Shop Floor Scheduling with Setup Times

-Efficiency versus Leadtime Performance-

Marco Schutten

ISBN 90-9009511-X

Printed by Febodruk B.V., Enschede, The Netherlands

SHOP FLOOR SCHEDULING WITH SETUP TIMES

-EFFICIENCY VERSUS LEADTIME PERFORMANCE-

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de Rector Magnificus,
prof.dr. Th.J.A. Popma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 20 juni 1996 te 16.45 uur.

door

Johannes Marius Jacobus Schutten

geboren op 6 januari 1969

te Hellendoorn

Dit proefschrift is goedgekeurd door de promotor:

prof.dr. W.H.M. Zijm

Preface

During my study at the Faculty of Applied Mathematics at the University of Twente, I started to appreciate doing research for practical problems. At that time, I was working on a vehicle routing problem for a Dutch parcel service. The problem was analyzed thoroughly and the proposed solutions were implemented in practice. This gave me much satisfaction. Since I liked doing research, I investigated the possibility of becoming a research assistant. Henk Zijm, the chairman of the Production and Operations Management group of the University of Twente, gave me a chance to study practical problems. I accepted this opportunity.

This thesis is the result of four years of research in the area of shop floor scheduling in small batch parts manufacturing shops. It concerns extending existing procedures for shop floor scheduling with practical features. In particular, it focuses on integral scheduling, aiming at an optimal delivery performance under tight capacity constraints, including setup times. Setup times occur, e.g., when machines must be cleaned between two operations.

Many people contributed to my research in one way or another. I thank them all. The people below, I like to thank in particular.

In the first place, I express my gratitude to Henk Zijm. He gave me the opportunity to work in his group. His ideas and enthousiasm for scientific research inspired me a lot. Second, I thank Steef van de Velde. His knowledge of scheduling theory and presenting results can be found throughout this thesis. Both Henk and Steef helped me a lot in writing this thesis. The third man that read early versions of parts of this thesis is Piet Weeda. His comments on the first chapters were very useful. Unfortunately, he was not able to comment on early versions of the other chapters.

Until February of this year, Geatse Meester was my room mate at the university. I appreciated his presence very much. Geatse helped me many times in getting more familiar with the concepts used in shop floor control and almost always had an answer to my questions about production techniques.

Next, I thank all other members of the Production and Operations Management group. They contribute to a very pleasant working atmosphere and the needed interruptions of a working day.

Several students contributed to my research. I like to mention Alexander Belderok, Dinand Reesink, Corina van Unen, Robert Leussink, Marcel Wildschut, Peter Klein Lebbink, and Anneleen van Beek. Thanks to you all.

I also like to mention here the people that work at FLEX, Engineers in Logistic Systems, especially the three managers Wik Heerma, Nico Lok, and Gerrit-Jan Steenbergen. I found the cooperation with them always a pleasant and fruitful experience. Thanks to them as well.

Last, but certainly not least, I like to thank my family, in particular my father. Although life has not been easy for him, he always did his utmost best to make life as easy as possible for me. I like to finish this preface with one Dutch sentence for him: Pa, hartstikke bedankt voor al je goede zorgen, met name in de afgelopen zeven jaar!

Summary

In this thesis, we analyze and develop algorithms for scheduling problems in small batch parts manufacturing shops. Short leadtimes and reliable delivery dates have become dominant market performance indicators. We show that *integral* shop floor planning and scheduling systems may result in shorter leadtimes and more reliable delivery dates. The scheduling problems that arise are in fact classical job shop problems with additional side constraints. After a brief introduction to scheduling theory, we discuss the classical job shop problem and algorithms to solve it.

One of the algorithms we discuss is the Shifting Bottleneck (SB) procedure of Adams et al. [2], which decomposes the problem of scheduling a job shop into a series of single-machine scheduling problems. Vaessens et al. [79] show that in general the SB procedure yields solutions of good quality in reasonable time. Each instance of the classical job shop problem can be represented by a disjunctive graph. We show that by changing the properties of this graph and by changing the algorithms for the single-machine scheduling problems, the SB procedure can be extended to deal with various practical features, including release and due dates of jobs, parallel machines, assembly steps, simultaneous resource requirements, and setup times.

The occurrence of setup times on one or more machines in the shop results for those machines in single-machine scheduling problems in which setup times have to be considered. In these problems, we have to find a trade-off between efficiency and leadtime performance. For the special case of family setup times, we develop a branch-and-bound algorithm. For lower bounding purposes, we introduce the concept of setup jobs and derive sufficient conditions to introduce them without loosing the optimal solution. We generalize the branch-and-bound algorithm to solve the parallel-machine scheduling problem with family setup times and characterize an optimal solution for a broad class of parallel-machine scheduling problems.

Our computational experiments indicate that the SB procedure significantly outperforms priority rules for a shop in which setup times and assembly of parts to end products occur. The SB procedure with extensions is part of a commercial shop floor scheduling system called JOBPLANNER. We discuss this system and its use in a company producing printed circuit boards. The introduction of JOBPLANNER in this company has been most satisfactory: the output and the delivery performance increased, whereas the throughput times decreased significantly.

Contents

Chapter 1

Introduction

Baker [6] defines *scheduling* as the allocation of resources over time to perform a collection of tasks. This chapter starts with the introduction of a simple scheduling problem, in which conference visitors must perform a number of activities. We use this example mainly to clarify notions used in scheduling theory. Section 1.2 introduces *machine scheduling* problems, a class of scheduling problems in which the resources, or *machines*, can perform at most one task, or *operation*, at a time. This thesis concentrates on *deterministic* machine scheduling problems in which we know all data in advance. In Section 1.3, we give an overview of the contents of this thesis. For an extensive introduction to scheduling theory, see, e.g., the text books by Baker [6, 7] and French [28], the survey of Lawler et al. [52], and the books by Morton and Pentico [59], Brucker [12], and Pinedo [63].

1.1 Introductory example[1]

Pete, Steve, Art, and Hank, four scientists, go to the same OR conference. The organizing committee has a special low budget option for staying in a student flat. Of course, the four scientists decide to take this stroke of luck. Now they must share a bathroom, a kitchen, a telephone, and an iron heater. Last night, the four conference visitors decided to walk together to the conference site. Before they leave, the

[1]The story, all names, characters and incidents in this example are fictitious. No identification with or similarity to actual persons, living or dead, or to actual events is intended or should be inferred.

men like to take a shower, make breakfast, call their wives, and iron their shirts, each in a particular order. The bathroom and the kitchen are very small: only one person at a time can use the bathroom and make breakfast. If a person starts a certain activity, e.g., taking a shower, then he finishes this activity without interruption: *preemption* of activities, i.e., interruption of activities and resumption later on, is not allowed.

Pete gets up at 7.00 AM, and the first thing he likes to do is to call his wife for 10 minutes. Then, he has breakfast for 20 minutes. After breakfast, Pete wants to iron his shirt. Since he is unfamiliar with domestic tasks, this will take him 25 minutes. Finally, he takes a shower for 20 minutes. Steve gets up at 7.15 AM. He also likes first to call his wife. Steve, an investor with some successful moments, also needs to call his local internet supplier to check out the stock market. All in all, Steve needs the telephone for 25 minutes. After this, he needs 10 minutes to iron his shirt, 20 minutes to eat, and 15 minutes to take a shower, in this order. Art also gets up at 7.15 AM. The first thing he likes to do is to take a shower for 20 minutes. Then, Art likes to have breakfast and, since this is the most important meal of the day, he does this in a relaxed and comprehensive way for 40 minutes. After this, Art quickly irons his shirt and calls his wife. Both activities take 10 minutes. Hank, finally, wakes up at 7.30 AM. After having breakfast for 25 minutes, he irons his shirt for 15 minutes, and calls his wife for 10 minutes. In the bathroom, Hank trims his beard and takes a shower. His time in the bathroom totals 30 minutes. Table 1.1 summarizes the order in which each person performs the activities and the amount of time each activity takes. In this table, the name of each person is

person	gets up at	order of activities				required time (minutes)			
P	7.00	T	K	I	B	10	20	25	20
S	7.15	T	I	K	B	25	10	20	15
A	7.15	B	K	I	T	20	40	10	10
H	7.30	K	I	T	B	25	15	10	30

Table 1.1: The data of the introductory example.

abbreviated by the first letter. The first letter of the *resources*, that is, the telephone, the kitchen, the iron heater, and the bathroom, are used to indicate the activities that take place using these resources. The conference program starts at 8.30 AM, but given the data in Table 1.1, Art and Hank will unavoidably be late. Since the four conference visitors decided to make the long walk to the conference hotel *together* and the conference program is very appealing this morning, their objective is to leave as soon as possible. This means that *the time to complete all activities must be minimized.*

The order in which the men perform a certain activity is called a *sequence*. Table 1.2 displays for each activity the sequence the men came up with. For example, P-S-A-H is the sequence for using the telephone.

activity	1st	2nd	3rd	4th
T	P	S	A	H
K	H	P	A	S
I	H	S	P	A
B	A	P	S	H

Table 1.2: Possible sequence for each activity.

This means that Pete is the first to call his wife. He is followed by Steve, Art, and Hank, in this order. A *schedule* specifies the time slot in which each activity is performed. Figure 1.1 is a graphical representation of a possible schedule, of which the sequence for each activity is displayed in Table 1.2. Such a graphical representation of a schedule is called a *Gantt chart*, after its developer Gantt [30]. In this chart, time is shown along the horizontal axis and the resources are shown along the vertical axis. Each rectangle represents an activity involving the corresponding resource during the corresponding time interval. Steve, for example, uses the telephone from 7.15 AM until 7.40 AM. The schedule depicted in Figure 1.1 is *feasible*: the first activity of each researcher starts after he gets up, the order in which the men want to perform their activities is followed, and no activity is performed by more than one person at the same time. Note that we cannot start any activity earlier without changing the sequences in which the men perform the activities. Schedules with this property are called *left-justified* schedules. In this schedule, the men leave for the conference at 10.00 AM. Although the scientists are supposed to have a profound knowledge of scheduling theory, they did not solve this practical problem very well. It is left to the reader to find a feasible schedule in which the men leave as early as possible.

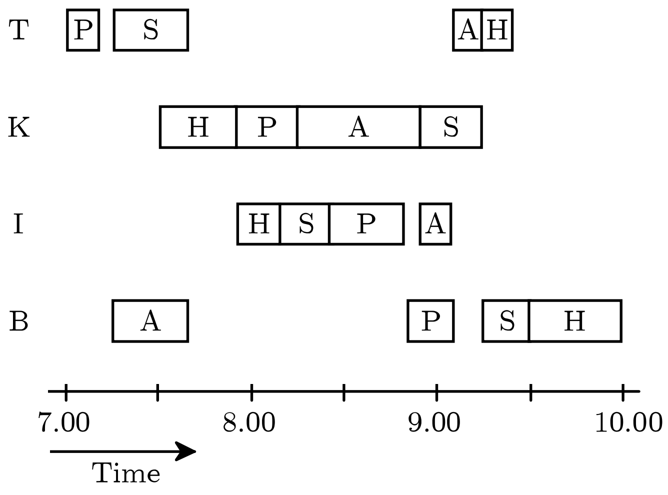


Figure 1.1: Graphical representation of a possible schedule.

1.2 Machine scheduling

If scheduling was limited to determining when conference visitors should take a shower and iron their shirts, then no one would study it. In practice, the resources to be scheduled include machines, processors, personnel, and vehicles. In the example, the activities to be scheduled are taking a shower, having breakfast, ironing a shirt, and making a phone call. In practice, the activities to be scheduled are, e.g., drilling, adding and subtracting, repairing a flat tire, and transporting people. Most of these scheduling problems can be modeled as scheduling *jobs* on *machines.* The generic term for the theory that studies these models is *machine scheduling theory.* The processing of a job on a machine is called an *operation.* The order in which a job must visit the machines for processing is called the *routing* of this job.

In this thesis, we concentrate on scheduling problems in *small batch parts manufacturing shops*; these are shops that produce a large variety of parts in small *batches.* In this context, a batch is a number of identical parts that are produced contiguously. Due to this variety, dedicated production lines are usually not beneficial. These shops therefore often have a functional layout in which operations are performed by versatile machines. In order to perform such a variety of operations, machines must be prepared, or *set up*, for some specific operation, e.g., by loading appropriate cutting tools or by altering certain process parameters, usually at the cost of a loss of time and, hence, of capacity. In the past, setup times were quite large and economies-of-scale principles dictated large production batches. Large batches, however, induce long manufacturing throughput times, in particular if the number of machines involved in producing a part is large.

In the last twenty years, however, short manufacturing leadtimes and a high delivery performance have become dominant market performance indicators; cf. Blackburn [11]. At the same time, the introduction of computer technology on the shop floor, such as in CNC machines and assembly robots, has reduced setup times. Many companies use priority rules for scheduling their machine shops. Priority rules are myopic, however, and may result in poor solutions, particularly when setup times are still significant. Clustering jobs with similar setup characteristics, a rule of thumb to pursue efficiency, may result in a poor delivery performance for other jobs. Even overall efficiency may decrease, since clustering jobs on one machine can cause idleness of another machine, because the latter does not receive the right jobs in time.

Computer technology and in particular advanced information systems (see, e.g., Tiemersma [76] and Arentsen [5]) provide the basis for implementing *integral shop floor planning and scheduling systems*. In this thesis, we concentrate on the design and analysis of algorithms for practical shop floor scheduling problems. We show that an integral shop floor planning and scheduling system results in a better delivery performance and smaller manufacturing leadtimes.

This section continues as follows. First, we introduce in Section 1.2.1 two single-machine scheduling problems. The first problem is easy to solve and we introduce it to examplify a generic technique for proving optimality of a scheduling rule. The second problem is closely related to the first, but is much more difficult to solve. This problem is important, because it appears as a subproblem in decomposition based approaches for scheduling machine shops. It is also used to schedule a shop with a single critical, or *bottleneck*, machine. Due to the large variety of machine scheduling problems, we need a classification scheme. We discuss such a scheme in Section 1.2.2. To distinguish between easy-to-solve and intractable problems, Section 1.2.3 reviews the basic concepts of complexity theory.

1.2.1 Two single-machine scheduling problems

In this section, we consider machine scheduling problems in which we only have a single machine. Each job consists of one operation. In the first problem, a set $\mathcal{J}$ of n jobs $J_1, J_2, \ldots, J_n$ needs to be scheduled on a single machine. The machine is continuously available for processing from time 0 onwards, and can process at most one job at a time. Each job J_j ($j = 1, 2, \ldots, n$) is available from time 0 onwards and needs uninterrupted processing during p_j time units, also called the *processing time*. Associated with each job J_j, there is a *due date* d_j by which the job should be completed. A schedule σ specifies for each job J_j a *completion time* $C_j(\sigma)$, i.e., the time at which this job finishes. The *lateness* $L_j(\sigma)$ of J_j in schedule σ is defined as the difference between its completion time in σ and its due date, i.e.,

$$L_j(\sigma) = C_j(\sigma) - d_j.$$

The objective is to find a schedule σ^* for which the *maximum lateness* $L_{\max}(\sigma^*)$ is minimal, i.e.,

$$L_{\max}(\sigma^*) = \min_{\sigma \in \Omega} L_{\max}(\sigma),$$

with Ω the set of feasible schedules and

$$L_{\max}(\sigma) = \max_{j=1,\ldots,n} L_j(\sigma).$$

σ^* is called an *optimal* schedule. Without loss of generality, we may assume for this example (and all other scheduling problems we consider) that all data are integral numbers. Note that we may restrict ourselves to *left-justified* schedules, i.e., schedules in which no job can start earlier without changing the sequence of jobs. Each sequence of jobs uniquely induces a left-justified schedule and vice versa.

Consider the following rule: *schedule the jobs in order of non-decreasing due dates.* This rule is known as the Earliest Due Date (EDD) rule or Jackson's rule (Jackson [44]). The following theorem proves that the EDD produces optimal schedules. We have included the proof of this theorem to introduce a generic technique for proving optimality of a scheduling rule.

Theorem 1.1 (Jackson [44]) *The EDD rule yields an optimal schedule.*

Proof. Let σ_1 be any optimal schedule and suppose that σ_1 differs from the schedule that results from Jackson's rule, which we denote by σ_J. Reindex the jobs so that J_j is the jth job in σ_J. We then have that $d_1 \leq d_2 \leq \ldots \leq d_n$. We will transform σ_1 into σ_J without losing optimality. Denote the index of the ith job in σ_1 by $[i]$, i.e., $J_{[i]}$ is the ith job in σ_1. Let $[j]$ be the index of the first job in σ_1 such that $[j] > [j+1]$ and therefore $d_{[j]} \geq d_{[j+1]}$. Let σ_2 be the schedule that results from σ_1 by swapping $J_{[j]}$ and $J_{[j+1]}$; see Figure 1.2.

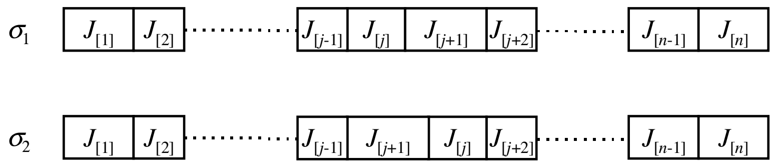


Figure 1.2: The schedules σ_1 and σ_2.

The relation $C_{[i]}(\sigma_1) = C_{[i]}(\sigma_2)$ holds for $1 \leq i \leq n$, $i \neq j, j+1$ and therefore $L_{[i]}(\sigma_1) = L_{[i]}(\sigma_2)$ for these jobs. Also, $L_{[j+1]}(\sigma_2) = C_{[j+1]}(\sigma_2) - d_{[j+1]} \leq C_{[j+1]}(\sigma_1) - d_{[j+1]} = L_{[j+1]}(\sigma_1)$ and $L_{[j]}(\sigma_2) = C_{[j]}(\sigma_2) - d_{[j]} = C_{[j+1]}(\sigma_1) - d_{[j]} \leq C_{[j+1]}(\sigma_1) - d_{[j+1]} = L_{[j+1]}(\sigma_1)$. Thus, we have that $L_{\max}(\sigma_2) \leq \max_{1 \leq i \leq n, i \neq j} L_{[i]}(\sigma_1) \leq \max_{1 \leq i \leq n} L_{[i]}(\sigma_1) = L_{\max}(\sigma_1)$ and σ_2 must also be an optimal schedule. By repeating this argument, we can transform σ_1 into σ_J without losing optimality. $\square$

The second problem we consider is the same as the first problem, except that the jobs have *release dates*, before which they cannot be processed. We denote the release date of job J_j by r_j. Consider the extended Jackson rule: *at any time T that the machine is available for processing, process without interruption an unprocessed, available job that has the smallest due date.* A job J_j is available at time T, if $r_j \leq T$. In general, the extended Jackson rule does not yield an optimal solution. In fact, it is very unlikely that a simple rule like this can be guaranteed to deliver an optimal solution for all instances of this problem. A problem *instance* is formed by specific choices for the parameters of the problem *type.* We will point out that there exists strong circumstantial evidence for this claim in Section 1.2.3. Consider the instance of which the data are given in Table 1.3. For this instance, the extended Jackson rule produces the schedule $\sigma = J_1 - J_2 - J_3 - J_4 - J_5 - J_6 - J_7$ with $L_{\max}(\sigma) = L_5(\sigma) = 3$; see Figure 1.3. An optimal schedule σ^* has $L_{\max}(\sigma^*) = 0$,

J_j	J_1	J_2	J_3	J_4	J_5	J_6	J_7
r_j	0	10	13	11	20	30	30
p_j	6	5	6	7	4	3	2
d_j	33	43	24	26	29	42	50

Table 1.3: A seven-job instance for the second problem.

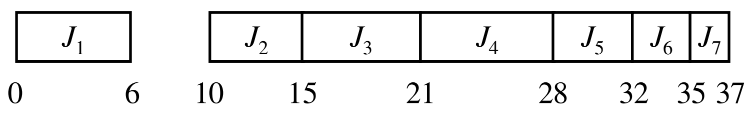


Figure 1.3: Schedule produced with the extended Jackson rule.

however; see Figure 1.4. We see that it is advantageous to keep the

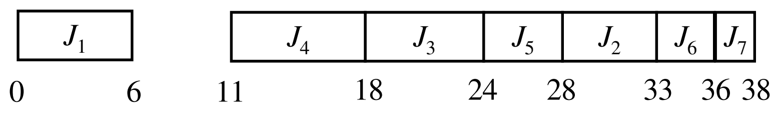


Figure 1.4: An optimal schedule.

machine idle at time 10, to be able to perform jobs with small due dates early. The difficulty is to decide when to keep the machine idle, although there are jobs ready to be processed.

Although the extended Jackson rule is an *approximation algorithm*, not an *optimization algorithm*, it has some agreeable properties, such as a performance guarantee. Carlier [17] uses the rule to develop an optimization algorithm for this problem. To clarify how this algorithm works, let us now analyze the extended Jackson rule. Let $\mathcal{U}$ be any subset of $\mathcal{J}$. Using a critical path argument, it easily follows that (cf. Carlier [17])

$$h(\mathcal{U}) = \min_{J_j \in \mathcal{U}} r_j + \sum_{J_j \in \mathcal{U}} p_j - \max_{J_j \in \mathcal{U}} d_j$$

is a *lower bound* on the optimal solution value $L^*_{\max}$, i.e.,

$$h(\mathcal{U}) \leq L^*_{\max}, \text{ for any } \mathcal{U} \subseteq \mathcal{J}.$$

We use this lower bound to prove the next theorem.

Theorem 1.2 (Carlier [17]) *If the schedule σ produced by the extended Jackson rule is not optimal, then there exists a critical job J_c and a critical job set $\mathcal{C} \subseteq \mathcal{J}$ such that*

1. *$h(\mathcal{C}) > L^*_{\max} - p_c$;*
2. *in any optimal schedule, job J_c is scheduled either before all jobs in $\mathcal{C}$, or after all jobs in $\mathcal{C}$.*

Proof. First of all, reindex the jobs in order of increasing completion times in σ. Suppose that σ is not an optimal schedule. Let J_k be the last job in σ for which $L_k(\sigma) = L_{\max}(\sigma)$ and let J_j be the first job in σ for which we have that $C_k(\sigma) = r_j + \sum_{i=j}^{k} p_i$. Note that $r_j = \min_{j \le i \le k} r_i$. Also, note that $d_k < \max_{j \le i \le k} d_i$, and that $J_k \neq J_j$, because otherwise we would have that

$$\begin{aligned} L_k(\sigma) &= r_j + \sum_{i=j}^{k} p_i - d_k \\ &= \min_{j \le i \le k} r_i + \sum_{i=j}^{k} p_i - \max_{j \le i \le k} d_i \\ &= h(\{J_j, \ldots, J_k\}) \\ &\le L^*_{\max} \end{aligned}$$

and therefore σ would be an optimal schedule.

Let J_c be the last job in σ belonging to the set $\{J_j, \ldots, J_k\}$ for which $d_c > d_k$ and define $\mathcal{C} = \{J_{c+1}, \ldots, J_k\}$. Note that $d_c > d_i$ for all $J_i \in \mathcal{C}$. Due to the way σ is constructed, we have that at start time $S_c(\sigma)$ of J_c in σ no job belonging to $\mathcal{C}$ is available for processing, i.e., $r_i > S_c(\sigma)$ for all $J_i \in \mathcal{C}$. Therefore, we have that $\min_{J_i \in \mathcal{C}} r_i > S_c(\sigma) = r_j + \sum_{i=j}^{c-1} p_i$. Moreover, we have that $d_k = \max_{J_i \in \mathcal{C}} d_i$. It then follows that

$$\begin{aligned} h(\mathcal{C}) &= \min_{J_i \in \mathcal{C}} r_i + \sum_{i=c+1}^{k} p_i - \max_{J_i \in \mathcal{C}} d_i \\ &> r_j + \sum_{i=j}^{k} p_i - p_c - d_k \\ &= L_k(\sigma) - p_c \end{aligned}$$

$$\begin{aligned} &= L_{\max}(\sigma) - p_c \\ &\geq L^*_{\max} - p_c. \end{aligned}$$

We now prove the second part of the theorem that characterizes the set of optimal schedules. Let σ^* be any optimal schedule without Property 2, that is, with J_c scheduled somewhere in-between the jobs in $\mathcal{C}$. Let $J_q \in \mathcal{C}$ be the job that is processed later in σ^* than any other job in $\mathcal{C}$. Then we have that

$$\begin{aligned} L_q(\sigma^*) &\geq \min_{J_i \in \mathcal{C}} r_i + \sum_{J_i \in \mathcal{C}} p_i + p_c - d_q \\ &\geq \min_{J_i \in \mathcal{C}} r_i + \sum_{J_i \in \mathcal{C}} p_i - \max_{J_i \in \mathcal{C}} d_i + p_c \\ &= h(\mathcal{C}) + p_c \\ &> L^*_{\max}, \end{aligned}$$

which is a contradiction. □

Based on Theorem 1.2, Carlier [17] develops an effective optimization algorithm for this problem. In this algorithm, we start with scheduling the jobs with the extended Jackson rule and check whether we can verify the optimality of the schedule σ using the arguments in the first part of the proof of Theorem 1.2. If we cannot verify the optimality of σ, then we determine the critical job J_c and the critical job set $\mathcal{C}$ and compute the lower bound $h(\mathcal{C})$. If $h(\mathcal{C}) = L_{\max}(\sigma)$, then σ is yet verified to be optimal, otherwise two new problems are created: one in which J_c precedes the jobs in $\mathcal{C}$ and one in which J_c succeeds the jobs in $\mathcal{C}$. These subproblems are solved in the same way as the original problem: the extended Jackson rule is used to schedule the jobs, the schedule is checked for optimality, the critical job and job set are determined, and, if necessary, two new subproblems are created. This process continues until all subproblems are solved. The best found schedule is an optimal schedule. Based on the next two theorems, Carlier tries to force that the critical job precedes or succeeds the critical job set by changing the release date and the due date of the critical job.

Theorem 1.3 (Carlier [17]) *If J_c precedes all jobs in $\mathcal{C}$ in any optimal solution, then the optimal solution value and the optimal solution are* *preserved if we let*

$$d_c = d_k - \sum_{J_i \in \mathcal{C}} p_i.$$

Proof. Let σ^* be any optimal solution and let J_q be the last job in σ^* that belongs to $\mathcal{C}$. Then we have that

$$C_c(\sigma^*) \leq C_q(\sigma^*) - \sum_{J_i \in \mathcal{C}} p_i,$$

and hence that

$$\begin{aligned} C_c(\sigma^*) - (d_k - \sum_{J_i \in \mathcal{C}} p_i) &\leq C_q(\sigma^*) - d_k \\ &\leq C_q(\sigma^*) - d_q \\ &\leq L^*_{\max}, \end{aligned}$$

since $d_q \leq d_k$. □

Theorem 1.4 (Carlier [17]) *If J_c succeeds all jobs in $\mathcal{C}$ in any optimal solution, then the optimal solution value and the optimal solution are preserved if we let*

$$r_c = \min_{J_i \in \mathcal{C}} r_i + \sum_{J_i \in \mathcal{C}} p_i.$$

Proof. Since J_c cannot start before all jobs in $\mathcal{C}$ are completed, we must also have that $S_c(\sigma^*) \geq \min_{J_i \in \mathcal{C}} r_i + \sum_{J_i \in \mathcal{C}} p_i$ in any optimal schedule σ^*. □

Note that this type of adjustment does not guarantee that J_c will be scheduled before or after the jobs in $\mathcal{C}$, but aims at strengthening the lower bounds $h(\mathcal{C})$ of the subproblems.

Let us now return to the instance of Table 1.3. Note that we do not need to reindex the jobs: they are already indexed in order of increasing completion times in σ. We cannot conclude that σ is an optimal solution: $L_5(\sigma) = L_{\max}(\sigma) = 3$, $j = 2$ is the first job index for which we have that $C_5(\sigma) = r_j + \sum_{i=j}^{5} p_i$, and $d_5 < \max\{d_2, d_3, d_4, d_5\}$. The critical job is job J_2 and the critical job set $\mathcal{C} = \{J_3, J_4, J_5\}$ with $h(\mathcal{C}) = 11 + 17 - 29 = -1$. Since $h(\mathcal{C}) < L_{\max}(\sigma)$, two subproblems P_1

and P_2 are created. In P_1, J_2 is supposed to precede the jobs J_3, J_4, and J_5; in P_2, J_2 is supposed to succeed them.

For problem P_1, we set $d_2 = d_5 - \sum_{i=3}^{5} p_i = 12$. The extended Jackson rule produces then the same schedule as before, but now $d_5 = \max\{d_2, d_3, d_4, d_5\}$ and hence this schedule is optimal for problem P_1, where we have to schedule J_2 before J_3, J_4, and J_5. We do not need to create subproblems from P_1.

For the problem P_2, we set $r_2 = \min_{3\leq i\leq 5} r_i + \sum_{i=3}^{5} p_i = 28$. The extended Jackson rule results in de schedule $\sigma^{(2)} = J_1 - J_4 - J_3 - J_5 - J_2 - J_6 - J_7$ with $L_{\max}(\sigma^{(2)}) = L_3(\sigma^{(2)}) = 0$. For this schedule, the critical job is J_4 and the critical job set $\mathcal{C} = \{J_3\}$ with $h(\mathcal{C}) = -6 < L_{\max}(\sigma^{(2)}) = 0$. We therefore create two subproblems P_3 and P_4, but it appears that these problems do not result in a better solution. The schedule $\sigma^{(2)}$ is therefore an optimal schedule.

1.2.2 Problem classification

Due to large variety of machine scheduling problems, it is convenient to have a classification scheme. We follow the three-field classification $\alpha|\beta|\gamma$, which was introduced by Graham et al. [38] and revised by Lawler et al. [51]. In this classification, a machine scheduling problem is identified by machine characteristics, job characteristics, and an objective function.

The first field, $\alpha = \alpha_1\alpha_2\mu$, specifies the machine characteristics of a problem. The parameter $\alpha_2 \in \{\circ, 1, 2, \ldots\}$ indicates the number of machines that must be scheduled, where $\circ$ is the symbol to denote an unspecified parameter value. In this case, $\circ$ means that the number of machines is unspecified. The parameter $\alpha_1 \in \{\circ, P, Q, R, J, F, O\}$ represents the machine environment. If $\alpha_1 = \circ$ and $\alpha_2 = 1$, then we have a single-machine scheduling problem. If $\alpha_1 \in \{P, Q, R\}$, then we have a parallel-machine scheduling problem in which we must schedule each job on one of the parallel machines. We use P, Q, and R to denote that we have identical parallel machines, uniform parallel machines, and unrelated parallel machines, respectively. If the machines are *identical*, then the processing time of each job is the same on all machines. *Uniform* machines work at different speeds, i.e., the processing time of each job differs by a constant factor for the individual machines. If the machines are *unrelated*, then there is no relation between the processing times of the jobs and the machines. J, F, and O indicate that we have a job shop, flow shop, and open shop problem, respectively. In *job shops*, each job has its own routing through the shop, whereas in *flow shops* the machines can be indexed so that each job visits the machines in order of increasing machine indices. In job shops and flow shops, the sequence in which a job visits the machines is given. This sequence is free for jobs in *open shop* problems: the sequence in which a job visits the machine can be determined by the scheduler. We use the parameter μ for further machine characteristics; $\mu = down$, e.g., indicates that machines may have *down times*: these are intervals in which a machine is not available for processing jobs.

The second field, $\beta \subseteq \{\beta_1, \beta_2, \ldots\}$, describes the job characteristics of a problem. We discuss only some of them. We use $\beta_1 = \circ$ to indicate that the jobs have the same release date and $\beta_1 = r_j$ to indicate that the jobs have different release dates. $\beta_2 = \circ$ specifies that the jobs must be processed without interruption, whereas $\beta_2 = pmtn$ specifies that preemption is allowed, i.e., we may start with the processing of a job, interrupt it, and finish it later on. The last characteristic we discuss, indicates whether setup times occur between the processing of different jobs. $\beta_3 = \circ$ indicates that no setup times occur. $\beta_3 = s_{ij}$ and $\beta_3 = s_i$ specify that sequence dependent and sequence independent setup times occur. In problems with *sequence dependent* setup times, the setup time between the processing of two consecutive jobs depends on both jobs. In problems with *sequence independent* setup times, the setup time between the processing of two consecutive jobs of different types depends only on the type of the second job.

Finally, the last field, γ, specifies the objective function that we want to minimize. Some important values for γ are $C_{\max}$, $L_{\max}$, $\sum w_j C_j$, and $\sum U_j$, which represent that the time to process all jobs, or *makespan*, the maximum lateness, the weighted sum of completion times, and the number of *tardy* jobs, i.e., jobs that finish after their due dates, respectively, must be minimized. Combinations of objective functions are also possible.

We end this section with giving the notation of two machine scheduling problems. In the second problem of Section 1.2.1, we must minimize the maximum lateness of jobs with release dates that need to be scheduled without preemption on a single machine. This problem is denoted by $1|r_j|L_{\max}$. In the problem $J3|pmtn, s_i|\sum U_j$, we must minimize the number of late jobs in a job shop with three machines. All jobs have the same release date, preemption of jobs is allowed, and sequence independent setup times occur.

1.2.3 Complexity theory

Machine scheduling problems belong to the class of *combinatorial optimization* problems in which there is a finite number of relevant solutions. Each solution is measured by an objective function and we want to find the solution with the best solution value. Since there is a finite number of relevant solutions, an obvious way to find a best solution seems to be to enumerate all solutions and store a best. This is, however, a feasible method only for very small problems. Consider, for example, the first problem of Section 1.2.1, the problem $1||L_{\max}$. For this problem, there are $n!$ relevant schedules, namely the $n!$ left-justified schedules: we have n possible jobs to schedule as the first job, then $n-1$ possible jobs for the second position in the schedule, and so on. Suppose that we could evaluate one million schedules per second. For a problem with 10 jobs, we then need less than four seconds to evaluate all possible schedules. For a problem with 15 jobs, we would already need more than 15 days to evaluate all possible schedules. For a problem with 20 jobs, we would need more than 770 centuries....

So, there is good reason to look for more efficient algorithms than complete enumeration. The *running time* of an algorithm is measured by an upper bound on the number of basic arithmetic operations, such as additions and multiplications, it needs as a function of the size $S(I)$ of an instance I. The *size of an instance* is defined as the number of symbols to represent it. For machine scheduling problems, the size of an instance is often a polynomial of n, the number of jobs, and m, the number of machines. The data to specify an instance of a problem is called the *input*. The string $p_1, d_1, p_2, d_2, \ldots, p_n, d_n$, for example, can be used to specify an instance of the problem $1||L_{\max}$. We say that the running time of an algorithm is *of order* $f(S(I))$, denoted by $O(f(S(I)))$, if there are constants c and S_0 such that the number of basic arithmetic operations is bounded from above by $c \cdot f(S(I))$ for any problem instance with $S(I) \geq S_0$. In the same way, we can measure how much *memory space* an algorithm needs for storing data. We say that a problem is *polynomially solvable*, if there exists an algorithm that requires $O(f(S(I))$ time for every instance I, with f a polynomial. Note that if an algorithm runs in polynomial time, then the space requirement is of polynomial size also.

A *decision* problem is a question to which the answer is either 'yes' or 'no'. Note that an optimization problem can easily be transformed into a finite series of decision problems by binary search over the interval $[lb, ub]$, where lb and ub are an integral lower and an integral upper bound on the optimal solution value. For the $1|r_j|L_{\max}$ problem, e.g., the associated decision problem is: 'Does there exist a feasible solution with $L_{\max} \leq k$?', with $k \in [lb, ub]$. If the logarithm of the difference between ub and lb is bounded from above by a polynomial in $S(I)$ for every instance I, then an optimization problem is polynomially solvable if the associated decision problem is polynomially solvable.

For the $1|r_j|L_{\max}$ problem, we can verify *easily*, i.e., in polynomial time, whether a *given* schedule σ has $L_{\max}(\sigma) \leq k$. The class $\mathcal{NP}$ consists of the decision problems for which we can easily verify whether a given solution is a 'yes' or a 'no' answer. An input that results in a 'yes' answer is called a *'yes' input*. The class $\mathcal{P}$ consists of all decision problems that are polynomially solvable. The decision problem associated with the problem $1||L_{\max}$ belongs to $\mathcal{P}$, because a simple sorting algorithm, which takes $O(n \log n)$ time, solves the optimization problem; see Section 1.2.1. Note that $\mathcal{P} \subseteq \mathcal{NP}$. It is widely assumed that $\mathcal{P} \neq \mathcal{NP}$: $\mathcal{NP}$ contains many difficult problems for which no polynomial algorithm has been found, in spite of all the research effort spent on them. The following definition is given by Lawler et al. [52].

Definition 1 *A problem A is polynomially reducible to problem B if and only if there exists a polynomial-time computable function τ that transforms inputs for A into inputs for B such that x is a 'yes' input for A if and only if $\tau(x)$ is a 'yes' input for B.*

Note that reducibility is a *transitive* relation: if A is polynomially reducible to B and B is polynomially reducible to C, then A is polynomially reducible to C. The following definition is useful to classify hard problems.

Definition 2 *A problem is $\mathcal{NP}$-complete if it is a member of the class $\mathcal{NP}$ and every problem in $\mathcal{NP}$ is polynomially reducible to it.*

The $\mathcal{NP}$-complete problems are the hardest problems in $\mathcal{NP}$: if *one* of the $\mathcal{NP}$-complete problems is polynomially solvable, then *all* problems in $\mathcal{NP}$ are polynomially solvable. An optimization problem is not a member of the class $\mathcal{NP}$, but it is called $\mathcal{NP}$-hard if the associated decision problem is $\mathcal{NP}$-complete. For instance, the problem $1|r_j|L_{\max}$ is $\mathcal{NP}$-hard, and hence it is very unlikely that it is polynomially solvable. Indeed, Carlier's algorithm is effective on the average but comes down to complete enumeration for certain instances.

Complexity theory is useful for machine scheduling for at least two reasons. First, if a problem is $\mathcal{NP}$-hard, then for optimization it is justified to develop enumerative algorithms, because we cannot expect to find a polynomial optimization algorithm. Enumerative algorithms, however, may take much computation time. Second, if a problem is $\mathcal{NP}$-hard, then it is justified to use approximation algorithms, too. We then may find good quality solutions in reasonable time. For more information on complexity theory, we refer to Garey and Johnson [31], Cook [22], and Karp [47].

1.3 Overview of the thesis

The problem of scheduling jobs in a machine shop is often modelled as a job shop scheduling problem. Chapter 2 introduces this problem and discusses optimization as well as approximation algorithms. Among the latter is the Shifting Bottleneck (SB) procedure of Adams et al. [2]. It is an intuitively appealing algorithm that decomposes the job shop scheduling problem into a series of $1|r_j|L_{\max}$ problems, which can be solved effectively by Carlier's algorithm (Carlier [17]). The SB procedure generates schedules of good quality and requires relatively little computation time, which is a necessary condition for most practical applications. The job shop scheduling problem is, however, “clean” in that it ignores most of the practical side constraints, such as simultaneous resource requirements, setup times, and convergent job routings. A nice feature of the SB procedure is that it can be extended to include such practical side constraints without much algorithmic adjustment. We call a machine shop with practical features a *practical* job shop. Chapter 3 reviews various extensions to the SB procedure. For each extension, we describe how to model it. For some extensions, the decomposition of a practical job shop problem results in machine scheduling problems other than the $1|r_j|L_{\max}$ problem. For example, if one of the machines is a machine with sequence independent setup times, then the decomposition results for this machine in the problem $1|r_j, s_i|L_{\max}$. Chapter 4 discusses an optimization algorithm for this problem in detail, whereas Chapter 5 deals with the identical parallel machine equivalent of this problem, i.e., the problem $P|r_j, s_i|L_{\max}$. An important practical extension of the SB procedure is the extension that allows assembly of components to other components or end products. Chapter 6 presents our computational experiences with the SB procedure with this extension. The SB procedure with extensions is part of a commercial scheduling system called JOBPLANNER. In Chapter 7, we discuss this system and experiences with it at Cityprint B.V., a manufacturer of printed circuit boards. Finally, in Chapter 8, we end with some conclusions and recommendations for further research.

Chapter 2

Job shop scheduling

2.1 Introduction

In job shop scheduling, we consider a shop consisting of m machines $M_1, M_2, \ldots, M_m$ on which a set of n jobs $J_1, J_2, \ldots, J_n$ needs to be processed. Each machine is available for processing from time 0 onwards and can process at most one job at a time. Each job J_j consists of a *chain* of operations $O_{1j}, O_{2j}, \ldots, O_{n_j,j}$, where n_j denotes the number of operations of job J_j. Operation O_{1j} is available from time 0 onwards, whereas operation O_{ij} can only be processed after the completion of operation $O_{i-1,j}$ $(i = 2, \ldots, n_j)$. Operation O_{ij} $(j = 1, 2, \ldots, n; i = 1, 2, \ldots, n_j)$ needs uninterrupted processing on a given machine μ_{ij} during a given non-negative time p_{ij}. The objective usually considered in the literature is to find a schedule that minimizes the *makespan* $C_{\max}$, that is, to find a schedule in which the time to process all jobs is minimal. Note that we may restrict ourselves again to left-justified schedules. This job shop problem is denoted as $J||C_{\max}$.

The job shop scheduling problem is $\mathcal{NP}$-hard and also difficult to solve to optimality from an empirical point of view. In fact, it is one of the hardest combinatorial optimization problems. For example, a problem with only 10 jobs and 10 machines, proposed by Fisher and Thompson [27], remained unsolved for almost 25 years, in spite of the research effort spent on it. Many solution approaches have been proposed to solve the job shop scheduling problem. Most of them make use of a disjunctive graph to represent an instance. In the next section, we describe this graph, which is due to Roy and Sussman [68], in detail. In

Section 2.3, we discuss the most prominent algorithms for the job shop scheduling problem. These algorithms include branch-and-bound algorithms and local search algorithms based on taboo search and simulated annealing. The Shifting Bottleneck (SB) procedure of Adams et al. [2] is a local search algorithm that decomposes the job shop problem into a series of single-machine scheduling problems. Since the SB procedure is the basis of the next chapters, we devote a separate section to this procedure. In Section 2.5, we end this chapter with some conclusions.

2.2 Disjunctive graph representation

Each instance of the job shop scheduling problem of minimizing makespan can be represented by a disjunctive graph $G = (V, A, E)$, with V a set of nodes, A a set of arcs, and E a set of orientable edges. For each operation O_{ij}, V contains a node v_{ij} with weight p_{ij}; V also contains two auxiliary nodes s and t, both with weight 0. V is equal to $\{v_{ij} \mid j = 1, 2, \ldots, n; i = 1, 2, \ldots, n_j\} \cup \{s, t\}$. If we denote an *arc* from node α to node β by $\langle \alpha, \beta \rangle$, then $A = \{\langle s, v_{1j} \rangle \mid j = 1, 2, \ldots, n\} \cup \{\langle v_{ij}, v_{i+1,j} \rangle \mid j = 1, 2, \ldots, n; i = 1, 2, \ldots, n_j - 1\} \cup \{\langle v_{n_j,j}, t \rangle \mid j = 1, 2, \ldots, n\}$, that is, A contains an arc from the source s to each node that represents any first operation, from each node representing an operation to the node that represents the next operation of the same job, and from each node that represents any last operation to the sink t. We denote an *edge* between nodes α and β by (α, β). Note that we have $(\alpha, \beta) = (\beta, \alpha)$, because an edge is not oriented. E consists of edges between nodes corresponding to operations that need to be processed on the same machine, i.e., $E = \{(v_{ij}, v_{gh}) \mid j, h = 1, 2, \ldots, n; i = 1, 2, \ldots, n_j; g = 1, 2, \ldots, n_h; v_{ij} \neq v_{gh}; \mu_{ij} = \mu_{gh}\}$. The weights of all arcs and edges are 0.

Table 2.1 shows the data of an instance of the job shop problem with three machines and three jobs, where each job consists of three operations. The objective is to minimize the makespan. Figure 2.1

J_j	μ_{1j}	μ_{2j}	μ_{3j}	p_{1j}	p_{2j}	p_{3j}
J_1	M_1	M_3	M_2	4	7	6
J_2	M_2	M_1	M_3	3	5	8
J_3	M_3	M_2	M_1	2	6	7

Table 2.1: Data for example instance.

shows the graph corresponding to this instance.

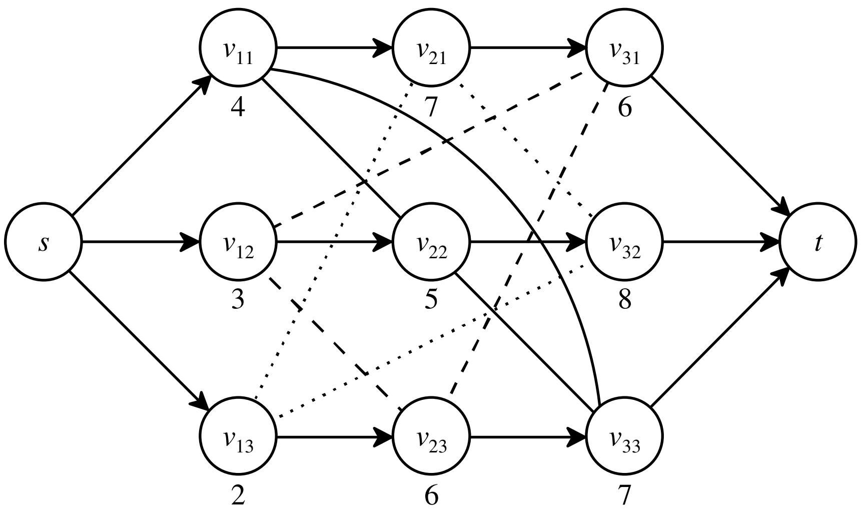


Figure 2.1: Graph representing example instance.

Let $\mathcal{S} = \langle a_1, a_2, \dots, a_k \rangle$ be any sequence of arcs, where $k \geq 1$ and $a_i = \langle \alpha_i, \beta_i \rangle$ $(i = 1, 2, \dots, k)$. $\mathcal{S}$ is called a *path from* α_1 *to* β_k if $\beta_i = \alpha_{i+1}$ for $i = 1, 2, \dots, k-1$. Assume now that $\mathcal{S}$ is a path. We also use the notation $\langle \alpha_1, \alpha_2, \dots, \alpha_k, \beta_k \rangle$ to denote $\mathcal{S}$. The length $l(\mathcal{S})$ of $\mathcal{S}$ is defined as $l(\mathcal{S}) = l_{a_1} + \sum_{i=2}^{k} (l_{a_i} + w_{\alpha_i})$, where l_{a_i} denotes the weight (or length) of arc a_i and w_{α_i} the weight of node α_i. Note that the length of $\mathcal{S}$ does not include the weights of the nodes α_1 and β_k and that for the job shop problem $l_a = 0$ for all arcs $a \in A$. In the next chapter, we will see some examples with $l_a \neq 0$. A *cycle* is a path from a node α to itself. A graph is *acyclic* if it contains no cycle. Let now $E' \subseteq E$. A set A' of arcs is an *orientation* of E' if

$$(\alpha, \beta) \in E' \Leftrightarrow \langle \alpha, \beta \rangle \in A' \text{ or } \langle \beta, \alpha \rangle \in A'.$$

A *complete* orientation is an orientation of E; a *partial* orientation is an orientation of some $E' \subsetneq E$. An orientation A' is *feasible* if the directed graph $D_{A'} = (V, A \cup A')$ is acyclic. An arc $\langle v_{ij}, v_{gh} \rangle$ in $D_{A'}$ represents that operation O_{ij} is processed before operation O_{gh}. The crux is now that each feasible complete orientation induces a unique left-justified schedule for the job shop problem. Also, each left-justified schedule for the job shop problem uniquely induces a feasible complete orientation.

The arcs of A' are called the *machine arcs*, because they represent the sequences of the operations on the machines. The length of a *longest* path from s to t, that is, a path from s to t which has maximal length, equals the makespan of the induced schedule. Figure 2.2 displays a feasible schedule with the sequence $O_{11} - O_{22} - O_{33}$ on machine M_1, the sequence $O_{12} - O_{23} - O_{31}$ on machine M_2, and the sequence $O_{13} - O_{21} - O_{32}$ on machine M_3 for the instance for which the data can be found in Table 2.1. For convenience, we left out the machine arcs that

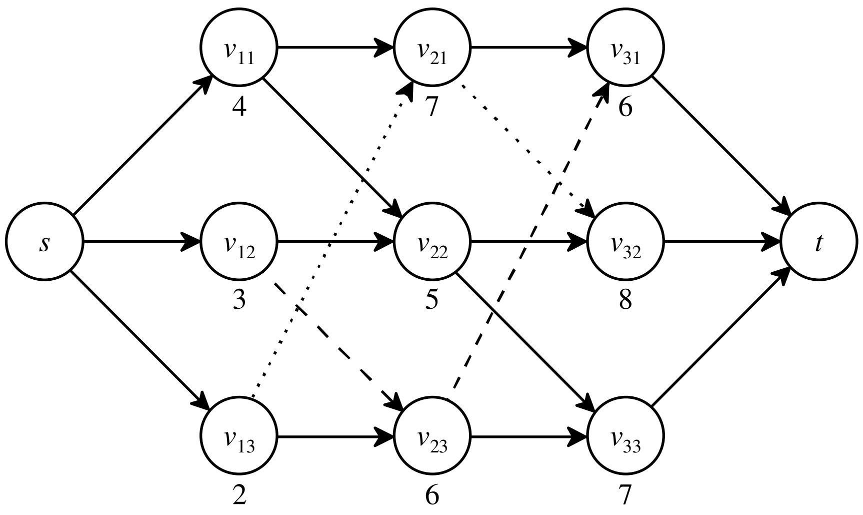


Figure 2.2: Graph representing a solution.

are induced by *transitivity*; for example, we left out the arc $\langle v_{11}, v_{33} \rangle$, because machine arcs $\langle v_{11}, v_{22} \rangle$ and $\langle v_{22}, v_{33} \rangle$ imply that operation O_{11} is processed before operation O_{33}.

2.3 Solution procedures

Algorithms for optimization problems can be divided in two classes: *exact* algorithms and *approximation* algorithms. Exact algorithms produce optimal solutions, but their running time cannot be bounded from above by a polynomial in the size of an instance for $\mathcal{NP}$-hard problems if $\mathcal{P} \neq \mathcal{NP}$. Approximation algorithms generally produce solutions in relatively little computation time, but the solutions need not be optimal. In the next subsection, we discuss exact algorithms for the job shop problem. Section 2.3.2 reports on approximation algorithms.

2.3.1 Exact solution procedures

The most straightforward procedure to find an optimal solution is *complete enumeration*. This approach evaluates all feasible solutions and stores a best. At the completion of the algorithm, an optimal solution has been stored. For the job shop problem, this comes down to enumerating each feasible complete orientation and evaluating the length of a longest path from s to t in the resulting directed graph.

The algorithm in Figure 2.3 enumerates all feasible complete orientations. We assume that $E = \{e_1, e_2, \ldots, e_{|E|}\}$, with $|E|$ the number of elements in E. The two possible orientations of edge e_i are denoted by $a_{i,1}$ and $a_{i,2}$, respectively. We denote the length of a longest path from s to t in the graph D_{A_i} by $l(A_i)$. During the execution of the algorithm, A_i is an orientation, $\mathcal{SP}$ a set of orientations that need to be examined, and ub the best solution value found so far. In the first three steps, the algorithm initializes these variables. The other steps constitute the main loop of the algorithm. The algorithm stops when all feasible orientations have been examined, i.e., when $\mathcal{SP} = \emptyset$. After choosing an orientation $A_i \in \mathcal{SP}$, it checks whether A_i is a feasible orientation, that is, whether D_{A_i} is acyclic. If A_i is feasible, then the algorithm also checks whether A_i is a complete orientation and, if so, whether A_i is better than the best solution found so far. If A_i was not a complete orientation, then a non-oriented edge e_j is chosen and the two orientations A_{2i+1} and A_{2i+2} are added to $\mathcal{SP}$. A_{2i+1} is orientation A_i plus the first orientation $a_{j,1}$ of e_j. In the same way, A_{2i+2} is A_i plus the second orientation $a_{j,2}$ of e_j. The process of creating the orientations A_{2i+1} and A_{2i+2} from A_i can be visualized in a graph. Each set A_i has a node i in this graph; there are edges between A_i and A_{2i+1} and between A_i and A_{2i+2}, indicating that A_{2i+1} and A_{2i+2} are created from A_i by adding one arc to A_i. If the feasibility check applies only to complete orientations, then we would obtain the graph in Figure 2.4, which is called the *search tree*. The complete enumeration algorithm needs an algorithm to determine whether the graph D_{A_i} contains a cycle and the length of a longest path from s to t if A_i is a feasible complete orientation. The algorithm in Figure 2.5 does both. We assume that $V = \{v_0 = s, v_1, v_2, \ldots, v_N, v_{N+1} = t\}$. During the execution of the

Algorithm 2.1

Enumeration of all feasible complete orientations.

```
 1. $A_0 \leftarrow \emptyset$;
 2. $\mathcal{SP} \leftarrow \{A_0\}$;
 3. $ub \leftarrow \infty$;
 4. WHILE $\mathcal{SP} \neq \emptyset$ DO
 5. BEGIN
 6.   choose $A_i \in \mathcal{SP}$;
 7.   $\mathcal{SP} \leftarrow \mathcal{SP} \setminus \{A_i\}$;
 8.   IF $D_{A_i}$ is acyclic THEN
 9.   BEGIN
10.     IF $A_i$ is a complete orientation THEN
11.       IF $l(A_i) < ub$ THEN $ub \leftarrow l(A_i)$;
12.     ELSE
13.     BEGIN
14.       choose non-oriented edge $e_j$;
15.       $A_{2i+1} \leftarrow A_i \cup \{a_{j,1}\}$;
16.       $A_{2i+2} \leftarrow A_i \cup \{a_{j,2}\}$;
17.       $\mathcal{SP} \leftarrow \mathcal{SP} \cup \{A_{2i+1}, A_{2i+2}\}$;
18.     END
19.   END
20. END
```

Figure 2.3: Algorithm for enumerating all feasible complete orientations.

algorithm, a node v_j is labeled when the length of a longest path from s to v_j has been computed. We say that a node v_j *has been considered* if v_j has been chosen in Step 5 of the algorithm. This means that if node v_j has been considered, then its weight and the length of its outgoing arcs $\langle v_j, v_k \rangle \in A \cup A_i$ are taken into account for computing the length of longest paths to the other nodes. Also, we then say that the arcs $\langle v_j, v_k \rangle \in A \cup A_i$ *have been considered.* We denote the set of labeled nodes that have *not* yet been considered by $\mathcal{NC}$. The algorithm initializes by labeling s and setting $\mathcal{NC} = \{v_0\}$, with $v_0 = s$. The main loop of the algorithm iterates until $\mathcal{NC} = \emptyset$, i.e., until there are no more nodes that are labeled but not considered. After choosing a node $v_j \in \mathcal{NC}$, we

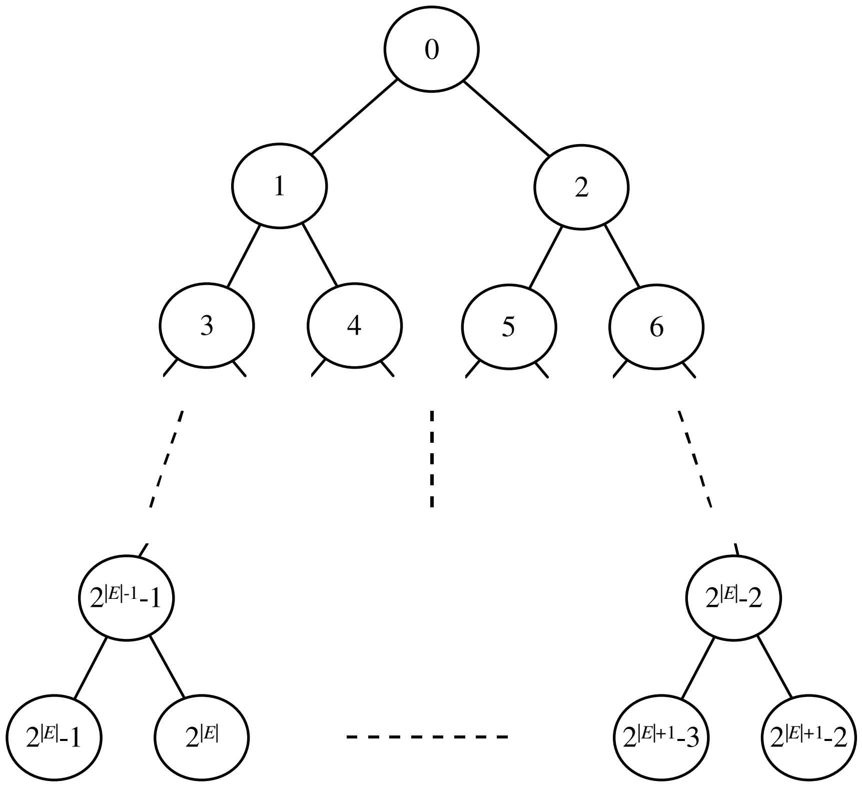


Figure 2.4: Search tree.

update the longest paths from s to nodes v_k for which $\langle v_j, v_k \rangle \in A \cup A_i$. If all nodes v_l with $\langle v_l, v_k \rangle$ are labeled and considered, then we label v_k and add it to $\mathcal{NC}$, because now the length of a longest path from s to v_k has been definitively computed.

Theorem 2.1 *The graph D_{A_i} is acyclic if and only if the algorithm in Figure 2.5 ends with node t labeled.*

Proof. $(\Rightarrow)$: Suppose that t is not labeled. Let C denote the set of unlabeled nodes, that is $v_k \in C$ if v_k has an ingoing arc that has not been considered. Let $\langle v_j, v_k \rangle$ be such an arc. Then also $v_j \in C$, because otherwise we would have considered $\langle v_j, v_k \rangle$. Since each $v_k \in C$ has at least one ingoing arc $\langle v_j, v_k \rangle$ that has not been considered, the number of arcs that have not been considered is at least $|C|$. Also, if $\langle v_j, v_k \rangle$ is

Algorithm 2.2

Computing longest paths in D_{A_i}.

```
 1. label $v_0$;
 2. $\mathcal{NC} \leftarrow \{v_0\}$;
 3. WHILE $\mathcal{NC} \neq \emptyset$ DO
 4. BEGIN
 5.   choose $v_j \in \mathcal{NC}$;
 6.   $\mathcal{NC} \leftarrow \mathcal{NC} \setminus \{v_j\}$;
 7.   FOR $\forall v_k$ with $\langle v_j, v_k\rangle \in A \cup A_i$ DO
 8.   BEGIN
 9.     update longest path from $s$ to $v_k$;
10.     IF $\forall v_l$ with $\langle v_l, v_k\rangle \in A \cup A_i$ : $v_l$ is labeled THEN
11.     BEGIN
12.       $\mathcal{NC} \leftarrow \mathcal{NC} \cup \{v_k\}$;
13.       label $v_k$;
14.     END
15.   END
16. END
```

Figure 2.5: Algorithm for computing longest paths.

an unconsidered arc, then $v_j \in C$ and $v_k \in C$. C must then contain a cycle.

($\Leftarrow$) : Suppose that D_{A_i} contains a cycle $K = \langle v_{j_1}, v_{j_2}, \ldots, v_{j_p}, v_{j_1}\rangle$. Then, $s \notin K$, because s has no incoming arcs. Initially, all nodes $v_j \neq s$ are unlabeled. If the algorithm would label node v_{j_1}, then node v_{j_p} must be labeled. If node v_{j_p} is labeled, then also node $v_{j_{p-1}}$ must be labeled. If we continue this reasoning, then the algorithm would label v_{j_1}, only if it has been labeled already. This is a contradiction. Therefore, all nodes in K are not labeled. Since there exists a path from each node to t, t cannot be labeled. □

Note that this algorithm computes not only the length of a longest path from s to t, but also the length of a longest path from s to node v_j $(j = 1, 2, \ldots, N)$. The time complexity of the algorithm is $O(|E|)$.

A serious disadvantage of complete enumeration for job shop scheduling is that much computation time is spent on partial orientations that will not result in a better solution than the current best. A class of exact algorithms that tries to avoid this is the class of *branch-and-bound* algorithms. In Figure 2.6, a simple branch-and-bound algorithm is given in which *ub* denotes an *upper bound* on the optimal makespan. Initially, *ub* can be set to the solution value of any feasible solution.

Algorithm 2.3

Branch-and-bound algorithm

```
1.  $ub \leftarrow$ solution value of any feasible solution;
2.  $\mathcal{SP} \leftarrow \{\Omega_0\}$;
3.  WHILE $\mathcal{SP} \neq \emptyset$ DO
4.  BEGIN
5.    Choose $SP \in \mathcal{SP}$;
6.    $\mathcal{SP} \leftarrow \mathcal{SP} \setminus \{SP\}$;
7.    IF $SP$ is a promising set of solutions THEN
8.    BEGIN
9.      IF a better solution found THEN
10.       $ub \leftarrow$ new solution value;
11.     partition $SP$ into subsets $SP_1, SP_2, \ldots, SP_k$;
12.     $\mathcal{SP} \leftarrow \mathcal{SP} \cup \{SP_1, SP_2, \ldots, SP_k\}$;
13.   END
14. END
```

Figure 2.6: Branch-and-bound algorithm.

Clearly, the optimal solution value is at most *ub*. During the execution of the algorithm, *ub* is the solution value of the best feasible solution found so far. A feasible solution can easily be found, for example by priority rules, which we discuss in the next subsection. Ω_0 denotes the set of all feasible solutions. In the steps of the WHILE loop, a set SP of feasible solutions is examined. First, we decide whether SP may contain an optimal solution. A necessary condition for this is that SP is a non-empty set. Also, a *lower bound* on the solution values of solutions in SP is computed. If this lower bound is at least *ub*, then no solution in SP has a better solution value than *ub*. Computing a lower bound is the *bounding* part of branch-and-bound algorithms. If we now have a solution with a solution value smaller than *ub*, then *ub* is set to this solution value: it is the best solution found so far. The *branching* part of the branch-and-bound algorithm is the partitioning of solutions in SP into subsets of solutions $SP_1, SP_2, \ldots, SP_k$. Of course, we only do this if SP contains more than one solution and if it may contain an optimal solution. Otherwise, it is *fathomed*: we do not examine solutions in SP any more.

There are some important implementation aspects of the branch-and-bound algorithm in Figure 2.6, including the effort to be spent on computing upper and lower bounds. We can say that the more time spent on computing bounds, the more nodes can be fathomed, and hence the fewer sets of solutions are examined. There is a trade-off involved, however: if the computation of bounds takes a lot of time, then the overall computation time might be larger. Another important aspect is the branching strategy, i.e., the strategy to choose the set of solutions to be examined next. If we make a good choice, then early in the execution of the algorithm we may find a better upper bound ub, which also may help to restrict the growth of the tree.

Let us now discuss a possible implementation of the branch-and-bound algorithm for the job shop problem. Let Ω_0 denote the set of all feasible complete orientations. We branch from this set by orienting e_1: Ω_1 is the set of all feasible complete orientations where e_1 is oriented as $a_{1,1}$, i.e., $\Omega_1 = \{\bar{A} \in \Omega_0 \mid \bar{A} \supset \{a_{1,1}\}\}$, and $\Omega_2 = \{\bar{A} \in \Omega_0 \mid \bar{A} \supset \{a_{1,2}\}\}$. Then, $\Omega_0 = \Omega_1 \cup \Omega_2$. In the same way, we branch from Ω_1 by orienting e_2. We then get the sets Ω_3 and Ω_4, where $\Omega_3 = \{\bar{A} \in \Omega_1 \mid \bar{A} \supset \{a_{2,1}\}\}$ and $\Omega_4 = \{\bar{A} \in \Omega_1 \mid \bar{A} \supset \{a_{2,2}\}\}$. Then, $\Omega_1 = \Omega_3 \cup \Omega_4$. This process continues until we get the sets $\Omega_{2^{|E|}-1}, \Omega_{2^{|E|}}, \ldots, \Omega_{2^{|E|+1}-2}$; each of these sets contains one complete orientation. Note that not all complete orientations are feasible; therefore, Ω_i might be an empty set. Note also that this process is the same as we discussed earlier for complete enumeration. The nodes $2i+1$ and $2i+2$ in the search tree are called *immediate child nodes* of node i; node i is the *parent node* of $2i+1$ and $2i+2$. A node j is called a *child node* of node i, if we obtain node j by branching one or more times from node i. Node 0 is called the *root* of the search tree which is also called the *branch-and-bound tree*.

Complete enumeration algorithms fathom a node i only if Ω_i is empty. In the branch-and-bound algorithm, we also compute a lower bound on the makespan for solutions in Ω_i. If the lower bound is large enough, that is, if the lower bound is at least ub, then we stop examining this node and all of its childs. Moreover, most branch-and-bound algorithms use *dominance criteria* that specify conditions to conclude that a set Ω_i of solutions does not contain a better solution than the best one in Ω_j for some i and j. The node that corresponds to Ω_i can then be fathomed.

Carlier and Pinson [19] were the first to solve the famous 10×10 instance proposed by Fisher and Thompson [27]. Let A_i be the set of oriented edges that is considered in node i of the branch-and-bound tree. Carlier and Pinson use the well-known lower bound that is found by solving for each machine in the job shop an instance of $1|r_j, pmtn|L_{\max}$ in which the release and due dates for the operations follow from longest path computations in the graph D_{A_i}. They branch from each node by orienting one of the edges. If possible, an edge is chosen between operations on the machine that had the largest lower bound in the root of the branch-and-bound tree. Carlier and Pinson derive conditions that ensure that in any optimal complete orientation $\bar{A}$ with $\bar{A} \supset A_i$ some edges have a fixed orientation. If this is the case for some edge e, then we do not need to branch on this edge. Instead, we add the fixed orientation of e to A_i. The sequence in which they examine the nodes of the branch-and-bound tree is the sequence in which the associated subproblems are created.

Applegate and Cook [4] use the same branching scheme and lower bounds as Carlier and Pinson. Let A_i again be the set of oriented edges that is considered in node i. They restrict their attention to edges to branch on to edges between operations on the machine that had the worst lower bound in the root of the branch-and-bound tree, until that machine is completely scheduled; after that, they consider the other edges. Suppose that we may branch on edge e_k, which results in the sets $A_i \cup \{a_{k,1}\}$ and $A_i \cup \{a_{k,2}\}$. The edge e_k that we may branch on and for which $\min\{lb(A_i \cup \{a_{k,1}\}), lb(A_i \cup \{a_{k,2}\})\}$ is maximum is chosen to branch on, where $lb(A_i)$ is a lower bound for solutions $\bar{A}$ with $\bar{A} \supset A_i$. The next subproblem they examine is the one that has the smallest lower bound. Applegate and Cook extend the idea of Carlier and Pinson to fix the orientation of some unoriented edges.

Brucker et al. [13] propose a different branching scheme to solve the job shop problem. Each node in their branch-and-bound tree corresponds to a feasible complete orientation. This branching scheme is based on the *block* approach of Grabowski et al. [36] for a single-machine scheduling problem with release and due dates. Brucker et al. prove that if the schedule in a node of the search tree is not optimal, then at least one operation in a block has to be processed either before the first, or after the last operation of the corresponding block. Apart from some simple lower bounds, they use the preemptive single-machine bound.

State-of-the-art branch-and-bound algorithms are able to solve instances with up to 15 jobs and 15 machines. The main problem in solving larger instances seems to be the lack of good lower bounds.

2.3.2 Approximation algorithms

Since exact algorithms can solve only relatively small instances of the job shop problem, various approximation algorithms have been proposed. We discuss two classes of approximation algorithms: priority rules and local search algorithms.

The algorithm in Figure 2.7 is a generic algorithm for scheduling jobs in a job shop by a *priority rule*. In this algorithm, $\mathcal{T}$ is a set of completion times of operations. The sets Q_{M_k} are sets of operations available to be processed on machine M_k, that is, unprocessed operations O_{ij} for which $\mu_{ij} = M_k$, while the operations $O_{1j}, O_{2j}, \ldots, O_{i-1,j}$ have already been processed. At each completion time on a machine M_k, the next operation to be processed is chosen by means of a function that assigns a priority to each available operation. Common functions are earliest due date, first in-first out, most work remaining, and shortest processing time. Priority rules require little computation time, are easy to implement, but in general generate schedules of poor quality. In Chapter 6 where we consider assembly shops with setup times, we will see that priority rules are significantly outperformed by a more sophisticated approach. For a survey of priority rule based scheduling, we refer to Haupt [39].

The second class of approximation algorithms we discuss is the class of *local search* algorithms. Local search algorithms usually start with a feasible solution and try to improve this solution by making small changes to it, e.g., reversing the sequence of two consecutive operations or giving a single operation a new position in the schedule. A solution that we obtain after making one such a change to a solution σ is called a *neighbor* of σ. The set of all neigbors of σ is called the *neighborhood*

Algorithm 2.4

Priority rule algorithm for scheduling jobs in a job shop

```
1.  $\forall M_i : Q_{M_i} \leftarrow \{O_{1j} \mid 1 \leq j \leq n, \mu_{1j} = M_i\}$;
2.  $N \leftarrow$ total number of operations;
3.  $\mathcal{T} \leftarrow \{0\}$;
4.  WHILE $N > 0$ DO
5.  BEGIN
6.    $T \leftarrow \min\{t \in \mathcal{T}\}$;
7.    IF at time $T$ an operation $O_{ij}$ finishes on a machine THEN
8.      IF $O_{ij}$ is not the last operation of job $J_j$ THEN
9.        $Q_{\mu_{i+1,j}} \leftarrow Q_{\mu_{i+1,j}} \cup \{O_{i+1,j}\}$;
10.   IF at time $T$ a machine $M_k$ is available with operations waiting
11.   THEN
12.   BEGIN
13.     Choose operation $O_{gh}$ from $Q_{M_k}$ with highest priority;
14.     Schedule $O_{gh}$ in the interval $[T, T + p_{gh}]$;
15.     $\mathcal{T} \leftarrow \mathcal{T} \cup \{T + p_{gh}\}$;
16.     $Q_{M_k} \leftarrow Q_{M_k} \setminus \{O_{gh}\}$;
17.     $N \leftarrow N - 1$;
18.   END
19.   ELSE $\mathcal{T} \leftarrow \mathcal{T} \setminus \{T\}$;
20. END
```

Figure 2.7: Priority rule algorithm for the job shop problem.

of σ. The simplest local search algorithm is *iterative improvement* that starts with a feasible solution and searches the neighborhood of this solution for a better one. If a better solution is found, then the neighborhood of this solution is searched for a better solution. This process continues until we have a solution that is better than all its neighbors. Such a solution is a *local optimum*. Local optima found with iterative improvement are often poor quality solutions. An approach to resolve this situation may be to run the same algorithm more than once, each time with a different starting solution.

Another type of local search algorithm is *taboo search*. The idea is to avoid bad local optima by moving to a worse solution if no better exists in the neighborhood of the current solution. To avoid jumping from one solution to the other and back again, a *taboo list* is introduced. We cannot move to a solution on the taboo list, unless some *aspiration criterion* is satisfied. After each move the taboo list is updated. A typical taboo list consists of the seven last changes made. The criterion to stop the algorithm is often an upper bound on the running-time or on the number of iterations without improvement. For details, we refer to Glover [32, 33] and Glover et al. [34]. For applications of taboo search to job shop problems, see, e.g., Dell'Amico and Trubian [24] and Nowicki and Smutnicki [60].

Simulated annealing is a local search algorithm that randomly picks a neighbor of the current solution. If the neighbor is a better solution than the current one, then this neighbor becomes the current solution. Otherwise, it becomes the current solution with a certain probability. This probability depends on the difference between the value of the current solution and the value of the neighbor, and on a certain control parameter, often called the temperature, whose value decreases during the execution of the algorithm. A low temperature makes it unlikely that we move to a neighbor with a worse solution value. For applications of simulated annealing to job shop problems, see, for instance, Van Laarhoven et al. [49] and Aarts et al. [1].

We refer to Vaessens et al. [79] for a computational study of the performance of the most prominent local search algorithms for the job shop problem. They discuss a number of other local search algorithms, such as genetic algorithms, variable-depth search, and a number of variants of the Shifting Bottleneck procedure of Adams et al. [2]. The strength of the SB procedure is that it is an intuitive algorithm that generates good schedules in reasonable time. Also, it can easily be generalized to deal with practical features, such as transportation and non-availability times of the machines. The next section discusses the original Shifting Bottleneck procedure.

2.4 Shifting Bottleneck procedure

The Shifting Bottleneck (SB) procedure decomposes the job shop problem into a series of single-machine subproblems. It schedules the machines one by one and focuses on bottleneck machines. Like branch-and-bound algorithms for the job shop problem, the SB procedure heavily relies on longest path computations in the graph $D_{A'}$ discussed in Section 2.2:

- the length of a longest path from node s to node v_{ij} defines the earliest possible starting time of operation O_{ij}, that is, it defines a *release date* r_{ij} for operation O_{ij};
- the length of a longest path from node v_{ij} to node t equals the minimum time the shop needs to process all jobs after the completion of operation O_{ij}, that is, it defines a *run-out time* q_{ij} for operation O_{ij};
- if all machines are scheduled, then the length of a longest path from s to t equals the makespan of this schedule.

In Figure 2.8, a description of the SB procedure is given. The initial-

Algorithm 2.5

Shifting Bottleneck procedure.

```
1. label every machine as a non-bottleneck machine;
2. FOR $i = 1$ TO $M$ DO
3. BEGIN
4.    compute longest paths, resulting in release dates and run-out times;
5.    schedule the non-bottleneck machines;
6.    let $M_k$ be the machine with the largest resulting makespan;
7.    label $M_k$ as a bottleneck machine;
8.    fix schedule of $M_k$;
9.    optimize bottleneck machines;
10. END
```

Figure 2.8: Shifting Bottleneck procedure.

ization of the procedure is to label every machine as a non-bottleneck machine. During each iteration of the main loop, one new bottleneck machine is chosen.

First, the longest paths are computed in the directed graph $D_{A'}$, where A' consists of the machine arcs representing the schedules on the bottleneck machines. We slightly modify the definition of A': until now, we represented the schedule $O_{gh} - O_{ij} - O_{kl}$ on a machine by the arc set $\{\langle v_{gh}, v_{ij}\rangle, \langle v_{gh}, v_{kl}\rangle, \langle v_{ij}, v_{kl}\rangle\}$; now, we represent it by the arc set $\{\langle v_{gh}, v_{ij}\rangle, \langle v_{ij}, v_{kl}\rangle\}$. The arc $\langle v_{gh}, v_{kl}\rangle$ is induced by transitivity. In this way, each node v_{ij} has *outdegree* at most two, i.e., there are at most two arcs that start in v_{ij}. Therefore, there are $O(N)$ arcs in $D_{A'}$, and the longest path algorithm in Figure 2.5 runs then in $O(N)$ time. The longest path computations result in release dates and run-out times for all operations.

Next, all non-bottleneck machines are scheduled. A number of $1|r_j, q_j|C_{\max}$-problems need then to be solved. This problem is $\mathcal{NP}$-hard, but it can be solved effectively with the algorithm proposed by Carlier [17]. Adams et al. use this optimization algorithm to solve these single-machine scheduling problems. Note that if we give each operation O_{ij} a due date $d_{ij} = -q_{ij}$, then each problem is equivalent to the $1|r_j|L_{\max}$ problem. The machine with the largest resulting makespan, or maximum lateness, becomes the new bottleneck machine. The schedule of this machine is fixed by adding the machine arcs representing it to A'.

The bottleneck machines are *rescheduled* in a bottleneck optimization procedure, which consists of a number of cycles. In each cycle, every bottleneck machine is rescheduled once. In the first cycle, the sequence in which the bottleneck machines are rescheduled is the sequence in which they were labeled as a bottleneck machine. In the other cycles, we use the sequence we get by reordering the bottleneck machines according to non-increasing solution values of the single-machine problems in the latest cycle. If a machine is rescheduled, then its machine arcs are deleted from A', release and due dates for operations on this machine are recomputed, the machine is scheduled according to the current release and due dates, and the machine arcs representing the new schedule are added to A'. If an improvement was found during a cycle, then another cycle is performed. If not all machines are bottleneck machines, however, then Adams et al. limit the number of cycles to three. The last step in the bottleneck optimization procedure is to reschedule successively a number of *non-critical* bottleneck machines, that is, bottleneck machines that have no operations on a longest path from s to t in the graph $D_{A'}$. The authors suggest to set the number of non-critical machines to reschedule equal to the $\min\{\sqrt{M_0}, M_0'\}$, where M_0 is the number of bottleneck machines and M'_0 the number of non-critical machines.

2.4.1 An example

In this subsection, we demonstrate the SB procedure by applying it to an instance with three machines and two jobs, where both jobs consist of three operations. The data of this instance can be found in Table 2.2. Figure 2.9 gives the graph G, representing this instance. For conve-

J_j	μ_{1j}	μ_{2j}	μ_{3j}	p_{1j}	p_{2j}	p_{3j}
J_1	M_1	M_2	M_3	2	3	1
J_2	M_1	M_3	M_2	3	2	3

Table 2.2: Data for example instance.

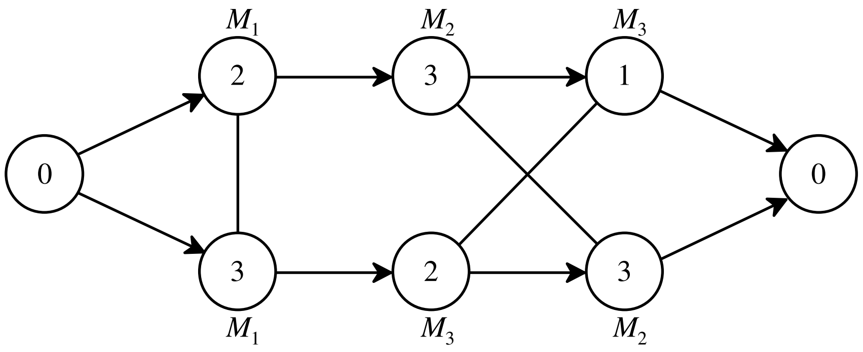


Figure 2.9: Graph representing example instance.

nience, we have included the weights in the nodes. The first step in the SB procedure is to label all machines as non-bottleneck machines. Next, the length of the longest paths are computed in the graph $D_{A'}$, with $A' = \emptyset$ since all machines are non-bottleneck machines. The graph $D_{A'}$ is the graph G with all edges removed. The data of the resulting single-machine problems are denoted in Table 2.3. The data in the columns 'r', 'p', and 'd' represent the release dates, the processing times, and the due dates of the operations, respectively. The optimal schedules for the single-machine problems are given underneath the data, together with the optimal maximum lateness. We see that M_1 has the largest resulting maximum lateness and we label M_1 as a bottleneck machine. Since

M_1	r	p	d
J_1	0	2	-4
J_2	0	3	-5

$\sigma_1 = \langle J_2 - J_1 \rangle$
$L_{\max}(\sigma_1) = 9$

M_2	r	p	d
J_1	2	3	-1
J_2	5	3	0

$\sigma_2 = \langle J_1 - J_2 \rangle$
$L_{\max}(\sigma_2) = 8$

M_3	r	p	d
J_1	5	1	0
J_2	3	2	-3

$\sigma_3 = \langle J_2 - J_1 \rangle$
$L_{\max}(\sigma_3) = 8$

Table 2.3: Data of the first single-machine problems.

there is only one machine labeled as a bottleneck machine, we do not perform the bottleneck optimization step. We fix the schedule on M_1 by adding the machine arc representing this schedule to A', see Figure 2.10.

The release and due dates are now recomputed, which results in the

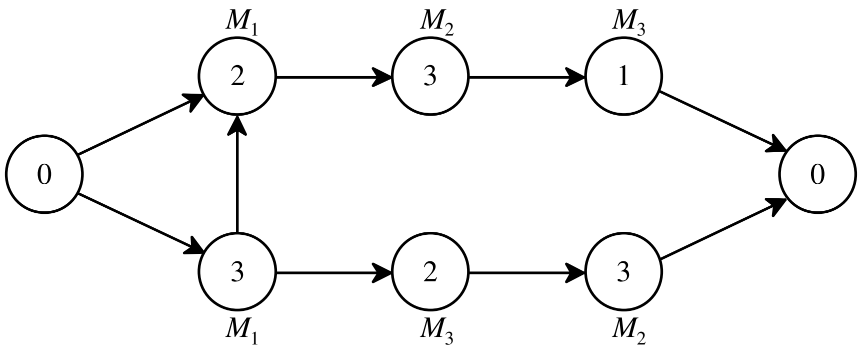


Figure 2.10: The graph $D_{A'}$ after fixing the schedule on M_1.

single-machine problems found in Table 2.4. Again, the optimal sched-

M_2	r	p	d
J_1	5	3	-1
J_2	5	3	0

$\sigma_2 = \langle J_1 - J_2 \rangle$
$L_{\max}(\sigma_2) = 11$

M_3	r	p	d
J_1	8	1	0
J_2	3	2	-3

$\sigma_3 = \langle J_2 - J_1 \rangle$
$L_{\max}(\sigma_3) = 9$

Table 2.4: Machine data after fixing schedule on M_1.

ules for the single-machine problems together with their solution values are given underneath the data. Now, M_2 is the machine with the largest resulting maximum lateness, hence it is labeled as a bottleneck machine and its schedule is fixed, see Figure 2.11. The length of a longest

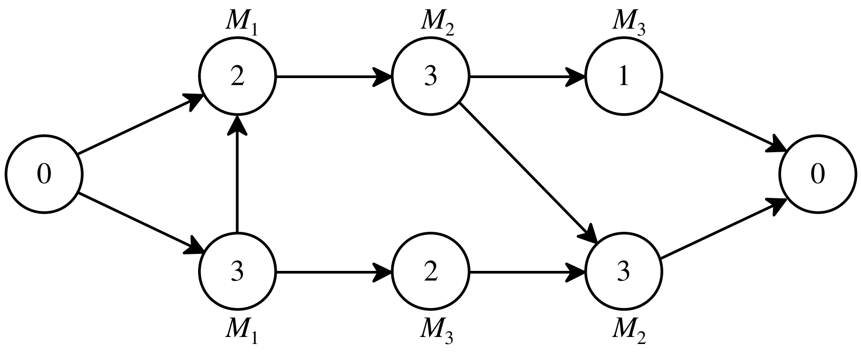


Figure 2.11: Graph after fixing the schedule on M_2.

path from s to t is now 11. Since we now have two bottleneck machines, we perform the bottleneck optimization step. In the first cycle, both machines are rescheduled in the order in which they were labeled as a bottleneck machine. This means that we first reschedule M_1 and then M_2. Releasing the schedule on M_1 results in the graph in Figure 2.12.

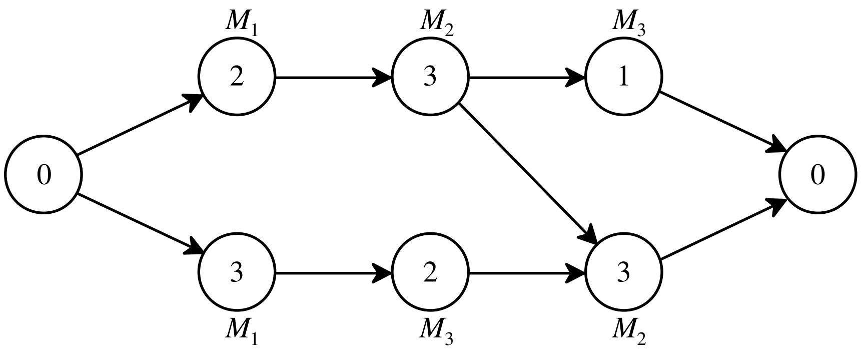


Figure 2.12: Graph after releasing the schedule on M_1.

The resulting machine data for M_1 are given in Table 2.5. The resulting schedule $J_1 - J_2$ on M_1 is better than the previous schedule $J_2 - J_1$, because the length of a longest path from s to t is now 10 instead of 11. Therefore, we accept the new schedule on M_1. The next step in the cycle is to release the schedule on machine M_2 and reschedule it. The schedule remains the same, however; the maximum lateness of this

M_1	r	p	d
J_1	0	2	-6
J_2	0	3	-5

$$\sigma_1 = \langle J_1 - J_2 \rangle$$
$$L_{\max}(\sigma_1) = 10$$

Table 2.5: Machine data after releasing schedule M_1.

schedule is 10. This completes the first cycle of the bottleneck optimization step. Since we have found an improvement during the last cycle, we perform another one. The bottleneck machines are now rescheduled in order of non-increasing solution values of the single-machine schedules. The schedules on both M_1 and M_2 have maximum lateness 10; we may therefore reschedule them in the sequence $M_1 - M_2$ or $M_2 - M_1$. In either case, we find no improvement of the length of a longest path from s to t. Now, we have to reschedule a number of non-critical bottleneck machines, but all bottleneck machines are critical. Also, there has been criticism about this step, because it rarely results in an improvement, cf. Dauzère-Peres and Lasserre [23]. We have chosen not to use this second part of the bottleneck optimization step. Figure 2.13 gives the resulting graph after the bottleneck optimization step. The longest

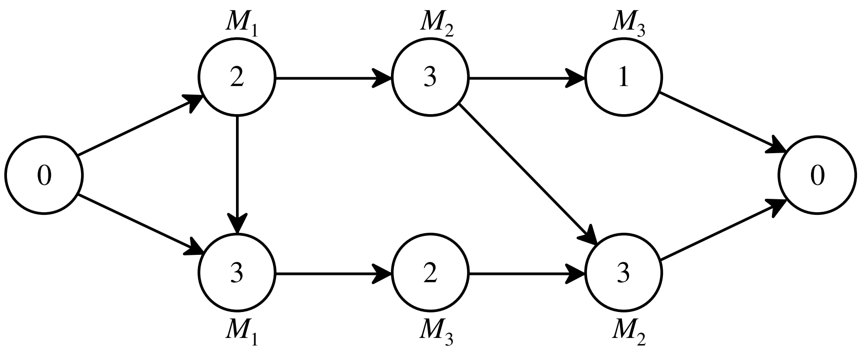


Figure 2.13: Graph after first bottleneck optimization step.

path computations now result in the machine data in Table 2.6. M_3 is labeled as a bottleneck machine and the resulting schedule on this machine is fixed. In the bottleneck optimization step, the bottleneck

M_3	r	p	d
J_1	5	1	0
J_2	5	2	-3

$\sigma_3 = \langle J_2 - J_1 \rangle$
$L_{\max}(\sigma_3) = 10$

Table 2.6: Machine data after first bottleneck optimization step.

machines are rescheduled in the sequence $M_1 - M_2 - M_3$, but this gives in no improvement. Hence, this is the end of the SB procedure. The final solution is represented in Figure 2.14. The makespan of this

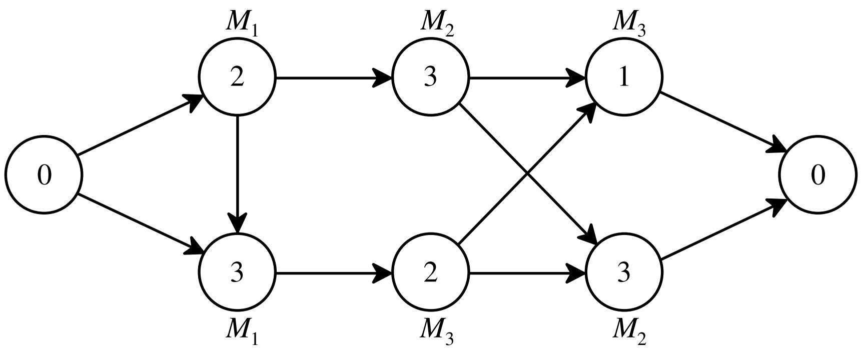


Figure 2.14: Final graph with $C_{\max} = 10$.

solution, which is equal to the length of a longest path from s to t, is 10.

2.5 Conclusion

In this chapter, we discussed the job shop shop scheduling problem and some optimization and approximation algorithms to solve it, including the SB procedure. In the next chapter, we discuss extensions to this procedure, such as setup times, simultaneous resource requirements, and minimization of maximum lateness instead of makespan. Some extensions involve only changes in the disjunctive graph; other extensions also require changes in the single-machine scheduling algorithms.

Chapter 3

Scheduling practical job shops

3.1 Introduction

In the operations research literature, machine scheduling problems, such as the job shop problem, are a popular area of research and a high level of algorithmic design and analysis has been achieved. These problems are, however, "clean" in that they ignore the nasty side constraints that occur in practice. In contrast, the production literature addresses such practical scheduling problems, but the emphasis is mainly on problem formulation and empirical analysis of priority rules.

This chapter tries to fill the gap between the operations research literature and the production literature by extending the Shifting Bottleneck (SB) procedure of Adams et al. [2] for the job shop problem to deal with practical features, such as transportation times and convergent job routings. Such practical features usually prohibit a theoretical analysis of the problem. For practical applications, computation time is very important. In general, the SB procedure produces good solutions for job shop problems in relatively little computation time; cf. Vaessens et al. [79]. This is why we use the SB procedure, instead of randomized local search methods like taboo search and simulated annealing that take more computation time.

The original paper by Adams et al. has prompted quite some research, which has taken two directions. The first concerns algorithmic improvements of the procedure; see, for instance, Dauzère-Peres and

Lasserre [23], Balas et al. [8], and Balas and Vazacopoulos [9]. The second direction concerns the adjustment of the SB procedure to other environments, e.g., job shops with practical features such as simultaneous resource requirements and setup times. We call job shops with such features *practical job shops*. Ovacik and Uzsoy [61] use an adapted SB procedure for scheduling semiconductor testing facilities. Their computational experiments with real-life data show that this procedure significantly outperforms dispatching rules both in terms of solution quality and robustness. Ivens and Lambrecht [43] discuss some practical extensions of the SB procedure, such as release and due dates, setup times, and convergent job routings.

In this chapter, we report on our research in the area of extending the SB procedure with practical features. This chapter is partly based on the work by Meester [56], who focuses on simultaneous resource requirements, and by Schutten [73]. The SB procedure with extensions is part of a commercial shop floor control system called JOBPLANNER, which we will discuss in Chapter 7.

The SB procedure consists of some generic steps, such as the computation of longest paths in the graph $D_{A'}$. Apart from the condition that $D_{A'}$ may not contain directed cycles, we have imposed no restrictions on it. The decomposition of a job shop scheduling problem as discussed in the previous chapter results in the single-machine scheduling subproblems $1|r_j|L_{\max}$. We show that by changing the properties of the graph G, and therefore of $D_{A'}$, and by changing the algorithms for the machine scheduling subproblems, we are able to handle various extensions of the classical job shop problem.

Practical instances may be very large and the machine scheduling subproblems can then often not be solved to optimality in reasonable time. In the SB procedure, it is possible to use heuristics for the machine scheduling subproblems. For example, we can use the extended Jackson [44] rule instead of Carlier's algorithm for the $1|r_j|L_{\max}$ problem. Below, we discuss possible extensions of the classical job shop problem and how to model them. Although all possible combinations of those extensions are allowed, we only consider one extension at a time. We stress that the decomposition principle proceeds along the lines Adams et al. proposed; we adjust only G and characterize the resulting machine scheduling subproblems.

3.2 Release and due dates

In the classical job shop problem, all jobs become available for processing at the same time. This is seldom true in practice, where jobs usually have different release dates. Suppose now that job J_j has a release date r_j. If we give the arc $\langle s, v_{1j} \rangle$ weight r_j, then the length of a longest path from s to v_{1j} is at least r_j. Thus, we ensure that $r_{1j} \geq r_j$ and that operation O_{1j} does not start before time r_j. Also, it is possible that we cannot process operation O_{ij} before some point in time t_{ij} ($i = 2, \ldots, n_j$), because some tool or material is not available before that time, with $t_{ij} > r_j + \sum_{k=1}^{i-1} p_{kj}$. We model this by adding an arc from s to v_{ij} with weight t_{ij}.

In the classical job shop problem, the objective is to minimize makespan. In practice, jobs for different customers have different due dates. It is then appropriate to have an objective function that measures the due date performance. Let d_j denote the due date of job J_j. Consider now the objective of minimizing *maximum lateness* $L_{\max}$, with $L_{\max} = \max_{j=1,\ldots,n}\{C_j - d_j\}$ and C_j the completion time of J_j. To cope with the objective of minimizing $L_{\max}$, we give the arcs $\langle v_{n_j,j}, t \rangle$ weight $-d_j$ ($j = 1, \ldots, n$). The length of a longest path from s to t is then equal to the maximum lateness of the corresponding schedule. Note that this objective generalizes minimizing makespan.

Thus, to deal with release and due dates, we need to change only the weights of some arcs in the graph G. If also the operations have release dates, then we must add some arcs to G.

3.3 Setup times

A machine may have to be set up before it can process the next operation. This happens, for instance, when tools must be switched off-line or when the machine must be cleaned between two operations. During a setup, the machine cannot process any operation.

Suppose that a partial schedule on the machine with setup times is $O_{gh} - O_{ij}$. The setup time between O_{gh} and O_{ij} is $s_{gh,ij}$. In $D_{A'}$, we model this setup time by giving weight $s_{gh,ij}$ to the machine arc $\langle v_{gh}, v_{ij} \rangle$. The length of a longest path from v_{gh} to v_{ij} in $D_{A'}$ is then at least $s_{gh,ij}$. This ensures that there are at least $s_{gh,ij}$ time units between the completion of O_{gh} and the start of O_{ij}, which leaves room for the needed setup.

In the standard SB procedure, we need to solve the single-machine problem $1|r_j|L_{\max}$ a number of times. Now, the setup times between the execution of the operations need to be taken into account, i.e., we have to solve the $1|r_j, s_{ij}|L_{\max}$ problem. For single-machine scheduling problems with family setup times, i.e., for the $1|r_j, s_i|L_{\max}$ problem, Schutten et al. [75] present a branch-and-bound algorithm that solves instances with up to 40 jobs to optimality. For details of this algorithm, we refer to Chapter 4.

Accordingly, the SB procedure can deal with setup times by changing the length of the machine arcs in $D_{A'}$, and by designing an algorithm for the machine scheduling subproblems with setup times.

In the Sheet Metal Factory of DAF trucks in Eindhoven (The Netherlands), setup times occur when changing the moulds of the presses. Belderok [10] tests the SB procedure with setup times in this factory. His computational experiments indicate that a significant leadtime reduction along with a better due date performance is possible, in particular for the Sheet Metal Press department. For example, Belderok tests the procedure on a real-life set of jobs that were processed in five days. The makespan of the schedule generated by the SB procedure is less than three and a half days.

3.4 Parallel machines

In the classical job shop, every operation requires a specific machine. In practice, an operation may sometimes be performed by any machine from a group of parallel machines. Parallel-machine scheduling comes down to assigning each operation to one of the machines and sequencing the operations assigned to the same machine.

The decomposition of the job shop scheduling problem with parallel machines results in a series of *parallel*-machine scheduling problems in which the jobs have release and due dates and the objective is again to minimize the maximum lateness. If the machines in a group are identical, then we may use Carlier's [18] algorithm to solve the resulting subproblems $P|r_j|L_{\max}$ to optimality.

If the machines in the group are not identical, that is, if the processing time of an operation depends on the machine, then the weight of the corresponding node changes during the execution of the SB procedure. If a parallel machine group is labeled as a bottleneck, then the weight of the corresponding node is equal to the processing time on the machine to which the operation has been assigned. Otherwise, the weight is equal to the smallest processing time of this operation.

In G, we have for each machine in this group a chain of arcs representing the sequence on this machine. In Figure 3.1, the bold arcs represent the schedule for a group consisting of two parallel machines with $O_{12} - O_{21}$ the schedule on the first machine and $O_{14} - O_{23} - O_{25}$ the schedule on the second machine in this group.

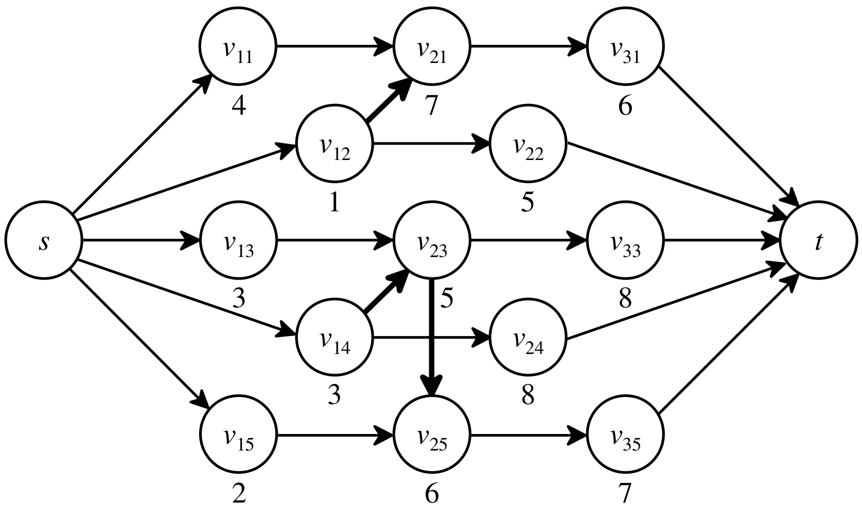


Figure 3.1: Representation of a parallel machine schedule.

3.5 Transportation times

In practice, it may be impossible to start operation O_{ij} immediately after the completion of operation $O_{i-1,j}$, because the product must first be transported from machine $\mu_{i-1,j}$ to machine μ_{ij}. If the transportation capacity is unlimited, i.e., the transportation of a product always starts immediately after the completion of the operation, then we model this by giving arc $\langle v_{i-1,j}, v_{ij} \rangle$ a weight that is equal to the transportation time. This creates enough time between the completion of $O_{i-1,j}$ and the start of O_{ij} to transport the product to the next machine. Note that we need not change any algorithm for the machine scheduling subproblems to deal with this type of transportation time.

Reesink [66] tests the SB procedure with transportation times at Stork Plastics Machinery in Hengelo, The Netherlands. He also uses transportation times to model operations that are subcontracted. These operations are assumed to have a fixed leadtime. Belderok [10] uses transportation times to make the resulting schedule more *robust*. A schedule is robust if a small increase in the processing time of an operation does not create the need to reschedule. Frequent rescheduling may lead to nervousness on the shop floor, if the operators have to deal with frequently changing schedules.

If transportation capacity is limited, then transportation is an operation that we need to schedule as well; it is then uncertain when transportation takes place and, accordingly, when the next operation can start. We model, for example, transportation performed by vehicles that can transport no more than one job at a time as a parallel machine group with setup times. Each vehicle is seen as a machine in this group, with the number of machines in this group equal to the number of transportation vehicles. Figure 3.2 shows the representation of an instance with two transportation operations and one vehicle. The nodes corresponding to the transportation operations are indicated by squares instead of circles. The figure represents the schedule $O_{21}-O_{32}$

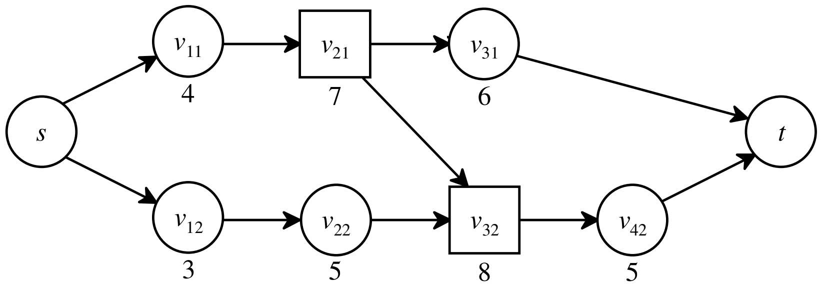


Figure 3.2: Schedule of transportation vehicle.

for the transportation vehicle. This means that the vehicle must pick up job J_1 at machine μ_{11} and transport it to machine μ_{31}. This transportation takes 7 time units; after this, the vehicle must pick up J_2 from machine μ_{22}. Since the vehicle can only transport one job at a time, it travels empty from μ_{31} to μ_{22}. We see this empty travel time for the vehicle as a setup time, and model it accordingly. An open question is how to adjust the SB procedure to deal with congestion and blocking of vehicles.

3.6 Unequal transfer and production batches

A job may represent an order to produce a batch of b identical products, not just a single product. An operation O_{ij} of this job is then actually a series of b identical operations: $O_{ij} = (O_{i,1,j}, O_{i,2,j}, \ldots, O_{i,b,j})$. If the b identical products need to be processed *contiguously* on each machine, then O_{ij} is called a *production batch.* We assume that a production batch needs to be processed continuously, i.e., without idle time, on the machines. Suppose now that we may transport $O_{i,k,j}$ $(k = 1, \ldots, b-1)$ to the next machine immediately after its completion. If we do this, then it may result in a smaller completion time on the next machine for the production batch. We call $O_{i,k,j}$ a *transfer batch.* For problems with $p_{ij} > p_{i+1,j}$, we shift the batches on machine $\mu_{i+1,j}$ to the right, such that no idle time between the batches on this machine exists; see Figure 3.3 for an example with $b = 4$. Note that the difference in time

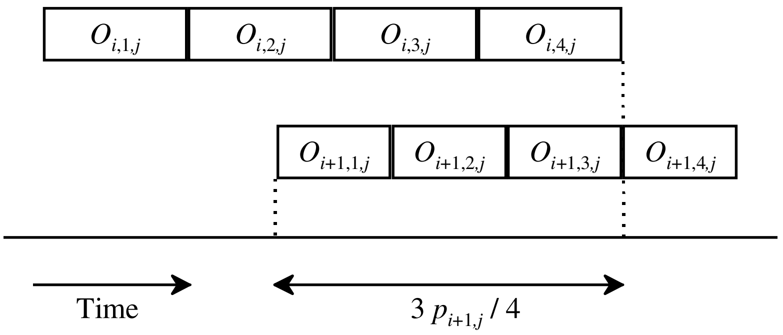


Figure 3.3: Transfer batches with $p_{ij} > p_{i+1,j}$ and $b = 4$.

between the completion of O_{ij} and the start of $O_{i+1,j}$ is at least $-\frac{b-1}{b} \cdot p_{i+1,j}$ time units. For problems with $p_{ij} \leq p_{i+1,j}$, the transfer batches may immediately be processed on the next machine after transporting it; see Figure 3.4. The difference between the start of $O_{i+1,j}$ and the

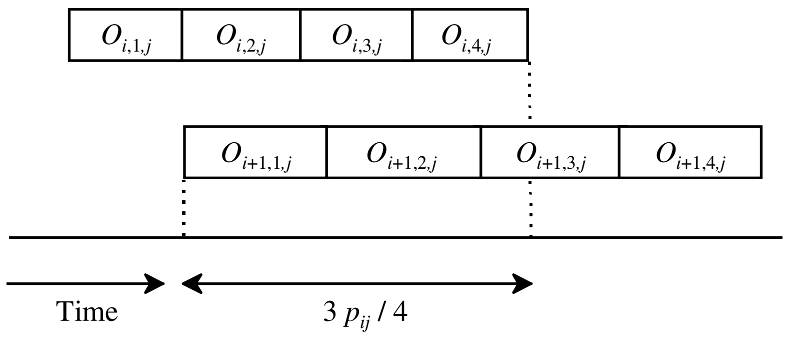


Figure 3.4: Transfer batches with $p_{ij} \leq p_{i+1,j}$ and $b = 4$.

completion of O_{ij} is now at least $-\frac{b-1}{b} \cdot p_{ij}$ time units. The SB procedure can therefore deal with unequal transfer and production batches if we give arc $\langle v_{ij}, v_{i+1,j} \rangle$ weight $-\frac{b-1}{b} \cdot \min\{p_{ij}, p_{i+1,j}\}$.

3.7 Multiple resources

An operation may need more than one resource simultaneously for its processing. Besides a machine, an operation may need a pallet on which it must be fixed, certain tools, or an operator at the machine. We model this by adding disjunctive edges to G that connect all operations that need the same resource. In Figure 3.5, operations O_{31}, O_{32}, and O_{33} need, besides the machines, the same additional resource. In the SB procedure, we orient those edges such that they represent the schedules on the additional resources. We distinguish two approaches to deal with multiple resources.

1. *The centralized approach.* In this approach, we treat every resource as a machine that needs to be scheduled. We do not differentiate between machines and other resources. Consequently, every resource becomes a bottleneck machine in the SB procedure. This approach is useful when the number of additional resources is limited. The modeling of this approach affects G only, not the machine scheduling subproblems. The interaction of the different resources is handled by the SB procedure; this is why we call this approach the centralized approach.

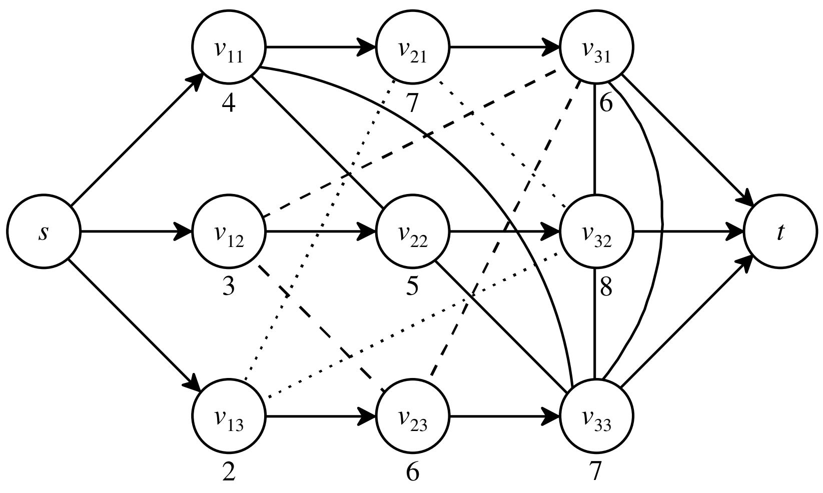


Figure 3.5: Graph with multi-resource aspects.

2. *The decentralized approach.* If the number of additional resources is large, then the centralized approach may be time-consuming, since we need to label each resource as a bottleneck machine. Sometimes, however, a group of resources may hardly be restrictive. This may be true for the cutting tools in a *Flexible Manufacturing Cell (FMC)*. Usually, an FMC consists of a parallel machine group and a large set of unique tools that can only be used by the machines of the FMC. If an FMC is part of the job shop, then the decentralized approach needs an algorithm to schedule the FMC, that is, we need an algorithm for the parallel-machine problem of minimizing the maximum lateness, subject to release dates and tooling restrictions. The interaction of the tools and the machines in the FMC is handled by the algorithm for scheduling the FMC, not the SB procedure. This is why we call it the decentralized approach.

   Meester and Zijm [57] present a hierarchical algorithm to schedule an FMC. They compare the performance of the algorithm with lower bounds obtained by relaxing the multi-resource constraints. As in the job shop scheduling problem, the gap between the lower and upper bounds is quite large. The authors feel that this is due to the weakness of the lower bounds. In contrast to the centralized approach, this approach affects both G and the machine scheduling subproblems.

Meester [56] tests both approaches on real-life instances. In one case, he tests the centralized approach in the machine shop of Ergon B.V. in Apeldoorn (The Netherlands), where each operation needs also an operator during processing. In another case, Meester tests the decentralized approach in the machine shop of El-o-Matic B.V. in Hengelo (The Netherlands). This machine shop consists of conventional and *Computer Numerically Controlled* (CNC) machines, among which one FMC with a large number of unique tools. Compared with the planning procedure used by the companies, the SB procedure shows a significant improvement of the due date performance in both cases.

Note that the outdegree of the nodes v_{ij} is not bounded by two any more. This means that the longest path computations in the graph $D_{A'}$ take $O(N^2)$ time, instead of $O(N)$, because there are now $O(N^2)$ arcs in $D_{A'}$. If, however, the outdegree of the nodes v_{ij} is bounded by k, e.g., the number of additional tools required for each operation is bounded by $k-2$, then the longest path computations take $O(kN)$ time.

3.8 Down times

The machines in a shop may have different availability times: some machines work 24 hours a day, other machines only work 8 hours a day. Also, machines might be unavailable due to scheduled maintenance. We call a period in which a machine is not available for processing a *down time*. We distinguish two types of down times: *preemptive* and *non-preemptive* down times. We call a down time preemptive, if an operation may start before and finish after it. A weekend, for example, is often a preemptive down time: it is often allowed that an operation starts before and finishes after a weekend, while no processing is performed during the weekend. We model preemptive down times by increasing the weight of a node with the length of the down time if the corresponding operation straddles the down time. We also need an algorithm that schedules a machine with preemptive down times. The objective is to minimize makespan and the operations have release dates and run-out times. Reesink [66] shows that Carlier's algorithm (see [17]) can easily be extended to solve this problem, see Appendix A.

If each operation needs to be completely processed either before, or after a down time, then this down time is called a non-preemptive down time. Maintenance, for example, is usually a non-preemptive down time: during maintenance, no job is allowed to be on the machine. The non-preemptive down times are modeled as operations that need to be processed in a prespecified interval. For each non-preemptive down time, we therefore add a node to G.

If the machines in a shop have non-preemptive down times, then the decomposition of the job shop problem results in subproblems in which the machines have down times. Westra [81] and Woerlee [85] propose algorithms to solve this machine scheduling subproblem. Either algorithm sees a down time as an operation O_{ij} with a release date r_{ij} equal to the start of the down time, a processing time p_{ij} equal to the length of the down time, and a due date $d_{ij} = r_{ij} + p_{ij} - Q$, with Q an appropriate constant. Let $L_{\max}(Q)$ denote the optimal value of the resulting problem; Carlier's algorithm (see [17]) can be used to find this value. Clearly, we have that $L_{\max}(Q) \geq r_{ij} + p_{ij} - d_{ij} = Q$ and $L_{\max}(Q)$ is monotonically non-decreasing in Q. Westra and Woerlee show that if $L_{\max}(Q) = Q$, then $L_{\max}(Q) \geq L^*_{\max}$, with $L^*_{\max}$ the optimal solution value of the problem in which each operation associated with a down time must start at its release date. If $L_{\max}(Q) > Q$, then $L_{\max}(Q) < L^*_{\max}$. The problem is then to find the smallest Q such that $L_{\max}(Q) = Q$. This can be done by binary search.

A complication with the modeling of both preemptive and non-preemptive down times is the following: suppose that M_i is a machine with preemptive down times and that M_i is the first bottleneck machine. Eventually, M_j $(j \neq i)$ will also be labeled as a bottleneck machine. The schedule of M_j is then fixed by adding the machine arcs representing this schedule to G, which may delay operations on M_i. It is then possible that an operation that *straddled* a down time can start now only *after* the down time. So, the weight of the corresponding node should no longer be increased with the length of the down time. To resolve this problem, we propose an 'intelligent' longest path procedure: when the length of a longest path to a node is computed, the procedure checks the down times of the machines and determines whether the weight of this node should be increased. A similar problem occurs for machines with non-preemptive down times. In this case, our longest path procedure checks whether an operation O_{ij} can be processed entirely before the next down time. If not, the procedure inserts an operation representing the down time just before O_{ij}.

3.9 Convergent and divergent job routings

In the job shop problem, each job is a *chain* of operations. In practice, job routings may be convergent or divergent. A *convergent* job routing occurs when some components are assembled. Figure 3.6 shows a representation of an instance with a convergent job routing. An ex-

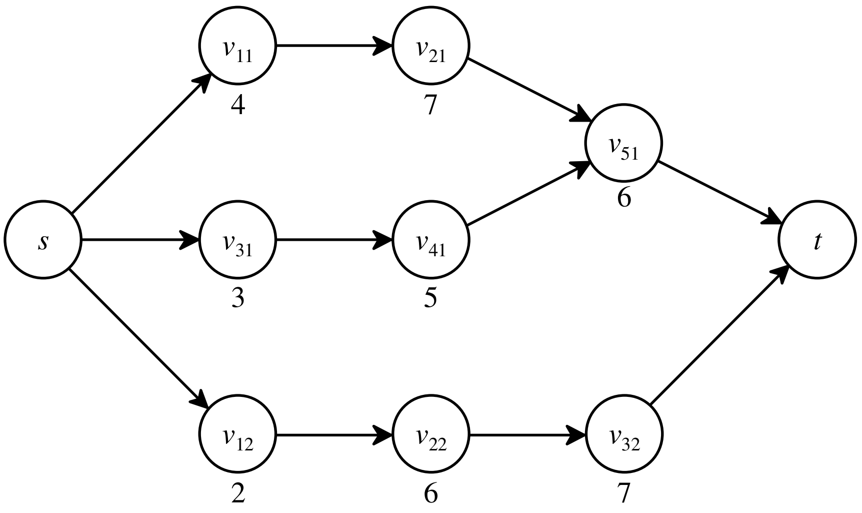


Figure 3.6: Instance in which J_1 has a convergent job routing.

ample of a *divergent* job routing is the routing of a metal sheet. Before cutting, the sheet needs some operations such as cleaning and surface treatments. After cutting, the different parts of the sheet have their own routings through the shop. We model this by allowing the nodes in G to have more than one ingoing and outgoing job arc. Note that this modeling of convergent and divergent job routings only influences the properties of G, not the machine scheduling subproblems. In Chapter 6, we compare the SB procedure with priority rules on the due date performance of an assembly shop. The effect of setup times and the arrival process is studied in this shop. Also, the effect of dynamic scheduling instead of static scheduling is studied. Test results show that the

SB procedure outperforms priority rules on the due date performance indicators maximum lateness and mean tardiness. The priority rules perform better for the performance indicator 'number of late jobs'.

3.10 Open shops

In the classical job shop, the sequence in which the operations of a job must be processed is given. In open shop problems, it is not: the operations can be performed in any order, although the operations of the same job cannot be processed simultaneously. We model this by introducing for each job a single, artificial machine on which the operations of this job must be processed. The schedule on the artificial machine dictates the sequence in which the operations of the corresponding job are processed. Each operation needs two resources: the artificial machine and the machine on which the actual processing takes place. Also, we need arcs $\langle s, v_{ij} \rangle$ $(j = 1, \ldots, n; i = 1, \ldots, n_j)$ to ensure a path from s to every other node. Analogously, we need an arc $\langle v_{ij}, t \rangle$ to ensure a path from node v_{ij} to t. Figure 3.7 shows the disjunctive graph model of an open shop problem with $n = m = 2$, the data of which are found in Table 3.1. The solid edges in the figure indicate that O_{11} and O_{12}

J_j	μ_{1j}	μ_{2j}	p_{1j}	p_{2j}
J_1	1	2	7	9
J_2	1	2	3	5

Table 3.1: Open shop problem.

as well as O_{21} and O_{22} need to be processed on the same machine, and therefore cannot be processed simultaneously. O_{11} and O_{21} as well as O_{12} and O_{22} also cannot be processed simultaneously, because they are operations of the same job. This is indicated by the non-solid edges.

3.11 Improvements of the SB procedure

Recently, some algorithmic improvements of the SB procedure have been proposed. In this section, we discuss three improvements reported in the literature.

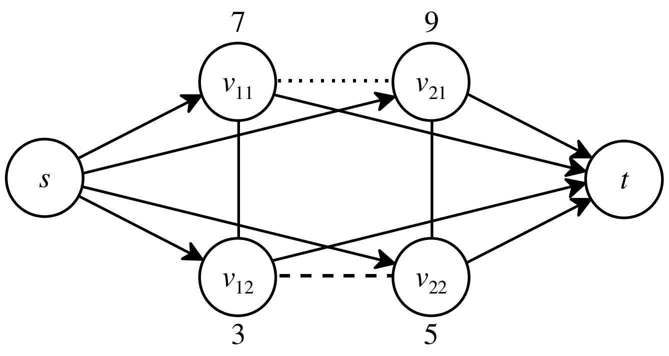


Figure 3.7: Representation of the instance of the open shop problem.

In the original SB procedure, the operations on a machine are treated independently. This may lead to infeasibilities. Figure 3.8 is the graph that we obtain after fixing schedule $O_{13} - O_{32} - O_{21}$ on machine M_3 for the instance of given in Table 2.1 in Section 2.2. O_{12} and O_{31} need to

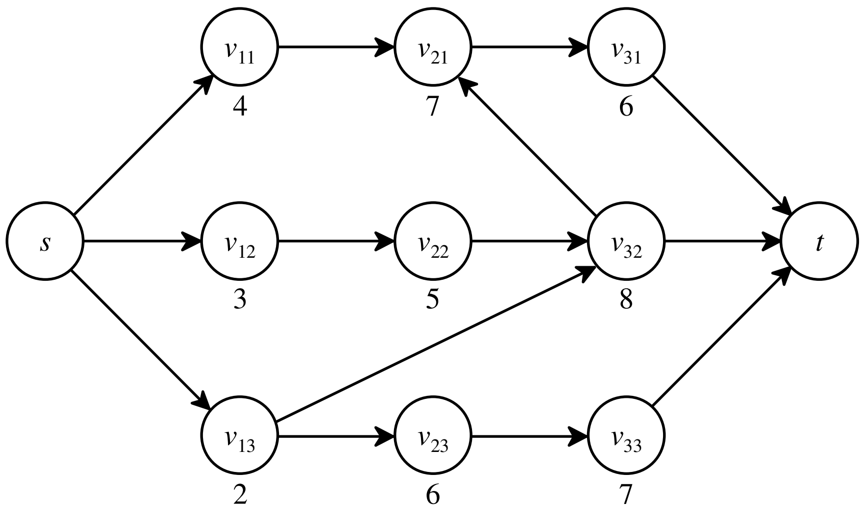


Figure 3.8: Graph with delayed precedence constraint.

be processed on machine M_2. If we want to schedule this machine, then we must schedule O_{12} before O_{31}, because otherwise a directed cycle would occur in the graph and the resulting schedule would be infeasible. What is more, after the completion of O_{12}, we must first process O_{22}, O_{32}, and O_{21}, respectively, before we can start the processing of O_{31}. So, there must be a gap of at least $p_{22}+p_{32}+p_{21}$ time units between the completion of O_{12} and the start of O_{31}. A precedence relation between two operations with the additional constraint that there is a certain delay between these two operations is called a *delayed precedence constraint*. Dauzère-Peres and Lasserre [23] were the first to incorporate delayed precedence constraints in the SB procedure. As a result, they ensure a monotonic decrease of the makespan in the bottleneck reoptimization step. They use approximation algorithms to solve the machine scheduling subproblems where the operations have release dates, run-out times, and delayed precedence constraints. Test results show that the quality of the schedules generated by this modified SB procedure is generally better than those generated by the standard SB procedure. If we want to incorporate delayed precedence constraints in the SB procedure with extensions, then each algorithm for the machine scheduling subproblems should be adapted to deal with these constraints. The computation of the delayed precedence constraints, however, takes $O(N^2)$ time, which may be too much for practical instances. A solution may then be to remove directed cycles from $D_{A'}$ by reversing the orientation of a machine arc in a cycle.

Balas et al. [8] propose an algorithm that solves the single-machine problems with delayed precedence constraints to optimality. The algorithm solves large instances that are randomly generated similar as Carlier [17] did. Balas et al. use this algorithm in the SB procedure with a modified bottleneck reoptimization step. Test results show that this variant of the SB procedure finds consistently better results than the standard SB procedure, at the expense of a considerable increase of computation time.

In the standard SB procedure, the bottleneck reoptimization step consists of rescheduling the bottleneck machines one by one. Balas and Vazacopoulos [9] propose to reoptimize *partial* schedules by applying a variable-depth search algorithm. This algorithm takes about the same computation time as the algorithm of Balas et al. [8] but performs better; cf. Vaessens et al. [79].

3.12 Conclusions

This chapter discussed extensions of the SB procedure to deal with practical features, such as setup times. To handle setup times within the SB procedure, we need an algorithm that takes the setup times into account. In the next chapter, we discuss a branch-and-bound algorithm for the problem $1|r_j, s_i|L_{\max}$ in detail. This algorithm is capable of solving instances with up to 40 jobs in reasonable time.

Chapter 4

Single-machine scheduling with family setup times

4.1 Introduction

In production environments, such as a part manufacturing shop, the combined goal of efficient and effective production may lead to complex control problems. Efficient production in such an environment is achieved by minimizing the loss of capacity due to setups and thus by combining jobs with similar setup characteristics. Effective production in an order-driven environment is achieved by completing jobs before their due dates, or at least by minimizing the lateness. Clearly, these two objectives may be conflicting: clustering jobs with similar setup characteristics may lead to the lateness of others. Any solution to these problems should therefore be based on a combination of batching and sequencing considerations. These problems are often dealt with hierarchically. On a higher level, batch sizes (or run lengths) of jobs of the same or similar nature are determined; sequencing these batches is then a lower level, short term decision. Maintaining this hierarchical approach under the current market conditions with increasing product diversity and decreasing product life cycles, however, may lead to unacceptable results, including a poor delivery performance and/or obsolete stocks. This creates the need to cluster jobs dynamically, depending on the workload.

This chapter addresses the combined setup/due date problem in a relatively simple but, in our experience, highly relevant setting. We consider the following problem, in which a set $\mathcal{J}$ of n independent jobs $J_1, \ldots, J_n$ need to be processed on a single machine. Each J_j ($j = 1, \ldots, n$) becomes available for processing at its release date r_j, needs uninterrupted processing during a given positive time p_j, and should be completed by its due date d_j. The machine is available from time 0 onwards and can process no more than one job at a time. The jobs are partitioned into families $\mathcal{F}_1, \ldots, \mathcal{F}_F$, while $f(j)$ denotes the index of the family to which job J_j belongs. If we schedule two jobs that belong to different families contiguously, then we need a given non-negative setup time s_i in between that is completely specified by the family $\mathcal{F}_i$ to which the second job belongs. We also assume that we need a setup for the first job of each family. No setup is needed when jobs of the same family are scheduled contiguously. During a setup time no processing of jobs is possible. The machine may be set up for a particular job prior to its release date. This type of setup times is called *sequence independent* setup times. The set of jobs between two subsequent setups are said to be scheduled in the same *batch*. The objective is to minimize the maximum lateness. This problem is denoted as $1|r_j, s_i|L_{\max}$. This problem is $\mathcal{NP}$-hard, even in the case of no family setup times (Lenstra et al. [54]) and in the case of equal release dates (Bruno and Downey [16]).

The presence of release dates is consistent with MRP-controlled environments. Also, the problem $1|r_j, s_i|L_{\max}$ appears as a subproblem in decomposition based approaches for job shop scheduling with setup times, such as the Shifting Bottleneck (SB) procedure of Adams et al. [2]; see Section 3.3. By having an algorithm for the *single-machine* problem $1|r_j, s_i|L_{\max}$, we are able to use the SB procedure to schedule *job shops* with setup times.

Although the interest in approaches that combine batching and scheduling in manufacturing is growing (see, e.g., Potts and Van Wassenhove [65]), we are not aware of any research addressing this particular problem. We feel therefore that this chapter fills an important gap in that it addresses a fundamental practical problem. Also, it makes a contribution in terms of algorithmic design for solving this type of $\mathcal{NP}$-hard problem by branch-and-bound, in general, and in terms of lower bound computing for problems with setup times, in particular. The lower bounds that work well for the problem without family setup times, $1|r_j|L_{\max}$, including Carlier's bound (Carlier [17]) and the preemptive lower bound obtained by allowing the interruption of the processing of a job and resumption later on, can be applied to our problem only if we ignore the setup times completely, which of course may result in weak lower bounds. For instance, the preemptive lower bound obtained by solving the $1|r_j, pmtn|L_{\max}$ problem is found by Horn's algorithm in $O(n \log n)$ time (Horn [42]); in contrast, the preemptive problem $1|r_j, s_i, pmtn|L_{\max}$ is $\mathcal{NP}$-hard, since $1|s_i|L_{\max}$ is (Bruno and Downey [16]). To clarify the last implication, we remark that for the problem $1|s_i, pmtn|L_{\max}$ always an optimal solution exists in which no job is preempted. The problems $1|s_i|L_{\max}$ and $1|s_i, pmtn|L_{\max}$ are therefore equivalent.

The organization of this chapter is as follows. Section 4.2 discusses the similarity of the $1|r_j, s_i|L_{\max}$ problem to the lot-sizing problem. In Section 4.3, we observe that for lower bounding purposes we can see setup *times* as setup *jobs* and discuss how to derive those setup jobs. Section 4.4 discusses a lower bound on the maximum lateness in a modified problem in which we consider the derived setup jobs and allow preemption. We show that this lower bound can be computed in $O(n \log n)$ time. Section 4.5 reports on the implementation of the branch-and-bound algorithm and on our computational experiments; our results show that we can solve instances with up to 40 jobs to optimality. Section 4.6 ends this chapter, which is based on Schutten et al. [75].

4.2 Similarity to the lot-sizing problem

In the *lot-sizing* model, we are given a set of jobs that need to be scheduled on one or more machines. Each job J_j may be split in up to a given number of *sublots*. Between sublots of different jobs, we need a major setup time, whereas between sublots of the same job we only need a minor setup time or no setup time at all. Potts and Van Wassenhove [65] review the literature on *batching*, i.e., clustering jobs to save setup times, and lot-sizing. They mention two advantages of job splitting. First, splitting jobs may result in a better delivery performance, because each sublot can be delivered to the customer immediately after its completion, instead of waiting for the whole job to complete. The second advantage happens for problems with more than one machine. In these problems it is possible that various operations of the same job overlap, by allowing that the next operation to a sublot starts immediately after a sublot has been processed. This process is called *lot-streaming*, a term introduced by Reiter [67]. Example 4.1 shows that job splitting can be advantageous.

Example 4.1

Consider the example in Table 4.1 with two jobs that need to be scheduled on a single machine. We assume that either job may be split into

J_j	b_j	r_j	p_j	d_j
J_1	1	0	2	7
J_2	1	3	2	5

Table 4.1: Instance with job splitting allowed.

two equal sublots. The column 'b_j' displays the required setup time when we switch to or start with processing a sublot of job J_j. The objective is to minimize the maximum lateness. The first two schedules in Figure 4.1 are the two possible left-justified schedules with no job split. The streaked boxes represent the setup times. We see that in the first

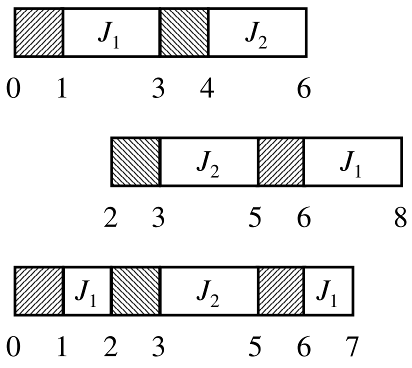


Figure 4.1: Three possible solutions for the lot-sizing problem.

two schedules one job finishes after its due date. In the last schedule, however, job J_1 is split and both jobs complete at their due date.

We can easily transform the instance in Example 4.1 into an equivalent instance of the $1|r_j, s_i|L_{\max}$ problem. Each *sublot* is then a *job* in the $1|r_j, s_i|L_{\max}$ problem with the same release and due dates as the job they belong to in the original instance. Note that in the lot-sizing problem we decide when to split a job at the cost of introducing extra setup times, whereas in the $1|r_j, s_i|L_{\max}$ problem we decide which jobs to cluster in order to save setup times.

4.3 Derivation of setup jobs

Our key observation is that we may regard any setup as the processing of an imaginary setup job of length equal to the setup time of the family associated with it. We will develop sufficient conditions for establishing that certain jobs belonging to the same family are *not* processed in the same batch. The implication is that these jobs are *separated* by a setup job for which we can specify precedence relations, a release time, and a due date. Let $\mathcal{S}$ be the set of setup jobs that are derived in this way. For any instance I of $1|r_j, s_i|L_{\max}$, we can then construct an instance I' of $1|r_j, prec|L_{\max}$ with job set $\mathcal{J} \cup \mathcal{S}$, where *prec* indicates the presence of precedence relations between the jobs. In fact, the precedence constraints in our application have a specific structure. The crux is that for any instance I and I' constructed in this way, we can show that

$$L^*_{\max}(I) \geq L^*_{\max}(I'),$$

with $L^*_{\max}(I)$ and $L^*_{\max}(I')$ the optimal solution values of these instances. Hence, a lower bound on $L^*_{\max}(I)$ can be computed by computing a lower bound on $L^*_{\max}(I')$.

We derive two types of setup jobs: *separating* setup jobs that have precedence relations, and *unrelated* setup jobs that have no precedence relations. We call the jobs in $\mathcal{J}$ the *real jobs* to distinguish them from the setup jobs. In the remainder, we let $\mathcal{S}$ be the set of setup jobs. Also, we let $\succ$ and $\prec$ mean 'has to follow' and 'has to precede', respectively.

In Section 4.3.1, we discuss the prerequisites of our approach to derive setup jobs, including a proof that a setup can indeed be seen as a setup job with a specific processing time, release date, due date, and precedence relations. We point out that the setup jobs should be consistent with each other and introduce a measure of the strength of a setup job. Finally, we also derive the so-called initial setup jobs. In Section 4.3.2, we discuss the logic behind the derivation of separating setup jobs and our two strategies to actually derive them. In Section 4.3.3, we derive unrelated setup jobs.

4.3.1 Preliminaries

Consider any instance I of $1|r_j, s_i|L_{\max}$ and let I' be the instance of $1|r_j, prec|L_{\max}$ obtained from I by ignoring the family setup times. Hence, we have that $L^*_{\max}(I') \leq L^*_{\max}(I)$. Suppose now that we have established, one way or the other, that in every optimal schedule for I all jobs in $\mathcal{A} \subset \mathcal{F}_i$ precede all jobs in $\mathcal{B} \subset \mathcal{F}_i$ ($\mathcal{B} \neq \emptyset$) and no job from $\mathcal{A}$ and no job from $\mathcal{B}$ are scheduled in the same batch. This then means that there must be at least one *separating* setup associated with family $\mathcal{F}_i$ between the last job belonging to $\mathcal{A}$ and the first job belonging to $\mathcal{B}$. Theorem 4.1 validates our key idea that this setup can be viewed as a *separating setup job* with a specific processing time, release date, due date, and precedence relations.

Theorem 4.1 *Suppose that in every optimal schedule for instance I all jobs in $\mathcal{A} \subset \mathcal{F}_i$ precede all jobs in $\mathcal{B} \subset \mathcal{F}_i$ ($\mathcal{B} \neq \emptyset$) and no job from $\mathcal{A}$ and no job from $\mathcal{B}$ are scheduled in the same batch. We still have that $L^*_{\max}(I') \leq L^*_{\max}(I)$, if we add a setup job J_s to I' with*

$$\begin{aligned}
p_s &= s_i, \\
J_s &\succ J_j, \text{ for all } J_j \in \mathcal{A}, \\
J_s &\prec J_j, \text{ for all } J_j \in \mathcal{B}, \\
r_s &= \min_{J_j \in \mathcal{F}_i \setminus \mathcal{A}} r_j - s_i, \\
d_s &= \min_{J_j \in \mathcal{B}} (d_j - p_j).
\end{aligned}$$

Proof. It only remains to be shown that the specification of r_s and d_s is correct. Consider any optimal schedule σ for I and any setup for family $\mathcal{F}_i$ that succeeds all jobs from $\mathcal{A}$ and precedes all jobs from $\mathcal{B}$ in this schedule. We associate the setup job J_s with this setup. We may assume that this setup occurs immediately before the execution of the job it is needed for. Since this may be any job in $\mathcal{F}_i \setminus \mathcal{A}$, the release date of J_s follows. Let σ' be the feasible schedule for I' obtained from σ in the following way: let the sequence of the real jobs in σ' concur with the sequence in σ, and replace the setup for family $\mathcal{F}_i$ between the jobs from $\mathcal{A}$ and $\mathcal{B}$ with its associated setup job J_s. Note that $C_j(\sigma') \leq C_j(\sigma)$ for all $J_j \in \mathcal{J}$, and therefore $L_j(\sigma') \leq L_j(\sigma) \leq L^*_{\max}(I)$. If we assign d_s as proposed, we have that $J_s \prec J_j$ and $d_s = d_j - p_j$ for some $J_j \in \mathcal{B}$, and hence that

$$\begin{aligned} L_s(\sigma') &= C_s(\sigma') - d_s \leq C_j(\sigma') - p_j - (d_j - p_j) \\ &\leq C_j(\sigma) - d_j = L_j(\sigma) \leq L^*_{\max}(I). \end{aligned}$$

Thus, we proved that $L_j(\sigma') \leq L^*_{\max}(I)$ for every job in I', and therefore $L^*_{\max}(I') \leq L_{\max}(\sigma') \leq L^*_{\max}(I)$. □

The crux is that the addition of this separating setup job may increase the value $L^*_{\max}(I')$, and thus to improve the lower bound on $L^*_{\max}(I)$. In the remainder, if we add a setup job to I separating some sets $\mathcal{A}$ and $\mathcal{B}$, then we implicitly assume that it has the release date, due date and precedence relations as specified in Theorem 4.1.

It is not sensible, even if it were possible, to consider all possible $\mathcal{A}$ and $\mathcal{B}$. The following subsets enable systematic procedures for deriving setup jobs. Let $J^i_{[j]} \in \mathcal{F}_i$ denote the job with the jth smallest release date in family $\mathcal{F}_i$. For any family $\mathcal{F}_i$ and any $a = 1, \ldots, |\mathcal{F}_i|$ and $b = a, \ldots, |\mathcal{F}_i|$ define

$$\mathcal{P}^i_{a,b} = \{J^i_{[a]}, \ldots, J^i_{[b]}\}.$$

From now on, we restrict our attention to subsets $\mathcal{A} = \mathcal{P}^i_{1,k}$ and subsets $\mathcal{B} = \mathcal{P}^i_{l,|\mathcal{F}_i|}$, with $1 \leq k < |\mathcal{F}_i|$ and $k < l \leq |\mathcal{F}_i|$, for deriving setup jobs.

We may not just derive setup jobs as we please. We have to make sure that the setup jobs are *consistent* with each other. For instance, if we have already derived a setup job between job sets $\mathcal{A}$ and $\mathcal{B}$, then we may not add another setup job between the subsets $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathcal{B}' \subseteq \mathcal{B}$. To ensure the derivation of consistent setup jobs, we introduce the notion of *induction*. We say that the jobs in $\mathcal{A}$ *left-induce* J_s, the jobs in $\mathcal{B}$ *right-induce* J_s, and the setup job J_s is induced by family $\mathcal{F}_i$. We construct a so-called *induction graph* $\mathcal{G} = (\mathcal{J} \cup \mathcal{S}, \mathcal{H})$, in which there is an arc (J_s, J_j) in $\mathcal{H}$ with $J_s \in \mathcal{S}$ and $J_j \in \mathcal{J}$ if and only if J_j right-induces J_s. Similarly, there is an arc (J_j, J_s) in $\mathcal{H}$ if and only if J_j left-induces J_s.

Observation 4.1 *If we only consider subsets $\mathcal{A} = \mathcal{P}^i_{1,k}$ and $\mathcal{B} = \mathcal{P}^i_{l,|\mathcal{F}_i|}$ for deriving setup jobs, then the induction graph corresponds to a set of consistent setup jobs if in its* transitive reduction, *obtained from $\mathcal{G}$ by removing all arcs that are implied by transitivity, each $J_j \in \mathcal{J}$ has at most one ingoing and at most one outgoing arc in $\mathcal{H}$.*

Accordingly, we may add a setup job to I' if this condition remains satisfied. Throughout this section, we assume that this consistency is preserved.

The *rank* of a setup job is defined as the number of jobs it separates. If $\mathcal{A} \cup \mathcal{B} = \mathcal{F}_i$, then the separation, and thereby the setup job J_s, is the strongest possible: we then say that J_s has *full rank*. If $|\mathcal{A} \cup \mathcal{B}| < |\mathcal{F}_i|$, then in fact J_s separates *at least* the job sets $\mathcal{A}$ and $\mathcal{B}$: we do not know yet on which side of J_s the other jobs in $\mathcal{F}_i$ will be scheduled. The rank of J_s is then equal to $|\mathcal{A} \cup \mathcal{B}|$. Intuitively, we prefer setup jobs of high rank. The aim of this section is to derive such setup jobs in the root node of the branch-and-bound tree.

One particular type of setup job of full rank is a sitting duck: for every family $\mathcal{F}_i$, we need a setup job just before the processing of its first job. Accordingly, we may introduce an *initial setup job* J_s for family $\mathcal{F}_i$ with

$$\begin{aligned} p_s &= s_i, \\ d_s &= \min_{J_j \in \mathcal{F}_i} (d_j - p_j), \\ J_s &\prec J_j, \text{ for all } J_j \in \mathcal{F}_i, \\ r_s &= \min_{J_j \in F_i} r_j - s_i. \end{aligned}$$

4.3.2 Deriving separating setup jobs

The separating setup jobs are derived through sufficient conditions for having an optimal schedule in which particular jobs of the same family are not scheduled in the same batch. We stipulate these conditions in terms of a lower bound lb and an incumbent upper bound ub on $L^*_{\max}(I)$, each proceeding from the assumption that $L^*_{\max}(I) < ub$. It is irrelevant how these lb and ub are obtained. However, the tighter lb and ub are, the more effective these conditions will be. In fact, there is a strong interaction between deriving setup jobs and computing lower bounds;

after all, the more setup jobs are derived, the stronger the lower bound is likely to be.

The logic behind the derivation of separating setup jobs is the following. Suppose we want to put two jobs in the same batch. If the release and due dates of these jobs prohibit that these jobs are scheduled contiguously, then the machine is idle in between their processing, if no other job belonging to the same family is available for processing. If this idle time period T is too long, then *saving a single setup does not make up for what is essentially a loss of machine capacity.* We have two strategies to conclude that T is effectively too long: (i) if T is so long that we can perform a setup for family $\mathcal{F}_i$ in the meantime, and (ii) if a lower bound for the case that we leave the machine idle during period T is equal to or larger than the incumbent upper bound. We formalize these strategies below.

In any optimal schedule, each J_j is scheduled somewhere in the interval $[r_j, d_j + L^*_{\max}(I)]$ $(j = 1, \ldots, n)$. Accordingly, if $L^*_{\max}(I) < ub$, then the largest possible completion time of J_j is $\bar{d}_j = d_j + ub - 1$. We call job J_j *safely scheduled* if $r_j + p_j \leq C_j \leq d_j + lb$; note that if each job is safely scheduled, then we have an optimal schedule σ, since $L_{\max}(\sigma) \leq lb \leq L^*_{\max}(I)$. For any job set $\mathcal{A}$, let $r(\mathcal{A}) = \min_{J_j \in \mathcal{A}} r_j$, and $\bar{d}(\mathcal{A}) = \max_{J_j \in \mathcal{A}} \bar{d}_j$; note that a necessary condition for having $L^*_{\max}(I) < ub$ is that all jobs in $\mathcal{A}$ are completed by time $\bar{d}(\mathcal{A})$.

We are now ready to make the following observation, which plays a key role in the derivation of the setup jobs.

Observation 4.2 *Consider disjoint subsets $\mathcal{A} \subset \mathcal{F}_i$ and $\mathcal{B} \subset \mathcal{F}_i$ with $\bar{d}(\mathcal{A}) < r(\mathcal{B})$. If there exists a schedule σ with $L_{\max}(\sigma) < ub$ that puts jobs from both $\mathcal{A}$ and $\mathcal{B}$ in the same batch, then it has the following properties:*

- *The machine is idle during the period $T = [\bar{d}(\mathcal{A}), r(\mathcal{B})]$, if there is no job $J_j \in \mathcal{F}_i \setminus (\mathcal{A} \cup \mathcal{B})$ available for processing during period T. This means that the machine is definitely idle during period T if $\mathcal{A} \cup \mathcal{B} = \mathcal{F}_i$.*
- *The batch spans at least the interval $[\max_{J_j \in \mathcal{A}}(\bar{d}_j - p_j), \min_{J_j \in \mathcal{B}}(r_j + p_j)]$.*

As pointed out before, a long idle time period T makes it unlikely that there indeed exists an optimal schedule in which some job from $\mathcal{A}$ and some job from $\mathcal{B}$ are scheduled in the same batch. Or equivalently, a long period T makes it likely that there exists an optimal schedule in which no job from $\mathcal{A}$ and no job from $\mathcal{B}$ are scheduled in the same batch.

The following theorem gives an effective means for deriving setup jobs. It says that if T is large enough to accommodate a setup for family $\mathcal{F}_i$, then we may introduce a setup job of full rank.

Theorem 4.2 *Suppose $L^*_{\max}(I) < ub$. If there is a family $\mathcal{F}_i$ ($i = 1, \ldots, F$) and an index k ($k = 1, \ldots, |\mathcal{F}_i| - 1$), for which*

$$\bar{d}(\mathcal{P}^i_{1,k}) + s_i \leq r(\mathcal{P}^i_{k+1,|\mathcal{F}_i|}), \tag{4.1}$$

then we may introduce a setup job J_s of full rank that separates $\mathcal{P}^i_{1,k}$ from $\mathcal{P}^i_{k+1,|\mathcal{F}_i|}$.

Proof. Let σ be any optimal schedule. There are two cases to consider:

1. There is no batch in σ that contains a job from $\mathcal{P}^i_{1,k}$ as well as a job from $\mathcal{P}^i_{k+1,|\mathcal{F}_i|}$. In this case, there is a setup between $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+1,|\mathcal{F}_i|}$.
2. There is a batch in σ that contains a job from $\mathcal{P}^i_{1,k}$ as well as a job from $\mathcal{P}^i_{k+1,|\mathcal{F}_i|}$. In this case, the machine is idle between $\bar{d}(\mathcal{P}^i_{1,k})$ and $r(\mathcal{P}^i_{k+1,|\mathcal{F}_i|})$; see Observation 4.2. We can then transform σ into an equivalent schedule in which a setup, performed during period $T = [\bar{d}(\mathcal{P}^i_{1,k}), r(\mathcal{P}^i_{k+1,|\mathcal{F}_i|})]$, splits this batch into two consecutive batches of the same family.

Therefore, we may assume that in every optimal solution a setup separates $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+1,|\mathcal{F}_i|}$. □

The next theorem is a generalization of Theorem 4.2 to derive setup jobs of smaller rank. If we cannot separate the sets $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+1,|\mathcal{F}_i|}$, then we may try to separate the sets $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+l,|\mathcal{F}_i|}$, for some $l \geq 2$. After all, the larger l is, the longer the idle time period T becomes if we want to put some jobs belonging to these sets in the same batch. The condition for testing if T gets too long is similar to condition (1), albeit period T should also have room to accommodate the 'intermittent jobs' $J^i_{[k+1]}, \ldots, J^i_{[k+l-1]}$.

Theorem 4.3 *Suppose $L^*_{\max}(I) < ub$. If there is a family $\mathcal{F}_i$ $(1 \leq i \leq F)$, an index k $(1 \leq k \leq |\mathcal{F}_i| - 1)$, and an index l $(1 \leq l \leq |\mathcal{F}_i| - k)$ such that the interval*

$$[\bar{d}(\mathcal{P}^i_{1,k}), r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})] \tag{4.2}$$

is large enough to safely schedule each of the jobs $J^i_{[k+1]}, \ldots, J^i_{[k+l-1]}$ and a setup for family $\mathcal{F}_i$ in it, then we may introduce a setup job J_s of rank $|\mathcal{F}_i| - l + 1$ that separates the job sets $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+l,|\mathcal{F}_i|}$. □

We now come to our second strategy to derive setup jobs. Suppose $lb(\mathcal{A}, \mathcal{B})$ is a lower bound for the case that some unspecified job from $\mathcal{A}$ and some unspecified job from $\mathcal{B}$ are scheduled in the same batch. If $lb(\mathcal{A}, \mathcal{B}) \geq ub$, then the sets $\mathcal{A}$ and $\mathcal{B}$ are obviously separated in any optimal schedule if $L^*_{\max}(I) < ub$. In Section 4.4, we show how to compute such a bound.

Theorem 4.4 *Suppose $L^*_{\max}(I) < ub$. If $\bar{d}(\mathcal{P}^i_{1,k}) < r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})$ and*

$$lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+l,|\mathcal{F}_i|}) \geq ub, \tag{4.3}$$

for some i, k and l with $1 \leq i \leq F$, $1 \leq k < |\mathcal{F}_i|$ and $1 \leq l \leq |\mathcal{F}_i| - k$, then we may introduce a setup job of rank $|\mathcal{F}_i| - l + 1$ that separates $\mathcal{P}^i_{1,k}$ from $\mathcal{P}^i_{k+l,|\mathcal{F}_i|}$. □

4.3.3 Deriving unrelated setup jobs

The derivation of unrelated setup jobs, which have no precedence relations, proceeds by the premise that batches of different families cannot be processed simultaneously. Suppose that $\bar{d}(\mathcal{P}^i_{1,k}) < r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})$ and $\bar{d}(\mathcal{P}^h_{1,a}) < r(\mathcal{P}^h_{a+b,|\mathcal{F}_h|})$ and the intervals $[\bar{d}(\mathcal{P}^i_{1,k}), r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})]$ and $[\bar{d}(\mathcal{P}^h_{1,a}), r(\mathcal{P}^h_{a+b,|\mathcal{F}_h|})]$ overlap in time; that is, there is a point in time t such that $\bar{d}(\mathcal{P}^i_{1,k}) \leq t \leq r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})$ and $\bar{d}(\mathcal{P}^h_{1,a}) \leq t \leq r(\mathcal{P}^h_{a+b,|\mathcal{F}_h|})$, with at least one $\leq$ sign holding as a strict inequality. The conclusion must then be that we may at least separate either $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+l,|\mathcal{F}_i|}$, or $\mathcal{P}^h_{1,a}$ and $\mathcal{P}^h_{a+b,|\mathcal{F}_h|}$, since the machine can process no more than one batch at a time. We may therefore introduce a setup job, but it has no precedence relations, since we cannot associate the setup job with either family. For this reason, these unrelated setup jobs are quite weak. They have rank 0, and their release and due dates are not very tight either.

Theorem 4.5 *If there are two families $\mathcal{F}_i$ and $\mathcal{F}_h$ and indices k, l, a and b for which the time intervals $[\bar{d}(\mathcal{P}^i_{1,k}),\, r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})]$ and $[\bar{d}(\mathcal{P}^h_{1,a}),\, r(\mathcal{P}^h_{a+b,|\mathcal{F}_h|})]$ overlap, then we may introduce a setup job J_s of rank 0 with*

$$\begin{aligned} p_s &= \min\{s_i, s_h\}, \\ r_s &= \min\{r(\mathcal{P}^i_{k+1,|\mathcal{F}_i|}) - s_i,\; r(\mathcal{P}^h_{a+1,|\mathcal{F}_h|}) - s_h\}, \\ d_s &= \max\{\min_{J_j \in \mathcal{P}^i_{k+l,|\mathcal{F}_i|}} (d_j - p_j),\; \min_{J_j \in \mathcal{P}^h_{a+b,|\mathcal{F}_h|}} (d_j - p_j)\}. \end{aligned}$$

□

Obviously, any number of families may be involved in this type of derivation, but the resulting setup jobs will then be even weaker.

4.4 Lower bounds

In this section, we first present the preemptive lower bound for the $1|r_j, prec|L_{\max}$ problem. Then, we show how to compute the bound $lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+l,|\mathcal{F}_i|})$ needed in condition (4.3) to derive setup jobs.

First of all, however, we characterize the acyclic directed precedence graph G induced by any set $\mathcal{S}$ of consistent setup jobs. We assume that $\mathcal{S}$ contains for each family at least the initial setup job. Let $\mathcal{S}_i$ be the set of separating setup jobs induced by the jobs in $\mathcal{F}_i$. We have as vertex set $V = \mathcal{J} \cup \mathcal{S}$ and there is an arc (J_j, J_k) if and only if $J_j \prec J_k$. If there is an arc (J_j, J_k), then J_j is an *immediate predecessor* of J_k and J_k is an *immediate successor* of J_j. If there is a path in G from J_j to J_k, then J_j is a predecessor of J_k; J_k is then a successor of J_j. There are no arcs between the unrelated setup jobs and the real jobs.

Let $G' = (V, A)$ be the transitive reduction of this graph, where A denotes the remaining arc set. The release and due dates jobs may not be consistent with the precedence constraints; e.g., we may have that $r_k < r_j + p_j$ for some $J_j \prec J_k$. We therefore modify the release and due dates in the following way:

$$r_j \leftarrow \max\{r_j, \max_{J_k \prec J_j} (r_k + p_k)\} \text{ for all } J_j \in \mathcal{J} \cup \mathcal{S}$$

and

$$d_j \leftarrow \min\{d_j, \min_{J_j \prec J_k} (d_k - p_k)\} \text{ for all } J_j \in \mathcal{J} \cup \mathcal{S}.$$

This modification neither affects the optimal solution, nor the optimal solution value.

The graph G' then has the following properties:

- It decomposes into m arc-disjoint connected subgraphs, one for every family, on the one hand, and isolated vertices representing the unrelated setup jobs, on the other hand.
- For any arc $(J_j, J_k) \in A$, we have that $r_j + p_j \leq r_k$ and $d_j \leq d_k - p_k$.
- For any arc $(J_j, J_k) \in A$, we have that if $J_j \in \mathcal{J}$, then $J_k \in \mathcal{S}$, and, conversely, if $J_j \in \mathcal{S}$, then $J_k \in \mathcal{J}$.
- Each job in $\mathcal{J}$ has at most one immediate successor and at most one immediate predecessor.
- There are $O(n)$ arcs; this means that the release and due date modification can be carried out in $O(n)$ time.

Due to the specific structure of the precedence constraints in our application, the derivation of setup jobs induce instances of what we term the $1|r_j, \mathit{setup\text{-}prec}|L_{\max}$ problem.

4.4.1 The preemptive bound

The $1|r_j, \mathit{prec}, \mathit{pmtn}|L_{\max}$ problem is solvable by Horn's rule (Horn [42]) after release and due date modification in $O(n^2)$ time. For the problem $1|r_j, \mathit{setup\text{-}prec}, \mathit{pmtn}|L_{\max}$, the modification of the release and due dates takes $O(n)$ time only. Hence, we have the following result, the proof of which is included for sake of completeness.

Theorem 4.6 *The problem $1|r_j, \mathit{setup\text{-}prec}, \mathit{pmtn}\,|L_{\max}$ is solvable in $O(n \log n)$ time by the following rule: at any time schedule an available job with the smallest due date.*

Proof. First of all, note that Horn's rule generates a feasible schedule for the problem $1|r_j, \mathit{setup\text{-}prec}, \mathit{pmtn}|L_{\max}$. This is true, since if $J_j \prec J_k$, then we have that $r_j + p_j \leq r_k$ and $d_j \leq d_k - p_k$.

Let now π be the schedule produced by Horn's rule, and let σ be any optimal schedule. We shall prove that we can transform σ into π by rescheduling jobs while preserving feasibility and optimality. Compare σ with π from time 0 onwards, and let t be the first time at which the schedules start to differ: suppose J_j is scheduled between time t and t_1 in σ and J_k is scheduled between time t and t_2 in π. Let $\tau = \min\{t_1, t_2\}$. Find time $s > \tau$ that designates the smallest interval $[t, s]$ in which J_k is processed for exactly $\tau - t \le p_k$ units of time, according to σ. Let $\mathcal{A}$ be the set of successors of J_j in G that are scheduled between t and s in σ. We then have that

$$d_j \le d_l \text{ for all } J_l \in \mathcal{A}.$$

Also, since J_k is scheduled at time t in π, not J_j, we have that

$$d_k \le d_j,$$

Hence, the following transformation of σ retains both feasibility and optimality:

- Remove all portions of J_j, J_k and the jobs in $\mathcal{A}$ between time t and s, but leave the other jobs intact.
- Schedule J_k in the time slot $[t, \tau]$.
- Schedule J_j and the jobs in $\mathcal{A}$ in the remaining available time slots between τ and s in the same order as before.

The optimality of the resulting schedule follows from an interchange argument, similar to the one used in Section 1.2.1. Now let $t \leftarrow \tau$, and repeat the argument till we reach the end of the schedule; both schedules are then identical. □

This rule can evidently be implemented in $O(n \log n)$ time, since there are n real and no more than n setup jobs, there are $O(n)$ preemptions, and the release and due dates of the available jobs need to be maintained in a partial order only.

4.4.2 Computing the bound $lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+l,|\mathcal{F}_i|})$

The bound $lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+l,|\mathcal{F}_i|})$, needed for condition (4.3), is a lower bound resulting from scheduling some unspecified job in $\mathcal{P}^i_{1,k}$ and some unspecified job in $\mathcal{P}^i_{k+l,|\mathcal{F}_i|}$ in the same batch, say, B ($i = 1, \ldots, F$, $1 \leq k < |\mathcal{F}_i|$, $0 \leq l \leq |\mathcal{F}_i| - 1$). We assume that some separating and unrelated setup jobs already have been derived and that the setup job that may be induced by this bound is consistent with them.

If we decide to schedule some job from $\mathcal{P}^i_{1,k}$ and some job from $\mathcal{P}^i_{k+l,|\mathcal{F}_i|}$ in the same batch, say, B, then B spans at least the interval $T = [t_1, t_2]$, where

$$t_1 = \max_{J_j \in \mathcal{P}^i_{1,k}} (\bar{d}_j - p_j),$$

and

$$t_2 = \min_{J_j \in \mathcal{P}^i_{k+l,|\mathcal{F}_i|}} (r_j + p_j);$$

see Observation 1. We assume that $t_2 > t_1$. If not, then we let $lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+l,|\mathcal{F}_i|}) = -\infty$.

Let I' be any instance of the $1|r_j, \mathit{setup\text{-}prec}|L_{\max}$ problem with the condition that we schedule those unspecified jobs in the same batch. To compute a lower bound, we construct an instance I'' with the additional constraint that the machine is not available for processing during the interval $T = [t_1, t_2]$. We initialize $I'' = I'$ and then remove all jobs $\mathcal{J}_j \in \mathcal{F}_i \cup \mathcal{S}_i$ from I'' for which the time intervals $[t_1, t_2]$ and $[r_j, \bar{d}_j]$ overlap, with $\mathcal{S}_i$ the setup jobs associated with family $\mathcal{F}_i$. We do this to ensure that $L^*_{\max}(I'')$ is a valid lower bound on $L^*_{\max}(I')$.

Moreover, we try to derive more separating setup jobs for each family other than $\mathcal{F}_i$. If the machine is not available during the period $T = [t_1, t_2]$, then any two jobs J_j and J_k cannot be in the same batch if $r_j > t_1 - p_j$ and $\bar{d}_k < t_2 + p_k$; after all, J_j must then be processed after period T and J_k before period T. So, if $\mathcal{C}_h = \{J_j \in \mathcal{F}_h \mid \bar{d}_j < t_2 + p_j\}$ and $\mathcal{D}_h = \{J_j \in \mathcal{F}_h \mid r_j > t_1 - p_j\}$ and $\mathcal{C}_h \neq \emptyset$ and $\mathcal{D}_h \neq \emptyset$, then we may add a setup job J_s to I'' that separates the sets $\mathcal{C}_h$ and $\mathcal{D}_h$ for any family $\mathcal{F}_h \neq \mathcal{F}_i$, if this setup job is consistent with the other setup jobs.

We now compute the preemptive lower bound for I'' subject to the condition that the machine is not available during period T. We can easily cope with this condition by adding an independent dummy job J_0 to I'' with $r_0 = t_1$, $p_0 = t_2 - t_1$, and $d_0 = \min_{J_j \in \mathcal{J} \cup \mathcal{S}} d_j - 1$. Horn's rule schedules J_0 then in period T, and we compute $lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+l,|\mathcal{F}_i|})$ as $\max_{J_j \in I'' \setminus \{J_0\}} L_j$.

4.5 Implementation and computational experiments

4.5.1 Implementation

Our branch-and-bound algorithm uses a forward sequencing branching rule, in which a node at level k ($k = 0, \ldots, n$) corresponds to an active partial schedule consisting of k jobs. A node at level k has $n-k$ descendant nodes, one for each unscheduled job. We branch from the nodes in order of non-decreasing release dates of the jobs associated with the nodes.

In the root node of the tree, we run a two-phase randomized local-search algorithm to find a good initial upper bound *ub*; it uses simulated annealing first and then tries to improve the solution by tabu search. The neighborhood of a feasible sequence is in either phase defined as the set of sequences obtained by either relocating any single job, or swapping any two jobs in the sequence. In fact, both the simulated annealing and the tabu search subroutines are straightforward implementations of the basic principles, as outlined in for instance Van Laarhoven and Aarts [48] and Glover [32]. Given this upper bound, we iteratively derive as many and as strong as possible consistent setup jobs. Deriving setup jobs is computationally expensive; for this reason, it is carried out only in the root node of the branch-and-bound tree. Although it takes only $O(n \log n)$ time, it is too time-consuming to compute the preemptive bound in each node of the tree. We use Carlier's lower bound [17] for the problem $1|r_j|L_{\max}$; this bound requires only $O(n)$ time in each node.

Given an upper bound *ub*, we derive the setup jobs in the following way. First of all, we specify the initial setup jobs, and then we compute the preemptive lower bound *lb* for the instance of the corresponding $1|r_j, \textit{setup-prec}|L_{\max}$ problem. Recall that we like to have setup jobs of full rank. In addition, we make sure that we derive only setup jobs that are consistent with those already derived. We first check condition (4.1) for all families $\mathcal{F}_i$ ($i = 1, \ldots, F$) and subsets $\mathcal{P}^i_{1,k}$ and $\mathcal{P}^i_{k+1,|F_i|}$ ($k = 1, \ldots, |\mathcal{F}_i| - 1$). If at least one setup job has been derived in this way, compute *lb* again, and if *lb* has improved, then check condition (4.1) again, and so on. If no setup job has been derived, or if *lb* has not improved, then check out whether $lb(\mathcal{P}^i_{1,k}, \mathcal{P}^i_{k+1,|F_i|}) \geq ub$, for $i = 1, \ldots, F$, $k = 1, \ldots, |\mathcal{F}_i| - 1$, which also induces setup jobs of full rank. If this test is successful and lb has improved, then go back to condition (4.1) and repeat until no setup job of full rank can be derived any more. In the same fashion, we try next to generate setup jobs of rank $|\mathcal{F}_i| - 1$ by alternately checking conditions (4.2), stipulated in Theorem 4.3, and conditions (4.3) with $l = 2$, stipulated in Theorem 4.4. If this improves lb, then again we check the conditions for deriving setup jobs of full rank, and so on up to a certain upper bound on l. If no separating setup job can be derived any more, then we try to derive unrelated jobs. If this succeeds, then we compute lb again, and once more go through the entire process. The process is terminated if no separating and no unrelated setup job can be derived any more.

Also, we use several simple but effective dominance criteria to restrict the growth of the branch-and-bound tree. Let π be a partial schedule corresponding to an unfathomed node of the search tree and let $\{\pi\}$ be the set of jobs in π. If there is a $\pi^* \neq \pi$ with $\{\pi^*\} = \{\pi\}$ such that

$$L_{\max}(\pi^*\sigma) \leq L_{\max}(\pi\sigma), \tag{4.4}$$

for any sequence σ for the jobs in $\mathcal{J} \setminus \{\pi\}$, then we can discard π and fathom the node; π is then *dominated* by π^*. If condition (4.4) holds with equality for all σ, then we discard either π, or π^*. If $\mathcal{P} \neq \mathcal{NP}$, however, we can not verify in polynomial time whether this condition indeed holds. A strong *sufficient* condition for having π dominated by $\pi^* \neq \pi$ with $\{\pi^*\} = \{\pi\}$ is that

$$\begin{aligned} L_{\max}(\pi^*) &\leq \max\{lb, L_{\max}(\pi)\}, & (4.5)\\ C(\pi^*) &\leq C(\pi), & (4.6) \end{aligned}$$

and

$$C(\pi^*) + s(f'(\pi^*), f'(\pi)) \leq \max\{C(\pi), \min_{J_j \in \mathcal{F}_{f'(\pi)} \setminus \{\pi\}} r_j\}, \tag{4.7}$$

where $f'(\pi)$ denotes the family index of the last job in π and

$$s(f'(\pi^*), f'(\pi)) = \begin{cases} s_{f'(\pi)} & \text{if } f'(\pi^*) \neq f'(\pi) \\ 0 & \text{otherwise.} \end{cases}$$

Conditions (4.6) and (4.7) ensure that any job in $\mathcal{J} \setminus \{\pi\}$ can start in case of π^* as an initial sequence at least as soon as in case of π as an initial sequence. Of course, finding out whether there exists such a π^* is an $\mathcal{NP}$-complete problem. The following dominance rule is an easy-to-check sufficient condition for the existence of such a π^*.

Dominance Rule 4.1 *The partial schedule πJ_j can be discarded if there is some $J_k \in \mathcal{J} \setminus \{\pi J_j\}$ such that*

$$C(\pi J_k) + s(f(k), f(j)) \leq r_j.$$

On the other hand, verifying whether a given π^* satisfies conditions (4.5), (4.6), and (4.7) can be done in polynomial time. In the branch-and-bound tree, we consider therefore three promising options for π^*:

- If $\pi = \pi_1 J_j J_k$, then consider $\pi^* = \pi_1 J_k J_j$.
- If $\pi = \pi_1 J_j \pi_2 J_k$, with J_j the last job belonging to the same family as J_k, then consider $\pi^* = \pi_1 J_k \pi_2 J_j$ and $\pi^* = \pi_1 J_j J_k \pi_2$.

4.5.2 Computational experiments

The performance of the branch-and-bound algorithm was evaluated for instances with up to 50 jobs. All parameters were randomly generated from discrete uniform distributions, except for the release times: jobs arrive at the machine according to a Poisson process. The processing times were drawn from the discrete uniform distribution $[1, 100]$, the number of families F from the uniform distribution $[2, \lfloor n/5 \rfloor]$, and the family indices of the jobs from the uniform distribution $[1, F]$. Let $\bar{p}$ denote the average job processing time. In addition to n, there are four input parameters:

- s, defining the interval $[1, s \cdot \bar{p}]$ from which the setup times are uniformly drawn,
- a and k, defining the mean interarrival time $(\bar{p} + a \cdot \bar{s})/k$ of the jobs, where $\bar{s}$ is the average setup time, and
- d, defining the interval $[r_j + p_j, r_j + p_j + d \cdot \bar{p}]$ from which d_j is uniformly drawn.

We generated instances for $n = 30, 40, 50$, $s = 0.25, 0.50, 0.75$, $a = 0.25, 0.33, 0.5$, $k = 0.8, 0.9$ and $d = 2, 4, 6$. For each combination of n, s, a, k, and d, we generated 15 instances. Table 4.2 summarizes of our computational results for varying values of n, the number of jobs, and k, determining the arrival intensity. Crudely speaking, we

n	k	$\#opt$	$\#nodes$	$seconds$
30	0.8	401	35,614	0.7
30	0.9	395	48,688	0.9
40	0.8	385	40,357	0.9
40	0.9	355	107,832	2.4
50	0.8	358	83,544	2.1
50	0.9	295	131,112	3.1

Table 4.2: Performance of the branch-and-bound algorithm.

can say that k determines the workload in the shop: the larger k, the higher the workload. We found that the performance of the branch-and-bound algorithm does not significantly vary with the other parameters. The column '$\#opt$' gives the number of instances out of 405 for which the branch-and-bound algorithm found an optimal solution within one minute on an HP 9000/710 workstation. It shows that we solve almost all instances with $n = 30$. The next two columns give averages only for the instances solved to optimality within one minute. The column '$\#nodes$' gives the average number of nodes searched, and the column '$seconds$' gives the average computing time in seconds that the algorithm takes. The time for the preprocessing phase, i.e., for running the approximation algorithms and deriving the setup jobs, is not included here. The preprocessing phases typically takes about 2 to 4 seconds on the HP. Table 4.2 shows that the instances become more difficult with increasing number of jobs, as expected, and with increasing value of k. If the workload is high, i.e., if there are many jobs available at the same time for processing, then it is more difficult to derive setup jobs of high rank, and consequently, our lower bounds become less effective with increasing value of k. Table 4.2 also shows that the instances that we can solve within the time limit take little time on average. This suggests a considerable watershed between computationally easy and hard instances.

Table 4.3 presents, for varying n and k, the results of the preprocessing step for those instances that were solved to optimality within one minute. The column '*lb*1' gives the average preemptive lower bound in the root node of the search tree without the addition of derived setup jobs. The column '*lb*2' gives this lower bound with the addition of the setup jobs. The average value of the initial solution found by our approximation algorithm is found in the column '*ub*'. The average optimal solution value is given in the column '*opt*'. We see that the gap between the initial lower bound *lb*1 and the optimal solution value *opt* is approximately halved by the addition of the setup jobs. The average number of derived setup jobs is given in the column '*derived*', whereas the average number of setups in the optimal solution we found is given in the column '*setups*'. Note that in general there exist more than one optimal solution and each may have a different number of setups.

n	k	*lb*1	*lb*2	*ub*	*opt*	*derived*	*setups*
30	0.8	96.4	125.1	154.0	152.8	14.4	18.4
30	0.9	120.8	152.7	188.1	186.5	12.8	17.4
40	0.8	118.3	153.9	185.0	183.9	19.6	25.7
40	0.9	150.5	188.5	226.7	224.7	17.2	24.2
50	0.8	126.5	171.3	204.3	202.6	25.5	33.5
50	0.9	158.5	200.1	240.4	237.7	22.4	31.2

Table 4.3: Results of preprocessing: solvable instances.

Table 4.4 presents the same information for those instances for which the algorithm failed to find an optimal solution within one minute. Since

n	k	*lb*1	*lb*2	*ub*	*ub**	*derived*
30	0.8	342.2	409.2	468.0	468.0	8.8
30	0.9	375.1	433.1	522.3	520.5	7.6
40	0.8	292.4	374.2	471.7	470.1	13.2
40	0.9	334.5	425.0	517.3	515.8	12.8
50	0.8	284.4	377.5	471.6	468.9	19.1
50	0.9	332.8	430.4	533.5	532.1	17.8

Table 4.4: Results of preprocessing: hard instances.

we do not have the optimal solution values, we have added the column ub^*, which gives the average value of the incumbent upper bound after one minute of computation time.

Our computational results did not reveal any relation between the difficulty of an instance and the choices of the parameters a, s, and d; the difficulty of an instance primarily depends on how close the release dates are to each other. Close release dates are most likely to occur in case of a high workload parameter k. The performance of the algorithm deteriorates in case of close release dates for two reasons. First, such release dates in combination with the almost agreeable due dates lead to a considerable lateness. This makes that $\bar{d}_j$, the latest possible completion time of job J_j, is relatively large; we then may expect to have few sets for which $\bar{d}(\mathcal{P}^i_{1,k}) < r(\mathcal{P}^i_{k+l,|\mathcal{F}_i|})$, and as a result, fewer setup jobs and thereby weaker lower bounds. Second, if the release dates are close to each other, then certain dominance criteria in our branch-and-bound algorithm are less effective. Indeed, Table 4.4 confirms our expectations: it shows that difficult instances have larger $L^*_{\max}$ and permit fewer setup jobs than the solvable instances.

4.6 Conclusions

This chapter has discussed our branch-and-bound algorithm for the problem $1|r_j, s_i|L_{\max}$. For lower bounding purposes, we have observed that setup times can be seen as setup jobs. We have derived two types of setup jobs: separating setup jobs and unrelated setup jobs. Some simple dominance rules effectively restrict the growth of the branch-and-bound tree. The computational experiments indicate that the branch-and-bound algorithm solves instances with up to 40 jobs in reasonable time. The next chapter discusses the parallel-machine counterpart of the problem $1|r_j, s_i|L_{\max}$.

Chapter 5

Parallel-machine scheduling with family setup times

5.1 Introduction

This chapter discusses the parallel-machine version of the problem discussed in the previous chapter: the problem $P|r_j, s_i|L_{\max}$. Parallel-machine scheduling comes down to assigning each job to one of the parallel machines and sequencing the jobs assigned to the same machine.

The plan of this chapter is as follows. First, we present in Section 5.2 a characterization of an optimal schedule for a class of parallel-machine scheduling problems. The problem $P|r_j, s_i|L_{\max}$ belongs to this class. In Section 5.3, we describe a branch-and-bound algorithm for this problem. This algorithm builds on the algorithm for the single-machine case discussed in the previous chapter. Section 5.4 reports on some implementation aspects and computational results. Section 5.5 concludes this chapter. This chapter is based on Schutten and Leussink [74] and Schutten [71].

5.2 Characterization of an optimal schedule

In this section, we consider the problem of scheduling a set $\mathcal{J}$ consisting of n independent jobs $J_1, J_2, \ldots, J_n$ on m identical parallel machines $M_1, M_2, \ldots, M_m$. In contrast to the remainder of this chapter, we consider in this section *sequence dependent* setup times, i.e., there is a setup time s_{ij} between the processing of job J_i and J_j. Before the first job J_j on each machine, we need a setup time s_{0j}. Each job J_j $(j = 1, \ldots, n)$ must be processed without preemption on exactly one of the machines during a given non-negative time p_j and may have a release date and a due date. Each machine M_k $(k = 1, \ldots, m)$ is available from a given non-negative time S_k onwards and can process at most one job at a time. A *schedule* σ specifies for each job J_j a completion time $C_j(\sigma)$. The quality of schedule σ is measured by a *regular* objective function $f(\sigma)$ that needs to be minimized. An objective function f is called regular if $f(\sigma_1) > f(\sigma_2)$ implies that $C_j(\sigma_1) > C_j(\sigma_2)$ for at least one j $(j = 1, \ldots, n)$; cf. Baker [6]. For any scheduling problem with a regular objective function, there always exists a left-justified schedule that is optimal.

In Section 5.2.1, we explain the concept of list scheduling. We point out that list schedules need not be dominant for problems with sequence dependent setup times if the list scheduling algorithm focuses on the starting times of the jobs. In Section 5.2.2, we prove that the list schedules are dominant if the list scheduling algorithm focuses on the completion times of the jobs.

5.2.1 Standard list scheduling

Given a certain list or permutation π of the job set $\mathcal{J}$, a standard list scheduling algorithm constructs a schedule for the parallel machines in the following way: the next job on the list is scheduled on the machine that becomes available first. If a tie exists, then the job is usually scheduled on the machine with the smallest index. Thus, this algorithm focuses on the starting times of the jobs. The schedule that results from the list π is denoted by LIST(π). Several authors analyze the worst-case performance of list scheduling algorithms. For instance, Graham [37] analyzes the worst-case performance of the list scheduling algorithm with the jobs sorted in order of non-increasing processing times for the problem $P||C_{\max}$. List schedules are also used in branch-and-bound algorithms for problems in which the set of list schedules is *dominant*, i.e., contains at least one optimal solution. Woerlee [85], for instance, develops such a branch-and-bound algorithm for the problem $P|r_j|L_{\max}$.

A schedule in which no machine is kept idle when there is a job available for processing is called a *non-delay* schedule. Given a non-delay schedule σ, there is exactly one list π such that $\mathrm{LIST}(\pi) = \sigma$: the ith job of π is the job with the ith smallest starting time in σ. Therefore, if for a certain problem the non-delay schedules are dominant, then enumerating all possible lists and evaluating the resulting list schedules yields an optimal solution; see also Elmaghraby and Park [25].

For problems with setup times, however, the set of non-delay schedules is not dominant. Ovacik and Uzsoy [62] present an instance of the problem $P|s_{ij}|C_{\max}$ for which the set of list schedules contains no optimal solution. The data of this instance with two machines and three jobs are given in Table 5.1. Due to the large setup times, it is evident

J_j	p_j
J_1	1
J_2	2
J_3	3

s_{ij}	j		
i	1	2	3
0	1	1	10
1	0	10	10
2	10	0	1
3	10	10	0

Table 5.1: Processing times and setup times for counter example.

that J_3 must be scheduled last. So, the only relevant lists are $J_1 - J_2 - J_3$ and $J_2 - J_1 - J_3$, but neither list gives the optimal solution, as shown in Figure 5.1. For problems with general release and due dates the

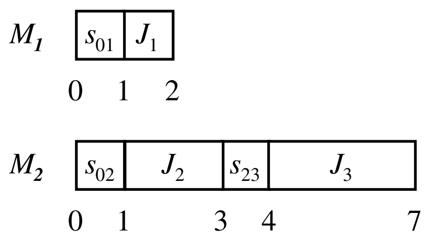


Figure 5.1: Optimal solution.

non-delay schedules are not dominant either.

In the following section, we present an alternative list scheduling algorithm. This algorithm focuses on the completion times of the jobs instead of the starting times. We show that the list schedules obtained in this way are dominant, even for problems with sequence dependent setup times. For problems without setup times, this alternative list scheduling algorithm is equal to the standard list scheduling algorithm.

5.2.2 An alternative list scheduling algorithm

Recall that standard list scheduling algorithms assign the next job of the list to the machine that becomes *available* first. In the alternative list scheduling algorithm we assign the next job of the list to the machine on which it will be *completed* first. The schedule that results from list π using the latter algorithm is denoted by $g(\pi)$. Note that for problems without setup times $\text{LIST}(\pi) = g(\pi)$ for every list π. Note also that $g(\pi)$ is constructed in $O(nm)$ time, whereas $\text{LIST}(\pi)$ takes $O(n \log m)$ time. Cho and Sahni [20] also use a list scheduling algorithm that focuses on the completion times. They derive a worst-case performance ratio of this algorithm for the problem $Q||C_{\max}$ and special cases of it.

The following theorem proves that the list schedules obtained with the alternative list scheduling algorithm are dominant for a broad class of parallel-machine scheduling problems; cf. Schutten [71].

Theorem 5.1 *Suppose that a set of jobs needs to be scheduled without preemption on identical parallel machines. The jobs have release dates, setup times need to be taken into account, and some regular cost function needs to be minimized. Then, there exists a list π such that $g(\pi)$ is an optimal schedule.*

Before we prove this theorem, we introduce some additional notation. Let π be any list of a subset of $\mathcal{J}$. $g(\pi)$ is then a *partial* schedule. Denote by $n_i(\sigma)$ $(i = 1, \ldots, m)$ the number of jobs that are scheduled on machine M_i in the (partial) schedule σ and let Ω be the set of *optimal* complete schedules. We say that a partial schedule σ' *deviates* from a complete schedule σ if one of the following conditions holds:

1. $n_i(\sigma') > n_i(\sigma)$ for at least one i $(1 \leq i \leq m)$;
2. the jth job on machine M_i in σ' is not the jth job on machine M_i in σ for some i and j $(1 \leq i \leq m; 1 \leq j \leq n_i(\sigma'))$.

Let $\tau(\sigma)$ be the maximum number of jobs that any list π can contain such that $g(\pi)$ does not deviate from a given σ, that is,

$$\tau(\sigma) = \max_{\pi \in \Pi}\{|\pi| \mid g(\pi) \text{ does not deviate from } \sigma\},$$

where Π is the set of all possible lists of subsets of $\mathcal{J}$ and $|\pi|$ is the number of jobs in π. We are now ready to prove the theorem. We do this by contradiction.

Proof of Theorem 5.1. Let $\sigma^* \in \Omega$ be any optimal schedule for which

$$\tau(\sigma^*) = \max_{\sigma \in \Omega} \tau(\sigma).$$

If the theorem does not hold, then $\tau(\sigma^*) < n$. Let π^* be such that it contains $\tau(\sigma^*)$ jobs and $g(\pi^*)$ does not deviate from σ^*.

Consider the directed graph $D = (V, A)$ with a node $v_i \in V$ for each machine M_i $(i = 1, \ldots, m)$. Suppose that $n_i(\sigma^*) > n_i(g(\pi^*))$. Let $J_{[i]}$ be the first job on M_i in σ^* that is not in $g(\pi^*)$ and let R_i be the job sequence that consists of $J_{[i]}$ and its successors on M_i in σ^*. Note that R_i contains precisely those jobs on M_i in σ^* that are not in $g(\pi^*)$. Suppose that $J_{[i]}$ is scheduled on machine M_j in $g(\pi^* J_{[i]})$. Then we have that $j \neq i$, because $\tau(\sigma^*)$ is maximal. Draw an arc in D from v_i to v_j. Do this for every machine M_i with $n_i(\sigma^*) > n_i(g(\pi^*))$. We can distinguish two cases:

1. There is a $v_j \in V$ with an incoming arc and no outgoing arc, e.g., $(v_i, v_j) \in A$. Then we know that in σ^* and in $g(\pi^*)$ exactly the same jobs are scheduled on M_j, because otherwise v_j would have had an outgoing arc. Also,

$$C_{[i]}(g(\pi^* J_{[i]})) \leq C_{[i]}(\sigma^*),$$

   due to the way $J_{[i]}$ is assigned to a machine in $g(\pi^* J_{[i]})$. Let σ' be the schedule obtained from σ by moving the sequence R_i to M_j. σ' is also optimal, because $C_{[i]}(\sigma') \leq C_{[i]}(\sigma)$, and therefore $C_j(\sigma') \leq C_j(\sigma)$ for $j = 1, \ldots, n$. What is more, $g(\pi^* J_{[i]})$ does not deviate from $\sigma' \in \Omega$, which is a contradiction with the maximality of $\tau(\sigma^*)$.

2. There is no $v_j \in V$ with an incoming arc and no outgoing arc. This means that each $v_j \in V$ has either an outgoing arc, or neither an incoming nor, an outgoing arc. Then must D contain at least one directed cycle K. Without loss of generality, we assume that $K = v_1 v_2 \ldots v_p v_1$. Let σ' be the schedule obtained from σ by moving the sequences $R_1, \ldots, R_p$ as follows: move sequence R_1 to

machine M_2, R_2 to M_3,...,R_{p-1} to M_p, and R_p to M_1. Using the same arguments as in case 1, we conclude that σ' is optimal, too. Since $\pi^* J_{[1]}$ does not deviate from σ', this is, again, a contradiction with the maximality of $\tau(\sigma^*)$. □

Theorem 5.1 implies that a set of at most $n!$ schedules is dominant for many parallel-machine scheduling problems. Note that different lists may result in the same schedule. Dominance rules that prevent exploring several lists that result in the same schedule can even further reduce this set.

The proof of Theorem 5.1 depends on the fact that $C_{[i]}(\sigma') \leq C_{[i]}(\sigma^*)$ implies that $C_j(\sigma') \leq C_j(\sigma^*)$ for all jobs J_j in the sequence R_i. This condition does not hold for non-identical parallel-machine scheduling problems. Hence, Theorem 5.1 does not hold for uniform and unrelated parallel-machine scheduling problems.

5.3 A branch-and-bound algorithm

In this section, we discuss the branch-and-bound algorithm for the problem $P|r_j, s_i|L_{\max}$. This algorithm exploits Theorem 5.1. In Sections 5.3.1-5.3.4, the main ingredients of the algorithm are discussed, such as the lower and upper bounds and the dominance rules.

5.3.1 The search tree

We adopt a *forward branching rule:* each node at level k of the search tree corresponds to a permutation π consisting of k jobs ($k = 0, \ldots, n$). A node at level k has $n-k$ descendant nodes: one for each unscheduled job. We employ an *active node search:* we branch only from one node at a time, thereby adding some unscheduled job J_j to π, which leads to the sequence πJ_j. We branch from the nodes in order of non-decreasing release dates. We backtrack at level n, or if we can discard the active node.

5.3.2 Upper bounds

We used several constructive heuristics to generate upper bounds on $L^*_{\max}$. Some of these heuristics are based on dispatching rules and some are based on cheapest insertion. Test results for these algorithms can be found in Section 5.4.2.

5.3.3 Lower bounds

An important issue for computing lower bounds on $L^*_{\max}$ is how to take into account the necessary setups. In the previous chapter, we have seen that setups can be viewed as setup jobs with specific release and due dates and processing times. Also, sufficient conditions are given that ensure when a setup job may be introduced. For the problem $P|r_j, s_i|L_{\max}$, we use setup jobs as well to compute lower bounds. We derive setup jobs in the same way as in the previous chapter. We use two kinds of lower bounds. One that follows by allowing preemption, i.e., a job may be interrupted and resumed later on. The second lower bound is based on a lower bound for the problem $P|r_j|L_{\max}$ given by Carlier [18].

The preemptive lower bound

The problem $1|r_j, pmtn|L_{\max}$ is easy to solve by Horn's rule (Horn [42]). This rule schedules at each moment in time the job with the smallest due date among all available jobs; see Section 4.4.1. One might expect that this rule also results in an optimal schedule for the problem $P|r_j, pmtn|L_{\max}$. This is not true. To see this, consider the example from Table 5.2 with four jobs and two machines.

J_j	r_j	p_j	d_j
J_1	0	10	10
J_2	3	10	35
J_3	4	30	40
J_4	0	15	15

Table 5.2: Data for a counter example

Horn's rule gives the schedule J_1J_2 on machine M_1 and J_4J_3 on machine M_2. The maximum lateness of this schedule is 5. The optimal schedule, however, is J_1J_3 on machine M_1 and J_4J_2 on machine M_2 with maximum lateness 0.

Suppose that we want to compute a lower bound in a node of the branch-and-bound tree. We are interested only whether this lower bound is at least ub, where ub is an upper bound on $L^*_{\max}$. Suppose π is the permutation of a subset of $\mathcal{J}$ that is associated with this node. We now contruct a graph D such that there exists a flow in D of value $\sum_{J_j \in \mathcal{J} \setminus \pi} p_j$ if and only if a preemptive schedule for the unscheduled jobs exists with maximum lateness smaller than ub.

Let c_i be the completion time of the last job on machine M_i in $g(\pi)$. Without loss of generality, we assume that $c_1 \leq c_2 \leq \ldots \leq c_m$. Each job $J_j \in \mathcal{J} \setminus \pi$ must be completed before $\overline{d}_j = d_j + ub - 1$; otherwise, the lateness of J_j is at least ub, and π does not lead to an improvement of ub. Let T be the collection of points in time $\{r_j | J_j \in \mathcal{J} \setminus \pi\} \cup \{\overline{d}_j | J_j \in \mathcal{J} \setminus \pi\} \cup \{c_1, \ldots, c_m\}$. Define $q := 2 \cdot (n - |\pi|) + m$; q is the maximum number of elements in T. Let t_i $(1 \leq i \leq q)$ be such that $t_i \in T$, $\bigcup_{i=1}^{q} t_i = T$ and $t_1 \leq t_2 \leq \ldots \leq t_q$. We may assume that no job is interrupted at any moment in time, except, possibly, at $t_i \in T$.

Construct a directed graph $D = (V, A)$ with:

$$V = \{s_1\} \cup \{v_j | J_j \in \mathcal{J} \setminus \pi\} \cup \{w_1, w_2, \ldots, w_{q-1}\} \cup \{s_2\},$$

where s_1 is the source and s_2 the sink; v_j is a node associated with the unscheduled job J_j; $w_i \in V$ is associated with the interval $[t_i, t_{i+1}]$. D has the following arcs:

- (s_1, v_j) with capacity p_j;
- (v_j, w_i) if $r_j \leq t_i$ and $\overline{d}_j \geq t_{i+1}$ with capacity $t_{i+1} - t_i$;
- (w_i, s_2) with capacity $\max\{k | 1 \leq k \leq m, c_k \leq t_i\} \cdot (t_{i+1} - t_i)$.

The number $\max\{k | 1 \leq k \leq m, c_k \leq t_i\}$ is the number of machines that is available in the interval $[t_i, t_{i+1}]$ for processing jobs from $\mathcal{J} \setminus \pi$.

Example 5.2

Figure 5.2 shows the graph D for the instance in Table 5.2 with $ub = 5$ and $\pi = J_1 J_2$. This means that J_1 is scheduled on machine M_1 and completes at time 10. J_2 is scheduled on machine M_2 and completes at time 13; therefore, $c_1 = 10$ and $c_2 = 13$. The deadlines

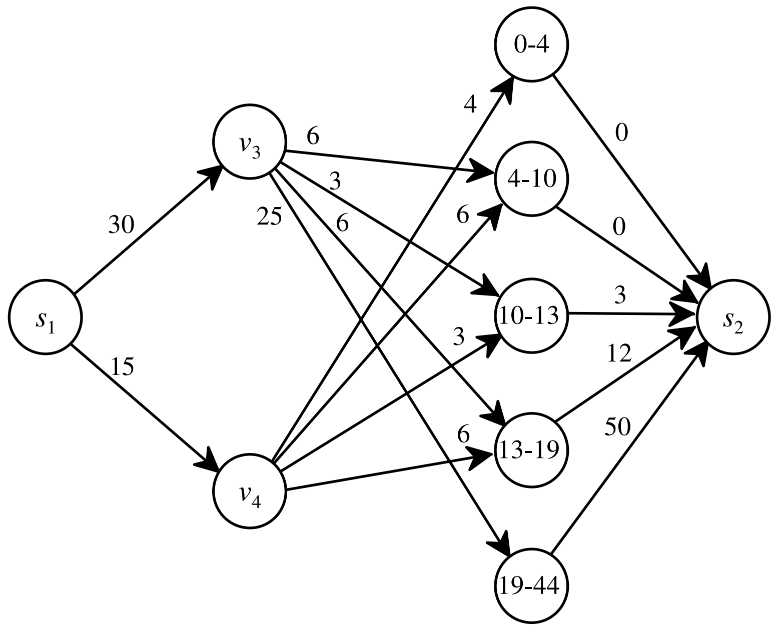


Figure 5.2: The graph D.

for the jobs J_3 and J_4 with $ub = 5$ are 44 and 19, respectively. Then, $T = \{r_3, r_4, \overline{d}_3, \overline{d}_4, c_1, c_2\} = \{4, 0, 44, 19, 10, 13\} = \{0, 4, 10, 13, 19, 44\}$. We denoted the node w_i associated with time interval $[t_i, t_{i+1}]$ by t_i-t_{i+1}; so, 13-19 is the node corresponding to the interval $[13, 19]$. The numbers at the arcs denote their capacities. Note that during the intervals $[0, 4]$ and $[4, 10]$ no machine is available for processing either job J_3 or J_4. We could have left the nodes corresponding to these intervals out of the graph.

In D, there exists a flow of value $\sum_{J_j \in \mathcal{J} \setminus \pi} p_j$ if and only if a preemptive schedule for the unscheduled jobs exists with maximum lateness smaller than ub (see, e.g., Labetoulle et al. [50] and Federgruen and Groenevelt [26]). So the problem of determining whether the maximum lateness in an optimal preemptive schedule is smaller than ub is equivalent to finding a maximum flow in the constructed graph D. Therefore, the preemptive lower bound can be computed in $O(n^3)$ time.

Carlier's lower bound

Carlier [18] gives a lower bound on the optimal makespan for the parallel-machine scheduling problem in which the jobs have heads and tails. Based on this lower bound, one can derive a lower bound for the problem $P|r_j|L_{\max}$.

Let $\mathcal{A} \subseteq \mathcal{J} \setminus \pi$ and $w := \min(m, |\mathcal{A}|)$. Suppose $J_{i_1}, \ldots, J_{i_w}$ are the jobs in $\mathcal{A}$ with the w smallest release dates and $J_{j_1}, \ldots, J_{j_w}$ the jobs in $\mathcal{A}$ with the w largest due dates. Then,

$$G(\mathcal{A}) := \frac{(r_{i_1} + \ldots + r_{i_w}) + \sum_{J_j \in \mathcal{A}} p_j - (d_{j_1} + \ldots + d_{j_w})}{w}$$

is a lower bound on $L^*_{\max}$, for any $\mathcal{A} \subseteq \mathcal{J}$. We refer to this lower bound as Carlier's lower bound.

We have chosen to compute Carlier's lower bound for subsets $\mathcal{B}_k$ and $\mathcal{D}_k$ of $\mathcal{J}$, with $\mathcal{B}_k$ containing the k jobs with the k largest release dates, and $\mathcal{D}_k$ containing jobs with the k smallest due dates $(1 \leq k \leq |\mathcal{J} \setminus \pi|)$.

Of course, $\max_{J_j \in \mathcal{J} \setminus \pi} \{r_j + p_j - d_j\}$ is also a lower bound on the optimal maximum lateness of jobs in $\mathcal{J} \setminus \pi$.

5.3.4 Dominance rules

The dominance rules presented in Chapter 4 for the $1|r_j, s_i|L_{\max}$ problem are valid for the individual machines in the current problem. The most important dominance rule states that whenever a release date of a job J_j is "too large", then πJ_j is dominated by πJ_k for some $J_k \in \mathcal{J} \setminus \pi$. "Too large" can be expressed more formally by $r_j \geq C_k(g(\pi J_k)) + s(f(k), f(j))$. If this dominance rule holds for J_j and J_k, then there is idle time before the processing of J_j in $g(\pi J_j)$. This idle time can be used for processing J_k without interfering with the processing of J_j.

Next, we mention a dominance rule that is specific for parallel-machine scheduling problems. Suppose that π_1 and π_2 are different permutations of the same subset of $\mathcal{J}$. If $g(\pi_1) = g(\pi_2)$, then π_1 is dominated by π_2, and vice versa. So, we may discard one of them. We can sharpen this rule slightly by saying that two schedules are equal (possibly after renumbering the machines) if they have the same maximum lateness and each machine has the same completion time and ends with a job from the same family in both schedules.

5.4 Implementation and computational experiments

5.4.1 Implementation

In Section 5.3.3, we presented two lower bounds: one based on a max-flow algorithm for the preemptive case and one based on Carlier's [18] lower bound. In terms of quality, the preemptive lower bound dominates Carlier's lower bound. In terms of speed, however, it is the other way around: Carlier's lower bound takes $O(n \log m)$ time and the preemptive lower bound $O(n^3)$ time. We tested three versions of our branch-and-bound algorithm: one that uses the preemptive lower bound only, one that uses Carlier's lower bound only, and one that first computes Carlier's lower bound and then, if necessary, the preemptive lower bound.

5.4.2 Computational experiments

The performance of the branch-and-bound algorithm was evaluated for instances with up to 25 jobs and two or three machines. All parameters were randomly generated from discrete uniform distributions, except for the release dates that come from a Poisson process. The processing times were drawn from the interval $[1, 100]$, the number of families F from the interval $[2, \lfloor n/5 \rfloor]$, and the family indices of the jobs from $[1, F]$. Let $\overline{p}$ denote the average processing time for the jobs. In addition to n and m, there are three input parameters:

- k, defining the mean interarrival time $k \cdot \overline{p}/m$,
- t, defining the interval $[r_j + p_j, r_j + p_j + t \cdot \overline{p}]$ from which the due date of job J_j is drawn,
- s, defining the interval $[1, s \cdot \overline{p}]$ from which the setup times are drawn.

For $n = 10, 15, 20, 25$, $m = 2, 3$, $k = 1, 1.2, 1.4$, $t = 1, 3$ and $s = 0.2, 0.6, 1$ we generated instances. For each combination of n, m, k, t and s, we generated 15 instances. Table 5.3 summarizes our computational results for varying values of n and k.

n	k	heur	Car	%Car	pmtn	%pmtn	both	%both
10	1.0	30	180	100	180	100	180	100
10	1.2	36	180	100	180	100	180	100
10	1.4	47	180	100	180	100	180	100
15	1.0	15	175	97.2	167	92.8	169	93.9
15	1.2	23	179	99.4	174	96.7	178	98.9
15	1.4	40	180	100	180	100	180	100
20	1.0	8	134	74.4	111	61.7	114	63.3
20	1.2	17	166	92.2	147	81.9	153	85.0
20	1.4	28	177	98.3	167	92.8	171	95.0
25	1.0	10	95	52.8	71	39.4	79	43.9
25	1.2	8	128	71.1	113	62.8	115	63.9
25	1.4	16	154	85.6	132	73.3	139	77.2

Table 5.3: Test results for varying n and k.

The parameter k determines the workload on the machines: the smaller k, the higher the workload. The column 'heur' gives the number of times out of 180 that one of the heuristics found an optimal solution. The column 'Car' gives the number of times the algorithm found an optimal solution within one minute on an HP 9000/710 workstation using Carlier's lower bound only. The column '%Car' gives this number as a percentage of the 180 instances. The columns 'pmtn', '%pmtn', 'both' and '%both' give the same data, but now for the algorithm that uses the preemptive lower bound only, and for the algorithm with both lower bounds. We see that the algorithm using Carlier's lower bound only gives the best results: it is the only algorithm that solves almost all instances with up to 15 jobs. Also, the instances with a workload less than 100% ($k = 1$) are solved quite often for problems with up to 20 jobs. The reason for this is that although more nodes are searched, the time per node is much smaller in this algorithm, compared with the algorithms that use the preemptive lower bound. This can also be concluded from Table 5.4. In this table, the column 'n_Car' gives the mean number of nodes searched in the algorithm with Carlier's lower bound only and the column 's_Car' gives the mean computation time in seconds for this algorithm. The means are taken over those instances that are solved within one minute. The columns 'n_pmtn', 's_pmtn',

n	k	n_Car	s_Car	n_pmtn	s_pmtn	n_both	s_both
10	1.0	811	0.2	484	0.5	484	0.3
10	1.2	481	0.1	343	0.3	343	0.2
10	1.4	242	0.0	182	0.1	182	0.1
15	1.0	14,936	2.4	3,695	5.1	4,377	3.9
15	1.2	7,314	1.0	2,789	3.0	3,674	2.9
15	1.4	3,615	0.6	2,181	2.5	2,181	1.8
20	1.0	40,822	6.1	6,272	10.0	7,610	7.6
20	1.2	36,054	5.2	6,712	7.7	8,831	7.0
20	1.4	15,685	2.2	4,633	4.5	5,642	4.3
25	1.0	52,642	6.6	6,323	7.7	11,053	9.2
25	1.2	37,391	4.4	6,464	7.6	7,401	5.9
25	1.4	34,786	4.3	6,287	6.6	8,844	7.0

Table 5.4: Mean number of nodes investigated and mean computation time.

'n_both' and 's_both' give the corresponding data for the algorithm with the preemptive lower bound only and the algorithm with both lower bounds.

For the instances that were solved to optimality with the algorithm using Carlier's lower bound only, Table 5.5 displays the mean optimal maximum lateness in the column '$L^*_{\max}$', and the mean lower and upper bound in the root of the search tree in the columns 'lb' and 'ub', respectively.

5.5 Conclusions

This chapter discussed the branch-and-bound algorithm we have developed for the problem $P|r_j, s_i|L_{\max}$. This algorithm is based on the algorithm for the single-machine case discussed in Chapter 4. In the single-machine case, a dominance rule, concerning the release date of a job being too large, was effective. This dominance rule does not work as well in this problem, because the mean interarrival time in this problem is, of course, smaller. We have observed that Carlier's lower bound is more effective than the preemptive lower bound.

n	k	lb	$L^*_{\max}$	ub
10	1.0	7.0	11.6	29.0
10	1.2	-0.1	2.6	19.0
10	1.4	-3.1	-2.4	14.3
15	1.0	24.4	35.2	51.3
15	1.2	11.5	20.0	37.9
15	1.4	5.2	9.2	26.1
20	1.0	37.6	59.1	77.4
20	1.2	17.4	29.9	46.8
20	1.4	8.7	15.8	33.0
25	1.0	53.0	83.5	99.0
25	1.2	28.1	51.6	64.5
25	1.4	19.3	33.9	51.5

Table 5.5: The optimal maximum lateness and lower and upper bounds.

In the next chapter, we test the Shifting Bottleneck (SB) procedure in a machine shop in which setup times occur. The decomposition for such shops results in single-machine scheduling problems with setup times. We use optimization algorithms and some heuristics for solving these single-machine scheduling problems. For both cases, the performance of the SB procedure is evaluated.

Chapter 6

Assembly shop scheduling

6.1 Introduction

In Section 1.2, we motivated the need for integral shop floor scheduling procedures, referring to changing market conditions, in particular the demand for short and reliable delivery times. Up to now, we have restricted our attention to machine shops producing jobs with *linear* routings, i.e., each job consists of a chain of operations. After completion, these jobs are delivered, or used in the assembly department. Scheduling systems usually do not consider the interaction between manufacturing and assembly departments. Current practice in discrete manufacturing is to use a global capacity planning and materials coordination system, such as Manufacturing Resource Planning (MRP II; see Wight [84]), for the coordination between departments, whereas shop floor scheduling systems, if used at all, are implemented at department level.

Shorter and more reliable leadtimes can be realized if the scheduling system includes both manufacturing and assembly, as well as materials procurement, engineering, and so on. MRP II systems use a *fixed* off-set leadtime for manufacturing a part or assembling a product. Manufacturing leadtimes depend, however, on the actual work load, product mix, and lot-sizes. MRP II systems try to circumvent coordination problems by making the fixed off-set leadtime large, e.g., by allowing a pre-set waiting or queueing time for each operation. In MRP II, the capacity planning function is usually a simple "bucket filling" procedure that ignores job interference due to different routings. This approach is simple, but does not lead to a flexible production system, able to respond quickly to a variable market demand. Indeed, practice shows that MRP II systems generally perform reasonably well in a stable make-to-stock or assemble-to-stock environment, but are inadequate in small batch parts manufacturing shops. A natural question is therefore whether for such make-to-order companies intelligent scheduling systems can be developed that include both manufacturing and assembly operations.

Problem complexity has often been used as an argument against integral scheduling, together with the observation that many a disturbance would quickly render such an integral schedule obsolete. As to the first argument, we now have adequate hardware, software, and scheduling algorithms to cope with the complexity. As to the other argument, it appears that many of the disturbances are caused by inaccurate scheduling. For instance, if the availability of scarce secondary resources, such as cutting tools and fixtures, are not considered, then it is no surprise that operations are delayed due to the absence of these resources; cf. Meester [56] and Chapter 3 in this thesis. In addition, *rescheduling*, the most obvious way to deal with disturbances, is now a feasible option thanks to the increased computer power.

Specifically, the Shifting Bottleneck (SB) procedure, as outlined in Chapters 2 and 3, provides a good basis for the development of intelligent heuristics that allow integration of parts manufacturing and assembly scheduling. To demonstrate this, we study in this chapter some simple assembly scheduling problems and show that the extended SB procedure yields satisfactory results.

Literature on *assembly shop scheduling*, i.e., machine scheduling where jobs may have convergent routings, mainly concentrates on empirical analysis of priority rules; see, for example, Maxwell and Mehra [55], Russell and Taylor [69], Goodwin and Weeks [35], and Miller et al. [58]. Recently also a number of papers have appeared that study the complexity of assembly scheduling problems and present heuristics with a performance guarantee as well as optimization algorithms. Lee et al. [53] consider a machine shop with three machines. Two machines produce components, whereas the third machine assembles those components. The objective is to minimize the makespan. They prove the $\mathcal{NP}$-hardness of this problem, propose a branch-and-bound algorithm, and develop heuristics based on Johnson's algorithm (Johnson [45]) for the problem $F2||C_{\max}$. One heuristic has a *worst-case performance ratio* of $\frac{3}{2}$, i.e., the makespan of the schedule generated with this heuristic is at most $\frac{3}{2}$ times the optimal makespan. Potts et al. [64] consider a similar problem, but now with m machines that produce components. They also prove the $\mathcal{NP}$-hardness of the problem and develop a heuristic, which is basically a generalization of the heuristic of Lee et al. [53], with a worst-case performance ratio of $2 - \frac{1}{m}$. Another heuristic has an *absolute* performance guarantee of $\frac{5}{4}p_{\max}$, i.e., the difference of the makespan of the schedule generated by this heuristic and the optimal makespan is at most $\frac{5}{4}p_{\max}$, where $p_{\max}$ is the maximum processing time of any operation. Brucker and Thiele [14] develop a branch-and-bound algorithm for machine shops in which sequence dependent setup times occur and with arbitrary precedence relations between operations. This algorithm is based on the algorithm of Brucker et al. [13], discussed in Section 2.3.1. The assembly shop scheduling problem is a special case of the problem they consider. Due to the hardness of the problem, they can solve only relatively small instances. Coffman et al. [21] consider a single-machine scheduling problem in which the machine produces two component types and assembles them to end products. Whenever the machine switches from producing one component type to the other, a setup time is needed. Coffman et al. present efficient algorithms that minimize the *total flow time* $\sum C_j$.

In the next section, we start to investigate some small parts manufacturing and assembly scheduling problems and show how the techniques developed in preceding chapters can be used in an overall scheduling procedure. Next, we turn to more realistic, larger problems in which we necessarily have to use heuristics instead of exact algorithms for the single-machine scheduling problems. We maintain the iterative decomposition approach of the SB procedure for the overall coordination of the activities, however. We compare the empirical performance of the extended SB procedure with priority rules that reportedly perform well. In Section 6.3, we end this chapter with conclusions. This chapter is based on Schutten [72].

6.2 Computational experiments

The SB procedure has proven to be an effective algorithm for the classic job shop problem, cf. Vaessens et al. [79]. In this section, we analyze the empirical performance of the procedure in assembly shops. We study the effect of static and dynamic scheduling, setup times, batch sizes, and the job arrival process.

6.2.1 A small test shop

The first test shop that we consider has four machines $M_1, \ldots, M_4$. M_1 produces component A, M_2 produces component B, and M_3 produces component C. A job is an order to produce a product consisting of a single component that is no part of an assembly, or an order to produce a product consisting of different components that need to be assembled. M_4 is used to assemble components. An assembly can start only when all the required components are available. Jobs arrive for producing the job types A, B, C, AB, AC, and BC. Jobs arrive at the shop according to a Poisson process with a mean interarrival time of 60 time units. Upon arrival, we determine what type of product is ordered. Each type has an equal probability of being chosen. We draw the processing times of operations on M_1, M_2, and M_3 from the discrete uniform distribution on the interval $[85, 115]$. For operations on M_4, we draw the processing times from the discrete uniform distribution on the interval $[65, 85]$. Whenever M_4 switches to assembling other components, we need a setup time of 120 units. The due date of job J_j is set to $d_j = r_j + P_j + n_j \cdot D_j$, with r_j the release date (equal to the arrival time), P_j the total processing time, and n_j the number of operations of J_j. We draw D_j from the discrete uniform distribution on the interval $[200, 400]$. The generated instances contain 60 jobs.

Priority rules

A practical way to schedule jobs in a shop is by use of priority rules. In the literature, priority rules are extensively tested for assembly shops, cf. Russell and Taylor [69] and Fry et al. [29]. We compare the performance of the SB procedure with two types of priority rules which have been shown to perform well for assembly shops without setup times:

1. $\rho_{ij} = -d_j$,

and

2. $\rho_{ij} = -slack_{ij}$,

with ρ_{ij} the priority of operation O_{ij}, and $slack_{ij}$ the slack of this operation, that is, $slack_{ij} = d_j - P_{ij}$, where P_{ij} denotes the total processing time of O_{ij} and of operations of job J_j that need to be processed after operation O_{ij}. So, the first priority rule focuses on the due date of the job; the second one focuses on the slack that operations have. For operations on a machine with setup times, we decrease the priority of an operation with $\beta \cdot setup$, with *setup* the required setup time. We vary β from 0 to 20 with steps of 0.5, which results in 82 combinations.

Shifting Bottleneck procedure

The decomposition of the machine shop scheduling problem results in a series of $1|r_j|L_{\max}$ problems for the machines M_1, M_2, and M_3. For machine M_4, it results in the problem $1|r_j, s_i|L_{\max}$. Recall that the convergent job routings do not change the single-machine scheduling problems. We test two variants of the extended SB procedure: if possible, the first uses exact algorithms for the single-machine scheduling problems; the second uses heuristics for these problems. Note that delayed precedence constraints do not occur in the problems that we consider.

The first variant of the extended SB procedure uses Carlier's algorithm (Carlier [17]) for solving the $1|r_j|L_{\max}$ problems and the branch-and-bound algorithm developed in Chapter 4 for the $1|r_j, s_i|L_{\max}$ problem. In that chapter, we observed that if the branch-and-bound algorithm solves an instance to optimality, then the computation time is in general small. We therefore stop computation when the algorithm does not find the optimal solution within 10 seconds and use the best solution found so far. In this case, the single-machine schedule may not be optimal.

The second variant of the extended SB procedure uses heuristics for the single-machine scheduling subproblems. For the machines without setup times, we use the extended Jackson rule (Jackson [44]). For the machine with setup times, we use priority rules. In this case, the priority of operation O_{ij} is $\rho_{ij} = late - \beta \cdot NP$, with *late* the lateness of this operation if O_{ij} would be the next operation to be scheduled, and NP the *non-productive machine time*. The non-productive machine time consists of setup time and idle time, i.e., NP is the difference between the start of O_{ij} and the completion of the previous operation. Again, we vary β from 0 to 20 with steps of 0.5 and take the delayed precedence constraints into account. Whenever the machine with setup times must be scheduled within the SB procedure, we evaluate the 41 combinations. The schedule delivered by the best combination is used by the coordination mechanism of the SB procedure.

Results

Table 6.1 summarizes our computational experiments for the small test shop. Each row in this table presents the average performance over

Algorithm	time	$L_{\max}$	# late jobs	mean tardiness
pri	14.4	115.3	3.2	12.3
sb1	20.9	-39.9	3.1	8.3
sb2	1.1	17.8	3.6	10.6

Table 6.1: Computational results for small test shop.

30 instances. The first column indicates the algorithm we used: 'pri' stands for the priority rules, 'sb1' stands for the extended SB procedure with exact algorithms for the single-machine scheduling subproblems, and 'sb2' stands for the extended SB procedure that uses heuristics for the single-machine scheduling subproblems. The column with the header 'time' shows the mean computation time in seconds on an HP 9000/710 workstation. For the priority rules, this is the total time to evaluate all 82 combinations. The column '$L_{\max}$' displays the average value of the maximum lateness. The column '# late jobs' lists the average number of late jobs, i.e., the number of jobs that finish after their due dates, whereas the last column displays information on the mean tardiness, which is defined as $\sum_{j=1}^{n} T_j/n$, with $T_j = \max\{0, L_j\}$ and n the number of jobs. For the priority rules, the data in the row 'pri' of Table 6.1 show for each performance criterion the average best value of all schedules generated by the priority rules.

We see that the SB procedure with the exact algorithms for the single-machine scheduling problems (sb1) clearly outperforms the other algorithms, albeit at the expense of more computation time. The SB procedure with heuristics for the single-machine scheduling problems (sb2) performs better on the performance criteria "maximum lateness" and "mean tardiness" and slightly worse on the criterium "# late jobs" than the best priority rule. In the next section, we further compare sb2 to priority rules on larger instances. Due to the size of the instances involved, we do not use exact algorithms for the single-machine scheduling subproblems any more.

6.2.2 Large test shops

The first large test shop that we consider is similar to the small test shop. It has four machines $M_1, \ldots, M_4$. M_1 produces component A, M_2 produces component B, and M_3 produces component C. M_4 is used to assemble components. Now, all possible combinations of components A, B, and C are considered. So, jobs arrive for producing A, B, C, AB, AC, BC, and ABC. Jobs arrive at the shop according to a Poisson process with a mean interarrival time of 70 time units. Upon arrival, we determine what type of product is ordered. Each type has an equal probability of being chosen. We draw the processing times of operations on M_1, M_2, and M_3 from the discrete uniform distribution on the interval $[85, 115]$. We consider problems with and without setup times on M_4. If no setup times occur, then we draw the processing times for operations on M_4 from the discrete uniform distribution on the interval $[90, 130]$. Otherwise, the interval we draw from is $[65, 85]$. Table 6.2 lists the setup times on M_4 in the latter case. The logic

		to			
		AB	AC	BC	ABC
	–	200	200	200	300
	AB	0	120	120	100
from	AC	120	0	120	100
	BC	120	120	0	100
	ABC	20	20	20	0

Table 6.2: Setup times on M_4.

behind these setup times is the following. Whenever a component is assembled, a specific tool is needed. If a component is not assembled, this tool should not be present at the machine. Mounting a tool on the machine takes 100 time units; unmounting a tool takes 20 time units. Note that we now consider sequence *dependent* setup times on M_4. The due date of job J_j is set in the same way as in the small test shop.

The second large test shop has two machines, M_1 and M_2. M_1 produces components A and B, whereas M_2 assembles these components to end products. Again, jobs arrive at the shop following a Poisson proces, but now with a mean interarrival time of 100 time units. Each job that arrives at the shop is an order to produce a product consisting of components A and B that need to be assembled. We consider problems with and without setup times on the first machine. For problems with setup times, a setup time is needed whenever the machine switches from producing components A to producing components B and the other way around. The required setup time is 60 time units in both cases. We draw the processing times for operations on the first machine from the discrete uniform distribution on the interval $[10, 60]$. For problems without setup times, the processing times on the first machine come from the discrete uniform distribution on the interval $[10, 70]$. The processing times for operations on the second machine are drawn from the discrete uniform distribution on the interval $[60, 100]$. Again, the due dates of the jobs are set in the same way as in the small test shop.

We also want to study the effect of static and dynamic scheduling on the performance of the SB procedure. Static scheduling means that all jobs of an instance are scheduled in one go. Dynamic scheduling means that we split a set of jobs into smaller sets. In the tests, we do this in the following way. First, let T be such that on average 200 jobs arrive at the shop during the interval $[0, T]$. Then, $T = 200 \cdot 70 = 14,000$ for the first test shop, and $T = 20,000$ for the second test shop. Schedule all jobs that arrive in the interval $[0, 2 \cdot T]$. We call this the first run. All operations that start in the interval $[0, T]$ are fixed. Now, all operations that were scheduled in the first run but not fixed, and all jobs that arrive in the interval $[2 \cdot T, 3 \cdot T]$ are scheduled. This is the second run. All operations that start in the interval $[T, 2 \cdot T]$ are fixed. This process continues until all operations are fixed. For static scheduling, we generate instances with 500 jobs. For dynamic scheduling, we generate instances with $2,000$ jobs.

Results

Table 6.3 shows information about the computation time. Each row in this table gives the average over 30 instances. The first column indicates whether setup times occur (S) or not (NS). The second column indicates whether the instances are scheduled statically (stat) or dynamically (dyn). The columns with the header 'pri' show the average com-

S/	stat/	shop 1		shop 2	
NS	dyn	pri	SB	pri	SB
NS	stat	1.0	4.8	2.0	6.2
NS	dyn	4.0	29.0	5.3	37.4
S	stat	49.1	16.3	70.0	21.0
S	dyn	173.3	102.9	410.2	156.7

Table 6.3: Average computation time in seconds.

putation time in seconds to evaluate all combinations on a HP 9000/710 workstation. For instances without setup times, we evaluate two combinations; for instances with setup times, we evaluate 82 combinations. The columns 'SB' show the average computation time of the SB procedure. Recall that the version of the SB procedure used here is equal to sb2 of Section 6.2.1. For instances that are scheduled dynamically, this time is the aggregate time of all the runs. The computation times of the SB procedure are acceptable. Even for problems with $2,000$ jobs, the mean computation time does not exceed three minutes. Computing 82 schedules with priority rules takes less than three minutes for the first shop and less than seven minutes for the second shop. Note that the problems for the second shop contain on average more operations than those for the first shop.

Table 6.4 reports on some performance measures of the different schedules for the first shop. Each row lists averages over 30 instances. The first two columns indicate again whether setup times occur and

S/	stat/	$L_{\max}$		# late jobs		mean tardiness	
NS	dyn	pri	SB	pri	SB	pri	SB
NS	stat	361.0	277.0	32.0	35.2	18.4	17.3
NS	dyn	1159.3	986.1	224.2	254.7	55.8	54.0
S	stat	604.1	354.8	46.4	51.0	25.1	20.6
S	dyn	1039.9	713.0	241.0	263.0	39.5	33.2

Table 6.4: Performance measures for the first shop.

whether the SB procedure solves the instances statically or dynamically. The third and the fourth column give information about the maximum lateness. The third column displays the average best value of the maximum lateness of all schedules generated by the priority rules. The fourth column gives the average value of the maximum lateness of the schedule generated with the SB procedure. We see that the SB procedure significantly outperforms the best priority rule. The columns with the common header '# late jobs' show information about the second performance measure we consider: the number of late jobs, i.e., the number of jobs that complete after their due dates. On average, the best priority rule gives better results than the SB procedure. The last criterion we consider is mean tardiness for which the columns with the common header 'mean tardiness' give information. For this criterion, the SB procedure is clearly better again. We could have anticipated that the SB procedure performs well for the criterion "maximum lateness" and worse for the criterion "the number of late jobs", because the SB procedure tries to minimize the maximum lateness and therefore prefers solutions with many jobs a little late to solutions with few jobs much late. We also see in Table 6.4 that the performance of the SB procedure remains satisfactory if we use it dynamically. This is useful in practice, because those problems might be too large to schedule statically.

Table 6.5 displays the same information as Table 6.4, but now for problems for the second shop. Note that for the problems without setup

S/	stat/	$L_{\max}$		# late jobs		mean tardiness	
NS	dyn	pri	SB	pri	SB	pri	SB
NS	stat	-10.5	-17.9	7.3	7.1	2.3	2.2
NS	dyn	281.5	281.7	25.3	24.8	2.7	2.7
S	stat	243.5	204.5	25.5	22.6	9.1	7.5
S	dyn	552.6	499.8	123.1	100.2	12.1	9.0

Table 6.5: Performance measures for the second shop.

times the performance of the SB procedure not really differs from the best priority rule. For problems with setup times, the SB procedure outperforms the best priority rule for all displayed performance measures.

Table 6.4 and 6.5 compare the SB procedure for different performance measures with the best priority rule for each measure. The best priority rule for, e.g., the maximum lateness may be another rule than the one that gives the best solution for the mean tardiness criterion. In practice, however, one is interested in a single solution with a good maximum lateness, few jobs too late, and a small mean tardiness. Table 6.6 compares the solution values of the SB procedure with the solution values of the priority rule that results in the best maximum lateness for the first shop, on which the remainder of this section focuses. Columns

S/ NS	stat dyn	# late jobs			mean tardiness		
		pri		SB	pri		SB
		best	ml		best	ml	
NS	stat	32.0	32.2	35.2	18.4	18.4	17.3
NS	dyn	224.2	225.4	254.7	55.8	56.1	54.0
S	stat	46.4	59.5	51.0	25.1	31.5	20.6
S	dyn	241.0	280.7	263.0	39.5	47.3	33.2

Table 6.6: Comparison of solution values.

three and six give again the solution value of the best priority rule. The columns that have the label 'ml' give the values of the performance measures of the schedule generated with the priority rule that results in the best maximum lateness. The best priority rule gives better results for the number of late jobs than the SB procedure. For the problems without setup times, the values in the column 'ml' deteriorate, but they are still better than the value of the SB schedule. For the problems with setup times, however, the values in the column 'ml' are worse than the values of the SB schedule. The mean tardiness of the best priority rule is worse than the value of the schedule generated by the SB procedure. Of course, the values deteriorate in the column 'ml'. We conclude that compared with priority rules, the SB procedure produces a schedule of good maximum lateness and mean tardiness. The number of late jobs in this schedule is acceptable.

In the tests above, a job is always an order to produce *one* component or product. An interesting effect to study is the impact of batch arrivals, that is, a job is an order to produce a number of the same components or products. In the remaining tests, we determine a batch size of the jobs that arrive at the shop. We draw the batch size of a job from a discrete uniform distribution on the interval $[1, 5]$. The mean batch size is therefore three. In Table 6.7, we present results for the case that the arrival time of the next job depends on the batch size of this job. More specifically, assume that q is the batch size of this job. Then, the interarrival time for the next job is drawn from a negative exponential distribution with a mean of $100 \cdot q$ time units. With this arrival process, we try to simulate the situation that the sales department of a company tries to relieve the production department after getting a large order. Note that we increased the mean processing time by a factor three, whereas we increased the mean interarrival time by a factor $\frac{100}{70} \cdot 3 > 4$. We see from Table 6.7 that the due date performance of the shop deteriorates, although we decreased the workload. A similar behavior is found in queueing theory. Whitt [82], for example, shows for the $GI|G|m$ queue that the mean waiting time of jobs in the queue increases with increasing variation in both the arrival and service processes. For the criteria "maximum lateness" and "the number of late jobs", the SB procedure performs comparable to the previous tests. Due to the large maximum lateness, the performance of the SB procedure is worse for the criterion "mean tardiness".

S/	stat/	$L_{\max}$		# late jobs		mean tardiness	
NS	dyn	pri	SB	pri	SB	pri	SB
NS	stat	1005.5	906.8	57.6	66.5	31.6	35.3
NS	dyn	1767.7	1644.3	244.6	274.7	39.0	42.7
S	stat	872.6	796.1	49.1	58.0	23.6	27.5
S	dyn	1368.8	1268.8	209.5	240.5	26.4	30.3

Table 6.7: Results for batch arrivals.

Due to the large variation in the arrival process, the due date performance of the shop in the previous test was poor. Now, we study the due date performance of the shop with a more regular arrival process. We use an Erlang-4 distribution for the interarrival times, which is the summation of four drawings from a negative exponential distribution. The variance of an Erlang-4 distribution with mean λ^{-1} is half the variation of a negative exponential distribution with mean λ^{-1}. The mean interarrival time is again 300 time units. Table 6.8 summarizes our test results for this situation. We see that indeed the due date performance of the shop improves significantly with a more regular arrival process. For a company it might therefore be advantageous to try to regulate the arrival process. The SB procedure still works comparable to the situation with no batch arrivals. The differences, however, are smaller,

S/	stat/	$L_{\max}$		# late jobs		mean tardiness	
NS	dyn	pri	SB	pri	SB	pri	SB
NS	stat	422.4	404.6	25.1	25.9	6.7	6.7
NS	dyn	728.7	689.7	102.2	107.7	7.2	7.3
S	stat	364.8	348.4	21.9	23.0	5.4	5.4
S	dyn	562.3	521.1	88.5	92.3	5.1	5.2

Table 6.8: Test results with Erlang-4 arrival process.

which is explained by the smaller workload of the shop.

6.3 Conclusions

We compared the performance of two versions of extended SB procedure and the most common priority rules with each other. Setup times occurred in the shops we considered, and the jobs had convergent routings. The SB procedure with algorithms that solve the single-machine scheduling problems to optimality if possible, outperforms both the SB procedure with heuristics for the single-machine scheduling problems and the priority rules. The SB procedure with heuristics outperformed the priority rules. For part manufacturing shops, due date performance is important. Therefore, it might be a good choice to replace the scheduling of a shop with priority rules by a more sophisticated approach like the SB procedure. Also, attention should be paid to the regulation of the arrival process.

In the next chapter, we discuss JOBPLANNER, a commercial shop floor scheduling system that is based on the extended SB procedure. We also discuss practical experiences with this scheduling system in a company producing printed circuit boards.

Chapter 7

A practical job shop scheduler

7.1 Introduction

In Chapter 3, we discussed various extensions to the SB procedure to deal with practical job shops. This extended SB procedure is part of the commercial shop floor control system JobPlanner. Heerma and Lok [41] use Figure 7.1 to position JobPlanner in its environment. The *production planning system*, which is often an MRP II system, informs JobPlanner *which* jobs need to be scheduled. The *engineering database* contains information on *how* a product is produced. The *shop floor data collection system* monitors the shop status, e.g., whether a machine is down, and gathers short term scheduling information. For frameworks for shop floor control, we refer to Tiemersma [76] and Arentsen [5].

In the next section, we briefly discuss the components of JobPlanner. Section 7.3 reports on some experiences with JobPlanner in a company producing printed circuit boards. Section 7.4 ends this chapter with some conclusions.

7.2 JobPlanner

As we see in Figure 7.1, JobPlanner consists of five main components: the database manager, the automatic scheduler, the graphical user interface, the schedule editor, and the evaluation component.

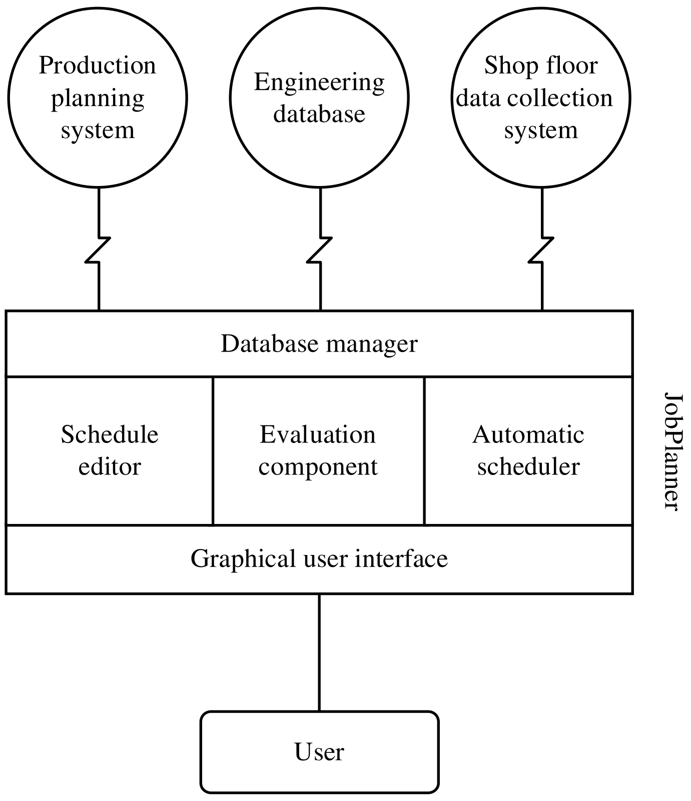


Figure 7.1: JobPlanner and its environment.

We briefly discuss these components in the following subsections.

7.2.1 Database manager

The database manager communicates with the production planning system, the engineering database, and the shop floor data collection system. It extracts data from these databases and stores it in a *local shop floor database*. In addition to the information on the jobs that must be produced and how this should be done, it contains data about the shop status. For example, when an operation is completed or a machine breaks down, this information is stored in the local shop floor database. This makes this database *dynamic*, i.e., it changes over time.

7.2.2 Automatic scheduler

The automatic scheduler is based on the SB procedure of Adams et al. [2] extended to deal with practical features, as discussed in Chapter 3. Through the database manager, it gets information like job routings and processing times from the local shop floor database. The scheduler proposes a schedule, which is stored in the database.

7.2.3 Graphical user interface

The graphical user interface is an important means for man-machine interaction. Figure 7.2 shows an example of a graphical user interface. In this figure, we see buttons on which the user can click with his

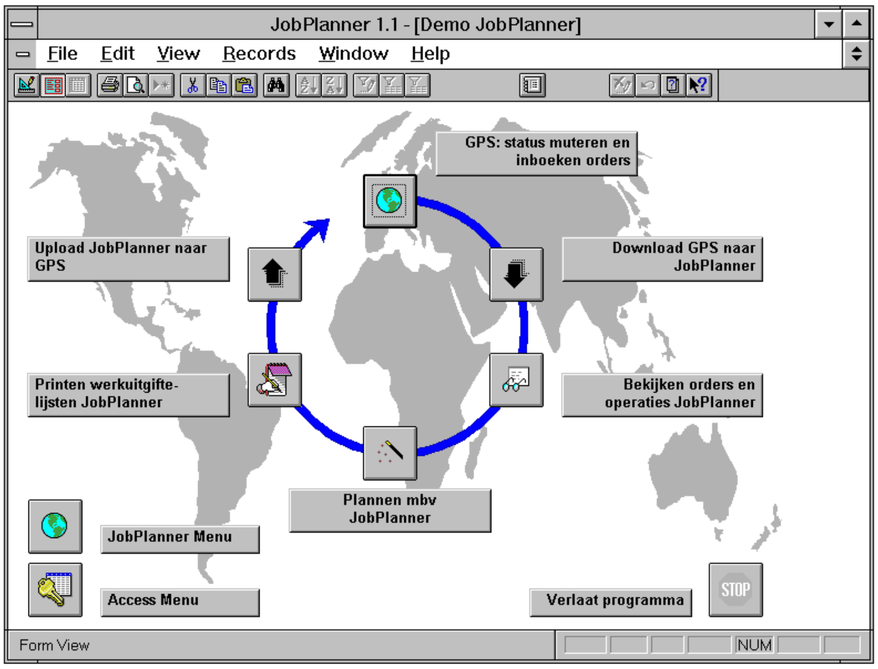

JobPlanner 1.1 - [Demo JobPlanner]
File
Edit
View
Records
Window
Help
GPS: status muteren en inboeken orders
Upload JobPlanner naar GPS
Download GPS naar JobPlanner
Printen werkuitgifte-lijsten JobPlanner
Bekijken orders en operaties JobPlanner
Plannen mbv JobPlanner
JobPlanner Menu
Access Menu
Verlaat programma
STOP
Form View
NUM

Figure 7.2: Example of a graphical user interface.

mouse device to perform actions like starting the automatic scheduler, downloading information from a database, printing of information, and so on. Also, the user can start actions by means of a menu.

An element in the graphical user interface for scheduling systems is the electronic planning board. The electronic planning board uses Gantt charts to represent schedules and provides the means to modify schedules by manipulating the Gantt chart; cf. Wennink [80]. Figure 7.3 depicts an electronic planning board that is divided in two parts. The

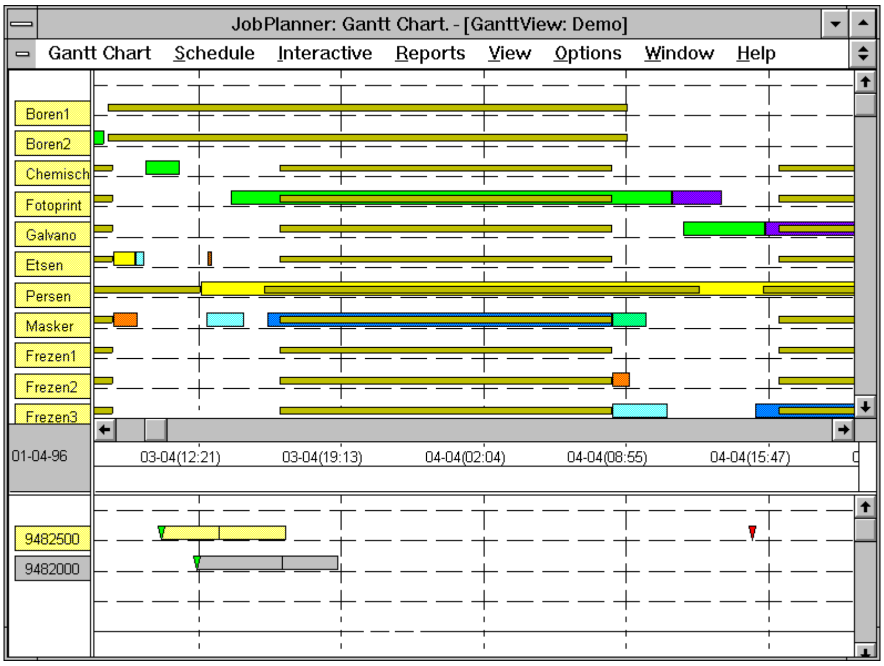
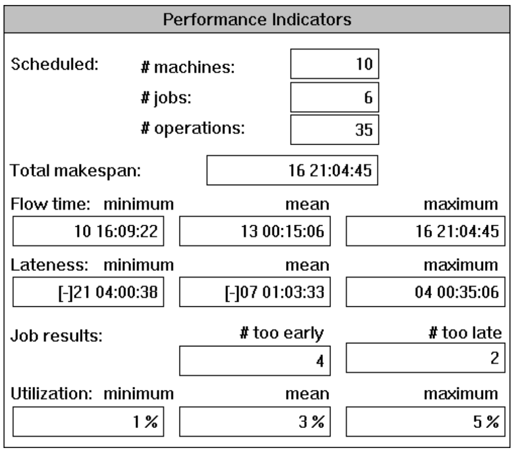


Figure 7.3: An electronic planning board.

upper part of this planning board is a Gantt chart in which the resources are shown along the vertical axis. The time is shown along the horizontal axis. The narrow rectangles represent the down times of the resources; the other rectangles represent scheduled operations. The lower part of the planning board shows information about unscheduled jobs. For a more elaborate discussion about the importance of planning boards in complex scheduling situations, we refer to Anthonisse et al. [3] and for the importance of representation and visualization in the context of optimization, we refer to Jones [46].

7.2.4 Evaluation component

Besides a graphical representation, a scheduler needs data to evaluate the quality of a schedule. Useful is, e.g., data on the delivery performance, machine utilization, makespan, and throughput times. Data can be represented in the form of tables, graphs, and throughput diagrams (see Wiendahl [83]), and so on. Figure 7.4 shows a report available in JobPlanner. For more information about envisioning information,



Performance Indicators

Scheduled:	# machines:	10	
	# jobs:	6	
	# operations:	35	
Total makespan:		16 21:04:45	
Flow time:	minimum	mean	maximum
	10 16:09:22	13 00:15:06	16 21:04:45
Lateness:	minimum	mean	maximum
	[-]21 04:00:38	[-]07 01:03:33	04 00:35:06
Job results:		# too early	# too late
		4	2
Utilization:	minimum	mean	maximum
	1 %	3 %	5 %

Figure 7.4: A report of JobPlanner.

we refer to Tufte [77, 78].

7.2.5 Schedule editor

In a scheduling system, the planner should have the ability to create, modify, store, and retrieve schedules. The scheduler oversees, for example, practical problems that have not been modeled. The *interaction* between the planner and the scheduling system is crucial to end up with a feasible schedule. In this context, interaction means the integration of human perception and mechanical algorithms; cf. Savelsbergh [70].

Bruggink [15] distinguishes six groups of schedule editing functions that a scheduling system should have: generate, edit, analyze, optimize, shop status, and extra functions. *Generate* functions enable the planner to generate schedules manually or automatically. *Edit* functions are functions to modify an existing schedule, such as exchanging and adding operations to a schedule. The *analyze* functions give information on the properties of the current schedule, such as delivery performance, makespan, operations on a longest path, and so on. *Optimize* functions help the planner to optimize a selected part of a schedule under the constraint that the unselected part is not affected. Also, they recover infeasibilities in a given schedule. The *shop status* functions provide information about the shop floor, such as completed operations and machine breakdowns. The *extra* functions contain functions that are not categorized in the first five groups. This group of extra functions includes the possibility to set precedence constraints between operations of different jobs.

7.3 Practical experiences

In the last few years, JobPlanner has been tested in various companies in the discrete manufacturing. In this section, which is partly based on work by Heerma [40], we report on experiences with JobPlanner at Cityprint B.V., which is located in Almelo, The Netherlands. Cityprint is a large Dutch producer of printed circuit boards (PCBs). Currently, there are about 140 employees, of which about 100 are production personnel or production engineers. The machine shop consists of about 40 machines. The annual turnover is more than 25 million guilders, of which more than 50% is due to export to countries in Europe. Production, and even engineering, is to customer order. The leadtimes are at most two to four weeks. Cityprint uses two base materials for producing PCBs: teflon and epoxy. Teflon is used, for example, for PCBs in satellite dishes and epoxy for PCBs in computers and television sets. The PCBs have one up to eight track layers.

The production system of Cityprint can be divided in six production groups: production engineering, cutting, through hole plating, imaging, pressing, and testing. For each customer order, *production engineering* makes an internal order. For each order, it determines the necessary production steps and the order and the machines on which they have to be performed. The *cutting* group cuts teflon and epoxy plates covered with copper foil to the required sizes and drills holes in them. The *through hole plating* group takes care that the faces of the drilled holes get plated with copper. Whenever there is a switch from plating teflon

PCBs to plating epoxy PCBs and vice versa, a setup time is needed. The *imaging* group makes the copper tracks on the PCBs. It uses a photo-sensitive material and a film of the track. At places that have not been exposed to light, the copper is removed through etching. The *pressing* group presses a number of plates to one multi-layered PCB. Finally, the *testing* group tests the produced PCBs for example on short-circuits. A typical job consists of 20 operations.

The introduction of JobPlanner at Cityprint has been satisfactory. First of all, the graphical representation of schedules has increased the insight of the planner into the actual workload of the production system, which has made it relatively easy to trace bottlenecks in the production system and respond appropriately. The sales department can also use the insight in the actual workload of the production system by quoting feasible and competitive delivery dates. The introduction of JobPlanner has reduced the time for production planning. It used to be a full-time job; now, it takes only one or two hours a day. Hence, there is more time to think about structural improvements of the production process. In the first year that JobPlanner was used, 95% of the jobs were delivered in time, the output increased by 15%, and the throughput times decreased by 25%.

7.4 Conclusions

We have discussed a commercial shop floor scheduling system. This system uses the extended Shifting Bottleneck procedure for scheduling jobs in machine shops. Experiences with the system have been satisfactory. We discussed practical experiences at Cityprint, where the time used for production planning is reduced significantly, along with an increase of the insight into the actual status of the shop increased. The delivery performance has been high, the output increased, and the throughput times decreased significantly after the introduction of the scheduling system in this company.

In the next chapter, some general conclusions and suggestions for further research will conclude this thesis.

Chapter 8

Epilogue

In this chapter, we summarize our results. We stress again the importance of an *integral* planning and scheduling approach, especially for machine shops in which setup times occur. We end this chapter with some conclusions.

8.1 Summary

We have motivated the development of an integral shop floor planning and scheduling system by pointing out the need for short leadtimes and reliable due dates. Based on the Shifting Bottleneck (SB) procedure of Adams et al. [2] for the *classic* job shop problem, we have designed and analyzed the algorithmic framework for a shop floor scheduling system for *practical* job shops. Our algorithms can handle job shops with release and due dates of jobs, setup times, parallel machines, transportation times, unequal transfer and production batches, multiple resource requirements, preemptive and non-preemptive down times, convergent and divergent job routings, and open shop characteristics.

For machine shops in which setup times occur, a scheduling system should find a trade-off between efficiency and leadtime performance; this is a common trade-off in practice. On the one hand, clustering jobs with the same setup characteristics on a machine leads to an efficient use of this machine. On the other hand, clustering jobs may lead to a poor leadtime performance of jobs with other setup characteristics. Shop efficiency may even be low, since other machines may not receive the right jobs in time. We therefore feel that an analytic approach to these problems in a complex job shop environment represents a fundamental contribution. For reasons explained earlier, the SB procedure appeared to be a natural candidate to be extended for scheduling machine shops with setup times.

The SB procedure decomposes the problem of scheduling a machine shop into a series of machine scheduling problems. The presence of setup times on one or more machines results for those machines in machine scheduling problems of minimizing maximum lateness in which we have to take the setup times into account. For the special case of *family*, or sequence independent, setup times, we have developed a branch-and-bound algorithm. Our computational experiments show that this algorithm solves most instances with up to 40 jobs to optimality within one minute. A major algorithmic novelty is the concept of *setup jobs* to compute strong lower bounds. We have developed sufficient conditions to separate jobs within one family by a setup time. If these conditions hold, then a setup job is added to the job set with a release and due date, and processing time. There exist precedence relations between this setup job and the real jobs. On average, the introduction of setup jobs halves the gap between the lower bound in the root of the search tree and the optimal solution value. We also studied the parallel machine equivalent of the single-machine scheduling problem with setup times. For a broad class of parallel-machine scheduling problems, we have characterized a set of at most $n!$ solutions that contains at least one optimal solution. The parallel-machine scheduling problem that we studied belongs to this class. For this problem, we are able to solve most instances with up to 20 jobs to optimality in reasonable time.

Although the modeling of convergent job routings is quite easy, it is important from a practical point of view. We empirically tested the SB procedure in shops with setup times and convergent job routings. In terms of delivery performance, the SB procedure significantly outperformed priority rules that reportedly perform well for assembly shops.

The commercial shop floor scheduling and planning system Jobplanner uses the SB procedure to schedule practical job shops. We discussed the experiences with this system at a manufacturer of printed circuit boards. The introduction of Jobplanner in this company was successful: the delivery performance increased and the throughput times decreased, while more PCBs were made.

8.2 Conclusions

We have developed the algorithmic framework of an integral shop floor planning and scheduling system. The system is able to handle a broad range of practical problems. Our computational results for randomly generated instances show that our algorithms significantly outperform simple priority rules. In practice, JOBPLANNER, with our algorithms embedded, has led to a significantly better delivery performance as well.

Further research on shop floor scheduling is a practical necessity, nonetheless. Here, we mention four areas.

First, further research needs be done to model other machine types, such as *batch processors* that can process more than one job at a time.

Second, better algorithms can be designed for scheduling machines with sequence dependent setup times. In Chapter 6, we studied the empirical performance of the SB procedure in machine shops with setup times. We observed that the quality of the algorithms for the machine scheduling subproblems has a significant impact on the performance of the SB procedure.

Third, further research needs to be done to incorporate alternative job routings in the SB procedure. In Section 3.10, we have shown that alternative processing orders of operations can be modeled easily. Further research needs to be done to incorporate alternative job routings that follow from different machines being able to perform an operation.

Finally, there is need for a better capacity planning function than the simple "bucket filling" algorithm used by MRP II systems, which ignores job routings. Alternatives for this algorithm might be based on a combination of linear programming and machine scheduling algorithms. The extended SB procedure that can handle convergent job routings is a candidate for such a scheduling algorithm.

Bibliography

[1] E.H.L. Aarts, P.J.M. van Laarhoven, J.K. Lenstra, and N.L.J. Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, 6:118–125, 1994.

[2] J. Adams, E. Balas, and D. Zawack. The Shifting Bottleneck procedure for job shop scheduling. *Management Science*, 34:391–401, 1988.

[3] J.M. Anthonisse, J.K. Lenstra, and M.W.P. Savelsbergh. Behind the screen: DSS from an OR point of view. *Decision Support Systems*, 4:413–419, 1988.

[4] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal on Computing*, 3:149–156, 1991.

[5] A.L. Arentsen. *A Generic Architecture for Factory Activity Control.* PhD thesis, University of Twente, Enschede, The Netherlands, 1995.

[6] K.R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley, New York, 1974.

[7] K.R. Baker. *Elements of Sequencing and Scheduling.* School of Business Administration, Dartmouth College, Hanover, 1992.

[8] E. Balas, J.K. Lenstra, and A. Vazacopoulos. The one-machine problem with delayed precedence constraints and its use in job shop scheduling. *Management Science*, 41:94–109, 1995.

[9] E. Balas and A. Vazacopoulos. Guided local search with shifting bottleneck for job shop scheduling. Management Science Research Report #MSRR-609, Carnegie-Mellon University, Graduate School of Industrial Administration, Pittsburg, Pennsylvania, 1994.

[10] G.A. Belderok. Ontwerp van een produktiebesturings- en informatiesysteem voor de produktie van zware persdelen in de plaat komponenten fabriek van DAF Trucks Eindhoven (in Dutch). Master's thesis, University of Twente, Faculty of Mechanical Engineering, Production and Operations Management Group, Enschede, The Netherlands, 1993.

[11] J.D. Blackburn. *Time-Based Competition, The Next Battle Ground in American Manufacturing.* Richard D. Irwin, Homewood, Ill., 1991.

[12] P. Brucker. *Scheduling Algorithms.* Springer-Verlag, Berlin, 1995.

[13] P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49:107–127, 1994.

[14] P. Brucker and O. Thiele. A branch & bound method for the general-shop problem with sequence dependent setup-times. To appear in: *OR Spektrum.*

[15] D.P. Bruggink. Interactieve scheduling (in Dutch). Master's thesis, University of Twente, Faculty of Mechanical Engineering, Production and Operations Management Group, Enschede, The Netherlands, 1994.

[16] J. Bruno and P. Downey. Complexity of task sequencing with deadlines, set-up times and changeover costs. *SIAM Journal on Computing*, 7:393–404, 1978.

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

[18] J. Carlier. Scheduling jobs with release dates and tails on identical machines to minimize the makespan. *European Journal of Operational Research*, 29:298–306, 1987.

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

[20] Y. Cho and S. Sahni. Bounds for list schedules on uniform processors. *SIAM Journal on Computing*, 9:91–103, 1980.

[21] E.G. Coffman, Jr., A. Nozari, and M. Yannakakis. Optimal scheduling of products with two subassemblies on a single machine. *Operations Research*, 37:326–336, 1989.

[22] S.A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971.

[23] S. Dauzère-Peres and J.-B. Lasserre. A modified shifting bottleneck procedure for job-shop scheduling. *International Journal of Production Research*, 31:923–932, 1993.

[24] M. Dell'Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research*, 41:231–252, 1993.

[25] S.E. Elmaghraby and S.H. Park. Scheduling jobs on a number of identical machines. *AIIE Transactions*, 6:1–13, 1974.

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

[27] 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 225–241. Prentice-Hall, Englewood Cliffs, NJ, 1963.

[28] S. French. *Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop.* John Wiley & Sons, New York, 1982.

[29] T.D. Fry, M.D. Oliff, E.D. Minor, and G.K. Leong. The effects of product structure and sequencing rule on assembly shop performance. *International Journal of Production Research*, 27:671–686, 1989.

[30] H.L. Gantt. *Organizing for Work.* Harcourt, Brace and Howe, New York, 1919.

[31] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, 1979.

[32] F. Glover. Tabu search - Part I. *ORSA Journal on Computing*, 1:190–206, 1989.

[33] F. Glover. Tabu search - Part II. *ORSA Journal on Computing*, 2:4–32, 1990.

[34] F. Glover, E. Taillard, and D. De Werra. A user's guide to tabu search. *Annals of Operations Research*, 41:3–28, 1993.

[35] J.S. Goodwin and J.K. Weeks. Evaluating scheduling policies in a multi-level assembly system. *International Journal of Production Research*, 24:247–257, 1986.

[36] J. Grabowski, E. Nowicki, and S. Zdrzalka. A block approach for single-machine scheduling with release dates and due dates. *European Journal of Operational Research*, 26:278–285, 1986.

[37] R.L. Graham. Bounds on multiprocessor timing anomalies. *SIAM Journal of Applied Mathematics*, 17:416–429, 1969.

[38] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[39] R. Haupt. A survey of priority rule-based scheduling. *OR Spektrum*, 11:3–16, 1989.

[40] W. Heerma. Jobplanner, Gereedschap voor betere logistieke prestaties (in Dutch). To appear in: *Via's, Magazine for the Electrotechnical Industry.*

[41] W. Heerma and N.R. Lok. Scheduling-systeem met IQ ontlast planner (in Dutch). *Logistiek Signaal*, 4:27–29, 1994.

[42] W.A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[43] Ph. Ivens and M. Lambrecht. Extending the shifting bottleneck procedure to real-life scheduling applications. In *Fourth International Workshop on Project Management and Scheduling*, pages 210–213, Leuven, Belgium, July 12-15 1994.

[44] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. Research report #43, Management Science Research Project, University of California, Los Angeles, 1955.

[45] S.M. Johnson. Optimal two and three-stage production schedule with setup times included. *Naval Research Logistics Quarterly*, 1:61–68, 1954.

[46] C.V. Jones. Visualization and optimization. *ORSA Journal on Computing*, 6:221–257, 1994.

[47] R.M. Karp. Reducibility among combinatorial problems. In R.E. Miller and J.W. Thatcher, editors, *Complexity of Computer Computations*, pages 85–103. Plenum Press, New York, 1972.

[48] P.J.M. van Laarhoven and E.H.L. Aarts. *Simulated Annealing: Theory and Applications.* Reidel, Dordrecht, The Netherlands, 1987.

[49] P.J.M. van Laarhoven, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40:113–125, 1992.

[50] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. In *Progress in Combinatorial Optimization*, pages 245–261. Academic Press, Canada, 1984.

[51] E.L. Lawler, J.K. Lenstra, and A.H.G. Rinnooy Kan. Recent developments in deterministic sequencing and scheduling: A survey. In M.A.H. Dempster, J.K. Lenstra, and A.H.G. Rinnooy Kan, editors, *Deterministic and Stochastic Scheduling*, pages 35–73. NATO Advanced Study and Research Institute, D. Reidel Publishing Company, Dordrecht, The Netherlands, 1982.

[52] E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, and D.B. Shmoys. Sequencing and scheduling: Algorithms and complexity. In S.C. Graves, A.H.G. Rinnooy Kan, and P. Zipkin, editors, *Hankbooks in Operations Research and Management Science, Volume 4: Logistics of Production and Inventory.* North-Holland, 1993.

[53] C.-Y. Lee, T.C.E. Cheng, and B.M.T. Lin. Minimizing the makespan in the 3-machine assembly-type flowshop scheduling problem. *Management Science*, 39:616–625, 1993.

[54] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[55] W.L. Maxwell and M. Mehra. Multiple-factor rules for sequencing with assembly constraints. *Naval Research Logistics Quarterly*, 15:241–254, 1968.

[56] G.J. Meester. *Multi-Resource Shop Floor Scheduling.* PhD thesis, University of Twente, Enschede, The Netherlands, 1996.

[57] G.J. Meester and W.H.M. Zijm. Multi-resource scheduling for an FMC in discrete parts manufacturing. In M.M. Ahmad and W.G. Sullivan, editors, *Flexible Automation and Integrated Manufacturing*, pages 360–370. CRC Press Inc., Atlanta, 1993.

[58] L.W. Miller, A.S. Ginsberg, and W.L. Maxwell. An experimental investigation of priority dispatching in a simple assembly shop. In M.A. Geisler, editor, *Logistics.* North-Holland, Amsterdam, 1975.

[59] T.E. Morton and D.W. Pentico. *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management.* John Wiley & Sons, New York, 1993.

[60] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop problem. Preprint 8/93, Instytut Cybernetyki Technicznej, Politnechniki Wroclawskiej, Wroclaw, 1993.

[61] I.M. Ovacik and R. Uzsoy. A shifting bottleneck algorithm for scheduling semiconductor testing operations. *Journal of Electronic Manufacturing*, 2:119–134, 1992.

[62] I.M. Ovacik and R. Uzsoy. Worst-case error bounds for parallel machine scheduling problems with bounded sequence-dependent setup times. *Operations Research Letters*, 14:251–256, 1993.

[63] M. Pinedo. *Scheduling: Theory, Algorithms, and Systems.* Prentice Hall, Englewood Cliffs, 1995.

[64] C.N. Potts, S.V. Sevast'janov, V.A. Strusevich, L.N. van Wassenhove, and C.M. Zwaneveld. The two-stage assembly scheduling problem: Complexity and approximation. *Operations Research*, 43:346–355, 1995.

[65] C.N. Potts and L.N. van Wassenhove. Integrating scheduling with batching and lot-sizing: a review of algorithms and complexity. *Journal of the Operational Research Society*, 43:395–406, 1992.

[66] F.F.J. Reesink. Werkplaatsbesturing met behulp van de Shifting Bottleneck Methode (in Dutch). Master's thesis, University of Twente, Faculty of Mechanical Engineering, Production and Operations Management Group, Enschede, The Netherlands, 1993.

[67] S. Reiter. A system for managing job shop production. *Journal of Business*, 34:371–393, 1966.

[68] B. Roy and B. Sussman. Les problèmes d'ordonnancement avec constraintes disjonctives. Note DS No. 9 bis, SEMA, Paris, 1964.

[69] R.S. Russell and B.W. Taylor III. An evaluation of sequencing rules for an assembly shop. *Decision Sciences*, 16:196–212, 1985.

[70] M.W.P. Savelsbergh. *Computer Aided Routing.* PhD thesis, Erasmus University, Rotterdam, 1988.

[71] J.M.J. Schutten. List scheduling revisited. To appear in: *Operations Research Letters.*

[72] J.M.J. Schutten. Assembly shop scheduling. Technical Report LPOM-95-15, University of Twente, Faculty of Mechanical Engineering, Production and Operations Management Group, Enschede, The Netherlands, 1995.

[73] J.M.J. Schutten. Practical job shop scheduling. Technical Report LPOM-95-12, University of Twente, Faculty of Mechanical Engineering, Production and Operations Management Group, Enschede, The Netherlands, 1995.

[74] J.M.J. Schutten and R.A.M. Leussink. Parallel machine scheduling with release dates, due dates and family setup times. To appear in: *International Journal of Production Economics.*

[75] J.M.J. Schutten, S.L. van de Velde, and W.H.M. Zijm. Single-machine scheduling with release dates, due dates and family setup times. To appear in: *Management Science.*

[76] J.J. Tiemersma. *Shop Floor Control in Small Batch Part Manufacturing.* PhD thesis, University of Twente, Enschede, The Netherlands, 1992.

[77] E.R. Tufte. *The Visual Display of Quantitive Information.* Graphics Press, Cheshire, 1983.

[78] E.R. Tufte. *Envisioning Information.* Graphics Press, Cheshire, 1990.

[79] R.J.M. Vaessens, E.H.L. Aarts, and J.K. Lenstra. Job shop scheduling by local search. To appear in: *Mathematical Programming.*

[80] M. Wennink. *Algorithmic Support for Automated Planning Boards.* PhD thesis, Eindhoven University of Technology, Eindhoven, The Netherlands, 1995.

[81] S.J. Westra. Het één-machine scheduling probleem met gefixeerde jobs (in Dutch). Master's thesis, University of Twente, Faculty of Mechanical Engineering, Production and Operations Management Group, Enschede, The Netherlands, 1994.

[82] W. Whitt. Approximations for the $GI|G|m$ queue. *Production and Operations Management*, 2:114–161, 1993.

[83] H. Wiendahl. *Belastungsorientierte Fertigungssteuerung.* Carl Hanser, Munich, 1987.

[84] O.W. Wight. *Manufacturing Resource Planning: MRP II.* Essex Junction, Vt.: Oliver Wight, Limited Publications, 1984.

[85] A.P. Woerlee. *Decision Support Systems for Production Scheduling.* PhD thesis, Erasmus University, Rotterdam, The Netherlands, 1991.

Appendix A

Single-machine scheduling with preemptive down times

In this appendix, we show that the algorithm of Carlier [17] for the problem $1|r_j|L_{\max}$ can easily be generalized to deal with *preemptive* down times. Let σ be the schedule produced by the extended Jackson rule. Reindex the jobs in order of increasing completion times in σ. Let J_k be the last job in σ for which $L_k(\sigma) = L_{\max}(\sigma)$ and let J_j be the first job in σ for which we have that $C_k(\sigma) = r_j + \sum_{i=j}^{k} p_i + DT(r_j, C_k(\sigma))$, where $DT(a, b)$ denotes the total down time in the interval $[a, b]$. Let J_c be the last job in σ belonging to the set $\{J_j, \ldots, J_k\}$ for which $d_c > d_k$ and define $\mathcal{C} = \{J_{c+1}, \ldots, J_k\}$.

Theorem A.1 *In any optimal schedule, J_c is scheduled either before all jobs in $\mathcal{C}$, or after all jobs in $\mathcal{C}$.*

Proof. Due to the way σ is constructed, we have that at start time $S_c(\sigma)$ of J_c in σ no job belonging to $\mathcal{C}$ is available for processing, i.e., $r_i > S_c(\sigma)$ for all $J_i \in \mathcal{C}$. In the interval $[S_c(\sigma), \min_{J_i \in \mathcal{C}} r_i]$, the machine processes J_c in σ during $\min_{J_i \in \mathcal{C}} r_i - S_c(\sigma) - D(S_c(\sigma), \min_{J_i \in \mathcal{C}} r_i) > 0$ time units. Let σ^* be any optimal schedule with J_c scheduled somewhere in-between the jobs in $\mathcal{C}$. Let $J_q \in \mathcal{C}$ be the job that is processed later in σ^* than any other job in $\mathcal{C}$. Since $C_q(\sigma^*) \geq C_k(\sigma) + \min_{J_i \in \mathcal{C}} r_i -$

$S_c(\sigma) - D(S_c(\sigma), \min_{J_i \in \mathcal{C}} r_i) > C_k(\sigma)$, and $d_q \leq d_k$, we have that

$$\begin{aligned} L_{\max}(\sigma^*) &\geq L_q(\sigma^*) \\ &= C_q(\sigma^*) - d_q \\ &> C_k(\sigma) - d_k \\ &= L_k(\sigma) \\ &= L_{\max}(\sigma), \end{aligned}$$

which is a contradiction. □

Index

Samenvatting

Bij machine schedulingproblemen moeten bewerkingen aan produkten verroosterd worden op machines met het doel een bepaalde prestatie-indicator te minimaliseren, rekening houdend met een aantal randvoorwaarden. Een machine kan bijvoorbeeld vaak maar één bewerking tegelijk uitvoeren en de bewerkingen aan een produkt moeten in een bepaalde volgorde uitgevoerd worden.

In dit proefschrift wordt algoritmiek ontwikkeld voor schedulingproblemen die opgelost moeten worden voor de logistieke besturing van werkplaatsen, met name in de kleinseriefabricage. Deze werkplaatsen worden geconfronteerd met de vraag naar korte doorlooptijden en een hoge leverbetrouwbaarheid. We tonen aan dat *integrale* werkplaatsbesturingssystemen kortere doorlooptijden en een hogere leverbetrouwbaarheid mogelijk maken. De schedulingproblemen die ontstaan zijn in feite jobshop schedulingproblemen met additionele randvoorwaarden. Na een algemene inleiding op de schedulingtheorie, bepreken we het jobshop schedulingprobleem en algoritmen om dit probleem op te lossen.

Een van de algoritmen die we bespreken is de Shifting Bottleneck (SB) procedure van Adams e.a. [2] die in redelijke tijd goede oplossingen genereert. Deze procedure decomponeert het jobshop schedulingprobleem in een aantal één-machine schedulingproblemen die relatief makkelijk op te lossen zijn. Iedere instantie van het jobshop schedulingprobleem kan worden gerepresenteerd door een disjuncte graaf. Door het veranderen van eigenschappen van deze graaf en het veranderen van de algoritmen voor de één-machine schedulingproblemen kan de SB procedure uitgebreid worden om met tal van praktijksituaties om te gaan. We noemen transporttijden, simultaan gebruik van hulpmiddelen en omsteltijden.

Omsteltijd is de tijd die nodig is voor het gereed maken van een

machine om een volgende bewerkingen uit te voeren, bijvoorbeeld omdat gereedschappen gewisseld moeten worden. Gedurende deze tijd is deze machine niet in staat om bewerkingen uit te voeren en dit betekent dus eigenlijk capaciteitsverlies. Er zal een balans gevonden moeten worden tussen efficiënt machinegebruik en een goede leverbetrouwbaarheid. Aan de ene kant is het heel aantrekkelijk bewerkingen met dezelfde instelkarakteristieken direkt na elkaar uit te voeren om zo de machine efficiënt te gebruiken. Dit kan echter ten koste gaan van de leverbetrouwbaarheid van produkten die bewerkingen vereisen met andere instelkarakteristieken. Bovendien kan het efficiënt gebruiken van één machine leegloop veroorzaken van andere machines, omdat deze niet voldoende aanvoer van produkten krijgen.

Indien de Shifting Bottleneck procedure gebruikt wordt voor het schedulen van een werkplaats waarin machines met omsteltijden voorkomen, dan resulteert de decompositie voor deze machines in machine schedulingproblemen waarbij rekening moet worden gehouden met deze omsteltijden. Voor het speciale geval van familieomsteltijden ontwikkelen we een algoritme dat in staat is om problemen met 40 bewerkingen in redelijke tijd optimaal op te lossen. Daarnaast generaliseren we dit algoritme voor het probleem waarin bewerkingen op een parallellemachinegroep gescheduled moeten worden. In dit probleem moet iedere bewerking toegewezen worden aan een van de identieke machines in de groep. Tussen bewerkingen op dezelfde machine zijn omsteltijden nodig. We vergelijken de prestatie van de SB procedure met die van prioriteitsregels voor werkplaatsen waarin omsteltijden voorkomen en onderdelen geassembleerd worden tot eindprodukten.

De SB procedure met uitbreidingen is onderdeel van het commerciële werkplaatsbesturingssysteem JOBPLANNER, dat wordt geleverd door het ingenieursbureau FLEX, Engineers in Logistic Systems. We bespreken dit systeem en het gebruik ervan bij Cityprint B.V., een printpanelenfabriek in Almelo. Het invoeren van JOBPLANNER bij Cityprint heeft veel succes gehad: de doorlooptijden gedaald, terwijl het aantal geproduceerde printpanelen en de leverbetrouwbaarheid zijn gestegen.