reflect the fact that they cannot start until k is completed. This creates an α-tight set for the remainder of the operations.

An interesting question is "Are the branches mutually exclusive?" The answer to that is "yes", provided α is not too large relative to the length of the short operations of the tight set.

Lemma 3. *If S is an α-tight set, and k and l are the two shortest operations of $\mathcal{O}^i$ then the branches are mutually exclusive if $\alpha < p_k + p_l$*

Computational Results The results of our branch-and-bound algorithms can be found in Table 1; when we did not obtain a result we report NR in the table.

Our first implementation ran Double Shave to reduce the windows and then branched on the next available operation to obtain a schedule. We used iterated Carlier-Pinson as the lower bounding technique for the branch-and-bound. This approach was successful in solving 11 of the 16 problems. We separate the time to find the optimal schedule into two parts: the time to reduce the windows using Double Shave and the time to find the schedule using branch-and-bound. In all cases where we found a schedule, the branch-and-bound portion took less than a second while the windows reduction phase usually took several thousand seconds. This approach was ineffective for instances where the critical path contained a long chain of operations on the same machine since, in this case, Double Shave performed badly.

Next we did not perform the windows reduction phase but instead used the stronger lower bound, C-P Shave. Using this technique we were able to solve fewer instances. But the amount of time required to solve some of the instances was greatly reduced. This technique seems to be appropriate only for relatively easy instances.

We also implemented α-tight set branching using C-P Shave as our lower bounding technique. We branched on the machine i with the highest one-machine bound and chose S to be the tightest subset of $\mathcal{O}^i$ subject to $|S| \geq n/2$. We indicate the running time to find the optimal schedule given the optimal T, the time to prove that the schedule is optimal and then the time to find the optimal schedule by starting T at the C-P Shave bound and increasing it by one until we find an optimal schedule. Using this technique we are able to solve 11 of the sixteen instances and obtain the best lower bounds known for three others. We were able to solve one other instance, LA27, when we used iterated Carlier-Pinson as our lower bound. The one instance that this technique has so far shown itself ineffective in solving is LA38: this might be due to the large gap between the one-machine bound and the optimal schedule length: consequently it is very difficult to find nearly tight sets.

There seem to be at least two very different classes of difficult job shop scheduling instances. In the first class, the one-machine bound is tight or nearly tight, but it is still very difficult to find an optimal schedule. LA27 is an example of this class: here the one-machine bound is tight but this problem was just recently solved when a new local search technique was finally able to reduce the upper bound to the one-machine bound. In the second class, the one-machine bound is very poor and it seems to be difficult to obtain good lower bounds. LA38 is an example of this sort; in this case the upper bound was established before the lower bound.

It seems likely that techniques that work for one type of problem might not work well for the other. The α-tight set branching seems to work well on the first type of problems. LA37 is a problem in this class. It seems to be difficult for local search techniques: only the guided local search techniques of Balas and Vazacopoulos are able to find an optimal schedule, but using α-tight set branching we are able to solve this problem in under two minutes. As we discussed above, this technique does not work as well on LA38, an instance at the other extreme.

Overall, our time-oriented branch-and-bound algorithms gave good results for the instances that we tested. We were able to solve to optimality four standard benchmark problems which were unsolved when we began our effort, including ABZ7 which to our knowledge we were the first to solve. For the other three unsolved problems that we examined, we were able to improve upon the best known lower bounds. In all cases, the performance of our algorithm is at least competitive with disjunctive graph based approaches and in many cases our algorithm performs significantly better.

It seems likely that further progress can be made by combing the purely combinatorial approaches with our time-oriented approach. From our insight into the dual nature of hard instances, we believe that it is likely such a “multiple-pronged attack” might be most effective.

References

1. J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Management Sci.*, 34:391–401, 1988.
2. D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA J. Comput.*, 3:149–156, 1991.

3. E. Balas. On the facial structure of scheduling polyhedra. *Math. Programming Stud.*, 24:179–218, 1985.
4. E. Balas and A. Vazacopoulos. *Guided Local Search with Shifting Bottleneck for Job Shop Scheduling*. Management Science Research Report #MSRR-609, Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1994.
5. P. Bratley, M. Florian, and P. Robillard. On sequencing with earliest starts and due dates with application to computing bounds for the $(n/m/G/F_{\max})$ problem. *Naval Res. Logist. Quart.*, 20:57–67, 1973.
6. P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Appl. Math.*, 49:107–127, 1994.
7. J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Sci.*, 35:164–176, 1989.
8. J. Carlier and E. Pinson. A practical use of Jackson's preemptive schedule for solving the job-shop problem. *Ann. Oper. Res.*, 26:269–287, 1990.
9. J. Carlier and E. Pinson. Adjustments of heads and tails for the job-shop problem. *European J. Oper. Res.*, 78:146–161, 1994.
10. H. Fisher and G.L. Thompson. Probabilistic learning combinations of local job-shop scheduling rules. In J.F. Muth and G.L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, Englewood Cliffs, NJ, 1963.
11. M.L. Fisher, B.J. Lageweg, J.K. Lenstra, and A.H.G. Rinnooy Kan. Surrogate duality relaxation for job shop scheduling. *Discrete Appl. Math.*, 5:65–75, 1983.
12. M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Math. Oper. Res.*, 1:117–129, 1976.
13. R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling. *Ann. Discrete Math.*, 5:287–326, 1979.
14. J.R. Jackson. An extension of Johnson's results on job lot scheduling. *Naval Res. Logist. Quart.*, 3:201–203, 1956.
15. S. Lawrence. *Resource Constrained Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*. Graduate School of Industrial Administration, Carnegie Mellon University, Pittsburgh, PA, 1984.
16. E. Nowicki and C. Smutnicki. A fast taboo search algoritm for the job shop problem. *Management Sci.* To appear.
17. S. A. Plotkin, D. B. Shmoys, and É. Tardos. Fast approximation algorithms for fractional packing and covering problems. *Math. Oper. Res.*, 20:257–301, 1995.
18. D.B. Shmoys. Solving scheduling problems via linear programming. Talk at *ORSA/TIMS*, Boston, MA, May 1994.
19. J.P. Sousa and L.A. Wolsey. A time-indexed formulation of non-preemptive single-machine scheduling problems. *Math. Programming*, 54:353–367, 1992.
20. C. Stein. *Approximation Algorithms for Multicommodity Flow and Shop Scheduling Problems*. PhD thesis, MIT/LCS/TR-550, Laboratory for Computer Science, MIT, Cambridge, MA, 1992.
21. R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. *Math. Programming B*. To appear.
22. J.M. Van den Akker. *LP-based solution methods for single-machine scheduling problems*. PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1994.

This article was processed using the LaTeX macro package with LLNCS style