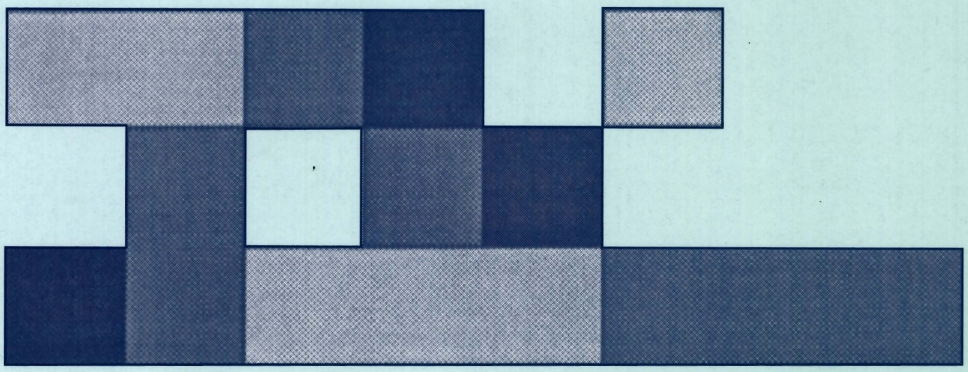
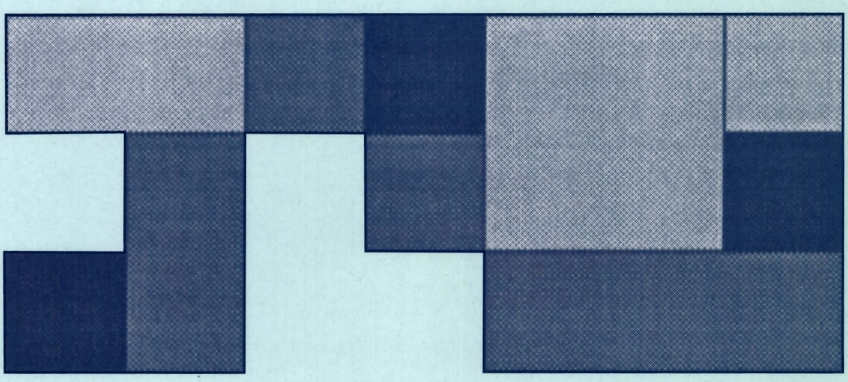
Generalized job shop scheduling : complexity and local search

Generalized Job Shop Scheduling: Complexity and Local Search





R.J.M. Vaessens

Generalized Job Shop Scheduling: Complexity and Local Search

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Vaessens, Robert Johannes Maria

Generalized Job Shop Scheduling: Complexity and
Local Search / Robert Johannes Maria Vaessens. -
Eindhoven: Eindhoven University of Technology
Thesis Technische Universiteit Eindhoven. -
With ref. - With summary in Dutch.
ISBN 90-386-0406-8
Subject headings: scheduling, local search.

Printed and bound by Ponsen & Looijen BV, Wageningen, The Netherlands

Generalized Job Shop Scheduling: Complexity and Local Search

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 22 september 1995 om 14.00 uur

door

ROBERT JOHANNES MARIA VAESSENS

geboren te Cadier en Keer

Dit proefschrift is goedgekeurd
door de promotoren:

prof.dr. J.K. Lenstra
en
prof.dr. E.H.L. Aarts

Preface

This thesis is the result of the research project 'Design, analysis and implementation of local search algorithms', which was financed by the Landelijk Netwerk Mathematische Besliskunde (Dutch Graduate School in the Mathematics of Operations Research), and was carried out at the Eindhoven University of Technology. An additional part of the research has been done at the Politecnico di Milano and was supported by Human Capital and Mobility Project ERB-CHRX-CT93-0087.

The process of writing this thesis has been completed successfully due to the help and support of various people. Therefore, I want to express my gratitude to all people who have contributed in some way to this process.

First of all, I want to thank my supervisors Jan Karel Lenstra and Emile Aarts. Our stimulating discussions, with each of them on his favorite topics, and their many comments on my manuscripts are invaluable to me. Working together has been a great experience for me, though it was sometimes difficult to realize the wishes of both of them simultaneously.

I am also indebted to Francesco Maffioli for having me as a guest in his group at the Politecnico in Milan. I will never forget our discussions on a complexity result in his car on our way back from a very pleasant weekend in the Dolomites.

I want to thank all colleagues who provided me with information for Chapter 5 about their computational work on the job shop scheduling problem and with comments on an earlier draft of a paper on which this chapter was based. I am very grateful to David Applegate and Bill Cook for making their codes available.

I am indebted to many of my colleagues in Eindhoven, who contributed in some way to this thesis. In particular, I want to thank Marc Wennink for the valuable discussions we had and his co-operation in writing our paper, Cor Hurkens for solving a lot of practical problems, my former roommate Patrick Zwietering for our discussions and his help to solve my problems with the computer, Wim Nuijten and Marco Verhoeven for our discussions on the job shop scheduling problem and for providing me with their LaTeX-styles, and Marjan van den Akker and Marc Sol for their co-operation in solving the LNMB exercises.

Furthermore, I want to thank some colleagues from Milan: Marco Trubian

for his co-operation during his stay in Eindhoven and Mauro Dell'Amico for his help and co-operation during my stay in Milan.

Finally, I want to thank my relatives and friends for their invaluable support during the period I was writing this thesis.

Rob Vaessens Eindhoven, July 1995

Contents

1

Introduction

In this thesis we study a particular scheduling problem, called the *generalized job shop scheduling* problem. Scheduling problems occur in situations where a set of activities has to be performed by a set of scarce resources [Baker, 1974]. Scheduling theory is concerned with the optimal assignment of these resources to the activities over time. Its applications can be found in various areas like production planning, personnel planning, computer system control, and time tabling.

Over the past decades, scheduling theory has been the subject of extensive research. Most attention has been paid to *deterministic* scheduling problems, in which all the information that defines a problem instance is known in advance with certainty. Most of the deterministic scheduling problems studied in the literature and also in this thesis are machine scheduling problems, in which the resources are usually called *machines* and the activities *operations*. The main restriction is that a machine can perform at most one operation at a time.

To solve a practical scheduling problem by mathematical means it is necessary to abstract a model from it. This abstraction must capture the essential elements of the practical problem in the sense that it should be possible to convert a solution obtained for the model into a solution of comparable quality for the practical problem. Another requirement is that solutions of satisfactory quality for the model can be found in a moderate amount of computation time. A major problem regarding the relation between the theory and practice of scheduling is that most models considered in the literature are either too simple to reflect re-

ality or too complex to be quickly solvable. In this thesis we intend to reduce the gap by proposing a model that is closer to practice and by making some first steps in the analysis of this model and in the design of efficient methods for its solution.

Our model contains some features which are not present in most models that have been studied in the literature and which lead to a drastic generalization. The most important extension of the model relaxes the requirement that each operation has to be processed by a single machine, which is known in advance. One important feature that has received little attention until now is that an operation may have to be performed by one machine out of a given operation-dependent set of machines, that is, before an operation can be scheduled over time it has to be assigned to a certain machine. Another neglected feature is that the processing of an operation may need the simultaneous cooperation of several machines. These two features are incorporated in the model studied in this thesis. In general, for a given operation some machine sets are given, each of which is capable of processing the operation. The selection of one such machine set for each operation is now part of the scheduling problem. Furthermore, arbitrary precedences between operations are included. On the other hand, only one optimality criterion is considered: the minimization of the maximum completion time.

There are many other aspects that are not reflected in most of the existing models and that are not considered in the generalized model studied in this thesis either. One could think of non-regular optimality criteria (that is, possibly decreasing in some of completion times of the operations), multiple optimality criteria, processing times that depend on the sequence in which operations are scheduled, or on the time at which they are scheduled, sequence-dependent setup times of machines, and preemptable operations. Several of these aspects are relatively easy to incorporate in the model studied here. For instance, non-regular or multiple optimality criteria do not lead to substantial changes in the description of the model. Furthermore, some more specialized problem areas like periodic scheduling and cyclic scheduling are excluded.

Since the generalized job shop scheduling is hard to solve, approximative solution methods are considered. Roughly speaking, one can divide approximative solution methods in two types. The first type consists of constructive methods and the second type of iterative methods. Constructive methods build a single schedule, which is usually done by using simple dispatch rules. Such methods may be easy to analyze in the sense that a worst-case bound can be derived on the length of any schedule obtained. In practice, however, constructive methods yield schedules of only moderate quality. This disadvantage is absent in many iterative methods, which generate many schedules instead of only a single one.

An important subclass of iterative methods is formed by the class of local search algorithms. Here, a sequence of schedules is generated, such that each schedule in this sequence is obtained by modifying the previous schedule in the sequence. In this thesis we extensively discuss local search methods, which have been proven to be quite successful for many optimization problems. We note that many hybrid approximative solution methods exist: constructive methods exist that use some form of local search, and iterative methods that use some form of construction.

The remainder of this introductory chapter is organized in the following way. Section 1.1 introduces the generalized job shop scheduling problem in an informal way. Section 1.2 deals with combinatorial optimization and local search. Finally, Section 1.3 gives an outline of the thesis.

1.1 The generalized job shop scheduling problem

One of the currently most complex machine scheduling models is that of job shop scheduling. Here, we are given a set of jobs and a set of machines. Each machine can handle at most one job at a time. Each job consists of a chain of operations, each of which needs to be processed during an uninterrupted time period of a given length on a given machine. The first operation of each job becomes available at time 0, and each other operation becomes available as soon as the processing of its predecessor in the chain has been completed. The purpose is to find a schedule, that is, an assignment of the operations to time intervals on the machines, such that a given optimality criterion is minimized.

The model we introduce here is more general and we therefore call it the *generalized job shop scheduling problem*. The main generalization is that an operation may be performed by several machines simultaneously. Such a set of machines that processes an operation simultaneously is called a *machine set*. A second generalization is that an operation may have several alternative machine sets, each being capable of processing the operation. One of these machine sets has to be selected to perform the operation. Here, the processing time of an operation may depend on the selected machine set. A third generalization is that an arbitrary precedence relation on the set of operations may be defined, instead of the simple precedence relation for the job shop scheduling problem that decomposes the set of operations into chains.

The model of the generalized job shop scheduling problem can now be described as follows. Given are a set of operations and a set of machines. For each operation, a set of machine sets is given; each of these machine sets is capable of processing the operation. For each operation and each of its machine sets, a processing time is given. Furthermore, a binary precedence relation is given

on the set of operations. A precedence between two operations denotes that the processing of the second operation cannot start before the processing of the first operation has been finished.

A *schedule* consists of two parts: an assignment of operations to machine sets and an assignment of operations to time intervals. Obviously, given the assignment of operations to machine sets, it is sufficient to know the starting time of each operation. A schedule is called *feasible* if each operation is processed by one of its machine sets for the required duration of time, if at any time instant no machine takes part in the processing of more than one operation, if the precedences are satisfied, and if each starting time is nonnegative. Our goal is to find a feasible schedule that minimizes the maximum completion time over all operations.

1.2 Combinatorial optimization and local search

The generalized job shop scheduling problem belongs to the class of combinatorial optimization problems. An *optimization problem* is either a maximization or a minimization problem specified by a class of problem instances. Without loss of generality we restrict ourselves to minimization problems. An *instance* is defined by the implicit specification of a solution space, a totally ordered space of possible cost values, and a cost function. The objective is to find a solution with minimum cost. An optimization problem is called a *combinatorial optimization problem* if for each instance the solution space is finite or countably infinite.

The *decision variant* of a combinatorial minimization problem considers the following question: does there exists a solution in the solution space, the cost of which does not exceed a given upper bound? To distinguish between the decision variant of a combinatorial minimization problem and the problem itself, the latter is often called the *optimization variant*. The decision variant of a combinatorial minimization problem may belong to $\mathcal{NP}$, the class of decision problems that can be solved non-deterministically in polynomial time. It also may belong to the subclass $\mathcal{P}$ of $\mathcal{NP}$, the class of decision problems that can be solved deterministically in polynomial time. A third possibility is that it does not belong to $\mathcal{NP}$. A decision problem is NP-complete if it belongs to $\mathcal{NP}$ and if it is as least as difficult as any other problem in $\mathcal{NP}$. If a decision problem is NP-complete, it is not possible to solve an arbitrary instance in polynomial time, unless $\mathcal{P}$ equals $\mathcal{NP}$. The corresponding optimization variant of such a problem is then called NP-hard. Solving an instance of an NP-hard optimization problem may need large and even impractical amounts of computation time.

If the problem under consideration is NP-hard, it may be that the instances one is interested in have some special structure. In this case it may be that the

subclass of instances in question is in fact solvable in polynomial time. Therefore, it is worth to consider the complexity of special cases of a given general problem. In this thesis we apply this approach to the generalized job shop scheduling problem. We consider several subproblems, for each of which we either prove that it is solvable in polynomial time or that it is still NP-hard.

If, after all, the instances one is interested in do not belong to a subproblem that can be solved in polynomial time, two options are left: optimization or approximation. Most optimization algorithms proceed by branch and bound. For most problems, small problem instances can still be solved in reasonable amounts of computation time, but solving larger instances may need enormous amounts of computation time, since the time needed to solve an arbitrary instance is superpolynomial in the size of the instance. The other option is to resort to approximation algorithms. An approximation algorithm tries to find a solution, the cost of which is near to the optimal cost. However, it is not guaranteed to find an optimal solution.

There are two different types of approximation algorithms. The first type consists of constructive methods. Such a method constructs in most cases only one solution using some problem specific rules. For many such rules one can prove that the ratio between the value of a solution found by this rule and the optimal value is bounded from above by a constant. But for many hard combinatorial optimization problems this difference may still be large. The second type of approximation algorithms consists of iterative methods, which generate various solutions, the best of which is often of better quality than the solutions found by constructive methods. An important subclass of iterative methods is formed by the class of so-called *local search* methods. A local search algorithm starts from a given initial solution, and then iteratively generates new solutions, each of which is obtained by modifying some parts of the previous solution. In such a way a sequence of solutions is obtained. A *neighborhood function* specifies which modifications are allowed. This function implicitly assigns to each solution a set of neighboring solutions or *neighbors* that can be reached from this solution. A *search strategy* specifies for each iteration which neighbor is selected from the neighborhood of the current solution as the next solution in the sequence. Most search strategies are such that solutions of good quality are preferred to solutions of lower quality. In this way many solutions are obtained and the best of these often have a relatively good quality. However, much depends on how the neighborhood function and the search process are defined.

Local search methods are generally applicable and obtain good results for many hard optimization problems. Local search approaches date back to the late 1950's, when Bock [1958] and Croes [1958] developed the first link exchange

procedures for the traveling salesman problem. Ever since, a large variety of local search algorithms has been proposed, each aiming at different remedies to the risk of getting stuck in poor local optima. Reeves [1993] provides a comprehensive survey of the area; a more detailed treatment is given by Aarts and Lenstra [1995]. Many local search methods have been developed for the job shop scheduling problem and some of them were quite successful. For this reason we may expect that local search will also give good results for the generalized job shop scheduling problem. In this thesis we discuss various existing local search methods for the job shop scheduling problem and we focus on some aspects of local search methods for the generalized job shop scheduling problem.

1.3 Outline of the thesis

The remainder of this thesis is organized as follows. Chapter 2 introduces the generalized job shop scheduling problem in a formal way. Chapter 3 deals with the computational complexity of this problem and of many of its subproblems. Chapter 4 describes several types of local search methods and provides a template that captures these methods. It is based on a paper by Vaessens, Aarts and Lenstra [1995a]. Chapter 5 reviews various local search methods that have been developed for the job shop scheduling problem and discusses their performance. It is based on a paper by Vaessens, Aarts and Lenstra [1995b]. Finally, Chapter 6 introduces several neighborhood functions for the generalized job shop scheduling problem and deals with some of their properties. Section 6.2 is for a considerable part based on a paper by Wennink and Vaessens [1995].

2

The generalized job shop scheduling problem

In this chapter we introduce a generalization of the job shop scheduling problem. In Section 2.1 we describe a model of this generalization. In Section 2.2 we give a classification of subproblems. In Section 2.3 we give the classification of some well-known shop scheduling problems.

2.1 The model

2.1.1 Basic definitions

In our model we are given a set of operations and a set of machines. Each operation needs processing on a subset of machines. For each operation a collection of such machine sets are given, one of which has to be selected for processing the operation. Furthermore, a precedence relation on the set of operations is given, denoting that for some pairs of operations the processing of the second operation cannot start before the processing of the first has been finished. A schedule is an allocation of operations to subsets of machines and to time intervals. We do not allow that the execution of an operation, once started, is interrupted. Our goal is to find a schedule such that the maximum completion time over all operations is minimized.

In a formal way the problem is defined as follows. We are given a finite set V of operations and a finite set M of machines. For each operation $v \in V$ there

is a non-empty set $\mathcal{H}(v)$, each consisting of non-empty subsets of M. For each operation $v \in V$ and each $H \in \mathcal{H}(v)$ a processing time $p(v, H) \in \mathbb{N}$ is given. On V a partial order A is defined, that is, a binary, antireflexive, and transitive relation.

A *schedule* (K, S) defines for each operation v a machine set $K(v)$ on which it will be processed and a start time $S(v)$. A schedule is *feasible* if the following restrictions hold:

$$\forall v \in V: \quad K(v) \in \mathcal{H}(v) \wedge S(v) \in \mathbb{N} \cup \{0\}; \tag{2.1}$$

$$\forall (v, w) \in A: \quad S(v) + p(v, K(v)) \leq S(w); \tag{2.2}$$

$$\forall v, w \in V, v \neq w: K(v) \cap K(w) \neq \emptyset \Rightarrow S(v) + p(v, K(v)) \leq S(w) \vee S(w) + p(w, K(w)) \leq S(v). \tag{2.3}$$

Here, requirement 2.1 stipulates that each operation must be processed on one of its possible machine sets and that it is not available before time 0. Requirement 2.2 stipulates that for pairs (v, w) for which a precedence exists, the processing of the second operation cannot start before the processing of the first has been completed. Finally, requirement 2.3 stipulates that all machines have capacity one, so that the processing of two operations with a common machine in their chosen machine sets cannot overlap in time.

Now the problem is to find a feasible schedule (K, S), such that its *length* (or *makespan*)

$$\max_{v \in V} S(v) + p(v, K(v))$$

is minimized. Such a schedule is called an *optimal* schedule.

We now introduce some additional notation, which is used throughout this thesis.

The number of operations is denoted by l and the number of machines by m. The set of all subsets of M is denoted by $\mathcal{M}$. If for an operation $v \in V$ the number of machine sets in $\mathcal{H}(v)$ is equal to 1, and if in addition this unique machine set consists of one machine only, this machine is denoted by $\mu(v)$.

The transitive reduction of the precedence relation A is denoted by $\tilde{A}$. The set of operations corresponding to a maximal connected component in the graph (V, A) is called a *job*. The set of all jobs is denoted by $\mathcal{J}$ and its size is denoted by n. The number of operations of a job $J \in \mathcal{J}$ is denoted by $|J|$. The unique job that contains operation v is denoted by $J(v)$.

Given a feasible schedule (K, S), the completion time $C(v)$ of an operation

v is defined as $C(v)=S(v)+p(v, K(v))$. Furthermore,

$$C_{\max} = \max_{v \in V} S(v)+p(v, K(v)).$$

Finally, in some cases we use indices to be able to distinguish between different operations, machines, or jobs. Then the set V of operations is written as $\{v_1, \ldots, v_l\}$, the set M of machines as $\{\mu_1, \ldots, \mu_m\}$, and the set $\mathcal{J}$ of jobs as $\{J_1, \ldots, J_n\}$.

2.1.2 Some elementary properties

In the following we discuss some elementary properties of schedules, which are used throughout this thesis.

Definition 2.1.

* A feasible schedule is called *left-justified* or *semi-active* if it is not possible to complete any operation earlier such that
  - ⋆ each operation is processed by the same machine set,
  - ⋆ each other operation is completed as least as early, and
  - ⋆ on each machine the processing order remains the same.
* A feasible schedule is called *(weakly) active* if it is not possible to complete any operation earlier such that
  - ⋆ each operation is processed by the same machine set,
  - ⋆ each other operation is completed as least as early, and
  - ⋆ on each machine the processing order of the other operations remains the same.
* A feasible schedule is called *strongly active* if it is not possible to complete any operation earlier such that
  - ⋆ each other operation is processed by the same machine set,
  - ⋆ each other operation is completed as least as early, and
  - ⋆ on each machine the processing order of the other operations remains the same.
* A feasible schedule is called *left-optimal* if it is not possible to complete any operation earlier such that
  - ⋆ each other operation is completed as least as early. □

Note that we have the following inclusions: each left-optimal schedule is strongly active, each strongly active schedule is active, and each active schedule is left-justified.

The following example illustrates the various properties introduced above.

Example 2.2. Consider the following instance with

$$\begin{aligned} V &= \{v_j \mid j = 1, \ldots, 8\}, \\ M &= \{\mu_i \mid i = 1, \ldots, 3\}, \text{ and} \\ A &= \{(v_1, v_2), (v_2, v_3), (v_1, v_3), (v_4, v_5), (v_6, v_7)\}. \end{aligned}$$

Let the machine sets and processing times be as follows:

$$\begin{array}{lll} \mathcal{H}(v_1) = \{\{\mu_1\}, \{\mu_2\}\} & p(v_1, \{\mu_1\}) = 2 & p(v_1, \{\mu_2\}) = 3; \\ \mathcal{H}(v_2) = \{\{\mu_1\}\} & p(v_2, \{\mu_1\}) = 3; & \\ \mathcal{H}(v_3) = \{\{\mu_3\}\} & p(v_3, \{\mu_3\}) = 1; & \\ \mathcal{H}(v_4) = \{\{\mu_1\}, \{\mu_2\}\} & p(v_4, \{\mu_1\}) = 2 & p(v_4, \{\mu_2\}) = 1; \\ \mathcal{H}(v_5) = \{\{\mu_2\}, \{\mu_3\}\} & p(v_5, \{\mu_2\}) = 1 & p(v_5, \{\mu_3\}) = 1; \\ \mathcal{H}(v_6) = \{\{\mu_1\}\} & p(v_6, \{\mu_1\}) = 2; & \\ \mathcal{H}(v_7) = \{\{\mu_2\}\} & p(v_7, \{\mu_2\}) = 2; & \\ \mathcal{H}(v_8) = \{\{\mu_3\}\} & p(v_8, \{\mu_3\}) = 4. & \end{array}$$

Now Figure 2.1 contains various schedules that illustrate the properties introduced in Definition 2.1. □

We first prove that the definitions of a left-justified schedule and a strongly active schedule can be simplified by deleting one of the restrictions.

Theorem 2.3. *A schedule is left-justified if and only if it is not possible to complete any operation earlier such that*

* *each operation is processed by the same machine set, and*
* *on each machine the processing order remains the same.*

Proof. Clearly, if in a schedule no operation can completed earlier such that each operation is processed by the same machine set and on each machine the processing order remains the same, then this schedule is left-justified. Now suppose that we are given a schedule (K, S) in which an operation v exists that can be completed earlier such that each operation is processed by the same machine set and on each machine the processing order remains the same. We prove that this schedule cannot be left-justified by showing that v can be completed earlier even without completing any other operation later. Let (K, S') denote a schedule with $C'(v) < C(v)$ for which the processing order on each machine is the same as in (K, S). We prove that there exists a schedule (K, S'') with $C''(v) < C(v)$, with $C''(w) \leq C(w)$ for all $w \in V$, and with the same processing order on each

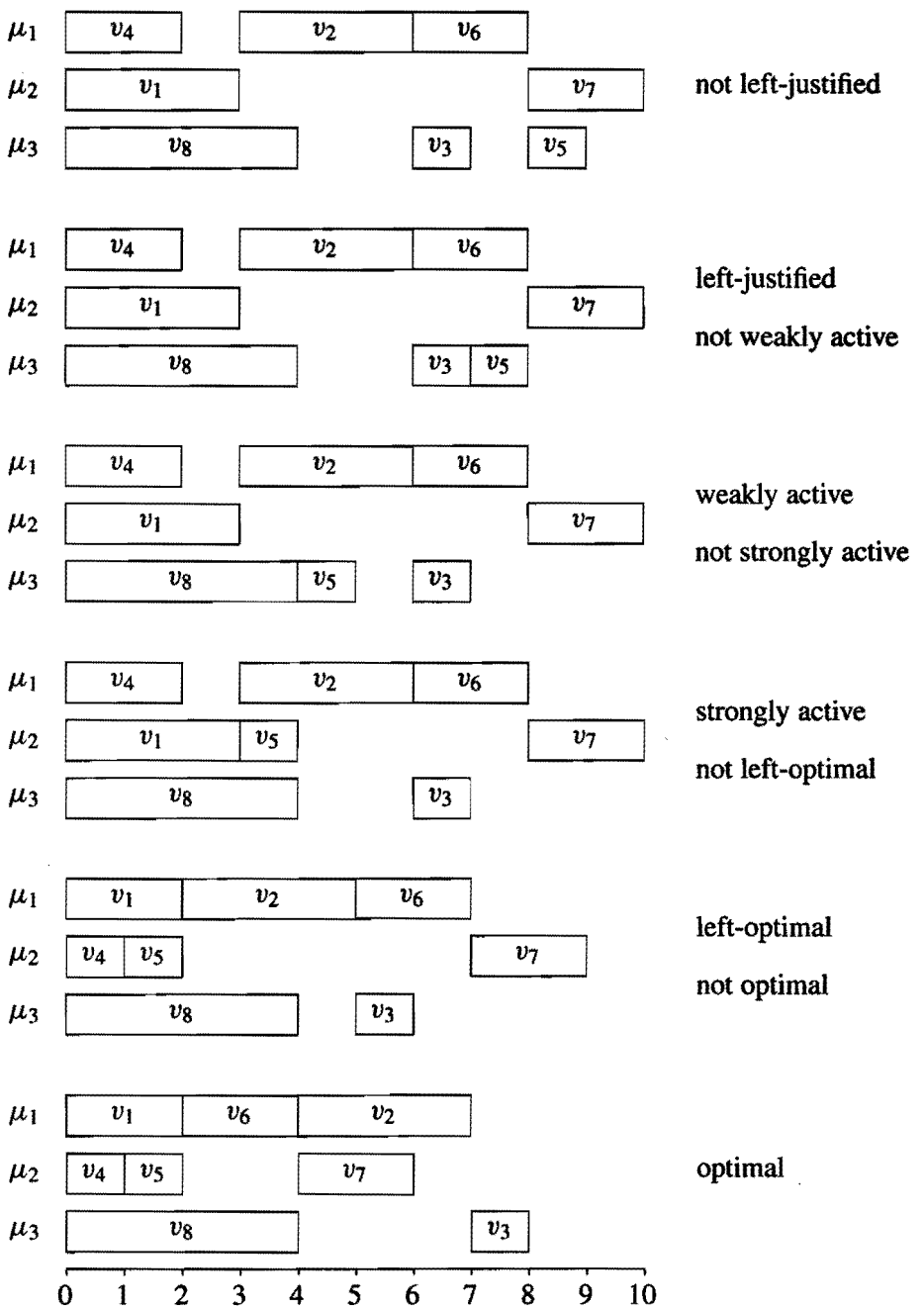


Figure 2.1: Illustration of the notions of Definition 2.1.

machine as in the schedule (K, S). Clearly, such a schedule proves that (K, S) is not left-justified. Define (K, S'') by $S''(w) = \min\{S(w), S'(w)\}$ for all $w \in V$. We will show that (K, S'') is feasible. Let $u, w \in V$ be two operations with $(u, w) \in A$ or with $K(u) \cap K(w) \neq \emptyset$ and $S(w) \geq S(u) + p(u, K(u))$. Since (K, S') is feasible and all orderings are the same as in (K, S), we also have that $S'(w) \geq S'(u) + p(u, K(u))$. But this implies that $S''(w) \geq S''(u) + p(u, K(u))$, which proves the feasibility of (K, S''). Now it is clear that $C''(v) < C(v)$, that $C''(w) \leq C(w)$ for all $w \in V$, and that all orderings in (K, S'') are the same as in (K, S). □

Theorem 2.4. *A schedule is strongly active if and only if it is not possible to complete any operation earlier such that*

* *each other operation is processed by the same machine set and*
* *each other operation is completed as least as early.*

Proof. Clearly, if in a schedule no operation can completed earlier such that each other operation is processed by the same machine set and each other operation is completed as least as early, then this schedule is strongly active. Now suppose that we are given a schedule (K, S) in which an operation v exists that can be completed earlier such that each other operation is processed by the same machine set and each other operation is completed as least as early. We prove that this schedule cannot be strongly active. If (K, S) is not weakly active, then it is also not strongly active, and we are done. So assume (K, S) is weakly active. Now, let (K', S') denote a schedule with $C'(v) < C(v)$ and with $K'(w) = K(w)$ and $C'(w) \leq C(w)$ for all operations $w \neq v$. If no operation x exists with $K(x) \cap K'(v) \neq \emptyset$ and $[S(x), C(x)) \cap [S'(v), C'(v)) \neq \emptyset$, then (K, S) is not strongly active, since v could be scheduled with smaller completion time without changing S and K for the other operations. If such an x does exist, it must be scheduled earlier in (K', S'). But, since $K'(x) = K(x)$, it is then also possible to find a schedule (K, S'') in which x is scheduled earlier. This contradicts the assumption that (K, S) is weakly active. □

The classes of weakly active and strongly active schedules coincide for problems in which each operation can only be processed by one fixed machine set. Furthermore, each of the schedule classes introduced above contains at least one optimal schedule. In contrast, the following classes of weakly and strongly undelayed schedules do not necessarily contain an optimal schedule.

Definition 2.5.

* A feasible schedule (K, S) is called a *(weakly) undelayed* schedule if there exists no schedule (K, S') in which there is an operation v with $S'(v) < S(v)$ such that all machines of the machine set $K(v)$ are idle (or starting an idle period) in (K, S) at time $S'(v)$.
* A feasible schedule (K, S) is called a *strongly undelayed* schedule if there exists no schedule (K', S') in which there is an operation v with $S'(v) < S(v)$ and $K'(w) = K(w)$ for all $w \neq v$, that is such that all machines of the machine set $K'(v)$ are idle (or starting an idle period) in (K, S) at time $S'(v)$. □

It is not difficult to see that the problem of deciding whether a given schedule is left-justified, weakly active, strongly active, weakly undelayed, or strongly undelayed can be solved in polynomial time. In contrast, the following theorem states that deciding whether a given schedule is not left-optimal is NP-complete in the strong sense. Therefore, the use of the notion of left-optimality is less practical.

Theorem 2.6. *Deciding whether a given schedule is not left-optimal is NP-complete in the strong sense.*

Proof. Suppose that we have another schedule in which all operations are completed at least as early as in the given schedule and in which at least one operation is completed earlier. This schedule proves that the given one is not left-optimal. Comparing the completion times requires polynomial time, and therefore the decision problem belongs to $\mathcal{NP}$.

To show completeness we give a polynomial-time reduction from the problem 3-PARTITION to this decision problem.

The problem 3-PARTITION is defined as follows: let $T = \{0, \ldots, 3t - 1\}$ for some $t \in \mathbb{N}$ and let $Z \in \mathbb{N}$; let, for each $i \in T$, a number $z_i \in \mathbb{N}$ be given with $\frac{Z}{4} < z_i < \frac{Z}{2}$ and such that

$$\sum_{i=0}^{3t-1} z_i = tZ;$$

the question is whether the set T can be partitioned into t pairwise disjoint sets T_j, $j \in \{0, \ldots, t - 1\}$, such that, for all $j \in \{0, \ldots, t - 1\}$,

$$\sum_{i \in T_j} z_i = Z.$$

Such a partition is called a 3-partition. 3-PARTITION has been proven NP-complete in the strong sense by Garey and Johnson [1975].

Suppose we are given an instance of the problem 3-PARTITION. We transform this instance into an instance of the generalized job shop scheduling problem and a schedule of this instance. This is done in such a way that the given schedule is not left-optimal if and only if a 3-partition exists.

The transformation is defined as follows. V, M, and A are defined as

$V = \{v_j | j = 0, \ldots, 3t\}$,
$M = \{\mu_i | i = 0, \ldots, 4t-1\}$, and
$A = \emptyset$.

The machine sets are taken as follows:

$$\mathcal{H}(v_j) = \{\{\mu_i\} | i = 0, \ldots, 4t-1\} \quad \text{if } j = 0, \ldots, 3t-1;$$
$$\mathcal{H}(v_{3t}) = \{\{\mu_i | i = 0, \ldots, 3t-1\}, \{\mu_i | i = 3t, \ldots, 4t-1\}\}.$$

The processing times are taken as follows:

$$p(v_j, \{\mu_i\}) = \begin{cases} Z & \text{if } i = 0, \ldots, 3t-1 \text{ and } j = 0, \ldots, 3t-1 \\ z_j & \text{if } i = 3t, \ldots, 4t-1 \text{ and } j = 0, \ldots, 3t-1 \end{cases}$$
$$p(v_{3t}, \{\mu_i | i = 0, \ldots, 3t-1\}) = Z$$
$$p(v_{3t}, \{\mu_i | i = 3t, \ldots, 4t-1\}) = Z+1.$$

Now, let the given schedule (K, S) be defined by:

$$K(v_j) = \{\mu_j\} \qquad S(v_j) = 0 \quad \text{if } j = 0, \ldots, 3t-1;$$
$$K(v_{3t}) = \{\mu_i | i = 3t, \ldots, 4t-1\} \qquad S(v_{3t}) = 0.$$

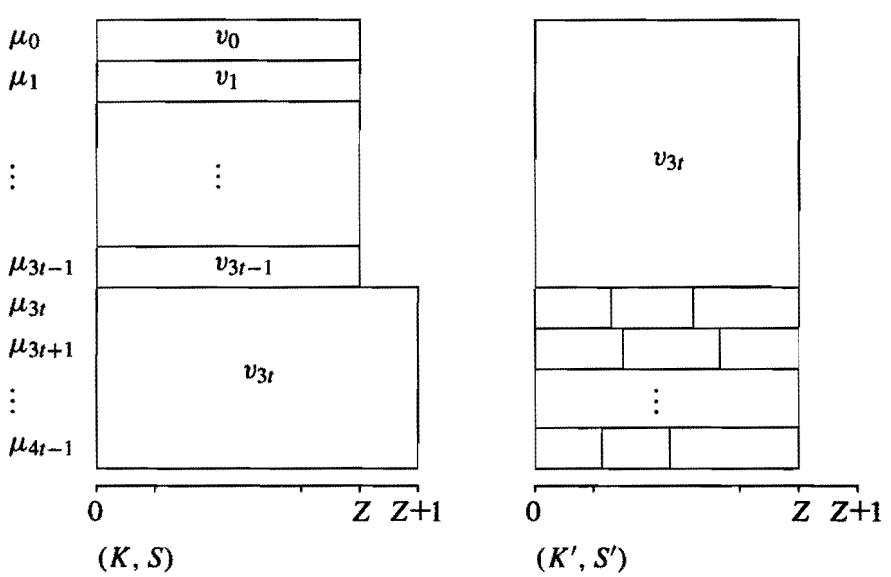


Figure 2.2: Schedules (K, S) and (K', S').

Clearly, v_{3t} has completion time $Z+1$ and all other operations have completion time Z. Figure 2.2 illustrates this schedule.

Now suppose that this schedule is not left-optimal. Then there exists a schedule (K', S') with $C'(v) \leq C(v)$ for all operations $v \in V$, and there is at least one such v for which inequality holds. Suppose that inequality holds for some v_j with $j \in \{0, \ldots, 3t-1\}$. Then v_j should be scheduled on $\{\mu_i\}$ for some $i \in \{3t, \ldots, 4t-1\}$, which implies that v_{3t} should be scheduled on the machine set $\{\mu_i | i = 0, \ldots, 3t-1\}$. On the other hand, when inequality holds for v_{3t}, it follows immediately that v_{3t} should be scheduled on $\{\mu_i | i = 0, \ldots, 3t-1\}$. In both cases it follows that each operation v_j with $j \in \{0, \ldots, 3t-1\}$ should be scheduled on a machine μ_i with $i \in \{3t, \ldots, 4t-1\}$. The sum of the processing times of these operations on the machines μ_i with $i \in \{3t, \ldots, 4t-1\}$ equals tZ. Since every such operation should be completed by Z, this is only possible when a 3-partition exists for the given instance of 3-PARTITION. Figure 2.2 also illustrates how such a schedule (K', S') must look like.

On the other hand, when a 3-partition exists for the given 3-PARTITION instance, a schedule (K', S') can be easily found, in which v_{3t} starts at time 0 on $\{\mu_i | i = 0, \ldots, 3t-1\}$ and in which each other operation is scheduled on a machine μ_i for some $i \in \{3t, \ldots, 4t-1\}$ with completion time at most Z. This schedule proves that (K, S) is not left-optimal. □

2.1.3 Graph representations

In this subsection it is shown how the generalized job shop scheduling problem can be described by a means of a graph.

Suppose that the machine set assignment K is known, that is, for each operation it has been decided by which machine set it will be processed. Then what remains is to find a function S. The problem of finding such a function S can be represented by means of a *disjunctive graph* $\mathcal{G}_K = (\mathcal{V}, \mathcal{A}, E_K)$ [Roy and Sussmann, 1964]. Here, the vertex set $\mathcal{V}$ consists of all operations in V, together with two dummy vertices s and t, which represent the start and the end of each schedule. The arc set $\mathcal{A}$ consists of the *precedence arcs* of A, which represent the given precedences between the operations, and *dummy arcs* (s, v) and (v, t) for each operation $v \in V$. The edge set E_K defined by

$$E_K = \{\{v, w\} \in \mathcal{P}_2(V) \mid K(v) \cap K(w) \neq \emptyset\}$$

represents the machine capacity constraints. Here, $\mathcal{P}_2(V)$ denotes the set of all subsets of V of size 2. Each vertex $v \in V$ has a weight, equal to the processing time $p(v, K(v))$; s and t have no weight.

For each edge $\{v, w\} \in E_K$ it has to be decided whether v will be processed

before w or w before v. These decisions are represented by a *complete orientation*. A complete orientation on E_K is a function $\Omega : E_K \to V \times V$ such that $\Omega(\{v, w\}) \in \{(v, w), (w, v)\}$ for each $\{v, w\} \in E_K$; furthermore, we write $\Omega(E_K) = \{\Omega(e) \mid e \in E_K\}$. Hence, $\Omega(E_K)$ contains for each edge $\{v, w\} \in E_K$ a so-called *machine arc*, which defines the relative position of v and w on their common machines. The corresponding digraph $\mathcal{G}_{(K,\Omega)} = (\mathcal{V}, \mathcal{A} \cup \Omega(E_K))$ is called the *solution graph*. Each arc $(v, w) \in \mathcal{A} \cup \Omega(E_K)$ in the solution graph represents a constraint of the form $S(w) \geq C(v)$. So, $\Omega(E_K)$ represents for each machine its *machine ordering*, that is, the order in which it processes the operations with this machine in their chosen machine set.

However, the solution graph $\mathcal{G}_{(K,\Omega)}$ may contain cycles. In this case it is not possible to find a schedule (K, S) in which each pair of operations is scheduled in an order corresponding to the orientation Ω. Therefore, a complete orientation Ω is defined to be *feasible* for machine set assignment K if the digraph $\mathcal{G}_{(K,\Omega)}$ is acyclic.

Clearly, each feasible schedule (K, S) uniquely determines a feasible complete orientation, which will be denoted by $\Omega_{(K,S)}$. Conversely, for each machine set assignment K and each feasible complete orientation Ω on E_K, there is a unique left-justified feasible schedule, in which the start time function will be denoted by $S_{(K,\Omega)}$. $S_{(K,\Omega)}$ can be computed by taking $S_{(K,\Omega)}(v)$ for all $v \in V$ equal to the length of a longest path from s to v in the digraph $\mathcal{G}_{(K,\Omega)}$. Here, the length of a path $(v_1, v_2, \ldots, v_k)$ is defined as the sum of the processing times of the operations v_2 up to and including v_{k-1}. Now, the length of the schedule $(K, S_{(K,\Omega)})$ equals the length of a longest path from s to t in the digraph. Finding an optimal left-justified schedule is now equivalent to finding a feasible complete orientation that minimizes the length of a longest s-t path in the corresponding digraph. Such a longest s-t path is also called *critical path*.

Note that as a result of this we obtained a one-to-one correspondence between left-justified schedules and complete feasible orientations. As we have noticed before, to find a schedule of minimum length it is sufficient to consider only left-justified schedules.

Many of the dummy arcs and precedence arcs in the disjunctive graph and many of the dummy arcs, precedence arcs, and machine arcs in a solution graph are redundant in the sense that they are implied by other arcs. Both graphs can be reduced by deleting such implied arcs. The resulting graph is called the *transitive reduction* of the original graph.

The following example gives an illustration of a disjunctive graph and a corresponding solution graph.

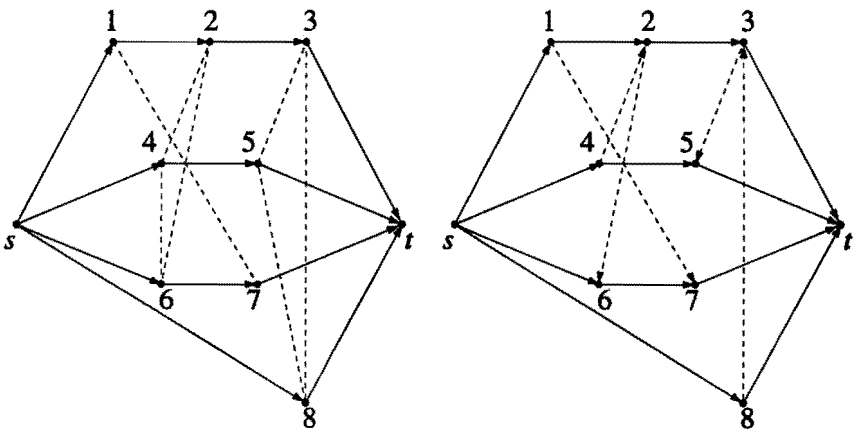


Figure 2.3: A disjunctive graph and a corresponding solution graph.

Example 2.7. Consider again the instance defined in Example 2.2. Figure 2.3 gives the transitive reduction of the disjunctive graph and the solution graph for the machine set assignment and the orientation corresponding to the upper solution depicted in Figure 2.1. The straight arrows denote the reduction of the arc set $\mathcal{A}$, the dotted lines denote the edge set E_K, and the dotted arcs denote the reduction of the arc set $\Omega(E_K)$. □

We also need to define *partial solutions* in terms of the graph representation described above. In a partial solution, only for some of the operations the chosen machine set is known, and only for some of these operations the relative order on their common machines is known. More formally, in a partial solution, $K(v)$ is defined for all $v \in W$ of a given subset $W \subseteq V$; this partial machine set assignment is denoted by $K|_W$. Orientations can only be defined for operations of W; the set

$$E_{K|_W} = \{\{v, w\} \in \mathcal{P}_2(W) \mid K(v) \cap K(w) \neq \emptyset\}$$

consists of all edges that can be oriented. $\Omega(\{v, w\})$ is defined for all $\{v, w\} \in F$ of a subset $F \subseteq E_{K|_W}$; this partial orientation on a subset $F \subseteq E_{K|_W}$ is denoted by $\Omega|_F$.

For the partial machine set assignment $K|_W$ the corresponding disjunctive graph equals $\mathcal{G}_{K|_W} = (\mathcal{V}, \mathcal{A}, E_{K|_W})$. Each vertex $v \in W$ has a weight, equal to the processing time $p(v, K(v))$; each vertex $v \in V \setminus W$ has weight 0.

For the partial machine set assignment $K|_W$ and the partial orientation $\Omega|_F$ the corresponding solution graph equals $\mathcal{G}_{(K|_W, \Omega|_F)} = (\mathcal{V}, \mathcal{A} \cup \Omega|_F(F))$, where

$\Omega|_F(F) = \{\Omega|_F(e) \mid e \in F\}$. The partial orientation $\Omega|_F$ is defined to be *feasible* for the partial machine set assignment $K|_W$ if the digraph $\mathcal{G}_{(K|_W, \Omega|_F)}$ is acyclic.

Clearly, if $W = V$ and $F = E_K$, then the notions and graphs introduced for partial solutions are identical to those for complete solutions.

2.2 Classification

A subproblem of our general model is specified by its machine characteristics, by its operation and precedence characteristics, and by its objective characteristics. These specifications are denoted in the form of a three-field classification $\alpha|\beta|\gamma$, which is introduced in this section. This classification is based on the one given by Graham, Lawler, Lenstra and Rinnooy Kan [1979]. The symbol $\circ$ is used to denote the empty symbol, which will be omitted when giving a specification. The symbol $\bullet$ is also used to denote the empty symbol, but this one will always be written in a specification.

2.2.1 Machine characteristics

The first field $\alpha = \alpha_1\alpha_2\alpha_3\alpha_4$ specifies the machine characteristics. It is defined as follows.

1. $\alpha_1 \in \{\circ, J, F, F_\pi, O\}$.
   If $\alpha_1 = J$, then the problem under consideration is a job shop scheduling problem. The problem is a flow shop scheduling problem if $\alpha_1 = F$ and a permutation flow shop scheduling problem if $\alpha_1 = F_\pi$. If $\alpha_1 = O$, then the problem is an open shop scheduling problem. The problem types for which $\alpha_1 \neq \circ$ will be defined in Section 2.3.
2. $\alpha_2 \in \mathbb{N} \cup \{m, \bullet\}$.
   If $\alpha_2 \in \mathbb{N}$, then it specifies the number m of machines as part of the problem type. If $\alpha_2 = m$, then the number of machines is also specified as part of the problem type, but its value is an unknown constant. If $\alpha_2 = \bullet$, the number of machines is specified as part the problem instance.
3. $\alpha_3 \in \mathbb{N} \cup \{\bullet\}$.
   If $\alpha_3 \in \mathbb{N}$, then it specifies an upper bound on the size of all possible machine sets: for all operations $v \in V$ and for all possible machine sets $H \in \mathcal{H}(v)$ we have $|H| \leq \alpha_3$. If $\alpha_3 = \bullet$, then no upper bound is specified on the size of the possible machine sets.
4. $\alpha_4 \in \mathbb{N} \cup \{\bullet\}$.
   If $\alpha_4 \in \mathbb{N}$, then it specifies an upper bound on the number of possible machine sets for each operation: for all operations $v \in V$ we have $|\mathcal{H}(v)| \leq \alpha_4$. If

$\alpha_4 = \bullet$, then no upper bound is specified on the number of possible machine sets.

2.2.2 Operation and precedence characteristics

The second field $\beta = \beta_1\beta_2\beta_3\beta_4$ specifies the operation and precedence characteristics. It is defined as follows.

1. $\beta_1 \in \{\circ, \text{empty}, \text{chain}, \text{intree}, \text{outtree}\}$.
   The entry β_1 specifies the precedence relation A. If $\beta_1 = \text{empty}$, then there are no precedences. If $\beta_1 \in \{\text{chain}, \text{intree}, \text{outtree}\}$, then the precedence relation has the structure of the union of several chains, intrees, or outtrees, respectively. If $\beta_1 = \circ$, no restrictions on the precedence relation are given.
2. $\beta_2 \in \{\circ, n\} \cup \{n = x\}$ with $x \in \mathbb{N}$.
   If β_2 equals n, then the number of jobs is specified as part of the problem type, but its value is an unknown constant. If β_2 is equal to $n = x$ for some $x \in \mathbb{N}$, then the number of jobs is specified as part of the problem type and is equal to x. Otherwise, no restriction on the number of jobs is given.
3. $\beta_3 \in \{\circ\} \cup \{|J| \leq x\}$ with $x \in \mathbb{N}$.
   If β_3 equals $|J| \leq x$ for some $x \in \mathbb{N}$, then all jobs consist of at most x operations. Otherwise, no restrictions on the size of the jobs are given.
4. $\beta_4 \in \{\circ, p(v)/q(H), p(v)/q(|H|), p(v), p(H), p(|H|), 1\}$.
   * If $\beta_4 = \circ$, then the processing times have no special structure.
   * If β_4 equals $p(v)/q(H)$, we have uniform machine sets. This means that each operation has a certain processing requirement, which is denoted by $p(v)$, and that each possible machine set has a certain speed, which is denoted by $q(H)$. The resulting processing time is then the ratio of the processing requirement of the operation and the speed of the machine set.
   * If β_4 equals $p(v)/q(|H|)$, we have identical machines. Now, in addition to the requirement for uniform machine sets, the speed of a machine set only depends on its size.
   * If β_4 equals $p(v)$, we have identical machine sets. This means that the processing time of an operation does not depend on the machine set that processes it, but only on the operation itself.
   * If $\beta_4 \in \{p(H), p(|H|), 1\}$, we have uniform machine sets, identical machines, or identical machine sets, respectively. But now in addition the processing requirement of each operation is 1. Hence, the processing time of an operation only depends on the machine set it is processed by, and not on the operation itself.

2.2.3 Objective characteristics

The third field $\gamma = \gamma_1\gamma_2$ specifies the characteristics of the objective.

1. γ_1 denotes the objective that is to be minimized. In our case this is always the makespan $C_{\max}$.

2. $\gamma_2 \in \{\circ, \leq x\}$ with $x \in \mathbb{N}$.

   If γ_2 equals $\leq x$ for some $x \in \mathbb{N}$, then x is a required upper bound on the objective, and is given as part of the problem type. In this case the given problem is a decision problem. If γ_2 equals $\circ$, no bound on the objective is required, in which case the given problem is an optimization problem.

2.3 Shop scheduling classification

In this section we define the classification of some well-known shop scheduling problems. First, we deal with the job shop and flow shop scheduling problem. Next, we discuss the open shop scheduling problem.

2.3.1 Job shop and flow shop scheduling

In a job and a flow shop problem there is exactly one possible machine set for each operation. In addition, this machine set has size one. Furthermore, each job consists of a chain of operations and for each pair of successive operations of a job it is required that they use different machines. So, in our terminology, the job shop scheduling problem is denoted by $J \bullet 1\ 1|\ \text{chain}\ |C_{\max}$. The job shop scheduling problem in which two successive operations may be processed by the same machine is denoted by $\bullet\ 1\ 1|\ \text{chain}\ |C_{\max}$.

For the flow shop problem, besides the requirements posed for the job shop scheduling problem, some additional requirements have to be satisfied: the operations of each job need to be processed on the machines in the same order and each job needs to be processed on each machine exactly once. This problem is denoted by $F \bullet 1\ 1|\ \text{chain}\ |C_{\max}$. If in addition it is required that each machine processes the jobs in the same order, then the problem is a permutation flow shop scheduling problem. This problem is denoted by $F_\pi \bullet 1\ 1|\ \text{chain}\ |C_{\max}$.

Note that, if $\alpha_1 \in \{J, F, F_\pi\}$, we always have that $\alpha_3 = \alpha_4 = 1$ and that $\beta_1 = \text{chain}$. Hence, not all elements of our classification can be chosen independently from their given sets of alternatives. We do not want to eliminate this redundancy, however, since we wish to keep the classification sufficiently close to the one of Graham, Lawler, Lenstra and Rinnooy Kan [1979]. This kind of redundancy occurs also in the definition of the open shop scheduling problem, which is subject of the next subsection.

2.3.2 Open shop scheduling

In the open shop scheduling problem each operation v has a unique machine $\mu(v)$ on which it must be processed. Furthermore, the set of operations is partitioned into parts. The set of all parts P is denoted by $\mathcal{P}$ and, for each $v \in V$, the part to which v belongs is denoted by $P(v)$. The partition is such that for each part P and each machine μ there is exactly one operation in part P that has to be processed by machine μ. Any two operations belonging to the same part are not allowed to be processed simultaneously. There is no precedence relation given between the operations.

In the following we show that the open shop problem can be incorporated in our model by considering it as a special case of $\bullet$ 2 1| empty $|C_{\max}$. For this purpose we define, besides the given set of machines, for each part $P \in \mathcal{P}$ an additional machine $\mu(P)$. Each operation $v \in V$ can be processed on only one machine set. This unique machine set has size two and consists of the original machine $\mu(v)$ on which operation v should have processed, and the additional machine $\mu(P(v))$ for the part to which v belongs. Therefore, we have $\mathcal{H}(v) = \{\{\mu(v), \mu(P(v))\}\}$. Clearly, the additional machines enforce that no two operations of the same part can be executed simultaneously. Therefore, each solution of this extended problem can be transformed into a solution for the original open shop problem by just disregarding the additional machines.

As a consequence, the open shop scheduling problem is denoted by the specification $O \bullet$ 2 1| empty $|C_{\max}$.

3

Computational complexity of subproblems

The generalized job shop scheduling introduced in the previous chapter is NP-hard in the strong sense, since it contains the job shop scheduling problem as a subproblem, which is already NP-hard in the strong sense [Garey, Johnson and Sethi, 1976].

In this chapter we investigate the computational complexity of some subproblems of the generalized job shop scheduling problem. We consider several types of subproblems, each of which is the subject of one section.

3.1 Fixed lengths

We discuss the decision problem of whether there exists a schedule of a fixed length, which is small in the majority of the cases considered. We consider several subproblems, some of which are proven to be solvable in polynomial time, and some of which are proven to be NP-complete.

The main interest for considering decision problems for which a fixed, small upper bound on the length is given, lies in the fact that, if the decision problem with a fixed upper bound L on the length is NP-complete, then finding a solution for the corresponding optimization problem of length smaller than $(L+1)/L$ times the optimum is NP-hard [Lenstra and Shmoys, 1995].

The following indicates why we do not consider problems with a fixed upper

bound that have a fixed number of machines. If such a problem has fixed bound L on the length and a fixed number m of machines, then there can be at most mL operations, since otherwise no schedule of length at most L can exist. Hence, the number of operations is bounded from above by a constant. Since also the number of possible machine sets and their sizes are bounded from above by a constant, the problem is solvable in constant time. Therefore, we assume that the number of machines is not fixed when considering problems with a fixed bound L on the length.

We first discuss a subproblem that has a fixed, but arbitrary upper bound on the length of the schedule; then we discuss subproblems with fixed upper bound 1, 2, 3, and 4. Note that an NP-completeness proof for a subproblem of the latter type is in most cases easily generalized to the similar subproblem in which the bound on the length is one larger. On the other hand, a subproblem that is polynomial solvable for a given upper bound on the length may become NP-complete if this upper bound is enlarged by one. Therefore, we consider the subproblems with fixed upper bound in order of increasing deadline.

3.1.1 Arbitrary fixed length

We prove the following result.

Theorem 3.1. *For every constant L, the problem $\bullet 1 \bullet \mid 1 \mid C_{\max} \leq L$ with the extra condition that for every pair $(v, w) \in \tilde{A}$ at least one of the operations v and w belongs to a chain of length at least L, is solvable in polynomial time.*

Proof. We give a polynomial-time reduction from this problem to BIPARTITE MATCHING, which problem is known to belong to $\mathcal{P}$.

Suppose we are given an instance of the type mentioned above. We transform this instance into an instance of BIPARTITE MATCHING in such a way that there is a matching of a specified size if and only if there exists a feasible schedule of length at most L.

The transformation is defined as follows. If there exists a chain of length larger than L, then construct a trivial bipartite graph for which a matching is required of impossible size. Otherwise, the vertex set of the bipartite graph is given by $V \cup (M \times \{0, \ldots, L-1\})$. Let V_L denote the set of operations that belong to a chain of length L, and let, for any $v \in V_L$, $S(v)$ denote the (forced) start time of v. There is an edge $\{v, (\mu, t)\}$ $(v \in V, \mu \in M, t \in \{0, \ldots, L-1\})$ if and only if

$$\{\mu\} \in \mathcal{H}(v) \text{ and } \max_{u \in V_L, (u,v) \in A} S(u) < t < \min_{w \in V_L, (v,w) \in A} S(w).$$

Now a matching is required of size l. We claim that there is a matching of size l if and only if there exists a feasible schedule of length at most L.

Suppose that there is a schedule of length at most L. For this schedule we define a subset B of the edges in the bipartite graph as follows:

$$\{v, (\mu, t)\} \in B \text{ if and only if } K(v) = \{\mu\} \text{ and } S(v) = t.$$

Note that B indeed is a subset of the set of edges as defined above. We prove that B is a matching of size l in the constructed bipartite graph. Since each operation v is scheduled, we must have $\{v, (\mu, t)\} \in B$ for some machine μ and some start time t. Hence, the size of B is l. B is a matching since for each machine μ and for each time $t \in \{0, \ldots, L-1\}$ there is at most one operation v with $K(v) = \{\mu\}$ and $S(v) = t$.

Now suppose that there is a matching B of size l. For this matching we define a schedule as follows:

$$K(v) = \{\mu\} \text{ and } S(v) = t \text{ if and only if } \{v, (\mu, t)\} \in B.$$

We have to prove that this schedule is feasible. Clearly, for each machine $\mu \in M$ and each start time $t \in \{0, \ldots, L-1\}$ there is at most one operation v with $K(v) = \{\mu\}$ and $S(v) = t$. The edge set of the bipartite graph is defined in such a way that all precedences between operations are satisfied automatically. □

3.1.2 Length 1

We prove the following results:

* $\bullet\, 1 \,\bullet \mid\mid C_{\max} \leq 1$ belongs to $\mathcal{P}$;
* $\bullet\, \bullet\, 2 \mid\mid C_{\max} \leq 1$ belongs to $\mathcal{P}$;
* $\bullet\, 2\, 3 \mid\mid C_{\max} \leq 1$ is NP-complete, even if for each operation v the set $\mathcal{H}(v)$ contains at most one possible machine set of size two.

Since the first two problems are solvable in polynomial time, $\bullet\, 2\, 3 \mid\mid C_{\max} \leq 1$ is the most restricted one that has to be considered next. We show that this problem is already NP-complete, even under some stronger conditions. First, we deal with the two polynomially solvable problems.

Theorem 3.2. *The problem* $\bullet\, 1 \,\bullet \mid\mid C_{\max} \leq 1$ *belongs to* $\mathcal{P}$.

Proof. Suppose we are given an instance of the problem $\bullet\, 1 \,\bullet \mid\mid C_{\max} \leq 1$. If $A \neq \emptyset$ or if there exists an operation with processing time larger than 1 on each of its machine sets, then no schedule of length at most 1 is possible. Hence, we may assume that the given instance is an instance of $\bullet\, 1 \,\bullet \mid \text{empty}, 1 \mid C_{\max} \leq 1$. By Theorem 3.1 this problem belongs to $\mathcal{P}$. □

Theorem 3.3. *The problem* $\bullet\bullet 2 \mid\mid C_{\max} \leq 1$ *belongs to* $\mathcal{P}$.

Proof. We present a polynomial-time reduction from $\bullet\bullet 2 \mid\mid C_{\max} \leq 1$ to the problem 2SAT, which is known to belong to $\mathcal{P}$.

Suppose we are given an instance of $\bullet\bullet 2 \mid\mid C_{\max} \leq 1$. We transform this instance into an instance of 2SAT in such a way that the formula of 2SAT is satisfiable if and only if there exists a feasible schedule of length at most 1. Without loss of generality we assume that there are no precedences between operations and that each operation has processing time 1 on each of its machine sets. Furthermore, we assume that $|\mathcal{H}(v)|$ equals 2 for all $v \in V$. If there is an operation v with $|\mathcal{H}(v)| = 1$, it is forced to be processed by its unique machine set $H(v)$. In this case we can delete operation v from the instance and furthermore we can delete for all other operations $w \in V \setminus \{v\}$ those machine sets $H(w)$ from $\mathcal{H}(w)$ that have a non-empty intersection with $H(v)$. By this process of repeatedly removing operations and machine sets we finally arrive at an instance in which either an operation v exists with $\mathcal{H}(v) = \emptyset$ or in which $\mathcal{H}(v)$ contains two machine sets for all operations v. Clearly in the first case no feasible schedule of length 1 exists.

Now let for every operation $v \in V$ the machine sets of $\mathcal{H}(v)$ be given by $H_1(v)$ and $H_2(v)$. The transformation is defined as follows. The set of variables of the instance of 2SAT is taken equal to V. For every two operations $v, w \in V$ we take the following clauses:

$$[v \vee w] \quad \Leftrightarrow \quad H_2(v) \cap H_2(w) \neq \emptyset; \tag{3.1}$$

$$[v \vee \overline{w}] \quad \Leftrightarrow \quad H_2(v) \cap H_1(w) \neq \emptyset; \tag{3.2}$$

$$[\overline{v} \vee w] \quad \Leftrightarrow \quad H_1(v) \cap H_2(w) \neq \emptyset; \tag{3.3}$$

$$[\overline{v} \vee \overline{w}] \quad \Leftrightarrow \quad H_1(v) \cap H_1(w) \neq \emptyset. \tag{3.4}$$

Now suppose there is a feasible schedule with length equal to 1. We define a corresponding truth assignment t by letting $t(v) = \text{true}$ for every $v \in V$ if and only if v is processed on machine set $H_1(v)$. We have to show that this truth assignment satisfies all clauses. Assume we are given a $v \in V$ that is processed by $H_1(v)$. Then $t(v) = \text{true}$ and therefore all clauses of type (3.1) and (3.2) that correspond to v are satisfied. Now let $w \in V \setminus \{v\}$ be such that $H_1(v) \cap H_2(w) \neq \emptyset$. Since v is processed by $H_1(v)$ operation w must be processed by $H_1(w)$, which means that $t(w) = \text{true}$. Hence, the clause of type (3.3) that corresponds to v and w is satisfied. In a similar way one can prove that for every $w \in V \setminus \{v\}$ with $H_1(v) \cap H_1(w) \neq \emptyset$ the clause of type (3.4) corresponding to v and w is satisfied. Similarly, one can prove that all clauses are satisfied that correspond to an operation $v \in V$ that is processed by $H_2(v)$.

Now suppose that there is a satisfying truth assignment t. Then we define a machine set assignment by taking for all $v \in V$ $K(v) = H_1(v)$ if and only if $t(v) = \text{true}$. We have to prove that $K(v) \cap K(w) = \emptyset$ for all operations $v, w \in V$. Suppose that there exist two operations $v, w \in V$ for which $K(v) \cap K(w) \neq \emptyset$. Then one of the clauses of type (3.1), (3.2), (3.3), or (3.4) cannot be satisfied. □

To prove that the problem $\bullet\ 2\ 3\ ||\ C_{\max} \leq 1$ is NP-complete, even if $\mathcal{H}(v)$ contains at most one possible machine set of size 2 for all operations v, we give a reduction from a restricted version of SATISFIABILITY, in which each clause contains two or three literals and in which each literal occurs exactly twice in the formula. We call this problem 2-LITERAL-2,3SAT. The following lemma states that this problem is NP-complete.

Lemma 3.4. *The problem* 2-LITERAL-2,3-SAT *is NP-complete.*

Proof. We give a polynomial-time reduction to 2-LITERAL-2,3SAT from the NP-complete problem 3-BOUNDED-$\leq$3SAT [Garey and Johnson, 1979, p. 259]. 3-BOUNDED-$\leq$3SAT is a restricted version of SATISFIABILITY, in which each clause contains at most three literals and each variable occurs (negated or not) at most three times in the formula.

Suppose we are given an instance of the problem 3-BOUNDED-$\leq$3SAT. We transform this instance into an instance of 2-LITERAL-2,3SAT in such a way that 2-LITERAL-2,3SAT is satisfiable if and only if 3-BOUNDED-$\leq$3SAT is. The transformation is described by the following algorithm.

1. If each variable occurs at least once unnegated and at least once negated in the formula, go to step 2. Otherwise, there is variable w that occurs only unnegated or only negated (or not at all) in the formula. Delete from the formula all clauses in which w occurs and delete w from the variable set; apply step 1 again.
2. If each clause has at least two literals, go to step 3. Otherwise, consider a clause with one literal. If there is another clause of size one that contains the complementary literal, replace the current variable set and formula by a trivial unsatisfiable instance of 2-LITERAL-2,3SAT and end the construction. Otherwise, delete all clauses that contain the literal and delete the complementary literal from every clause in which it is contained; remove the corresponding variable from the variable set and return to step 1.
3. If each literal occurs exactly twice, then end the construction. Otherwise, consider a literal w that occurs only once; introduce a new variable u_w to the variable set and add the tautological clauses $[w \vee u_w \vee \overline{u_w}]$ and $[u_w \vee \overline{u_w}]$ to the formula; apply step 3 again.

It it clear that by each application of step 1, 2, or 3 the current formula is replaced by an equivalent one. Furthermore, in step 1 each variable is deleted that occurs three times negated or three times unnegated in the formula. After the last application of step 2, every clause has two or three literals. When the construction ends in step 3, each variable occurs exactly twice. Note that this also holds for the new variables introduced in this step. □

We are now able to prove the following theorem.

Theorem 3.5. *The problem* $\bullet\ 2\ 3\ ||\ C_{\max} \leq 1$ *is NP-complete, even if for each operation* v *the set* $\mathcal{H}(v)$ *contains at most one possible machine set of size 2.*

Proof. We give a polynomial-time reduction from 2-LITERAL-2,3SAT to this scheduling problem.

Suppose we are given an instance of the problem 2-LITERAL-2,3SAT. We transform this instance into an instance of the given scheduling problem in such a way that there exists a feasible schedule of length at most 1 if and only if the formula of 2-LITERAL-2,3SAT is satisfiable.

The transformation is defined as follows. Let $U = \{u_1, \ldots, u_m\}$ be the set of m variables of the instance of 2-LITERAL-2,3SAT and let $\mathcal{C} = \{C_1, \ldots, C_n\}$ be the collection of n clauses over U. Let $r_j \in \{2, 3\}$ be the number of literals in clause C_j, $j \in \{1, \ldots, n\}$, and let the c_{jk} denote the kth literal of clause C_j, $j \in \{1, \ldots, n\}$, $k \in \{1, \ldots, r_j\}$. For each clause C_j we define three operations x_{j1}, x_{j2}, and x_{j3}, which correspond to the literals in the clause (except perhaps operation x_{j3}). For each clause C_j we also define two clause machines C_j^1 and C_j^2, and for each variable u_i we define four assignment machines μ_i^l, $l \in \{1, \ldots, 4\}$. For each operation x_{jk}, $j \in \{1, \ldots, n\}$ and $k \in \{1, 2\}$, we let $\mathcal{H}(x_{jk})$ contain the machine set $\{C_j^k\}$. For each operation x_{j3}, $j \in \{1, \ldots, n\}$, we let $\mathcal{H}(x_{j3})$ contain the machine sets $\{C_j^1\}$ and $\{C_j^2\}$. Now let $i \in \{1, \ldots, m\}$. The literal u_i occurs twice in a clause, say $u_i = c_{j_1k_1} = c_{j_2k_2}$, with $j_1 < j_2$ or $j_1 = j_2$ and $k_1 < k_2$. We let $\mathcal{H}(x_{j_1k_1})$ also contain the machine set $\{\mu_i^1, \mu_i^2\}$ and $\mathcal{H}(x_{j_2k_2})$ the machine set $\{\mu_i^3, \mu_i^4\}$. Similarly, the literal $\overline{u_i}$ occurs twice in a clause, say $\overline{u_i} = c_{j_3k_3} = c_{j_4k_4}$, with $j_3 < j_4$ or $j_3 = j_4$ and $k_3 < k_4$. We let $\mathcal{H}(x_{j_3k_3})$ also contain the machine set $\{\mu_i^1, \mu_i^3\}$ and $\mathcal{H}(x_{j_4k_4})$ the machine set $\{\mu_i^2, \mu_i^4\}$. Obviously, all processing times are 1 and no precedences are given.

Now suppose that we are given a satisfying truth assignment for the constructed formula. Then we define a feasible schedule of length 1 as follows. Obviously, every operation has to start at time 0. For every operation x_{jk} that corresponds to a true literal c_{jk} we take $K(x_{jk})$ equal to its unique machine set that contains two of the four assignment machines. Note that this is possible,

since for each variable u_i the two operations corresponding to the literal u_i can be processed simultaneously on the assignment machines for variable u_i. The same argument holds for the two operations corresponding to $\overline{u_i}$. For all other operations x_{jk} we take $K(x_{jk})$ equal to one of the machine sets $\{C_j^1\}$ or $\{C_j^2\}$. Since for each clause C_j at least one operation is scheduled on the assignment machines, there are at most two operations x_{jk} that have to be scheduled on the clause machine $\{C_j^1\}$ and $\{C_j^2\}$. The construction of the sets $\mathcal{H}(x_{jk})$ is such that it is always possible to choose for these two operations two different machine sets that contain a clause machine.

Now suppose that we are given a feasible schedule of length 1. We have to define a truth assignment that satisfies all clauses. We assign each variable u_i the value true if an operation corresponding to literal u_i is processed on a pair of assignment machines, and false if an operation corresponding to literal $\overline{u_i}$ is processed on a pair of assignment machines. Note that by the construction of the sets $\mathcal{H}(x_{jk})$ it is impossible for a variable u_i that an operation corresponding to literal u_i and an operation corresponding to literal $\overline{u_i}$ are processed on a pair of assignment machines simultaneously. Therefore this truth assignment is consistent. Since we have three operations and only two clause machines for each clause, at least one of these operations must be processed by a pair of assignment machines. This also holds if the clause contains only two literals, since the operation that does not correspond to a literal in the clause must be processed on a clause machine. Therefore, each clause is satisfied by the above truth assignment. Note that it is possible that for a variable u_i none of its assignment machines processes an operation. In this case the value of u_i may be chosen arbitrarily, since all clauses are already satisfied. □

3.1.3 Length 2

Given the results for the subproblems with upper bound 1 the remaining problems of interest are

* $\bullet\, 1 \bullet \mid\mid C_{\max} \leq 2$ and
* $\bullet\bullet 2 \mid\mid C_{\max} \leq 2$.

In this subsection we show the following results:

* $\bullet\, 1 \bullet \mid 1 \mid C_{\max} \leq 2$ belongs to $\mathcal{P}$;
* $\bullet\bullet 1 \mid\mid C_{\max} \leq 2$ belongs to $\mathcal{P}$;
* $\bullet\, 1\, 2 \mid$ empty, $p(v) \mid C_{\max} \leq 2$ is NP-complete;
* $\bullet\, 2\, 2 \mid$ empty, $1 \mid C_{\max} \leq 2$ is NP-complete.

The second result shows that all problems with deadline 2 that have only one machine set per operation are solvable in polynomial time. Therefore, each NP-complete problem will have operations with more than one possible machine set. The first result shows that $\bullet\ 1\ \bullet\ ||\ C_{\max} \leq 2$ is solvable in polynomial time if all processing times are 1. The third result shows that the problem becomes NP-complete if we drop this restriction, even if each operation has at most two possible machines it can be processed on, if the processing time of each operation does not depend on the machine it is processed on, and if no precedences exist. On the other hand, if we drop the condition that machine sets have size one, the problem becomes NP-complete, even if each operation has at most two possible machine sets of size two and if no precedences exist. This is shown by the fourth result.

Furthermore, note that the two problems that are obtained from the two NP-complete problems mentioned above by replacing the empty precedence structure by an arbitrary one are less restricted and therefore remain NP-complete.

The first result follows immediately from Theorem 3.1. The other results are subject of the following theorems.

Theorem 3.6. *The problem* $\bullet\ \bullet\ 1\ ||\ C_{\max} \leq 2$ *belongs to* $\mathcal{P}$.

Proof. It is quite straightforward to solve this problem in polynomial time. Let an instance of the problem be given. If there is chain of operations (possibly consisting of a single operation) for which the sum of the processing times is larger than 2, then no schedule of length 2 can exist. So, we may assume that for each chain of operations the sum of the processing times is at most 2. We now briefly describe the construction of a schedule of length at most 2. First, schedule all operations that are contained in a chain for which the sum of the processing times equals 2. Clearly, for each such operation there is only one start time possible. If it is not possible to schedule all these operations, then a schedule of length at most 2 does not exist for the given instance. Otherwise, consider the remaining operations with processing time 1 for which no precedences exist. As long as there is an operation for which at least one machine in its machine set is already occupied by a scheduled operation, try to schedule such an operation v. Clearly, at most one possible start time is left for v. If no start time is left, no schedule of length 2 is possible. Otherwise, schedule v on the unique interval that is still open.

Finally, we obtain a situation in which all unscheduled operations have processing time 1 and their machine sets have no machine in common with the machine set of any scheduled operation. Now we can schedule an arbitrary unscheduled operation without loss of generality at start time 0. Again, as long as there

is an operation for which at least one machine in its machine set is already occupied by a scheduled operation, try to schedule such an operation in the way described above. Such a situation, in which no operation is forced to be scheduled at a certain start time, may occur several times; each time an arbitrary operation is chosen that starts at time 0.

Clearly, the above method is correct and an algorithm can be designed that runs in polynomial time. Note that this method solves in essence the GRAPH 2-COLORABILITY PROBLEM, which problem is known to be solvable in polynomial time. □

Theorem 3.7. • 1 2 | empty, $p(v)$ | $C_{\max} \leq 2$ *is NP-complete.*

Proof. We give a polynomial-time reduction from 2-LITERAL-2,3SAT to this scheduling problem.

Let an instance of the problem 2-LITERAL-2,3SAT be given as defined in the proof of Theorem 3.5. We transform this instance into an instance of the given scheduling problem in such a way that there exists a feasible schedule of length at most 2 if and only if the formula of 2-LITERAL-2,3SAT is satisfiable.

For each clause C_j with $r_j = 2$ we define one operation x_j. For each clause C_j with $r_j = 3$ we define three operations x_{j1}, x_{j2}, and x_{j3}, which correspond to the three literals in the clause. Furthermore, for each variable $u \in U$ we define an operation u. For each clause C_j with $r_j = 3$ we define a clause machine μ_j. For each variable u we define two assignment machines u and $\bar{u}$. Now the machine sets by which the operations can be processed are defined as follows:

$$\begin{aligned}
&\mathcal{H}(x_j) = \{\{c_{j1}\}, \{c_{j2}\}\} \text{ and } p(x_j) = 1 && \text{if } C_j \in C \text{ and } r_j = 2;\\
&\mathcal{H}(x_{jk}) = \{\{c_{jk}\}, \{\mu_j\}\} \text{ and } p(x_{jk}) = 1 && \text{if } C_j \in C \text{ and } r_j = 3\\
&&& \quad \text{and } k \in \{1, 2, 3\};\\
&\mathcal{H}(u) = \{\{u\}, \{\bar{u}\}\} \text{ and } p(u) = 2 && \text{if } u \in U.
\end{aligned}$$

Now suppose that a satisfying truth assignment $t : U \to \{\text{true, false}\}$ for the constructed formula exists. For each clause C_j there is at least one literal c_{jk} with $t(c_{jk}) = \text{true}$. Choose for each clause C_j such a literal, say c_{jk^j}. We now define the machine set assignment K by:

$$\begin{aligned}
&K(x_j) = \{c_{jk^j}\} && \text{if } r_j = 2;\\
&K(x_{jk}) = \{c_{jk}\} && \text{if } r_j = 3 \text{ and } k = k^j;\\
&K(x_{jk}) = \{\mu_j\} && \text{if } r_j = 3 \text{ and } k \neq k^j;\\
&K(u) = \{\bar{u}\} && \text{if } t(u) = \text{true};\\
&K(u) = \{u\} && \text{if } t(u) = \text{false}.
\end{aligned}$$

Since every literal occurs in exactly two clauses, this machine set assignment is such that each machine has to process one operation of processing time 2 or at

most two operations of processing time 1. It is trivial to define the start times for the operations such that a feasible schedule of length 2 is obtained.

Now suppose that a feasible schedule of length at most 2 exists. If the length is smaller than 2, the set U is empty and the corresponding empty formula is satisfied. Otherwise, we define a truth assignment t by defining for all $u \in U$:

$$t(u) = \text{true} \Leftrightarrow K(u) = \{\overline{u}\}.$$

We have to show that this truth assignment satisfies all clauses. If $r_j = 2$, we have either $K(x_j) = \{c_{j1}\}$ or $K(x_j) = \{c_{j2}\}$. Without loss of generality assume that $K(x_j) = \{c_{j1}\}$. If $c_{j1} = u$ for some variable u, then we must have $K(u) = \{\overline{u}\}$ and thus $t(c_{j1}) = t(u) = \text{true}$. If $c_{j1} = \overline{u}$ for some variable u, then we must have $K(u) = \{u\}$; then $t(u) = \text{false}$ and thus $t(c_{j1}) = \text{true}$. Hence, in both cases the clause C_j is satisfied. If $r_j = 3$, there is at least one index $k \in \{1, 2, 3\}$ for which $K(x_{jk}) = \{c_{jk}\}$, since otherwise machine μ_j would be busy for 3 time units. Let k^j be such an index. In a similar way as for the case $r_j = 2$ it can be shown that we must have $t(c_{jk^j}) = \text{true}$. Hence clause C_j is satisfied. □

Theorem 3.8. *The problem* $\bullet\, 2\, 2 \mid \text{empty}, 1 \mid C_{\max} \leq 2$ *is NP-complete.*

Proof. We give a polynomial-time reduction from 2-LITERAL-2,3SAT to this scheduling problem.

Let an instance of the problem 2-LITERAL-2,3SAT be given as defined in the proof of Theorem 3.5. We transform this instance into an instance of the given scheduling problem as follows.

For each clause C_j we define three operations x_{j1}, x_{j2}, and x_{j3}, which correspond to the literals in the clause (except perhaps operation x_{j3}). Furthermore, for each variable u_i we define two operations v_{i1} and v_{i2}. For each clause C_j we define a clause machines C_j and for each variable u_i we define four assignment machines μ_i^l, $l \in \{1, \ldots, 4\}$. For each operation x_{jk}, $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, 3\}$, we let $\mathcal{H}(x_{jk})$ contain the machine set $\{C_j\}$. Each literal u_i, $i \in \{1, \ldots, m\}$, occurs twice in a clause, say $u_i = c_{j_1 k_1} = c_{j_2 k_2}$, with $j_1 < j_2$ or $j_1 = j_2$ and $k_1 < k_2$. We let $\mathcal{H}(x_{j_1 k_1})$ also contain the machine set $\{\mu_i^1, \mu_i^2\}$ and $\mathcal{H}(x_{j_2 k_2})$ the machine set $\{\mu_i^3, \mu_i^4\}$. Similarly, the literal $\overline{u_i}$ occurs twice in a clause, say $\overline{u_i} = c_{j_3 k_3} = c_{j_4 k_4}$, with $j_3 < j_4$ or $j_3 = j_4$ and $k_3 < k_4$. We let $\mathcal{H}(x_{j_3 k_3})$ also contain the machine set $\{\mu_i^1, \mu_i^3\}$ and $\mathcal{H}(x_{j_4 k_4})$ the machine set $\{\mu_i^2, \mu_i^4\}$. Finally, we take $\mathcal{H}(v_{i1}) = \{\{\mu_i^1, \mu_i^4\}\}$ and $\mathcal{H}(v_{i2}) = \{\{\mu_i^2, \mu_i^3\}\}$, $i \in \{1, \ldots, m\}$.

Now suppose that a satisfying truth assignment $t : U \to \{\text{true}, \text{false}\}$ for the constructed formula exists. We define a machine set assignment K and a start time assignment S as follows. Obviously, we have $K(v_{i1}) = \{\mu_i^1, \mu_i^4\}$ and $K(v_{i2}) = \{\mu_i^2, \mu_i^3\}$ for all $i \in \{1, \ldots, m\}$. We take the start time for these opera-

tions equal to 0. For the other operations we define K by:

$K(x_{jk}) = \{C_j\}$ if and only if $k > r_j$ or $t(c_{jk}) = \text{false}$.

Since for every clause C_j there is at least one true literal, the clause machine C_j has to process at most two operations. Their start times are chosen from $\{0, 1\}$ such that they are not equal. Now for every operation x_{jk} that corresponds to a literal c_{jk} with $t(c_{jk}) = \text{true}$ we take $K(x_{jk})$ equal to its unique machine set that contains two of the four assignment machines, and we take its start time equal to 1. Note that this is possible, since for each variable u_i the two operations corresponding to the literal u_i can be processed simultaneously on the assignment machines for variable u_i. The same argument holds for the two operations corresponding to $\overline{u_i}$. This schedule can be easily checked to be feasible.

Now suppose that there exists a feasible schedule of length at most 2. We have to define a truth assignment t that satisfies all clauses. Note that for every $i \in \{1, \ldots, m\}$ the operations v_{i1} and v_{i2} occupy each machine μ_i^l for one time unit ($l \in \{1, \ldots, 4\}$). Therefore, an operation corresponding to literal u_i and an operation corresponding to literal $\overline{u_i}$ cannot be processed both on their own pair of assignment machines. Now for each variable u_i we define $t(u_i) = \text{true}$ if an operation corresponding to literal u_i is processed on a pair of assignment machines, and $t(u_i) = \text{false}$ if an operation corresponding to literal $\overline{u_i}$ is processed on a pair of assignment machines. Since for each clause at least one operation is processed by a pair of assignment machines, each clause is satisfied by truth assignment t. If for a variable u_i its assignment machines process only the operations v_{i1} and v_{i2}, $t(u_i)$ is chosen arbitrarily. □

3.1.4 Length 3

Given the results for the subproblems with upper bound 2 the interesting problems with upper bound 3 are

* $\bullet\, 1\, \bullet \mid 1 \mid C_{\max} \leq 3$ and
* $\bullet\, \bullet\, 1 \mid\mid C_{\max} \leq 3$.

In this subsection we show the following results:

* $\bullet\, 1\, 1 \mid\mid C_{\max} \leq 3$ belongs to $\mathcal{P}$;
* $\bullet\, 2\, 1 \mid$ empty, $1 \mid C_{\max} \leq 3$ is NP-complete;
* $\bullet\, 1\, 2 \mid$ chain, $1 \mid C_{\max} \leq 3$ is NP-complete.

The first result shows that $\bullet\ \bullet\ 1 \mid\mid C_{\max} \leq 3$ is solvable in polynomial time if all machine sets have size one. The second result shows that, in case machine sets may have size two, the problem becomes NP-complete, even if no precedences are allowed and all processing times are 1. The third result shows that

the problem $\bullet\ 1\ \bullet\ |\ 1\ |\ C_{\max} \leq 3$ becomes NP-complete if operations may have two machines on which they can be processed and if in addition arbitrary chain precedences are allowed. Note that the problem becomes solvable in polynomial time again, if we replace these arbitrary chains by a precedence structure that satisfies the extra condition of Theorem 3.1. Therefore, in the NP-completeness proof for the third result chains have to be defined that do not satisfy this additional condition.

The second result was obtained by Krawczyk and Kubale [1985] (see also Kubale [1987], Hoogeveen, Van de Velde and Veltman [1994]) by a reduction from the EDGE 3-COLORING problem, which has been proven NP-complete by Holyer [1981]. The other results are dealt with in the following two theorems.

Theorem 3.9. *The problem* $\bullet\ 1\ 1\ ||\ C_{\max} \leq 3$ *belongs to* $\mathcal{P}$.

Proof. We present a method to determine whether for a given instance a schedule of length at most 3 exists. The method is based on the one for the more restricted problem $J\ \bullet\ 1\ 1\ |\ \text{chain}\ |\ C_{\max} \leq 3$ given by Williamson, Hall, Hoogeveen, Hurkens, Lenstra, Sevast'janov and Shmoys [1996]. It consists of two parts. First, it is determined whether a schedule of length at most 3 exists for the subset of operations that have processing time more than 1 or that take part in a precedence. Next, it is investigated whether the remaining operations, which have processing time 1 and do not take part in a precedence, can be added to this schedule without exceeding length 3. This is possible if and only if the total processing requirement of each machine is at most 3.

The first part is solved by giving a polynomial-time reduction to 2SAT, which problem is known to belong to $\mathcal{P}$. We transform the restricted instance, which only consists of the operations that have processing time more than 1 or that take part in a precedence, into an instance of 2SAT in such a way that the formula of 2SAT is satisfiable if and only if there exists a feasible schedule of length at most 3 for this restricted instance. Note that every operation of this restricted instance has at most two possible starting times.

Now the transformation is defined as follows. The set of variables of the instance of 2SAT consists of elements $x(v, k)$ with $v \in V$ and $k \in \{0, 1, 2\}$. The interpretation of $x(v, k) = \text{true}$ is that operation v starts at time k. Now the corresponding formula F of 2SAT is defined by the following algorithm.

1. Check whether there exists a sequence $v_1, \ldots, v_l$ of operations for some l with $(v_i, v_{i+1}) \in A$ for all $i < l$ and

$$\sum_{i=1}^{l} p(v_i) > 3.$$

If the answer is affirmative, no feasible schedule of length at most 3 is possible. Take F equal to a trivial unsatisfiable formula and stop. Otherwise, take for F the empty formula and go to step 2.

2. Check whether there exists a sequence $v_1, \ldots, v_l$ of operations for some l with $(v_i, v_{i+1}) \in A$ for all $i < l$ and

$$\sum_{i=1}^{l} p(v_i) = 3.$$

If the answer is affirmative, then for each $j \in \{1, \ldots, l\}$ the starting time $S(v_j)$ of operation v_j must be equal to

$$\sum_{i=1}^{j-1} p(v_i).$$

Add to F for each $j \in \{1, \ldots, l\}$ the singleton clauses $[x(v_j, S(v_j))]$ and $[\overline{x}(v_j, k)]$ for each $k \neq S(v_j)$ and apply step 2 again. Otherwise, go to step 3.

3. Check whether there exists a sequence $v_1, \ldots, v_l$ of operations for some l with $(v_i, v_{i+1}) \in A$ for all $i < l$ and

$$\sum_{i=1}^{l} p(v_i) = 2.$$

If the answer is affirmative, then for each $j \in \{1, \ldots, l\}$ the starting time $S(v_j)$ of operation v_j must be equal to $T(v_j)$ or $1 + T(v_j)$, where

$$T(v_j) = \sum_{i=1}^{j-1} p(v_i).$$

Now add to F a clause $[x(v_j, T(v_j)) \vee x(v_j, 1 + T(v_j))]$ and also a clause $[\overline{x}(v_j, T(v_j)) \vee \overline{x}(v_j, 1 + T(v_j))]$ for each $j \in \{1, \ldots, l\}$, and a clause $[\overline{x}(v_j, k)]$ for each $j \in \{1, \ldots, l\}$ and $k \notin \{T(v_j), 1 + T(v_j)\}$. Furthermore, add to F a clause $[x(v_j, T(v_j)) \vee x(v_{j+1}, 1 + T(v_{j+1}))]$ for each $j \in \{1, \ldots, l - 1\}$ to ensure that two operations v_j and v_{j+1} do not overlap in time. Apply step 3 again. Otherwise go to step 4.

4. Next, clauses have to be added to ensure that each machine processes at most one operation at a time. If operations v and w have to be processed on the same machine, their starting times have to satisfy $S(v) + p(v) \leq S(w)$ or $S(w) + p(w) \leq S(v)$. Thus, $-p(v) < S(v) - S(w) < p(w)$ cannot be valid. Therefore, for any two operations v and w that have to be processed by the same machine, add to F for every $k, k' \in \{0, 1, 2\}$ with $-p(v) < k - k' < p(w)$ the clause $[\overline{x}(v, k) \vee \overline{x}(w, k')]$. Go to step 5.

5. Finally, replace every singleton clause by a clause in which the element of the singleton clause occurs twice.

Now suppose that for the restricted instance a feasible schedule exists with length at most 3. We define a truth assignment t by letting $t(x(v, k)) = \text{true}$ if and only if v starts at time k. It follows immediately that each clause of F is satisfied.

Now suppose that a satisfying truth assignment t exists for F. Then, for each operation $v \in V$, we take its starting time $S(v)$ equal to k if and only if $t(x(v, k)) = \text{true}$. We have to show that this definition defines a feasible schedule. The clauses defined in steps 2 and 3 ensure that for each operation $v \in V$ at most one of $x(v, 0)$, $x(v, 1)$, and $x(v, 2)$ is true. Furthermore, these clauses ensure that each pair of operations for which a precedence is defined, are processed in the correct order and without overlap in time. The clauses defined in step 4 ensure that each machine processes at most one operation at a time. □

Theorem 3.10. *The problem* $\bullet\ 1\ 2 \mid \textit{chain}\ 1 \mid C_{\max} \leq 3$ *is NP-complete.*

Proof. We give a polynomial-time reduction from 2-LITERAL-2,3SAT to this scheduling problem.

Let an instance of the problem 2-LITERAL-2,3SAT be given as defined in the proof of Theorem 3.5. We transform this instance into an instance of the given scheduling problem as follows.

For each clause C_j we define a clause machine C_j and four operations $x_{j,k}$, $k \in \{1, \ldots, 4\}$. For $k \in \{1, \ldots, r_j\}$ the operations $x_{j,k}$ correspond to the literals $c_{j,k}$ in clause C_j. Furthermore, for each variable u_i we define eighteen assignment operations $v^i_{k,l}$, $k \in \{1, 2\}$ and $l \in \{1, \ldots, 9\}$, and eight assignment machines $\mu^i_{k,l}$, $k \in \{1, 2\}$ and $l \in \{1, \ldots, 4\}$. For each $i \in \{1, \ldots, m\}$ and $k \in \{1, 2\}$ we define the following precedences: $(v^i_{k,1}, v^i_{k,2})$, $(v^i_{k,3}, v^i_{k,4})$, $(v^i_{k,5}, v^i_{k,6})$, and $(v^i_{k,6}, v^i_{k,7})$.

For each $i \in \{1, \ldots, m\}$ and $k \in \{1, 2\}$ we define the possible machine sets for the operations $v^i_{k,l}$ as follows:

$$\begin{aligned}
\mathcal{H}(v^i_{k,1}) &= \{\{\mu^i_{k,1}\}\}, & \mathcal{H}(v^i_{k,5}) &= \{\{\mu^i_{k,4}\}\},\\
\mathcal{H}(v^i_{k,2}) &= \{\{\mu^i_{k,2}\}\}, & \mathcal{H}(v^i_{k,6}) &= \{\{\mu^i_{k,4}\}\},\\
\mathcal{H}(v^i_{k,3}) &= \{\{\mu^i_{k,1}\}\}, & \mathcal{H}(v^i_{k,8}) &= \{\{\mu^i_{k,2}\}\},\\
\mathcal{H}(v^i_{k,4}) &= \{\{\mu^i_{k,3}\}\}, & \mathcal{H}(v^i_{k,9}) &= \{\{\mu^i_{k,3}\}\};
\end{aligned}$$

for each $i \in \{1, \ldots, m\}$ we define

$$\mathcal{H}(v^i_{1,7}) = \{\{\mu^i_{1,2}\}, \{\mu^i_{2,3}\}\} \text{ and } \mathcal{H}(v^i_{2,7}) = \{\{\mu^i_{1,3}\}, \{\mu^i_{2,2}\}\}.$$

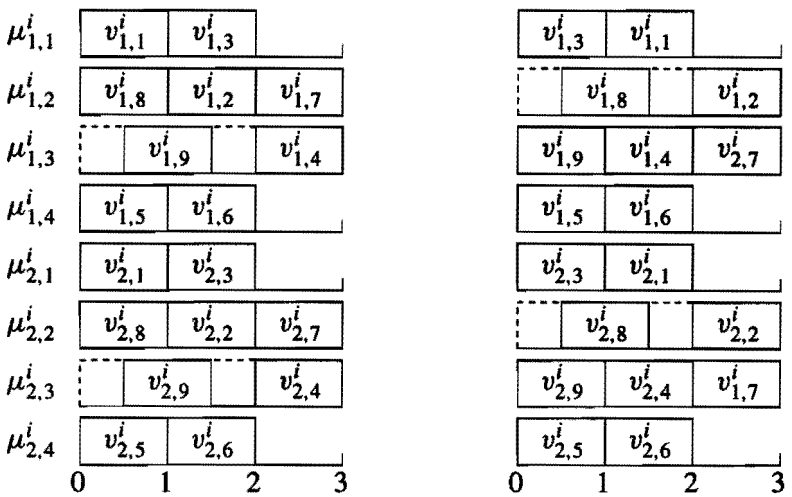


Figure 3.1: Two essentially different schedules for assignment operations.

For each operation $x_{j,k}$, $j \in \{1, \ldots, n\}$ and $k \in \{1, \ldots, 4\}$, we let $\mathcal{H}(x_{j,k})$ contain the machine set $\{C_j\}$. Each literal u_i, $i \in \{1, \ldots, m\}$, occurs twice in a clause, say $u_i = c_{j_1,k_1} = c_{j_2,k_2}$, with $j_1 < j_2$ or $j_1 = j_2$ and $k_1 < k_2$. Now we let $\mathcal{H}(x_{j_1,k_1})$ also contain the machine set $\{\mu^i_{1,3}\}$ and $\mathcal{H}(x_{j_2,k_2})$ the machine set $\{\mu^i_{2,3}\}$. Similarly, each literal $\overline{u_i}$ occurs twice in a clause, say $\overline{u_i} = c_{j_3,k_3} = c_{j_4,k_4}$, with $j_3 < j_4$ or $j_3 = j_4$ and $k_3 < k_4$. Now we let $\mathcal{H}(x_{j_3,k_3})$ also contain the machine set $\{\mu^i_{1,2}\}$ and $\mathcal{H}(x_{j_4,k_4})$ the machine set $\{\mu^i_{2,2}\}$.

Now suppose that a feasible schedule of length at most 3 exists. We have to define a truth assignment t that satisfies all clauses. For each $i \in \{1, \ldots, m\}$ there are two essentially different ways for scheduling the operations $v^i_{k,l}$; these are given by Figure 3.1. In the left schedule $v^i_{1,9}$ and $v^i_{2,9}$ are the only operations for which the start times are not fixed. But they can be chosen such that one additional operation can be scheduled on each of the machines $\mu^i_{1,3}$ and $\mu^i_{2,3}$. A similar statement holds for the operations $v^i_{1,8}$ and $v^i_{2,8}$ in the right schedule. Now for each variable u_i we define $t(u_i) =$ true if the assignment operations $v^i_{k,l}$ are scheduled as in the left schedule and $t(u_i) =$ false otherwise. We have to show that this truth assignment t satisfies all clauses. For each clause C_j there are four variables that may be scheduled on clause machine C_j. Since the given schedule has length 3, at least one operation is scheduled on an assignment machine. Let $x_{j,k}$ be such an operation. If it corresponds to an unnegated variable, say u_i, it is scheduled on machine $\mu^i_{1,3}$ or $\mu^i_{2,3}$, and hence the assignment operations for variable u_i must be scheduled as in the left schedule. Therefore, the clause is

satisfied. A similar argument holds in case an operation $x_{j,k}$ that is not scheduled on the clause machine C_j corresponds to a negated variable.

Now suppose that a satisfying truth assignment $t : U \rightarrow$ {true, false} for the constructed formula exists. It should be clear how a feasible schedule of length 3 has to be constructed. If $t(u_i) =$ true, the assignment operations corresponding to variable u_i are scheduled as in the left schedule; otherwise, they are scheduled as in the right schedule. All operations $x_{j,k}$ with $k \leq r_j$ for which $t(c_{j,k}) =$ true are scheduled on their assignment machine, and all remaining operations $x_{j,k}$ are scheduled on their clause machine. Since each clause C_j has at least one true literal, at most three operations remain to be scheduled on the clause machine C_j. Hence the length of the schedule is 3. The feasibility of this schedule can be easily checked. □

3.1.5 Length 4

Given the results for the subproblems with upper bound 3 the only remaining problem of interest with upper bound 4 is

$\bullet\ 1\ 1\ ||\ C_{\max} \leq 4.$

We have the following results:

* $\bullet\ 1\ 1$ | empty | $C_{\max} \leq 4$ belongs to $\mathcal{P}$;
* $J \bullet\ 1\ 1$ | chain, 1 | $C_{\max} \leq 4$ is NP-complete.

The first result is trivial: just check whether there is a machine for which the sum of the processing times of all operations that it has to process is larger than 4. Williamson, Hall, Hoogeveen, Hurkens, Lenstra, Sevast'janov and Shmoys [1996] show that the problem becomes NP-complete if the empty precedence structure is replaced by chains, even in the case that all processing times are 1 and the chains satisfy the requirement for the job shop scheduling problem. Clearly, this result also implies that $\bullet 1\ 1$ | chain, 1 | $C_{\max} \leq 4$ is NP-complete. Note however that this problem becomes solvable in polynomial time again if the precedence structure satisfies the extra condition of Theorem 3.1.

3.2 Unspecified length

For the subproblems discussed in this section, no fixed upper bound on the schedule length is given. Here, the subproblems are characterized by fixing an upper bound on the number of machines, the size of the machine sets, or the number of possible machine sets. In case precedences occur, also an upper bound on the number of jobs may be fixed. Furthermore, subproblems are considered for

which the processing time does not depend on the machine set it is processed on, or for which the processing time of each operation equals 1.

For each subproblem mentioned we indicate whether it is solvable in polynomial time or NP-hard. In the latter case we sometimes also mention when the problem is solvable in pseudo-polynomial time. For most of these results we give a reference where such an algorithm or proof can be found in the literature. We prove the remaining results.

First, we discuss subproblems in which no precedences occur. Next, we discuss subproblems in which the precedence graph consists of chains. We do not discuss other special types of precedence structures. Note however, that problems with intrees, with outtrees, or with a general precedence structure are harder than the corresponding problems without precedences or with chain precedences.

3.2.1 No precedences

Based on the results for the problems studied in the previous subsection we now study problems without a fixed bound on the length. The only problems of the previous subsection that proved to be solvable in polynomial time for each fixed upper bound are the problem of Theorem 3.1 and the problem

- $1\,|1\,|\,\text{empty}\,|\,C_{\max}$.

Clearly, the requirement for the precedence relation in the first problem depends in general on the fixed deadline. This requirement can only be satisfied for problems with arbitrary deadline in the case that no precedences exist. Hence, the problem

- $1\,|\bullet\,|\,\text{empty}, 1\,|\,C_{\max}$

is also still of interest. Both problems are solvable in polynomial time. For the latter problem this is shown in the following theorem.

Theorem 3.11. *The problem* $1\,|\bullet\,|\,\text{empty}, 1\,|\,C_{\max}$ *is solvable in polynomial time.*

Proof. The problem can be solved by constructing for a given instance several instances of the MAXIMUM-FLOW PROBLEM, in which a maximum flow through a network has to be determined. Note that the length of an optimal schedule is at most l, the number of operations. Now for each possible length L smaller than l an instance can be constructed as follows. There is a node for each operation and for each machine; furthermore, there is a source node and a sink node. An arc of capacity 1 is given between the source and each operation node; an arc of capacity 1 is given between each operation node and each machine node when the corresponding machine defines a machine set for the corresponding operation; an arc of capacity L is given between each machine node and the sink node. Now

the problem is to find the smallest L for which the constructed network allows a flow of size l. Each of these flow problems is solvable in a time that is polynomial in the number of nodes of the network. Since the number of nodes in each of the networks equals $l+m+2$, and at most $\log l$ different networks have to be constructed, the total running time is polynomial in l and m. Clearly, the size of an instance is at least $\min\{l, m\}$. □

The second problem, in which each operation has to be processed by a unique single machine, is trivial: just determine for each machine the sum of the lengths of all operations that must be processed on it; the optimal length equals the largest of these sums.

As a consequence of these results only problems with a fixed number of machines remain to be considered. The problem $\bullet\ 1\ \bullet$ | empty, 1 | $C_{\max}$ can only be made harder by allowing processing times larger than 1, by relaxing the constraint on the size of the machine sets, or by allowing more general precedences. Generalizations of the latter type are dealt with in the next subsection. Due to the fact that the problem $\bullet\ 1\ 1$ | empty | $C_{\max}$ is solvable in polynomial time,

$$2\ 1\ 2 \mid \text{empty}, p(v) \mid C_{\max}$$

is the minimal interesting problem with a fixed number of machines, with machine set size one, and with processing times that are not equal to 1 for all operations. This problem is already NP-hard since it contains PARTITION as a special case [Garey and Johnson, 1979]. However, it is generally known that the more general problem

$$m\ 1\ \bullet \mid \text{empty}, p(v) \mid C_{\max}$$

is solvable in pseudo-polynomial time.

We now consider relaxations of $\bullet 1 \bullet$ | empty, 1 | $C_{\max}$ in which machine sets may have sizes larger than one. For these type of problems we have the following results:

* $2\ 2\ 1$ | empty | $C_{\max}$ is solvable in polynomial time;
* $3\ 2\ 1$ | empty | $C_{\max}$ is strongly NP-hard;
* $m\ \bullet\ 1$ | empty, 1 | $C_{\max}$ is solvable in polynomial time.

It is trivial to solve the first problem: first schedule all operations that need both machines simultaneously, and then schedule all remaining operations. But with three machines instead of two this problem becomes strongly NP-hard. However, when all processing times are 1, then the problem becomes polynomially solvable again, even if a larger fixed number of machines and machine sets of arbitrary size are allowed. The second result has been proven by Blazewicz, Dell'Olmo, Drozdowski and Speranza [1992] and by Hoogeveen, Van de Velde

and Veltman [1994] by a reduction from 3-PARTITION. The third result has been obtained by Blazewicz, Drozdowski and Weglarz [1986] and by Hoogeveen, Van de Velde and Veltman [1994].

3.2.2 Chain precedences

We first consider problems in which each operation can be processed by exactly one machine set, and in which the size of each machine set equals one. Next, for each of these problems that have been proven to be solvable in polynomial time, we consider generalizations that are obtained by allowing machine set sizes larger than one. Thereafter, we consider generalizations of the same problems, but now by allowing more than one machine set per operation. Finally, for the polynomially solvable subproblems in the two latter classes we consider generalizations in which both machine sets of size larger than one and more than one machine set per operation are allowed.

Problems with chain precedences in which each operation can be processed by exactly one machine set and in which the size of this machine set equals one, are job shop scheduling problems or generalizations of job shop scheduling problems in which it is allowed that two consecutive operations of a chain need to be processed by the same machine. As we have seen in the previous subsection, the problem $\bullet\ 1\ 1 \mid$ empty $\mid C_{\max}$ is solvable in polynomial time. Replacing the empty precedence structure by chain precedences will make the problem harder. The complexity of many subproblems of $\bullet\ 1\ 1 \mid$ chain $\mid C_{\max}$ is well known.

The following problems are solvable in polynomial time:

* $1\ 1\ 1 \mid$ chain $\mid C_{\max}$;
* $2\ 1\ 1 \mid$ chain, $n \mid C_{\max}$ [Brucker, 1994];
* $\bullet\ 1\ 1 \mid$ chain, $n = 2 \mid C_{\max}$ [Akers, 1956; Brucker, 1988];
* $J2\ 1\ 1 \mid$ chain, $1 \mid C_{\max}$ [Hefetz and Adiri, 1982];
* $2\ 1\ 1 \mid$ chain, $|J| \leq 2 \mid C_{\max}$.

The first result is trivial. The last result is obtained by generalizing the algorithm of Jackson [1956] for the problem $J2\ 1\ 1 \mid$ chain, $|J| \leq 2 \mid C_{\max}$ by replacing a chain of two operations that have to be processed by the same machine by one single operation for which the processing time is the sum of the processing times of the two given operations.

On the other hand, the following problems are NP-hard:

* $J3\ 1\ 1 \mid$ chain, $n = 3 \mid C_{\max}$ [Alberton, 1988];
* $2\ 1\ 1 \mid$ chain, $1 \mid C_{\max}$ [Blazewicz, Lenstra and Rinnooy Kan, 1983; Hoogeveen, Van de Velde and Veltman, 1994];

* $J2\,1\,1 \mid \text{chain}, p(v, H) \leq 2 \mid C_{\max}$ [Lenstra and Rinnooy Kan, 1979];
* $J3\,1\,1 \mid \text{chain}, 1 \mid C_{\max}$ [Lenstra and Rinnooy Kan, 1979];
* $J2\,1\,1 \mid \text{chain}, |J| \leq 3 \mid C_{\max}$ [Lenstra, Rinnooy Kan and Brucker, 1977];
* $J3\,1\,1 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$ [Lenstra, Rinnooy Kan and Brucker, 1977].

Next, we consider problems that are obtained from the five polynomially solvable problems mentioned above by relaxing the constraint that each possible machine set consists of a single machine. Clearly, relaxing the first problem in this sense is not possible, and the relaxation of the fourth problem is NP-hard due to the fact that $2\,1\,1 \mid \text{chain}, 1 \mid C_{\max}$ is already NP-hard. The relaxations of the remaining three problems are

* $2\,2\,1 \mid \text{chain}, n \mid C_{\max}$,
* $\bullet\,\bullet\,1 \mid \text{chain}, n = 2 \mid C_{\max}$, and
* $2\,2\,1 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$,

and these remain solvable in polynomial time. Clearly, for the first and the third problem the size of each machine set is bounded by two.

The first result is obtained by generalizing the algorithm of Brucker [1994] for the problem $2\,1\,1 \mid \text{chain}, n \mid C_{\max}$. Brucker defines a block as a schedule of a special type for a subset of the operations. In such a block two operations start at the same time if and only if no other operations start earlier in the block, and an operation starts on a machine while leaving the other machine idle if and only if no operation starts earlier in the block. A precise definition of a block is given in Theorem 3.14. For the problem $2\,2\,1 \mid \text{chain}, n \mid C_{\max}$ we have to take into account also operations that are executed by the two machines simultaneously. This is solved by defining additional blocks consisting of a single operation that is executed by both machines simultaneously. In a way similar to the proof of Brucker, one can prove that Brucker's algorithm runs in polynomial time for the problem $2\,2\,1 \mid \text{chain}, n \mid C_{\max}$.

The second result is obtained by generalizing a graphical algorithm for the problem $\bullet\,1\,1 \mid \text{chain}, n = 2 \mid C_{\max}$ [Akers, 1956]. This algorithm has to find a shortest line consisting of horizontal, vertical and diagonal segments between two points in the plane, such that it does not cross certain rectangles in the plane. These rectangles are called obstacles and express that the two jobs cannot be processed simultaneously when they need the same machine. For the problem $\bullet\,\bullet\,1 \mid \text{chain}, n = 2 \mid C_{\max}$ we have to take into account that there may be operations that need to be processed on a machine set of size larger than one. This is done by defining an obstacle if the machine sets of two operations that do not

belong to the same job have a machine in common, instead of defining an obstacle when two operations have to be processed by the same machine. Clearly, checking whether two machine sets have a machine in common can be done in polynomial time.

The third result is obtained by generalizing the algorithm of Jackson [1956] for the problem $J2\,1\,1 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$. Besides the modification to obtain an algorithm for $2\,1\,1 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$, still further modifications are needed to take into account operations that are executed by two machines. This is done as follows. All first operations of a job with a machine set of size two are scheduled before all operations with a machine set of size one and all last operations of a job with a machine set of size two are scheduled after all operations with a machine set of size one. In between all remaining operations with a machine set of size one are scheduled in the same way as dictated by the algorithm for $2\,1\,1 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$. Here, each operation with a machine set of size two is neglected by lowering the size of the corresponding job by one.

Next, we consider problems that are obtained from the five polynomially solvable problems mentioned above by relaxing the constraint that each operation has only one possible machine set. Again, relaxing the first problem in this sense is not possible, and the relaxation of the fourth problem is NP-hard since the problem $2\,1\,1 \mid \text{chain}, 1 \mid C_{\max}$ is already NP-hard. The relaxations of the remaining three problems are

* $2\,1\,2 \mid \text{chain}, n \mid C_{\max}$,
* $\bullet\,1\,\bullet \mid \text{chain}, n = 2 \mid C_{\max}$, and
* $2\,1\,2 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$.

We show that the first problem is solvable in polynomial time when at most two jobs are given, but that it is NP-hard in the case of three jobs. Given these results, $3\,1\,2 \mid \text{chain}, n = 2 \mid C_{\max}$ is the minimal subproblem of the second problem that is still of interest. We show that this problem is already NP-hard. However, we also show that even the more general problem $3\,1\,3 \mid \text{chain}, n = 2 \mid C_{\max}$ is solvable in pseudo-polynomial time. Finally, the third problem is NP-hard.

Summarizing, we claim the following results:

* $2\,1\,2 \mid \text{chain}, n = 2 \mid C_{\max}$ is solvable in polynomial time;
* $2\,1\,2 \mid \text{chain}, n = 3 \mid C_{\max}$ is NP-hard;
* $3\,1\,2 \mid \text{chain}, n = 2 \mid C_{\max}$ is NP-hard;
* $3\,1\,3 \mid \text{chain}, n = 2 \mid C_{\max}$ is solvable in pseudo-polynomial time;
* $2\,1\,2 \mid \text{chain}, |J| \leq 2 \mid C_{\max}$ is NP-hard.

The first result is a corollary of Theorem 3.14, in which it is proven that the more general problem 2 2 3 | chain, $n = 2$ | $C_{\max}$ is still solvable in polynomial time. The second result is a corollary of the fact that the more restricted problem 2 1 2 | chain, $n = 3$, $p(v)$ | $C_{\max}$ is already NP-hard, which has been shown by Jurisch [1992] and by Brucker, Jurisch and Krämer [1994]. The final result is a corollary of the fact that the more restricted problem 2 1 2 | empty, $p(v)$ | $C_{\max}$ is already NP-hard (see the previous subsection). The other two results are proven in the following two theorems.

Theorem 3.12. *The problem* 3 1 2 | chain, $n = 2$ | $C_{\max}$ *is NP-hard.*

Proof. We give a polynomial-time reduction from the PARTITION problem to the decision variant of this scheduling problem.

The PARTITION problem is defined as follows: let $T = \{0, \ldots, t-1\}$ for some $t \in \mathbb{N}$ and let $Z \in \mathbb{N}$; let, for each $i \in T$, a number $z_i \in \mathbb{N}$ be given such that

$$\sum_{i \in T} z_i = 2Z;$$

the question is whether there exists a $T' \subseteq T$ such that

$$\sum_{i \in T'} z_i = Z.$$

Such a set T' and its complement form a partition of T. PARTITION is proven NP-complete by Karp [1972].

Suppose we are given an instance of the problem PARTITION. We transform this instance into an instance of the decision variant of the given scheduling problem in such a way that there exists a feasible schedule of a certain length if and only if a partition exists.

The transformation is defined as follows. V, M, and A are defined as

$$\begin{aligned} V &= \{v_i | i = 0, \ldots, 2t-1\} \cup \{w_i | i = 0, \ldots, 2t-1\}, \\ M &= \{1, 2, 3\}, \text{ and} \\ A &= \{(v_i, v_{i+1}) | i = 0, \ldots, 2t-2\} \cup \{(w_i, w_{i+1}) | i = 0, \ldots, 2t-2\}. \end{aligned}$$

The machine sets and the processing times are taken as follows ($i \in T$):

$$\begin{aligned} \mathcal{H}(v_{2i}) &= \{\{1\}, \{3\}\} & p(v_{2i}, \{1\}) &= 2Z + z_i \\ & & p(v_{2i}, \{3\}) &= 2Z \\ \mathcal{H}(v_{2i+1}) &= \{\{1\}\} & p(v_{2i+1}, \{1\}) &= 2Z \\ \mathcal{H}(w_{2i}) &= \{\{2\}, \{3\}\} & p(w_{2i}, \{2\}) &= 2Z + z_i \\ & & p(w_{2i}, \{3\}) &= 2Z \\ \mathcal{H}(w_{2i+1}) &= \{\{2\}\} & p(w_{2i+1}, \{2\}) &= 2Z. \end{aligned}$$

Now a schedule is required of length at most $(4t + 1)Z$.

Now suppose that a schedule (S, K) exists for which the length is not larger than $(4t + 1)Z$. Let

$$T_1 = \{i \in T | K(v_{2i}) = \{1\}\} \text{ and}$$
$$T_2 = \{i \in T | K(w_{2i}) = \{2\}\}.$$

We have to prove that T_1 and T_2 form a partition of T. First, we prove that their union is equal to T. Suppose that there exists a $j \in T \backslash (T_1 \cup T_2)$. Then we have $K(v_{2j}) = K(w_{2j}) = \{3\}$. Now, under the assumption $S(v_{2j}) < S(w_{2j})$ we have

$$\begin{aligned} C_{\max} &\geq \sum_{i=0}^{2j} p(v_i, K(v_i)) + \sum_{i=2j}^{2t-1} p(w_i, K(w_i)) \\ &\geq (2t + 1)2Z, \end{aligned}$$

which contradicts the assumption that the given schedule has length not larger than $(4t + 1)Z$. A similar argument holds when $S(v_{2j}) > S(w_{2j})$. Next we prove that $T_1 \cap T_2 = \emptyset$. We have

$$\begin{aligned} 2C_{\max} &\geq \sum_{i=0}^{2t-1} p(v_i, K(v_i)) + \sum_{i=0}^{2t-1} p(w_i, K(w_i)) \\ &= 4t \cdot 2Z + \sum_{i \in T_1} z_i + \sum_{i \in T_2} z_i \\ &= 4t \cdot 2Z + 2Z + \sum_{i \in T_1 \cap T_2} z_i \\ &\geq 2(4t + 1)Z. \end{aligned}$$

This can only be true if equality holds in the two inequalities. Hence, $T_1 \cap T_2 = \emptyset$. Furthermore, it follows that

$$\sum_{i=0}^{2t-1} p(v_i, K(v_i)) = \sum_{i=0}^{2t-1} p(w_i, K(w_i)) = (4t + 1)Z.$$

On the other hand we have

$$\sum_{i=0}^{2t-1} p(v_i, K(v_i)) = 2t \cdot 2Z + \sum_{i \in T_1} z_i,$$

and therefore

$$\sum_{i \in T_1} z_i = Z.$$

Now suppose that a partition $(T', T \backslash T')$ exists of the given instance of PARTITION. From the reasoning above it should be clear that we take for all $i \in T$

$$K(v_{2i}) = \{1\} \quad \text{if and only if} \quad i \in T' \text{ and}$$
$$K(w_{2i}) = \{2\} \quad \text{if and only if} \quad i \in T \backslash T'.$$

The starting times are taken as follows ($i \in T$):

$$\begin{aligned} S(v_{2i}) &= 2i \cdot 2Z + \sum_{j \in T', j \leq i-1} z_j; \\ S(v_{2i+1}) &= 2(i+1)i \cdot 2Z + \sum_{j \in T', j \leq i} z_j; \\ S(w_{2i}) &= 2i \cdot 2Z + \sum_{j \in T \setminus T', j \leq i-1} z_j; \\ S(w_{2i+1}) &= 2(i+1)i \cdot 2Z + \sum_{j \in T \setminus T', j \leq i} z_j. \end{aligned}$$

What remains is to prove that this schedule is feasible. Since the only operations on machine 1 are the v operations and the only operations on machine 2 the w operations, it is easy to check that on machine 1 and on machine 2 no two operations overlap. The following proves that for any v operation and any w operation on machine 3 the start time of one of them and the completion time of the other differ by at least Z. Let $i_1 \in T'$ and $i_2 \in T \setminus T'$. Then for v_{2i_2} and w_{2i_1}, which are both scheduled on machine 3, we have:

$$\begin{aligned} & |S(v_{2i_2}) - S(w_{2i_1})| \\ = \; & |2i_2 \cdot 2Z + \sum_{j \in T', j \leq i_2-1} z_j - 2i_1 \cdot 2Z - \sum_{j \in T \setminus T', j \leq i_1-1} z_j| \\ \geq \; & |2(i_2 - i_1) \cdot 2Z| - |\sum_{j \in T', j \leq i_2-1} z_j - \sum_{j \in T \setminus T', j \leq i_1-1} z_j| \\ \geq \; & 4Z - \max\{\sum_{j \in T'} z_j, \sum_{j \in T \setminus T'} z_j\} \\ = \; & 3Z. \end{aligned}$$

Since the processing times of both v_{2i_2} and w_{2i_1} on machine 3 are $2Z$, we have the required result. □

Theorem 3.13. *The problem* 3 1 3 | chain, $n = 2$ | $C_{\max}$ *is solvable in pseudo-polynomial time.*

Proof. We describe an algorithm that determines the length of a given instance in pseudo-polynomial time. This algorithm is a variant of the one of Giffler and Thompson [1960], in which all active schedules for the job shop scheduling problem are generated. In this variant we have to deal with the fact that an operation can be processed by several machines. Furthermore, to ensure that the algorithm runs in pseudo-polynomial time, we introduce a dominance relation. In this way we exclude some active schedules that cannot be optimal.

First we describe an enumeration algorithm that generates all strongly active schedules, but in which dominance is not yet taken into account. We assume that

the operations of job 1 are numbered $1, \ldots, l_1$ and those of job 2 are numbered $1, \ldots, l_2$. The enumeration algorithm generates partial schedules by constructing them from front to back, that is, for each job in a partial schedule all operations are scheduled that have index not larger than a particular (job dependent) index and no operations with larger index are scheduled. To extend a particular partial schedule the following is done. One considers the operation of job 1 that in the precedence relation immediately follows the last scheduled one of job 1, and the similar operation of job 2, provided they exist. For each of these operations the earliest possible completion time on any of the machines is determined. An operation v and a machine μ is selected that gives the minimal earliest possible completion time. The only operation that can prevent v of being scheduled on machine μ with this minimal earliest possible completion time, is the first unscheduled operation of the other job. Therefore, each partial schedule can be extended in at most two ways. Each time a partial schedule is extended by one operation of a particular job on a particular machine, we immediately schedule operations of the other job on the other machine as early as possible. This is done as long as the length of the extended partial schedule remains the same. In a similar way as is proven that the algorithm of Giffler and Thompson [1960] generates all active schedules for the job shop scheduling problem, one can prove that this algorithm generates all strongly active schedules.

Clearly, the number of different strongly active schedules generated in this way is $\mathcal{O}(2^l)$. To find a pseudo-polynomial algorithm, the introduction of a dominance relation is needed. To be able to compare two partial schedules as far as their possible extensions is concerned, the only relevant information needed is the number of scheduled operations of each job and the completion times of all jobs and all machines. Therefore, a partial schedule is represented by a 7-tuple $(q_1, q_2; cj_1, cj_2; cm_1, cm_2, cm_3)$. Here, q_1 and q_2 represent the number of scheduled operations of job 1 and job 2, respectively, cj_j denotes the completion time of the last operation scheduled on job j, and cm_μ denotes the completion time of the last operation scheduled on machine μ. Now we say that

$$T = (q_1, q_2; cj_1, cj_2; cm_1, cm_2, cm_3) \text{ dominates}$$
$$T' = (q_1', q_2'; cj_1', cj_2'; cm_1', cm_2', cm_3')$$

if

$$q_1 = q_1' \text{ and } q_2 = q_2' \text{ and}$$
$$\max(cj_j, cm_\mu) \leq \max(cj_j', cm_\mu') \text{ for all } j \in \{1, 2\} \text{ and } \mu \in \{1, 2, 3\},$$

that is, if on any of the machines neither the next operation of job 1 nor the next operation of job 2 can start earlier in the partial solution corresponding to T' than in the partial solution corresponding to T. Note that only partial schedules are compared in which the same operations are scheduled. Clearly, for each exten-

sion of a dominating partial schedule the same extension in a dominated schedule will lead to a length that is as least as long as in the dominating schedule. Here, two extensions are considered the same if the choices of the machines on which the remaining operations are scheduled and the sequence of the remaining operations on each of the machines are identical. Therefore, it is sufficient to consider only dominating partial schedules and to discard dominated partial schedules.

To be able to discard as many dominated schedules as possible, the partial schedules in the enumeration tree for which no further extensions have been considered yet have to be considered in a special order. It is not allowed that a partial schedule is considered for further extensions, when another partial schedule exists that has not been considered for further extensions and in which the number of scheduled operations of both job 1 and job 2 is not larger.

Now consider all possible partial schedules generated by the enumeration algorithm described above in which a particular fixed number of operations of job 1 and a particular fixed number of operations of job 2 are scheduled, and in which no schedule is dominated by another. Let $p_{\max}$ denote the maximum processing time of any of the operations on any of the machines. From the fact that in any of the partial schedules generated by the enumeration algorithm cj_1 and cj_2 differ by at most $p_{\max} - 1$, it follows that $6p_{\max}$ is an upper bound on the number of mutually undominating schedules in which a fixed number of operations of job 1 and a fixed number of operations of job 2 are scheduled. Since the number of possibilities to fix the number of operations of job 1 and job 2 is equal to $(l_1+1)\cdot(l_2+1)$, the total number of mutually undominating schedules is at most $(l_1 + 1) \cdot (l_2 + 1) \cdot 6p_{\max}$. Now, it follows that the algorithm described above runs in pseudo-polynomial time. □

We now briefly discuss how the complexity of the problems with machine set size one changes when we require for each operation that its processing time does not depend anymore on the machine set on which it is processed. Obviously, these problems become easier by posing this extra condition. The problem 2 1 2 | chain, $n = 3$ | $C_{\max}$ remains (weakly) NP-hard when adding this extra condition. It is solvable in pseudo-polynomial time, and this also holds for the more general problem with an arbitrary but fixed number of jobs and with no restrictions on the number of machines and the number of possible machine sets per operation. The problem 3 1 2 | chain, $n = 2$ | $C_{\max}$ becomes polynomially solvable when we add this extra condition. Even the more general subproblem • 1 • | chain, $n = 2$, $p(v)$ | $C_{\max}$ has been proven solvable in polynomial time. However, if we drop the condition that the number of jobs is fixed, then the problem with only two machines and at most two possible machine sets per operation

is already strongly NP-hard. Finally, the problem 2 1 2 | chain, $|J| \leq 2$ | C_{max} remains NP-hard if we require that the processing times do not depend on the machine set on which an operation is processed. Summarizing, we have the following results:

* 2 1 2 | chain, $n = 3$, $p(v)$ | C_{max} is NP-hard;
* • 1 • | chain, n, $p(v)$ | C_{max} is solvable in pseudo-polynomial time;
* • 1 • | chain, $n = 2$, $p(v)$ | C_{max} is solvable in polynomial time;
* 2 1 2 | chain, $p(v)$ | C_{max} is strongly NP-hard;
* 2 1 2 | chain, $|J| \leq 2$, $p(v)$ | C_{max} is NP-hard.

The first result is proven by Jurisch [1992] and by Brucker, Jurisch and Krämer [1994]. The second result is proven by Meyer [1992], the third by Brucker and Schlie [1990], and the fourth by Du, Leung and Young [1991]. The last result is trivial since the subproblem 2 1 2 | empty, $p(v)$ | C_{max} is already NP-hard.

Finally, we consider problems in which for each operation more than one machine set is given, and in which the size of each machine set may be larger than one.

Since 2 1 2 | chain, $n = 2$ | C_{max} is the most general polynomially solvable problem with machine sets of size one, with more than one possible machine set per operation, and without any restriction on the processing times, the only interesting problem that remains to be considered is the problem

$$2\,2\,3 \mid \text{chain}, n = 2 \mid C_{max}.$$

The following theorem shows that this problem is solvable in polynomial time.

Theorem 3.14. *The problem* 2 2 3 | chain, $n = 2$ | C_{max} *is solvable in polynomial time.*

Proof. We give an algorithm to determine the length for a given instance. This algorithm is based on the one for the problem 2 1 1 | chain, n | C_{max} given by Brucker [1994]. It constructs a directed graph with weighted arcs, in which a particular type of path of minimal length has to be found. This length equals the length of a schedule of minimal length.

Before giving the algorithm some notions have to be introduced. A *block* is a schedule on a non-empty subset B of V that satisfies the following conditions:

* B consists of some consecutive operations of job 1 and of some consecutive operations of job 2;
* for each job the first operation in B starts at time 0;

* for each operation in B and for each machine this operation is scheduled on, the start time is equal to 0 or to the completion time of another operation on this machine;
* at any time unequal to 0 at most one machine starts the processing of an operation;
* no operation starts at a time on which the other machine is idle or starts an idle period.

Obviously, there are three types of blocks:

* a block in which a single operation is processed simultaneously on both machines;
* a block in which all operations of job 1 in B are processed by machine 1 and all operations of job 2 in B by machine 2;
* a block in which all operations of job 1 in B are processed by machine 2 and all operations of job 2 in B by machine 1.

Clearly, each left-justified schedule on V can be composed by the juxtaposition of several blocks, so an optimal schedule can be found by considering all possible juxtapositions of blocks. Therefore, we define the *length* of a block as the completion time of the last operation in this block. Now, the length of a schedule on V, which is obtained by the juxtaposition of several blocks, equals the sum of all lengths of these blocks.

In order to define the directed weighted graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ mentioned above, we assume that the operations of job 1 are numbered $1, \ldots, l_1$ and those of job 2 are numbered $1, \ldots, l_2$. Now we take

$$\mathcal{V} = \{0, \ldots, l_1\} \times \{0, \ldots, l_2\} \text{ and}$$
$$\mathcal{A} \subseteq \{((q_1, q_2), (r_1, r_2)) \mid q_1 \leq r_1,\ q_2 \leq r_2,\ (q_1, q_2) \neq (r_1, r_2)\}.$$

A pair $((q_1, q_2), (r_1, r_2))$ constitutes an arc in $\mathcal{A}$ if a block exists on the operations $q_1 + 1, \ldots, r_1$ of job 1 and the operations $q_2 + 1, \ldots, r_2$ of job 2; the weight of this arc is equal to the smallest length of all such blocks. Clearly, each path from $(0, 0)$ to (l_1, l_2) corresponds to a schedule, and the length of this path equals the length of the corresponding schedule. This schedule can be obtained by the juxtaposition of the blocks associated with the arcs of this path. Furthermore, a shortest path from $(0, 0)$ to (l_1, l_2) corresponds to an optimal schedule. Thus, to solve the scheduling problem it suffices to find a path of minimal length in the graph $\mathcal{G}$.

It remains to show that constructing the graph $\mathcal{G}$ and determining a path of minimal length can be done in polynomial time. Clearly, the directed graph $\mathcal{G}$ has no cycles, so a path of minimal length can be determined in a time, which is polynomial in the number of nodes in $\mathcal{G}$. Obviously, this number of nodes is

equal to $(l_1+1)(l_2+1)$. The number of arcs in $\mathcal{G}$ is $\mathcal{O}((l_1+1)^2(l_2+1)^2)$. Computing the length of an arc takes $\mathcal{O}(l_1+l_2)$ time, since for each arc there are at most three types of blocks that can define this arc. Clearly, computing the length of each of these blocks takes $\mathcal{O}(l_1+l_2)$ time. □

Now we consider the case in which the processing times of operations do not depend on the machine set on which they are processed. The most general polynomially solvable problem with machine sets of size one, with more than one possible machine set per operation, and with $p(v, H)=p(v)$ is the problem $\bullet\, 1\, \bullet \mid \text{chain}, n=2, p(v) \mid C_{\max}$. The more general problem

$$\bullet\, \bullet\, \bullet \mid \text{chain}, n=2, p(v) \mid C_{\max},$$

in which no restrictions hold for the machine set sizes, is still solvable in polynomial time. This result can be obtained by generalizing the algorithm of Brucker and Schlie [1990] for the problem $\bullet\, 1\, \bullet \mid \text{chain}, n=2, p(v) \mid C_{\max}$. For this the definition of an obstacle has to be slightly modified, such that it is possible to deal with machine sets instead of single machines. Since these obstacles can still be computed in polynomial time, the problem still belongs to $\mathcal{P}$.

4

Local Search

Since the introduction of local search methods by Bock [1958] and Croes [1958], a large variety of local search algorithms has been proposed. Each of these variants tries to decrease the risk of getting stuck in poor local optima, which is the most important disadvantage of the simplest local search algorithm, the deterministic iterative improvement algorithm. At present, there is a proliferation of local search algorithms, which, in all their different guises, seem to be based on a few basic ideas only.

In this chapter a local search template is presented that has been designed to capture most of the variants proposed in the literature. The aim of the template is to provide a classification of the various existing local search algorithms. Furthermore, it should also be sufficiently general to capture new approaches to local search and thereby to suggest novel variants.

The organization of this chapter is as follows. Section 4.1 introduces some basic definitions. Section 4.2 considers deterministic iterative improvement algorithms. Section 4.3 presents the local search template, and Section 4.4 shows for a number of well-known local search algorithms how they fit into this template. Section 4.5 mentions some lesser known or new algorithms that fit into the template. Finally, Section 4.6 discusses some complexity issues of local search.

4.1 Preliminaries

An *optimization problem* is either a maximization or a minimization problem specified by a class of problem instances. Without loss of generality we restrict ourselves to minimization problems. An *instance* is defined by the implicit specification of a triple $(\mathcal{S}, \mathcal{X}, f)$, where the *solution space* $\mathcal{S}$ is the set of all (feasible) solutions, the *cost space* $\mathcal{X}$ is a totally ordered set of all possible cost values, and the *cost function* f is a mapping $f: \mathcal{S} \to \mathcal{X}$. The optimal cost f_{opt} of an instance is defined by $f_{\text{opt}} = \min\{f(s) | s \in \mathcal{S}\}$, and the set of optimal solutions is denoted by $\mathcal{S}_{\text{opt}} = \{s \in \mathcal{S} | f(s) = f_{\text{opt}}\}$. The objective is to find some solution $s_{\text{opt}} \in \mathcal{S}_{\text{opt}}$. An optimization problem is called a *combinatorial optimization problem* if for all instances $(\mathcal{S}, \mathcal{X}, f)$ the solution space $\mathcal{S}$ is finite or countably infinite.

A *neighborhood function* $\mathcal{N}$ is a mapping $\mathcal{N}: \mathcal{S} \to \mathcal{P}(\mathcal{S})$, which specifies for each $s \in \mathcal{S}$ a subset $\mathcal{N}(s)$ of $\mathcal{S}$ of neighbors of s. A solution $s \in \mathcal{S}$ is called a *local minimum with respect to* $\mathcal{N}$ if $f(s) \leq f(t)$ for all $t \in \mathcal{N}(s)$. Furthermore, to distinguish between local minima and elements of $\mathcal{S}_{\text{opt}}$, we call the latter ones *global* minima. A neighborhood function $\mathcal{N}$ is called *exact* if every local minimum with respect to $\mathcal{N}$ is also a global minimum.

In some applications of local search algorithms to optimization problems the search is not done directly on the solution space $\mathcal{S}$, but on a search space, in which solutions are represented in one way or another.

For problems with very few feasible solutions or for which it is even not known whether feasible solutions exist, the search space may also contain infeasible solutions in addition to feasible ones. In this case, the cost function f should be extended in such a way that the a solution of minimum cost automatically corresponds to a feasible solution. The same is true if one wants to include partial solutions in the search space.

Another possibility is that some solutions are represented by more than one element in the search space. This occurs very often in applications of genetic algorithms, which are discussed in Section 4.4.3. Here, solutions are represented by strings over a finite alphabet, and several such strings may represent the same solution.

However, since each problem described by a non-trivial representation can be seen as an optimization problem of its own, we consider in the remainder of this chapter only problems of the form introduced at the beginning of this section.

4.2 Deterministic iterative improvement

The basic local search algorithm is the so-called deterministic iterative improvement algorithm. We assume that an instance of an optimization problem and a neighborhood function are given. The deterministic iterative improvement algorithm starts from an initial solution and then continually searches the neighborhood of the current solution for a solution of better quality. If such a solution is found, it replaces the current solution. The algorithm terminates as soon as the current solution has no neighboring solutions of better quality, at which point a locally optimal solution is found.

The pseudo-Pascal procedure given in Figure 4.1 represents the basic part of the deterministic iterative improvement algorithm. Here, the procedure GENERATE NEIGHBOR deterministically generates a solution t from the neighborhood $\mathcal{N}(s)$ of the current solution s, such that every $t \in \mathcal{N}(s)$ is generated at most once as a neighbor of s. The procedure DETERMINISTIC ITERATIVE IMPROVEMENT returns a solution s that is locally optimal with respect to the neighborhood function $\mathcal{N}$.

The deterministic iterative improvement algorithm terminates at the first local optimum that is found and, in general, the quality of such a local optimum may be arbitrarily bad. To improve the quality one can consider applying the following ideas.

* Generating several or all neighbors of the current solution instead of just one neighbor in each iteration. If all neighbors are generated and a best one is accepted, one obtains a steepest descent or best improvement algorithm.
* Using more intricate functions to determine a new solution from the current solution and its neighbor, for instance, accepting solutions of quality worse

---

```
procedure DETERMINISTIC ITERATIVE IMPROVEMENT (s ∈ S);
  { input:  s ∈ S
    output: s ∈ S, s locally optimal w.r.t. N }
  begin
    repeat
      GENERATE NEIGHBOR (s, t, N);
      if f(t) < f(s) then s := t
    until ∀t ∈ N(s) : f(t) ≥ f(s)
  end
```

Figure 4.1: The procedure DETERMINISTIC ITERATIVE IMPROVEMENT.

than that of the current solution. The well-known simulated annealing and threshold accepting algorithms fall into this category.

* Replacing the single current solution by a population of current solutions. This is the basic idea of genetic algorithms.
* Alternating between two or more neighborhood functions. Such an algorithm has been proposed by Martin, Otto and Felten [1989].

Based on one or more of the above ideas, a considerable number of algorithms has been proposed in the literature. In the next section we present a generic local search template that captures most of these ideas.

4.3 A local search template

Our local search template generalizes the iterative improvement algorithm from the previous section in the following ways.

1. The search may proceed at several *levels*, each with its own specifications.
2. The single current solution is replaced by a *population* of current solutions.
3. The neighborhood function associated with a single solution is replaced by a neighborhood function associated with a *cluster* of solutions.

More formally, a *population* P at a given level l is a multi-set of p_l solutions from S. It represents the current state of the search at level l. We will talk about *point-based* local search if, at the first level, P contains one single solution, and about *population-based* local search otherwise. A *cluster* at level l is a c_l-tuple $C \in S^{c_l}$ of solutions, such that with each cluster a hyper-neighborhood is associated. That is, there is a *hyper-neighborhood function* $\mathcal{N}_l : S^{c_l} \rightarrow \mathcal{P}(S)$ which, for each cluster C, defines a set $\mathcal{N}_l(C)$ of neighboring solutions. Here c_l denotes the *cluster size*. In case $c_l = 1$, the hyper-neighborhood function reduces to the standard neighborhood function of Section 4.1.

The local search template is defined by the recursive procedure LOCAL SEARCH. At each level l, this procedure takes a population P as input and uses the hyper-neighborhood function $\mathcal{N}_l$ to produce a new population P as output. This is done in two nested loops: an outer loop of *generations* and an inner loop of *iterations*.

The generation loop creates a number of generations of populations until a stopping condition is satisfied. In each generation, the procedure GENERATE CLUSTERS assembles from the current population P a finite multi-set $\mathcal{C}$ of clusters $C \in P^{c_l}$. Hence, each of the c_l components $C_1, \ldots, C_{c_l}$ of a cluster C of $\mathcal{C}$ is a solution from P.

For each cluster $C \in \mathcal{C}$, the iteration loop applies a number of iterations until a stopping criterion is satisfied. Each iteration starts with a call of the procedure

GENERATE NEIGHBORS, which selects a finite multi-set Q of hyper-neighbors of C, that is, each component of Q is an element from $\mathcal{N}_l(C)$. The procedure LOCAL SEARCH is then called recursively, with level $l+1$ and population Q as its parameters. The result is a modified population Q. After this, the procedure REDUCE NEIGHBORS reduces the union of the original cluster C and the new population Q into a new cluster $C \in \mathcal{S}^{c_l}$, which then serves as input for the next iteration.

If, at level l, the iteration loop has terminated for all clusters $C \in \mathcal{C}$, the procedure CREATE collects the solutions found in (usually) the final iteration for each $C \in \mathcal{C}$ into a single multi-set $\hat{P}$ of $\mathcal{S}$. The procedure REDUCE POPULATION finally merges P and $\hat{P}$ into a new current population P.

We now give a short description of the procedures and functions used in the procedure LOCAL SEARCH, which is shown in Figure 4.2 in pseudo-Pascal.

* The Boolean function CONTINUE POPULATION GENERATION has the current level l as input. Based on additional information from previous generations, it returns the value TRUE as long as new generations of populations have to be generated and FALSE otherwise.
* The procedure GENERATE CLUSTERS has a level l and a population P as input and a finite multi-set $\mathcal{C} \subseteq \mathcal{S}^{c_l}$ as output. It clusters P into a collection $\mathcal{C}$ of c_l-tuples $C \in P^{c_l}$, either deterministically or probabilistically. In this way, the hyper-neighborhood function $\mathcal{N}_l$ can be applied indirectly to the given population P.
* The Boolean function CONTINUE ITERATION has a level l as input. Based on additional information from previous iterations it returns the value TRUE as long as iterations have to go on in the iteration loop and FALSE otherwise.
* The procedure GENERATE NEIGHBORS has a level l, a multi-set C of size c_l, and the hyper-neighborhood function $\mathcal{N}_l$ as input, and a multi-set Q of size p_{l+1} as output. It generates a multi-set Q of neighbors from C using $\mathcal{N}_l$. The basic part of the procedure prescribes how a neighbor of C is to be determined, that is, randomly or deterministically, and how many neighbors are to be determined.
* The procedure REDUCE NEIGHBORS has a level l, a c_l-tuple C, and a p_{l+1}-tuple Q as input, and a modified version of C as output. It determines how to merge the old cluster C and the collection Q of (modified) neighbors into a new cluster C.
* The procedure CREATE has a level l as input and a population $\hat{P}$ as output. It puts a population $\hat{P}$ together from solutions found in (usually) the final iteration for each $C \in \mathcal{C}$.

```
procedure LOCAL SEARCH (l: integer; P ∈ S^{p_l} );
  { input:  l ∈ ℕ, P ∈ S^{p_l}
    output: P ∈ S^{p_l}          }
  begin
    while CONTINUE POPULATION GENERATION (l) do
      begin
        GENERATE CLUSTERS (l, P, C);
        for all C ∈ C do
          begin
            while CONTINUE ITERATION (l) do
              begin
                GENERATE NEIGHBORS (l, C, N_l, Q);
                LOCAL SEARCH (l + 1, Q);
                REDUCE NEIGHBORS (l, C, Q)
              end
          end;
        CREATE (l, P̂);
        REDUCE POPULATION (l, P, P̂)
      end
  end;
```

Figure 4.2: The procedure LOCAL SEARCH.

* The procedure REDUCE POPULATION merges P and $\hat{P}$ into a new population P.

To make the recursive procedure finite, we need to define a bottom level l^*. At this level l^*, the Boolean function CONTINUE POPULATION GENERATION assumes the value FALSE. The levels $l < l^*$ are called *active* levels. Obviously, for the description of a local search algorithm only a specification of the active levels is needed. We know of no algorithms that use more than two active levels.

The next section shows how most local search algorithms proposed in the literature fit into our template.

4.4 Instantiations of the local search template

The local search template captures most types of local search algorithms proposed in the literature. This is shown by the specification of the bottom level l^* and, for each active level, by an instantiation of the procedures GENERATE

CLUSTERS, REDUCE NEIGHBORS, CREATE and REDUCE POPULATION. The other procedures are usually less characteristic of an algorithm; they are instantiated only if they constitute a relevant part of the algorithm.

In handling the various local search algorithms we distinguish between *point-based* and *population-based* local search and between local search with exactly one and more than one active level.

4.4.1 Single-level point-based local search

Among point-based local search algorithms with one active level, first the classes of *threshold* and *taboo search* algorithms are discussed. Next, *variable-depth search* is discussed.

Threshold algorithms and taboo search. Both threshold and taboo search algorithms are characterized by the fact that only one generation is created. Hence, they are completely determined by the iteration loop of the procedure LOCAL SEARCH. Both algorithms can be instantiated as follows.

* CONTINUE POPULATION GENERATION returns the value TRUE for the first generation and FALSE for each subsequent generation. In this way only one generation is created.
* GENERATE CLUSTERS generaties a single cluster C of size 1 that contains the single solution of the 1-tuple P.
* CREATE sets $\hat{P}$ equal to the current cluster C.
* REDUCE POPULATION sets the new population P equal to $\hat{P}$.

We now consider threshold and taboo search algorithms separately.

In *threshold algorithms*, a neighbor of a given solution becomes the new current solution if the cost difference between the current solution and its neighbor is below a certain threshold $t \in \mathbb{R}$. Depending on the nature of the thresholds one distinguishes three kinds of threshold algorithms: iterative improvement, simulated annealing [Kirkpatrick, Gelatt and Vecchi, 1983; Černy, 1985; Aarts and Korst, 1989], and threshold accepting [Dueck and Scheuer, 1990]. They are characterized by the following instantiations.

* Using the neighborhood function $\mathcal{N}_1$ GENERATE NEIGHBORS generates a multi-set Q that contains only one neighbor Q_1. In most cases a neighbor is generated randomly; sometimes this is done deterministically.
* REDUCE NEIGHBORS determines whether the unique component Q_1 in Q satisfies $f(Q_1) - f(C_1) < t$ for a certain threshold value $t \in \mathbb{R}$, where C_1 denotes the first (and only) component of C. If the answer is affirmative, Q_1 replaces the current solution C_1; otherwise, C_1 remains unchanged. Several types of threshold algorithms exist; each of these is characterized by a

particular type of threshold. In iterative improvement the thresholds are 0, so that only true improvements are accepted. The deterministic iterative improvement algorithm introduced in Section 4.2 has the further restriction that neighbors are generated deterministically. In threshold accepting the thresholds are nonnegative. They are large in the beginning of the algorithm's execution and gradually decrease to become 0 in the end. General rules to determine appropriate thresholds are lacking. In simulated annealing the thresholds are positive and stochastic. Their values equal $-T \ln u$, where T is a control parameter (often called 'temperature'), whose value gradually decreases in the course of the algorithm's execution according to a 'cooling schedule', and u is drawn from a uniform distribution on (0,1]. Each time a neighbor is compared with the current solution, u is drawn again. Under certain mild conditions simulated annealing is guaranteed to find an optimal solution asymptotically.

Taboo search [Glover, 1989; Glover, 1990; Glover, Taillard and De Werra, 1993] combines the deterministic iterative improvement algorithm with a possibility to accept cost increasing solutions. In this way the search is directed away from local minima, such that other parts of the search space can be explored. This is done by selecting at each iteration a solution of minimum cost from a subset of *permissible neighbors* of the current solution. In basic taboo search a neighbor is permissible if it is not on the 'taboo list' or satisfies a certain 'aspiration criterion'. The taboo list is often implicitly defined in terms of forbidden moves from the current solution to a neighbor. It is recalculated at each iteration. The aspiration criterion expresses possibilities to overrule the taboo-status of a neighbor. For details see Glover [1989; 1990] and Glover, Taillard and De Werra [1993]. Taboo search algorithms are characterized by the following instantiations.

* GENERATE NEIGHBORS selects deterministically all neighbors of the current solution C_1 with respect to $\mathcal{N}_1$ by inspecting these in a prespecified order.
* REDUCE NEIGHBORS determines among all permissible solutions in Q a solution $Q_j \neq C_1$ of minimum cost. C_1 is then replaced by Q_j.
* CONTINUE ITERATION returns the value TRUE as long as the best solution found so far is not of a prescribed quality, or as long as a prescribed number of iterations has not yet been reached. Furthermore, when the taboo list contains all neighbors of C_1 and none of these attains the aspiration level, the function CONTINUE ITERATION returns the value FALSE.

When the taboo list contains all neighbors of a current cluster C_1 and none of these attains the aspiration level, it is impossible to determine a neighbor of C_1. Some variants of taboo search solve this problem by letting the function CON-

TINUE ITERATION return the value FALSE. Other variants modify the taboo list in such a way that neighbors of the current cluster are removed from the taboo list. In this way neighbors of C_1 become available again.

In other variants, the procedure GENERATE NEIGHBORS selects only one neighbor per iteration and the function REDUCE NEIGHBORS accepts this neighbor when it is not on the taboo list or attains the aspiration level, and rejects it otherwise. But in that case the taboo list has to be significantly larger, so as to avoid that the procedure accepts a solution with a cost larger than the current solution too often. However, in this case unacceptably large amounts of memory space and computation time would be required.

Variable-depth search algorithms. In contrast to the above algorithms, a variable-depth search algorithm creates several generations. In each generation a finite sequence of iterations is generated, in each of which a neighbor of the previous solution is computed. In principle, each neighbor chosen is a minimum cost neighbor of the previous solution. However, in this approach the risk of cycling is large. To avoid cycling, a sort of taboo list is introduced, which prevents the search from generating a solution that has occurred in the sequence before. Before starting the first iteration in a generation the taboo list is emptied and the solution contained in the single cluster is chosen from the solutions that occurred in the previous generation.

There are two main variants to choose the solution that a new generation is started with. In the first variant a solution with smallest cost is chosen among those generated in the previous generation, but it is not allowed to choose the solution that this previous solution was started with. In the second variant the first solution is chosen among those generated in the previous generation that has smaller cost than the solution that the iteration loop was started with, provided that such a solution has been found. Otherwise, an arbitrary solution obtained in the previous generation is chosen.

The instantiations for variable-depth search are as follows.

* GENERATE CLUSTERS generaties a single cluster C of size 1 that contains the single solution of P.
* GENERATE NEIGHBORS selects all neighbors of the current solution deterministically by inspecting these in a prespecified order.
* REDUCE NEIGHBORS determines among all solutions in Q not on the taboo list a solution $Q_j \neq C_1$ of minimum cost. C_1 is then replaced by Q_j.
* CREATE sets $\hat{P}$ equal to the current $C = (C_1)$, where C_1 is a solution found in the last iteration loop that is different from the solution with which the iteration loop started. We mention the following possibilities for choosing

C_1, each of which also leads to a different choice for CONTINUE ITERATION.

1. In the first variant a solution is chosen that has smallest cost among those obtained in the last iteration loop. In this case CONTINUE ITERATION returns the value TRUE as long as the number of iterations has not yet reached a specified upper bound.

2. In the second variant the first solution is chosen with smaller cost than the solution that the iteration loop was started with, provided that such a solution has been found. Otherwise, an arbitrary solution obtained in the last iteration loop is chosen. As soon as a solution is found that has smaller cost than the solution that the iteration loop was started with, the loop is terminated by letting CONTINUE ITERATION return the value FALSE.

* REDUCE POPULATION simply selects the best of the two solutions in P and $\hat{P}$. Ties are broken arbitrarily.

* In some variants CONTINUE POPULATION GENERATION returns the value TRUE as long as the sequence of the costs of solutions in P for the subsequent generations is strictly decreasing. Other variants use different rules to stop the generation of new generations.

4.4.2 Multi-level point-based local search

We now discuss point-based local search algorithms with more than one active level. Very few algorithms of this type have been proposed in the literature, and the existing ones are often tailored to a specific problem type. Algorithms of this type can usually be composed from single-level point-based local search algorithms. For this reason we do not detail the corresponding procedures and functions here.

Nevertheless, since algorithms of this kind seem to give good results, we briefly discuss an example due to Martin, Otto and Felten [1989], which is one of the first multi-level algorithms. Their algorithm for the traveling salesman problem uses, in our terminology, two active levels.

At level 1 they use simulated annealing. Their neighborhood is a subset of the 4-exchange neighborhood. After selecting a single neighbor at level 1, at level 2 they determine a local minimum with respect to a special 3-exchange neighborhood, using any single-level point-based local search algorithm that is able to do so. Then this local minimum is compared with the current solution at level 1 and is accepted using simulated annealing. The authors attribute the power of their algorithm to the fact that, after making a single 4-exchange and then applying 3-exchanges until a local optimum is reached, typically many links

in the tour have been changed. Algorithms of this type, which use more levels in the local search template, seem to be powerful and deserve wider attention.

4.4.3 Single-level population-based local search

We now discuss a class of single-level population-based local search algorithms, called *genetic* algorithms. These were first introduced by Holland [1975] and have been well described in a textbook by Goldberg [1989].

In each generation, first some clusters C of the current population P are created. To each C, the hyper-neighborhood function $\mathcal{N}_1$ is applied to produce a set of new solutions. From these new solutions and the solutions of the current population, the low cost solutions are selected to form a new population, which then starts up a next generation. The generation loop terminates as soon as some stopping criterion, which is usually chosen heuristically, is satisfied. The instantiations for the class of genetic algorithms are as follows.

* GENERATE CLUSTERS generates from the population P of size p_1 a multiset $\mathcal{C}$ of clusters C of size c_1. In most cases the clusters are formed heuristically and in such a way that solutions with lower cost are contained in a cluster with higher probability. Note that a solution in P can occur in more than one cluster and even several times in the same cluster.
* GENERATE NEIGHBORS selects randomly a number of neighbors of the current cluster C using the hyper-neighborhood function $\mathcal{N}_1$. In many implementations, this number of neighbors also equals c_1.
* REDUCE NEIGHBORS takes from the current cluster C and from Q the c_1 best solutions to form a new cluster C.
* CONTINUE ITERATION usually returns the value TRUE for the first iteration and FALSE otherwise. In this way, only one set of neighbors is generated for each chosen cluster, after which the iteration loop is left. In this case, the function REDUCE NEIGHBORS can be skipped, since there is no reason to create a new current cluster C when there is one iteration only.
* CREATE sets $\hat{P}$ equal to the union of all current clusters $C \in \mathcal{C}$.
* REDUCE POPULATION merges P and $\hat{P}$ into a new population P. In most variants, this is done by choosing from P and $\hat{P}$ exactly p_1 elements, with a preference for low-cost solutions.
* CONTINUE POPULATION GENERATION gives the value TRUE for instance as long as a certain upper bound on the number of generations has not been exceeded, or as long as the population contains different solutions.

4.4.4 Multi-level population-based local search

Few examples of population-based local search algorithms with more than one active level are known. Here we discuss the so-called *genetic local search* approach [Ulder, Aarts, Bandelt, Van Laarhoven and Pesch, 1990], which is a variant of the class of genetic algorithms. The only difference is that there is now a second active level, in which a point-based hyper-neighborhood function $\mathcal{N}_2$: $\mathcal{S} \rightarrow \mathcal{P}(\mathcal{S})$ is used.

After the computation of a tuple Q of neighbors at the first level, a local minimum is computed for each solution Q_j at the second level, using the neighborhood function $\mathcal{N}_2$. After that, back at level 1 the function REDUCE NEIGHBORS is applied to the current Q, which now contains local minima with respect to the neighborhood function $\mathcal{N}_2$. The instantiations for the second level are as follows.

* CONTINUE POPULATION GENERATION gives the value TRUE for the first generation and FALSE for the subsequent generations. In this way, only one generation is created.
* GENERATE CLUSTERS creates for each solution $P_i \in P$ a cluster $C = (P_i)$.
* CREATE sets $\hat{P}$ equal to the set that, for each cluster, contains a local minimum obtained in the iteration loop for the corresponding cluster.
* REDUCE POPULATION sets the new population P equal to $\hat{P}$.

CONTINUE ITERATION, REDUCE NEIGHBORS and GENERATE NEIGHBORS are the same as the ones specified for the deterministic iterative improvement algorithm.

4.5 Open spots in the local search template

When looking at the template one can try to find types of local search algorithms that have not been proposed before, or that have no widespread application. In this section we mention some of these algorithms, which emerge in a more or less natural way from our template. We first deal with single-level point-based algorithms and then with single-level population-based algorithms. We do not consider multi-level algorithms, since these are composed of single-level ones. However, multi-level algorithms are important, as they may have an impressive performance.

4.5.1 Single-level point-based local search

In almost all known single-level point-based algorithms also the number of clusters equals one. In this case it often happens that several neighbors of a current solution are promising while only one neighbor is allowed to be chosen. One

would like to postpone the decision of chosing the neighbor for a while until after a few iterations it becomes clear which of the neighbors is most promising. This idea fits in our template in the following way.

At the beginning of a generation several clusters are generated, each containing a copy of the solution in the current one-element population P. Then for each cluster an iteration loop is started. When this loop terminates depends on how CONTINUE ITERATION is chosen. Reasonable points for terminating the iteration loop are when a local optimum has been found or when a given number of iterations has been executed. When for each cluster its corresponding iteration loop is terminated, a solution must be selected to become the new solution in P. It is reasonable to select the best solution that resulted from the various iteration loops in the last generation. Next, a new generation is started up.

4.5.2 Single-level population-based local search

Below we give some ideas for single-level population-based local search algorithms. A distinction is made between algorithms with cluster size 1, cluster size 2, and cluster size larger than 2.

When the cluster size is 1, only ordinary neighborhoods can be used. Since we have a population-based algorithm, it is reasonable to let the number of clusters be larger than 1. The idea of the following algorithm is that several parallel runs of a point-based algorithm are interrupted now and then, and that the runs with the worse results are stopped definitely. The remaining runs are continued in such a way that one run may proceed in several directions. It is therefore necessary that the point-based algorithm uses some type of randomization. This idea fits in our template in the following way.

At the beginning of a generation several clusters are generated, each containing a copy from one of the solutions in the current population P. Then for each cluster an iteration loop is started, using a point-based algorithm. When this loop terminates after the execution of a given number of iterations, a population $\hat{P}$ is formed from the final or from intermediate solutions obtained in the previous iterations. The populations P and $\hat{P}$ are then merged into a new population P, using a selection criterion like those used in genetic algorithms.

All present genetic algorithms use hyper-neighborhoods that are based on clusters of size 2. Furthermore, in a given generation, for each cluster there is exactly one iteration in which the hyper-neighborhood is applied. One can think of algorithms that use more than one iteration for the same cluster in a given generation. The execution of iterations in one generation can be stopped after a certain number of iterations. One can also use another stop criterion. Therefore, we define a cluster to be locally optimal with respect to a hyper-neighborhood $\mathcal{N}$ if it

has no neighbor in $\mathcal{N}$ that has lower cost than the solutions of this cluster. Now, one can stop the execution of iterations as soon as the current cluster is locally optimal. This idea fits into our template in an obvious way.

Until now, no genetic algorithms have been proposed that use hyper-neighborhoods that are based on clusters of size larger than 2. For many standard hyper-neighborhoods based on cluster size 2, it is not difficult to generalize them to hyper-neighborhoods based on cluster size 3 or on even larger cluster sizes. Whether these generalizations will give computational results of the same quality as those for hyper-neighborhoods based on cluster size 2, is not clear at present. Here again, this idea obviously fits in our template.

4.6 The complexity of local search

Complexity analyses have reveiled a marked difference between the theoretical and empirical performance of local search.

4.6.1 Theoretical results

Studies of the theoretical performance of local search have exhibited its limitations, at least from a worst-case point of view. The literature presents a number of bad examples, for which the following results hold:

* Minimum exact neighborhoods may be of exponential size.
* It may take an exponential number of steps to find a local optimum.
* Final solutions may deviate arbitrarily far from the optimum in cost.

Johnson, Papadimitriou and Yannakakis [1988] addressed the question how easy it is to find a local optimum. They introduced several notions. A local search problem L is given by a set of instances, each of which is defined by a finite set of solutions, a cost function, and a neighborhood function. This problem belongs to the complexity class $\mathcal{PLS}$ of 'polynomial-time local search' problems, if polynomial-time algorithms exist, only depending on L, for producing an arbitrary solution, for computing the cost of a given solution, and for determining for a given solution a neighbor that has lower cost (or reporting that no such neighbor exists). The problem now is to find a local minimum for any given instance. Informally speaking, $\mathcal{PLS}$ defines the class of local search problems for which local optimality can be verified in polynomial time. The class $\mathcal{PLS}$ is situated between the search problem variants of $\mathcal{P}$ and $\mathcal{NP}$. However, it has been shown that a problem in $\mathcal{PLS}$ cannot be NP-hard, unless $\mathcal{NP}$ =*co*-$\mathcal{NP}$ [Johnson, Papadimitriou and Yannakakis, 1988].

Furthermore, the concept of a PLS-reduction has been introduced, which is similar to the classical concept of polynomial-time reductions [Garey and Johnson, 1979]. A problem in $\mathcal{PLS}$ is PLS-complete if any problem in $\mathcal{PLS}$ is PLS-reducible to it. The PLS-complete problems are the hardest ones in $\mathcal{PLS}$, and if one of them can be shown to be solvable in polynomial time, then all the others can.

Since its introduction, the class $\mathcal{PLS}$ has received considerable attention, and many local search problems have been proven PLS-complete. It is even conjectured that PLS-completeness is the normal behavior of local search variants of NP-hard problems and that more than half of the problems mentioned in Garey and Johnson's [1979] NP-completeness catalogue have PLS-complete variants.

4.6.2 Empirical results

Over the years the empirical performance of local search has been extensively studied for a large variety of problems. A general conclusion is that local search algorithms can find good solutions within low order polynomial running times. This conclusion has been reached in studies for problems of a theoretical as well as practical origin. For instance, for the traveling salesman and job shop scheduling problems, local search algorithms have been shown to be the best approximation algorithms from an empirical point of view, and it appears that for large problem instances, the difference with other existing algorithms becomes even more pronounced. For many practical problems, for instance in the areas of logistics, VLSI design, and production planning, local search leads to good solutions in a reasonable amount of time. In addition, its ease of use makes it a flexible industrial problem solving tool.

5

Local search for the job shop scheduling problem

In this chapter we survey solution methods for the standard job shop scheduling problem with an emphasis on local search. Both deterministic and randomized local search methods as well as the proposed neighborhoods are discussed. We compare the computational performance of the various methods in terms of their effectiveness and efficiency on a standard set of problem instances.

In Chapter 3 it was mentioned that the job shop scheduling problem is difficult to solve to optimality. This is witnessed by the fact that a relatively small instance with ten jobs, ten machines and hundred operations due to Fisher and Thompson [1963] remained unsolved until 1986. Many solution methods have been proposed, ranging from simple and fast dispatching rules to sophisticated branch-and-bound algorithms. During the last decade many different types of local search algorithms have been developed, and some of them have proved to be very effective.

This chapter is structured as follows. Section 5.1 deals with some preliminaries. In Section 5.2 methods for the solution of the job shop scheduling problem are reviewed. The remainder of the chapter focuses on local search methods to tackle the job shop scheduling. Section 5.3 discusses representations and neighborhoods for the problem. Sections 5.4 and 5.5 describe constructive and iterative algorithms with local search, respectively; Section 5.6 describes some other

techniques. Section 5.7 contains computational results, and Section 5.8 gives some concluding remarks.

5.1 Preliminaries

The job shop scheduling problem is a special case of the generalized job shop scheduling problem, in which each operation v has to be processed by a given single machine $\mu(v)$. The precedence relation A decomposes V into chains of operations. It is not allowed that two consecutive operations of a chain have to be processed by the same machine.

Clearly, since each operation has to be processed by a machine that is known in advance, the only problem is to find a start time function S or, equivalently, an orientation Ω of the edge set $E = \{\{v, w\} \mid v, w \in V, v \neq w, \mu(v) = \mu(w)\}$.

In this chapter we use the notation introduced in Subsection 2.1.4, but we omit the machine set assignment K. For instance, a partial solution is only characterized by a partial orientation $\Omega|_F$ for some subset F of E; the corresponding solution graph is then denoted by $\mathcal{G}_{\Omega|_F} = (V, \mathcal{A} \cup \Omega|_F(F))$.

5.2 Solution approaches

We give a brief review of the lower bounds and enumeration schemes that are used in branch-and-bound methods, and of the approximative approaches that yield upper bounds on the optimum. Techniques of the latter type that proceed by local search are discussed in the later sections of this chapter. Results on the computational complexity of the job shop scheduling problem have been discussed in Chapter 3.

5.2.1 Lower bounds

Optimization algorithms for the problem employ some form of tree search. A node in the tree is usually characterized by a partial orientation Ω on a subset $F \subset E$. The question is then how to compute a lower bound on the length of any feasible schedule corresponding to a completion of Ω.

Németi [1964] and many subsequent authors obtained a lower bound by simply disregarding $E \setminus F$ and computing the longest path length in the digraph $\mathcal{G}_{\Omega|_F}$.

Bratley, Florian and Robillard [1973] obtained the stronger *single-machine bound* by relaxing the capacity constraints of all machines except one. Given a machine M', they propose to solve the job shop scheduling problem on the disjunctive graph $(V, \mathcal{A} \cup \Omega|_F(F), \{\{v, w\} \mid \mu(v) = \mu(w) = M'\} \setminus F)$. This is a single-machine problem, where the arcs in $\mathcal{A} \cup \Omega|_F(F)$ define release and delivery times for the operations on M' and precedence constraints between them. Lageweg, Lenstra and Rinnooy Kan [1977] pointed out that many other lower

bounds appear as special cases of this bound. For example, relaxing the capacity constraint of M' gives Németi's bound, and allowing preemption gives the bound used in current branch-and-bound codes. The bound itself is NP-hard to compute but can be found fairly efficiently [Baker and Su, 1974; McMahon and Florian, 1975; Lageweg, Lenstra and Rinnooy Kan, 1976; Carlier, 1982]. It has been strengthened by Carlier and Pinson [1990], who compute larger release and delivery times, and by Tiozzo [1988] and Dauzere-Peres and Lasserre [1993], who observe that the arcs also define delays between precedence-related operations; Balas, Lenstra and Vazacopoulos [1995] develop an algorithm for computing the bound subject to these delayed precedences.

Fisher, Lageweg, Lenstra and Rinnooy Kan [1983] investigated surrogate duality relaxations, in which either the machine capacity constraints or the precedence constraints among the operations of a job are weighted and aggregated into a single constraint. Balas [1985] described a first attempt to obtain bounds by polyhedral techniques. Applegate and Cook [1991] review the valid inequalities studied before and gave some new ones. The computational performance of surrogate duality and polyhedral bounds reported until now is disappointing in view of what has been achieved for other hard problems.

5.2.2 Enumeration schemes

The traditional enumeration scheme generates all active schedules by constructing them from front to back [Giffler and Thompson, 1960]. At each node a machine on which the earliest possible completion time of any unscheduled operation is achieved is determined, and all unscheduled operations that can start earlier than this point in time on that machine are selected in turn.

In recent branch-and-bound algorithms more flexible enumeration schemes are used. Carlier and Pinson [1989; 1990; 1994] and Applegate and Cook [1991] branch by selecting a single edge and orienting it in either of two ways. Brucker, Jurisch and Sievers [1994] follow Grabowski's 'block approach'. All these authors apply the preemptive single-machine bound and a host of elimination rules. For details we refer to the literature.

The celebrated 10×10 instance of Fisher and Thompson [1963] is within easy reach of these methods, but 15×15 instances seem to be the current limit. The main deficiency of the existing optimization algorithms for job shop scheduling is the weakness of the lower bounds. The situation is much brighter with respect to finding good upper bounds.

5.2.3 Upper bounds

Upper bounds on the optimum are usually obtained by generating a schedule and computing its length. An obvious first step is to apply a dispatch rule and to

schedule the operations according to some priority function. Haupt [1989] surveys such rules. They tend to exhibit an erratic behavior; the procedure 'bidir' proposed by Dell'Amico and Trubian [1993] is one of the safer alternatives. The next step is then to try to improve the schedule by some sort of local search.

An entirely different approach is taken by Sevast'janov [1994]. Using the vector sum theorem of Steinitz, he develops polynomial-time algorithms for finding an upper bound with an absolute error that is independent of the number of jobs. Shmoys, Stein and Wein [1994] improve on his results.

5.3 Solution representations and neighborhood functions

A local search algorithm consists of the following main ingredients: a representation of the solutions, a neighborhood function, and a search strategy to guide the search in the solution space by means of the exploration of the neighborhoods. In this section, several basic representations and neighborhood functions are introduced for the job shop scheduling problem. The next section deals with the search strategies.

For most threshold and taboo search algorithms, only left-justified or active schedules are represented. This is done by specifying the start times of the operations or, equivalently, the corresponding machine orderings of the operations. Also other representations are used, especially in the context of genetic algorithms.

To be able to define the neighborhood functions, we need some extra notions. Given an instance and an operation v, $jp(v)$ and $js(v)$ denote the immediate predecessor and successor of v in the precedence relation A, provided they exist. Given a feasible schedule S and an operation v, $mp_S(v)$ and $ms_S(v)$ denote the immediate predecessor and successor of v in the orientation Ω_S, provided they exist. If the schedule S is clear from context, we delete the superscript S. Furthermore, $jp^2(v)$ denotes $jp(jp(v))$, provided it exists, and a similar notation is used for js, mp_S and ms_S. Two operations v and w are *adjacent* when $S(v) + p(v) = S(w)$. A *block* is a maximal sequence of size at least one, consisting of adjacent operations that are processed on the same machine and belong to a longest path. An operation of a block is *internal* if it is neither the first nor the last operation of that block.

Several neighborhood functions have been proposed in the literature. Most of these are not defined on a schedule S itself but on the corresponding orientation Ω_S. If Ω_S is changed into another feasible orientation Ω', $S_{\Omega'}$ is the corresponding neighbor of S. In this way neighbors of a given schedule are always left-justified.

The following properties [Balas, 1969; Matsuo, Suh and Sullivan, 1988;

Nowicki and Smutnicki, 1995] are helpful in obtaining reasonable neighborhood functions.

1. Given a feasible orientation, reversing an oriented edge on a longest path in the corresponding digraph results again in a feasible orientation.
2. If reversing a non-critical oriented edge of a feasible orientation Ω results in a feasible orientation Ω', then $S_{\Omega'}$ is at least as long as S_{Ω}.
3. Given a feasible orientation Ω, reversing an oriented edge (v, w) between two internal operations of a block results in a feasible schedule at least as long as S_{Ω}.
4. Given is a feasible orientation Ω. Let v and w be the first two operations of the first block of a longest path, and let w be an internal operation. Reversing (v, w) results in a feasible schedule at least as long as S_{Ω}. The same is true in case v and w are the last two operations of the last block of a longest path and v is internal.

In view of these properties, the simplest neighborhood functions are based on the reversal of exactly one edge of a given orientation. Van Laarhoven, Aarts and Lenstra [1992] propose a neighborhood function N_1, which obtains a neighbor by interchanging two adjacent operations of a block. Matsuo, Suh and Sullivan [1988] use a neighborhood function N_{1a} with the same interchanges, except those involving two internal operations. Nowicki and Smutnicki [1995] use a neighborhood function N_{1b}, excluding from N_{1a} the interchange of the first two operations of the first block when the second one is internal and the interchange of the last two operations of the last block when the first is internal. For N_{1b}, neither a schedule with only one block nor one with only blocks of size one has a neighbor; note that such schedules are optimal.

Dell'Amico and Trubian [1993] propose several neighborhood functions that may reverse more than one edge. Their neighborhood function N_2 obtains, for any two operations v and $w = ms(v)$ on a longest path, a neighboring orientation by permuting $mp(v)$, v and w, or by permuting v, w and $ms(w)$, such that v and w are interchanged and a feasible orientation results. Their neighborhood function N_{2a} excludes from N_2 the solutions for which both v and $w = ms(v)$ are internal. Their neighborhood function N_3 considers blocks of size at least two: a neighbor is obtained by positioning an operation v immediately in front of or after the other operations of its block, provided that the resulting orientation is feasible; otherwise, v is moved to the left or to the right as long as the orientation remains feasible.

While the above neighborhood functions are based on adjacent interchanges or *swaps*, Balas and Vazacopoulos [1994] propose a neighborhood function N_4

that uses reinsertions or *jumps*. More precisely, N_4 considers any two operations v and w on the same machine such that v occurs prior to w on a longest path. A neighbor is obtained by inserting v immediately after w or w immediately before v. Sufficient conditions are derived under which these new schedules are feasible. If $mp(v)$ and $ms(w)$ are both on a longest path, they cannot improve the current schedule and are disregarded.

Adams, Balas and Zawack [1988] propose a neighborhood function N_5, in which one machine ordering may be changed completely. For every machine M' with an operation on a longest path, a neighbor is obtained by replacing the orientation on M' by any other feasible orientation.

In the following neighborhood functions a neighbor is obtained by changing several machine orderings at the same time. Relatively small modifications are made by the neighborhood function N_6 of Matsuo, Suh and Sullivan [1988], which reorients at most three edges simultaneously. A neighbor is obtained by interchanging two adjacent operations v and $w = ms(v)$ of a block (except when they are both internal) and in addition by interchanging $jp^t(w)$ and $mp(jp^t(w))$ for some $t \geq 1$ and by interchanging $js(v)$ and $ms(js(v))$. The latter interchanges are executed only if certain additional conditions are satisfied; see their paper for details. Aarts, Van Laarhoven, Lenstra and Ulder [1994] use a variant N_{6a} with $t = 1$.

Applegate and Cook [1991] propose a neighborhood function N_7 that drastically changes the given orientation. Their neighborhood contains all feasible orientations that can be obtained by simultaneously replacing the orientation on $m - t$ machines by any other feasible orientation. Here, t is a small number depending on m.

Storer, Wu and Vaccari [1992] use completely different representations of schedules. These are based on a modified version of the Giffler-Thompson algorithm (see Subsection 5.2.2). Suppose that at a certain point the earliest possible completion time of any unscheduled operation is equal to C and is achieved by operation v, and that T is the earliest possible start time on machine $\mu(v)$. Then all unscheduled operations on $\mu(v)$ that can start no later than $T + \delta(C - T)$ are candidates for the next position on $\mu(v)$. Here, δ is a priori chosen in [0,1) (in experiments 0, 0.05 or 0.1); if δ approaches 1 all active schedules can be generated, while $\delta = 0$ gives only so-called *undelayed* schedules. Two representations are defined.

The representation R_8 represents a schedule by modified processing times for the operations. Using these, the modified Giffler-Thompson algorithm with the shortest processing time rule as selection rule uniquely determines a feasible orientation Ω, and S_Ω, computed with the original processing times, is the

corresponding schedule. The neighborhood function N_8 now obtains a neighbor by increasing the processing times by amounts of time that are independently drawn from a uniform distribution on $(-\theta, \theta)$. Here, θ is a priori chosen (in experiments 10, 20 or 50). The representation R_9 represents a schedule by dividing the scheduling horizon into several time windows (in experiments 5, 10 or 20) and assigning one of a given set of dispatch rules to each window. The modified Giffler-Thompson algorithm determines a schedule by applying the dispatch rule of the corresponding window. The neighborhood function N_9 changes the dispatch rule for a window of a given schedule.

Genetic algorithms use two types of representations: the natural one, which is also used for the other algorithms, and the more artificial 'string representations'.

For the former type of representation Yamada and Nakano [1992] propose a hyperneighborhood function N_{h1}. Given two schedules S and S', N_{h1} determines a neighbor using the Giffler-Thompson algorithm. When this algorithm has to choose from two or more operations, it takes, for a small $\varepsilon > 0$, the operation that is first in S with probability $(1 - \varepsilon)/2$, the operation that is first in S' with probability $(1-\varepsilon)/2$, and a random operation from the other available operations with probability ε.

Aarts, Van Laarhoven, Lenstra and Ulder [1994] propose a hyperneighborhood function N_{h2}. Given two schedules S and S', N_{h2} determines a neighbor by repeating the following step $\lfloor nm/2 \rfloor$ times: choose a random arc (w, v) of S' and change S by reversing arc (v, w), provided it belongs to a longest path of S.

The latter type of representation encodes a schedule or its orientation into a string over a finite – usually binary – alphabet. Such representations facilitate the application of hyperneighborhood functions involving operations like 'crossover' and 'mutation'; see Goldberg [1989, pp. 166-175]. There are a number of drawbacks, however. A schedule or orientation may have several representatives, or none. Conversely, a string does not have to represent a schedule, and if it does, it may be nontrivial to calculate the corresponding schedule. Although attempts have been made to circumvent these difficulties, the hyperneighborhood functions that operate on strings often have no meaningful effect in the context of the underlying problem. We will consider genetic algorithms using string representations in less detail.

5.4 Constructive algorithms with local search

This section deals with the *shifting bottleneck procedure* and its variants. These algorithms construct a complete schedule and apply local search to partial schedules on the way.

The basic idea of the algorithms described here is as follows. The algorithm goes through m stages. At each stage, it orients all edges between operations on a specific machine. In this way, at the beginning of any stage all edges related to some machines have been oriented, while the edges related to the other machines are not yet oriented. Furthermore, at the end of each stage, it reoptimizes the current partial schedule. This is usually done by applying iterative best improvement using neighborhood function N_5, which revises the orientation on a machine scheduled before. Orienting or reorienting the edges related to one machine in an optimal way requires the solution of a single-machine problem, where the partial schedule defines release and delivery times and delayed precedence constraints. The algorithms discussed hereafter mainly differ by the order in which the m machines are considered, by the implementation of iterative best improvement, and by the single-machine algorithm used.

The original shifting bottleneck procedure SB1 of Adams, Balas and Zawack [1988] orients at each stage the edges related to the *bottleneck machine*. This is the unscheduled machine for which the solution value to the corresponding single-machine problem is maximum; the delays between precedence-related operations are not taken into account. After scheduling a machine, iterative best improvement is applied during three cycles. In each cycle each scheduled machine is reconsidered once. The first cycle handles the machines in the order in which they were sequenced. After a cycle is completed, the machines are reordered according to decreasing solution values to the single-machine problems in the last cycle. When all of the machines have been scheduled, the cycles continue as long as improvements are found. Furthermore, after a phase of iterative best improvement, the orientations on several machines that have no operations on a longest path are deleted, and then these machines are rescheduled one by one.

Applegate and Cook [1991] use almost the same algorithm. The main difference is that at each stage iterative improvement cycles continue until no improvement is found.

Dauzere-Peres and Lasserre [1993] were the first to take the delays between precedence-related operations into account. They develop a heuristic for the single-machine problem with delayed precedences and incorporate it into a shifting bottleneck variant.

Balas, Lenstra and Vazacopoulos [1995] use their optimization algorithm for the single-machine problem with delayed precedences to determine the bottleneck machine in their procedure SB3. Their local search strategy differs from the one of Adams, Balas and Zawack [1988] in some minor details; for instance, the number of cycles is limited to six. Again, after scheduling a new machine, they

first apply iterative improvement, then delete the orientations on several non-critical machines, and reschedule these machines one by one. Their extended procedure SB4 takes the best solution of SB3 and a variant of SB3 that reverses the order of the two reoptimizations procedures: first reschedule some non-critical machines, then apply regular iterative improvement.

The procedure SB-GLS proposed by Balas and Vazacopoulos [1994] is rather different. It reoptimizes partial schedules by applying their variable-depth search algorithm GLS (see Subsection 5.5.3) for a limited number of iterations using the jump neighborhood function N_4.

The shifting bottleneck procedure and its variants have been incorporated into other algorithms. Most of these employ some form of partial enumeration. Dorndorf and Pesch [1995] embed a variant in a genetic algorithm; see Section 5.5.4.

Adams, Balas and Zawack [1988] develop an algorithm PE-SB, which applies SB1 to the nodes of a partial enumeration tree. A node corresponds to a subset of machines that have been scheduled in a certain way. In each of its descendants one more machine is scheduled. The schedule is obtained by first solving the single-machine problem, with release and delivery times defined by the parent node, and then applying iterative improvement as in SB1. Descendants are created only for a few machines with highest solution values to the single-machine problem. A penalty function is used to limit the size of the tree. For details about the branching rule, the penalty function and the search strategy we refer the reader to the original paper.

Applegate and Cook [1991] develop an algorithm Bottle-t, which employs partial enumeration in a different way. Bottle-t applies their shifting bottleneck variant described above as long as more than t machine are left unscheduled. For the last t machines it branches by selecting each remaining unscheduled machine in turn. The values $t = 4$, 5 and 6 were tested.

5.5 Iterative algorithms with local search

The algorithms presented in this section start from one or more given feasible schedules and manipulate these in an attempt to find better schedules. They can naturally be divided into threshold algorithms, taboo search algorithms, variable-depth search algorithms, and genetic algorithms.

5.5.1 Threshold algorithms

The basic threshold algorithms are iterative improvement, threshold accepting, and simulated annealing. We also consider some closely related variants. Unless stated otherwise, a schedule is represented in the ordinary way by the starting

times or the orientation.

Iterative improvement is the simplest threshold algorithm. Aarts, Van Laarhoven, Lenstra and Ulder [1994] test iterative improvement with the neighborhood functions N_1 and N_{6a}. To obtain a fair comparison with other algorithms they apply a *multi-start* strategy, that is, they run the algorithm with several randomly generated start solutions until a limit on the total running time is reached, and take the best solution found over all individual runs.

The algorithm Shuffle of Applegate and Cook [1991] uses the neighborhood function N_7. At each iteration, the schedule on a small number of heuristically selected machines remains fixed, and the schedule on the remaining machines is optimally revised by their branch-and-bound algorithm 'edge finder'. As initial solution they take the result of Bottle-5.

Storer, Wu and Vaccari [1992] propose a variant of iterative improvement, called PS10, with representation R_8 and neighborhood function N_8. Given a solution, the function GENERATE NEIGHBORS determines a fixed number of neighbors (in experiments 100 or 200), the best one of which becomes the new solution. They also test a standard iterative first improvement algorithm, called HSL10, with representation R_9 and neighborhood function N_9. Neighbors are generated randomly; CONTINUE ITERATION gets the value FALSE after a fixed number of iterations (in experiments 1000 or 2000).

Threshold accepting has only been implemented by Aarts, Van Laarhoven, Lenstra and Ulder [1994]. Their algorithm TA1 uses the neighborhood function N_1. Threshold values are determined empirically.

Simulated annealing has been tested by several authors. Van Laarhoven, Aarts and Lenstra [1992] use the neighborhood function N_1. Aarts, Van Laarhoven, Lenstra and Ulder [1994] use N_1 (algorithm SA1) and N_{6a} (algorithm SA2).

Matsuo, Suh and Sullivan's [1988] 'controlled search simulated annealing' algorithm SA-II is a bi-level variant, which also incorporates deterministic iterative improvement. Given a schedule S, a neighbor S' is selected using the neighborhood function N_6. S' is accepted or rejected by the simulated annealing criterion. In the latter case, S' is subjected to deterministic iterative improvement using N_6 again, and if the resulting local optimum improves on S, it is accepted as the new solution. The algorithm also differs from most other implementations of simulated annealing in that the acceptance probability for a schedule that is inferior to the current schedule is independent of the difference in schedule length.

5.5.2 Taboo search algorithms

The taboo search algorithm TS1 of Taillard [1994] uses the neighborhood function N_1. After an arc (v, w) has been reversed, the interchange of w and its ma-

chine successor is put on the taboo list. Every 15 iterations a new length of the taboo list is randomly selected from a range between 8 and 14. The length of a neighbor is estimated in such a way that the estimate is exact when both operations involved are still on a longest path, and that it is a lower bound otherwise. Then, from the permissible neighbors the schedule of minimum estimated length is selected as the new schedule.

The algorithm TS2 of Barnes and Chambers [1995] also uses N_1. Their taboo list has a fixed length. If no permissible moves exist, the list is emptied. The length of each neighbor is calculated exactly, not estimated. A start solution is obtained by taking the best from the active and undelayed schedules obtained by applying seven dispatch rules.

The algorithm TS3 of Dell'Amico and Trubian [1993] uses the union of the neighborhoods generated by N_{2a} and N_3. The items on the taboo list are forbidden reorientations of arcs. Depending on the type of neighbor, one or more such items are on the list. The length of the list depends on the fact whether the current schedule is shorter than the previous one and the best one, or not. Furthermore, the minimal and maximal allowable lengths of the list are changed after a given number of iterations. When all neighbors are taboo and do not satisfy the aspiration criterion, a random neighbor is chosen as the next schedule. A start solution is obtained by a procedure called 'bidir', which applies list scheduling simultaneously from the beginning and the end of the schedule.

The algorithm TS-B of Nowicki and Smutnicki [1995] combines taboo search with a backtracking scheme. In the taboo search part of their algorithm, the neighborhood function is a variant of N_{1b}, which only allows reorientations of arcs on a single longest path. The items on the taboo list are forbidden reorientations of arcs. The length of the list is fixed to 8. If no permissible neighbor exists, the following is done. If there is one neighbor only, which as a consequence is taboo, this one becomes the new schedule. Otherwise, the oldest items on the list are removed one by one until there is one non-taboo neighbor, and this one is chosen. A start solution is obtained by generating an active schedule using the shortest processing time rule or an insertion algorithm.

The backtracking scheme forces the taboo search to restart from promising situations encountered before. Each such restart corresponds with a new generation in the generation-loop. Each generation consists of one ordinary taboo search. At the start of a generation the population P contains at most 5 schedules. At the start of the first generation it contains only one schedule. At the start of each generation GENERATE CLUSTERS selects the best schedule contained in P. With this schedule a taboo search is started. When this taboo search stops, since a maximum number of iterations was reached without improving the best

schedule found in the current generation, the following is done. CREATE stores all schedules in the population $\hat{P}$ that improved on the previous best schedule of this generation and had more than one neighboring schedule. REDUCE POPULATION merges P and $\hat{P}$ such that the updated population P contains the best solutions of P and $\hat{P}$. However, when for a schedule from this population each possible neighbor has been considered in some generated, this schedule is not allowed to participate in the updated population P. Furthermore, information is stored about these solutions, such that the taboo search in a next generation will be directed into direction that is different from the direction of the taboo search in the generation this solution was found for the first time.

5.5.3 Variable-depth search algorithms

Balas and Vazacopoulos' [1994] *guided local search* algorithm GLS is based on the neighborhood function N_4. It differs from the standard variable-depth search introduced in Chapter 4 in the sense that trees are used instead of sequences. Each node of the tree corresponds to an orientation, and each child node is a neighbor of its parent. The number of children of a parent is restricted by a decreasing function of the level in the tree. The children are selected using estimates of the lengths of the associated schedules. When a node is obtained by inserting an operation just before or after another one, their relative order remains fixed in the orientation of all of its descendants. Other parts of the orientation are fixed too to ensure that each orientation occurs in at most one node in the tree. The size of the tree is kept small by limiting the number of children, by fixing parts of the orientations, and by bounding the depth of the tree by a logarithmic function of the number of operations. From the nodes in the tree one with smallest length is chosen as the root node for the next tree, provided that it has a smaller length than the root of the current tree. Otherwise, a node is chosen that differs at least a certain number of reinsertions from the root, where a node with smaller length is chosen with higher probability. A start solution is generated by a randomized dispatch rule.

Balas and Vazacopoulos [1994] propose two variants, which we call *iterated* and *reiterated* guided local search, or IGLS and RGLS-k. Starting from a solution generated by SB-GLS (see Section 5.4), IGLS repeats reoptimization cycles until no improvement is found. Each of these cycles removes the orientation on one machine, applies GLS for a limited number of trees, then adds the removed machine again, and applies GLS to the complete schedule for a limited number of trees. RGLS-k starts from a solution obtained by IGLS and repeats k cycles of the following type: remove the orientations on $\lfloor\sqrt{m}\rfloor$ randomly chosen machines, apply GLS for a limited number of trees, then add the removed machines

again by applying SB1 (see Section 5.4), and finally apply IGLS.

5.5.4 Genetic algorithms

The genetic algorithm GA1 of Yamada and Nakano [1992] determines, for every chosen pair of schedules of the current population, two hyperneighbors by using N_{h1}. From these four schedules two are selected for the next population: first the best schedule is chosen, and next the best unselected hyperneighbor is chosen.

Aarts, Van Laarhoven, Lenstra and Ulder [1994] propose a genetic algorithm that incorporates iterative first improvement. In each iteration there is a population of solutions that are locally optimal with respect to either N_1 (algorithm GA-II1) or N_{6a} (algorithm GA-II2). The population is doubled in size by applying N_{h2} to randomly selected pairs of schedules of the population. Each hyperneighbor is subjected to iterative first improvement, using N_1 or N_{6a}, and the extended population of local optima is reduced to its original size by choosing the best schedules. Then a next iteration is started. Start solutions are generated randomly, and iterative first improvement is applied to them before the genetic algorithm is started.

In the work of Davis [1985], Falkenauer and Bouffouix [1991] and Della Croce, Tadei and Volta [1995] a string represents for each machine a preference list, which defines a preferable ordering of its operations. From such a list a schedule is calculated. Davis [1985] and Falkenauer and Bouffouix [1991] restrict themselves to undelayed schedules; Della Croce, Tadei and Volta [1995] are able to represent other schedules as well. Falkenauer and Bouffouix [1991] and Della Croce, Tadei and Volta [1995] use the linear order crossover as hyperneighborhood function. See the original papers for details.

Nakano and Yamada [1991] consider problem instances with exactly one operation for each job-machine pair. For each machine and each pair of jobs, they represent the order in which that machine executes those jobs by one bit. Thus, a schedule is represented by a string of $mn(n-1)/2$ bits. Since such a string may not represent a feasible orientation, they propose a method for finding a feasible string that is close to a given infeasible one. Two hyperneighbors are obtained by cutting two strings at the same point and exchanging their left parts.

Dorndorf and Pesch [1995] propose a 'priority rule based genetic algorithm' GA-P, which uses the Giffler-Thompson algorithm. Each element p_i in their string $(p_1, \ldots, p_{l-1})$ denotes a dispatch rule that resolves conflicts in the ith iteration of the algorithm. Two hyperneighbors are obtained by cutting two strings at the same point and exchanging their left parts. These authors also propose a second genetic algorithm, called GA-SB, which uses a shifting bottleneck procedure (see Section 5.4). A solution is represented by a sequence of the machines.

A corresponding schedule is generated by a variant of SB1: each time it has to select an unoriented machine, it chooses the first unoriented machine in this sequence. The hyperneighborhood function used is the cycle crossover; see Goldberg [1989, p. 175]. In contrast to SB1, reoptimization is applied only when less than six machines are left unscheduled.

5.6 Other techniques

5.6.1 Constraint satisfaction

Constraint satisfaction algorithms consider the decision variant of the job shop scheduling problem: given an overall deadline, does there exist a feasible schedule meeting the deadline? Most algorithms of this type apply tree search and construct a schedule by assigning start times to the operations one by one. A *consistency checking* process removes inconsistent start times of not yet assigned operations. If it appears that a partial schedule cannot be completed to a feasible one, a *dead end* is encountered, and the procedure has to undo several assignments. Variable and value ordering heuristics determine the selection of a next operation and its start time. The algorithm stops when a feasible schedule meeting the deadline has been found or been proved not to exist. Note that it is also possible to establish lower bounds on the optimum with this technique.

Sadeh [1991] developed an algorithm of this type, but its performance was poor. Nuijten and Aarts [1995] designed new variable and value orderings and extensive consistency checking techniques. They restart the search from the beginning when a dead end occurs, and they also randomize the selection of a next operation and its start time. Their 'randomized constraint satisfaction' algorithm RCS performs quite well.

5.6.2 Neural networks

Foo and Takefuji [1988a, 1988b] describe a solution approach based on the deterministic neural network model with a symmetrically interconnected network, introduced by Hopfield and Tank [1985]. The job shop scheduling problem is represented by a 2-dimensional matrix of neurons. Zhou, Cherkassky, Baldwin, and Olson [1991] develop a neural network algorithm which uses a linear cost function instead of a quadratic one. For each operation there is one neuron in the network, and also the number of interconnections is linear in the number of operations. The algorithm improves the results of Foo and Takefuji both in terms of solution quality and network complexity. Altogether, applications of neural networks to the job shop scheduling problem are at an initial stage, and the reported computational results are poor up to now.

5.7 Computational results

The computational merits of job shop scheduling algorithms have often been measured by their performance on the notorious 10×10 instance FT10 of Fisher and Thompson [1963]. Applegate and Cook [1991] found that several instances of Lawrence [1984] (LA21, LA24, LA25, LA27, LA29, LA38, LA40) pose a more difficult computational challenge. We have included the available computational results for these instances and, in addition, for two relatively easy instances (LA2, LA19) and for all remaining 15×15 instances of Lawrence (LA36, LA37, LA39). Each of these thirteen instances has exactly one operation for each job-machine pair.

Tables 5.1, 5.2, 5.3, 5.4, and 5.5 present the computational results for most algorithms discussed in Sections 5.4, 5.5, and 5.6, as far as these are available. All results were taken from the literature, with the exception of the results for the algorithms of Applegate and Cook [1993], which we obtained using their codes. Tables 5.1, 5.2, and 5.3 give the individual results for the thirteen instances, Table 5.4 aggregates these results, and Table 5.5 contains the results for four algorithms that were only tested on instance FT10. Schedule lengths are printed in roman, computation times have been measured in CPU-seconds and are printed in italic, and blank spaces denote that no results are available.

In Tables 5.1, 5.2, and 5.3, the values LB and UB are the best known lower and upper bounds on the optimal schedule lengths. We ran the 'edge finder' algorithm of Applegate and Cook [1993] for the instances LA21, LA29, and LA38 to obtain better lower bounds than were known before; for LA21 and LA38 this resulted in optimality proofs. The best upper bounds are obtained by one or more of the algorithms in the tables, except the one for LA27, which was found by Carlier and Pinson [1994], and for LA40, which is due to Applegate and Cook [1991]. Note that all instances but one have been solved to optimality. For each of the included algorithms, a superscript b followed by a number x indicates that the schedule lengths reported are the best ones obtained after x runs of the algorithm; the computation time is the total time over all runs. A superscript m indicates that the schedule lengths are means over several runs; in this case, the computation time is the average over these runs. A superscript 1 refers to a single run.

For each algorithm and each instance, we computed the relative error, that is, the percentage that the schedule length reported is above LB. Table 5.4 presents, for each algorithm, the mean and the standard deviation of these relative errors. Note that UB has already a mean relative error of 0.18. Table 5.4 also gives, for each algorithm, the sum of the computation times for the thirteen instances, the computer used, and a computer independent sum of computation times. The

Table 5.1: Results for three instances.

algorithm	authors	FT10		LA2		LA19	
n		10		10		10	
m		10		5		10	
LB		930		655		842	
UB		930		655		842	
shifting bottleneck							
SB1[1]	Adams et al.	1015	*10*	720	*2*	875	*7*
SB3[1]	Balas et al.	981	*6*	667	*1*	902	*4*
SB4[1]	Balas et al.	940	*11*	667	*1*	878	*9*
SB-GLS[1]	Balas & Vazacopoulos	930	*13*	666	*1*	852	*12*
PE-SB[1]	Adams et al.	930	*851*	669	*12*	860	*240*
Bottle-4[1]	Applegate & Cook	938	*7*	667	*1*	863	*10*
Bottle-5[1]	Applegate & Cook	938	*7*	662	*8*	847	*65*
Bottle-6[1]	Applegate & Cook	938	*8*			842	*201*
threshold algorithms							
Shuffle1[1]	Applegate & Cook	938	*25*	655	*8*	842	*73*
Shuffle2[1]	Applegate & Cook	938	*25*	655	*8*	842	*73*
TA1[m5]	Aarts et al.	1003	*99*	693	*19*	925	*94*
SA1[m5]	Aarts et al.	969	*99*	669	*19*	855	*94*
SA2[m5]	Aarts et al.	977	*99*	658	*19*	854	*94*
$SA1^{1}_{t\to\infty}$	Aarts et al.						
SA[m5]	Van Laarhoven et al.	985	*779*	663	*117*	853	*830*
SA[b5]	Van Laarhoven et al.	951	*3895*	655	*585*	848	*4150*
SA-H[1]	Matsuo et al.	946	*987*	655	*3*	842	*115*
taboo search							
TS1[b5]	Taillard	930					
TS2[b2]	Barnes & Chambers	930	*450*	655	*60*	843	*450*
TS3[m5]	Dell'Amico & Trubian	948	*156*	655	*19*	846	*104*
TS3[b5]	Dell'Amico & Trubian	935	*779*	655	*94*	842	*519*
TS-B[1]	Nowicki & Smutnicki	930	*30*	655	*8*	842	*60*
TS-B[b3]	Nowicki & Smutnicki	930		655		842	
variable-depth search							
GLS[b4]	Balas & Vazacopoulos	930	*153*	655	*33*	842	*134*
IGLS[1]	Balas & Vazacopoulos	930	*45*	655	*8*	842	*74*
RGLS-5[1]	Balas & Vazacopoulos	930	*247*	655	*8*	842	*269*
genetic algorithms							
GA-II1[m5]	Aarts et al.	978	*99*	668	*19*	863	*94*
GA-II2[m5]	Aarts et al.	982	*99*	659	*19*	859	*94*
$GA\text{-}II2^{1}_{t\to\infty}$	Aarts et al.						
GA2[m5]	Della Croce et al.	965	*628*	685	*284*	855	*651*
GA2[b5]	Della Croce et al.	946	*3140*	680	*1420*	850	*3255*
GA-P[1]	Dorndorf & Pesch	960	*933*	681	*108*	880	*191*
$GA\text{-}SB^{m2}_{40}$	Dorndorf & Pesch	938	*107*	666	*16*	863	*77*
$GA\text{-}SB^{m2}_{60}$	Dorndorf & Pesch					848	*161*
constraint satisfaction							
RCS[b5]	Nuijten & Aarts	930	*2955*	655	*110*	848	*1455*

schedule length in roman, computation time in seconds in italic

Table 5.2: Results for five instances.

algorithm	LA21		LA24		LA25		LA27		LA29	
n	15		15		15		20		20	
m	10		10		10		10		10	
LB	1046		935		977		1235		1130	
UB	1046		935		977		1235		1157	
shifting bottleneck										
SB1[1]	1172	*2*	1000	*25*	1048	*28*	1325	*45*	1294	*48*
SB3[1]	1111	*11*	976	*11*	1012	*13*	1272	*19*	1227	*21*
SB4[1]	1071	*20*	976	*20*	1012	*23*	1272	*38*	1227	*39*
SB-GLS[1]	1048	*25*	941	*26*	993	*26*	1243	*30*	1182	*44*
PE-SB[1]	1084	*362*	976	*434*	1017	*430*	1291	*837*	1239	*892*
Bottle-4[1]	1094	*17*	983	*26*	1029	*22*	1307	*31*	1220	*31*
Bottle-5[1]	1084	*46*	983	*63*	1001	*48*	1288	*92*	1220	*91*
Bottle-6[1]	1084	*301*	958	*200*	1001	*100*	1286	*666*	1218	*280*
threshold algorithms										
Shuffle1[1]	1055	*955*	971	*421*	997	*74*	1280	*98*	1219	*95*
Shuffle2[1]	1046	*87478*	965	*65422*	992	*98*	1269	*604*	1191	*15358*
TA1^{m5}	1104	*243*	1014	*235*	1075	*255*	1289	*492*	1262	*471*
SA1^{m5}	1083	*243*	962	*235*	1003	*255*	1282	*492*	1233	*471*
SA2^{m5}	1078	*243*	960	*235*	1019	*255*	1275	*492*	1225	*471*
SA1$^{1}_{t\to\infty}$	1053		935		983		1249		1185	
SA^{m5}	1067	*1991*	966	*2098*	1004	*2133*	1273	*4535*	1226	*4408*
SA^{b5}	1063	*9955*	952	*10490*	992	*10665*	1269	*22675*	1218	*22040*
SA-II[1]	1071	*205*	973	*199*	991	*180*	1274	*286*	1196	*267*
taboo search										
TS1^{b5}	1047						1240		1170	
TS2^{b2}	1050	*480*	946	*480*	988	*480*	1250	*600*	1194	*600*
TS3^{m5}	1057	*199*	943	*182*	980	*192*	1252	*254*	1194	*281*
TS3^{b5}	1048	*994*	941	*909*	979	*958*	1242	*1271*	1182	*1407*
TS-B[1]	1055	*21*	948	*184*	988	*155*	1259	*66*	1164	*493*
TS-B^{b3}	1047		939		977		1236		1160	
variable-depth search										
GLS^{b4}	1047	*222*	938	*243*	982	*330*	1236	*435*	1157	*627*
IGLS[1]	1048	*112*	937	*175*	977	*224*	1240	*210*	1164	*369*
RGLS-5[1]	1046	*612*	935	*682*	977	*616*	1235	*315*	1164	*1062*
genetic algorithms										
GA-II1^{m5}	1084	*243*	970	*235*	1016	*255*	1303	*492*	1290	*471*
GA-II2^{m5}	1085	*243*	981	*235*	1010	*255*	1300	*492*	1260	*471*
GA-II2$^{1}_{t\to\infty}$	1055		938		985		1265		1217	
GA2^{m5}	1113	*1062*	1000	*1045*	1029	*1052*	1322	*1555*	1257	*1550*
GA2^{b5}	1097	*5310*	984	*5275*	1018	*5260*	1308	*7775*	1238	*7550*
GA-P[1]	1139	*352*	1014	*352*	1014	*350*	1378	*565*	1336	*570*
GA-SB$^{m2}_{40}$	1074	*135*	960	*137*	1008	*134*	1272	*242*	1204	*241*
GA-SB$^{m2}_{60}$	1074	*293*	957	*289*	1007	*229*	1269	*446*	1210	*453*
constraint satisfaction										
RCS^{b5}	1069	*7600*	942	*7385*	981	*7360*	1285	*13950*	1208	*5660*

schedule length in roman, computation time in seconds in italic

Table 5.3: Results for five instances.

algorithm	LA36		LA37		LA38		LA39		LA40	
n	15		15		15		15		15	
m	15		15		15		15		15	
LB	1268		1397		1196		1233		1222	
UB	1268		1397		1196		1233		1222	
shifting bottleneck										
SB1[1]	1351	*47*	1485	*61*	1280	*58*	1321	*72*	1326	*77*
SB3[1]	1319	*28*	1425	*26*	1318	*30*	1278	*25*	1266	*26*
SB4[1]	1319	*56*	1425	*53*	1294	*59*	1278	*51*	1262	*52*
SB-GLS[1]	1268	*55*	1397	*37*	1208	*56*	1249	*48*	1242	*56*
PE-SB[1]	1305	*735*	1423	*837*	1255	*1079*	1273	*669*	1269	*899*
Bottle-4[1]	1326	*23*	1444	*14*	1299	*46*	1301	*42*	1295	*22*
Bottle-5[1]	1316	*153*	1444	*56*	1299	*96*	1291	*134*	1295	*24*
Bottle-6[1]	1299	*321*	1442	*562*	1268	*182*	1279	*192*	1255	*154*
threshold algorithms										
Shuffle1[1]	1295	*171*	1437	*64*	1294	*104*	1268	*178*	1276	*43*
Shuffle2[1]	1275	*3348*	1422	*1577*	1267	*17799*	1257	*6745*	1238	*150*
TA1[m5]	1385	*602*	1469	*636*	1323	*636*	1305	*592*	1295	*597*
SA1[m5]	1307	*602*	1440	*636*	1235	*636*	1258	*592*	1256	*597*
SA2[m5]	1308	*602*	1451	*636*	1243	*636*	1263	*592*	1254	*597*
$SA1^{1}_{t\to\infty}$					1208				1225	
SA[m5]	1300	*5346*	1442	*5287*	1227	*5480*	1258	*5766*	1247	*5373*
SA[b5]	1293	*26730*	1433	*26435*	1215	*27400*	1248	*28830*	1234	*26865*
SA-II[1]	1292	*624*	1435	*577*	1231	*672*	1251	*660*	1235	*603*
taboo search										
TS1[b5]					1202					
TS2[b2]	1278	*540*	1418	*540*	1211	*540*	1237	*540*	1228	*540*
TS3[m5]	1289	*238*	1423	*242*	1210	*257*	1254	*238*	1235	*237*
TS3[b5]	1278	*1192*	1409	*1211*	1203	*1283*	1242	*1189*	1233	*1183*
TS-B[1]	1275	*623*	1422	*443*	1209	*165*	1235	*325*	1234	*322*
TS-B[b3]	1268		1407		1196		1233		1229	
variable-depth search										
GLS[b4]	1269	*455*	1400	*268*	1208	*464*	1233	*577*	1233	*367*
IGLS[1]	1268	*179*	1397	*146*	1198	*299*	1233	*434*	1234	*332*
RGLS-5[1]	1268	*920*	1397	*822*	1196	*1281*	1233	*1131*	1224	*1590*
genetic algorithms										
GA-II1[m5]	1324	*602*	1449	*636*	1285	*636*	1279	*592*	1273	*597*
GA-II2[m5]	1310	*602*	1450	*636*	1283	*636*	1279	*592*	1260	*597*
$GA\text{-}II2^{1}_{t\to\infty}$					1248				1233	
GA2[m5]	1330	*1880*	1526	*1872*	1282	*1887*	1332	*1870*	1297	*1853*
GA2[b5]	1305	*9400*	1519	*9360*	1273	*9435*	1315	*9350*	1278	*9265*
GA-P[1]	1373	*524*	1498	*520*	1296	*525*	1351	*525*	1321	*526*
$GA\text{-}SB^{m2}_{40}$	1317	*336*	1484	*350*	1251	*336*	1282	*327*	1274	*348*
$GA\text{-}SB^{m2}_{60}$	1317	*688*	1446	*666*	1241	*666*	1277	*687*	1252	*698*
constraint satisfaction										
RCS[b5]	1292	*11165*	1411	*12760*	1278	*14075*	1233	*13620*	1247	*12875*

schedule length in roman, computation time in seconds in italic

Table 5.4: Summary of results.

algorithm	authors	m.r.e.	s.d.r.e.	s.c.t.	computer	c.i.s.c.t.
LB		0	0			
UB		0.18	0.66			
shifting bottleneck						
SB1[1]	Adams et al.	8.20	2.72	*483*	VAX 780/11	*60*
SB3[1]	Balas et al.	4.90	2.52	*222*	Sparc 330	*560*
SB4[1]	Balas et al.	3.87	2.24	*432*	Sparc 330	*1,100*
SB-GLS[1]	Balas & Vazacopoulos	1.12	1.23	*430*	Sparc 330	*1,100*
PE-SB[1]	Adams et al.	3.64	2.25	*10,742*	VAX 780/11	*1,300*
Bottle-4[1]	Applegate & Cook	4.77	2.23	*293*	Sparc ELC	*730*
Bottle-5[1]	Applegate & Cook	4.03	2.51	*884*	Sparc ELC	*2,200*
Bottle-6[1]	Applegate & Cook	3.12	2.09	*3,175*	Sparc ELC	*7,900*
threshold algorithms						
Shuffle1[1]	Applegate & Cook	3.04	2.63	*2,307*	Sparc ELC	*5,800*
Shuffle2[1]	Applegate & Cook	1.95	1.94	*198,685*	Sparc ELC	*500,000*
TA1[m5]	Aarts et al.	7.72	2.41	*4,971*	VAX 8650	*3,500*
SA1[m5]	Aarts et al.	3.39	1.87	*4,971*	VAX 8650	*3,500*
SA2[m5]	Aarts et al.	3.43	1.91	*4,971*	VAX 8650	*3,500*
$SA1^{1}_{t\to\infty}$	Aarts et al.	0.87	1.29	*40,000*	VAX 8650	*30,000*
SA[m5]	Van Laarhoven et al.	3.12	2.00	*44,143*	VAX 785	*8,400*
SA[b5]	Van Laarhoven et al.	2.06	1.88	*220,715*	VAX 785	*42,000*
SA-II[1]	Matsuo et al.	2.21	1.61	*5,378*	VAX 780/11	*670*
taboo search						
TS1[b5]	Taillard					
TS2[b2]	Barnes & Chambers	1.08	1.47	*5,820*	IBM RS 6000	*70,000*
TS3[m5]	Dell'Amico & Trubian	1.47	1.40	*2,598*	PC 386	*1,300*
TS3[b5]	Dell'Amico & Trubian	0.82	1.18	*12,989*	PC 386	*6,500*
TS-B[1]	Nowicki & Smutnicki	0.99	0.90	*2,895*	AT 386 DX	*1,400*
TS-B[b3]	Nowicki & Smutnicki	0.35	0.74	*8,685*	AT 386 DX	*4,300*
variable-depth search						
GLS[b4]	Balas & Vazacopoulos	0.43	0.68	*4,306*	Sparc 330	*11,000*
IGLS[1]	Balas & Vazacopoulos	0.38	0.84	*2,606*	Sparc 330	*6,500*
RGLS-5[1]	Balas & Vazacopoulos	0.24	0.83	*9,554*	Sparc 330	*24,000*
genetic algorithms						
GA-II1[m5]	Aarts et al.	4.94	3.08	*4,971*	VAX 8650	*3,500*
GA-II2[m5]	Aarts et al.	4.48	2.68	*4,971*	VAX 8650	*3,500*
$GA\text{-}II2^{1}_{t\to\infty}$	Aarts et al.			*40,000*	VAX 8650	*30,000*
GA2[m5]	Della Croce et al.	6.33	2.46	*17,189*	PC 486/25	*12,000*
GA2[b5]	Della Croce et al.	5.05	2.48	*85,945*	PC 486/25	*61,000*
GA-P[1]	Dorndorf & Pesch	8.01	3.99	*6,041*	DEC 3100	*9,700*
$GA\text{-}SB^{m2}_{40}$	Dorndorf & Pesch	3.54	1.63	*2,787*	DEC 3100	*4,500*
$GA\text{-}SB^{m2}_{60}$	Dorndorf & Pesch	3.25	1.55	*5,523*	DEC 3100	*8,800*
constraint satisfaction						
RCS[b5]	Nuijten & Aarts	2.06	2.42	*110,970*	Sparc ELC	*280,000*

mean (m.r.e.) and standard deviation (s.d.r.e.) of relative error in percents,
sum of computation times (s.c.t.) and computer independent sum of computation times (c.i.s.c.t.) in seconds; computer independent computation times are rough estimates.

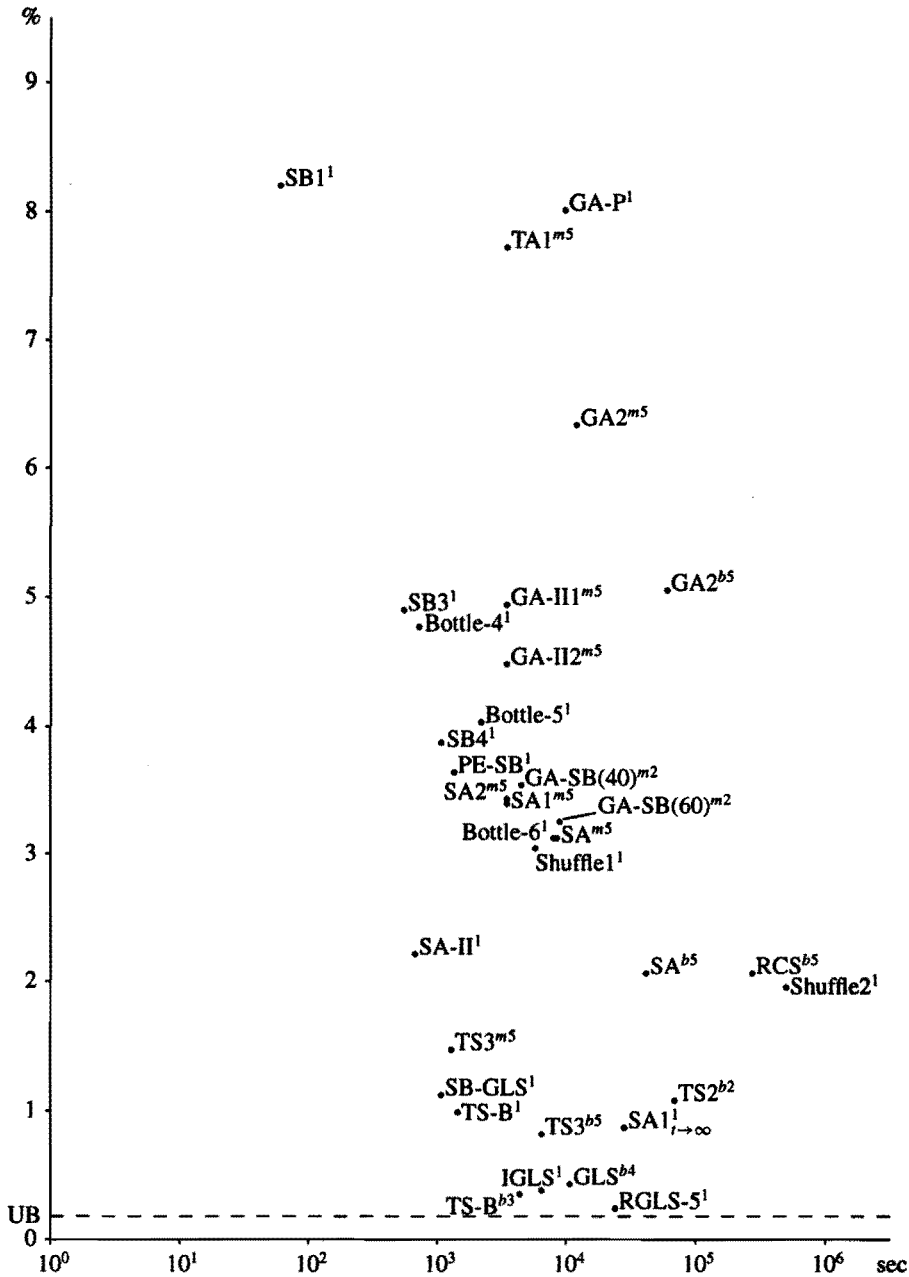


Figure 5.1: Relation between mean relative error and computer independent sum of computation times.

Table 5.5: Miscellaneous results for FT10.

algorithm	authors	length	time	computer
threshold algorithms				
PS10[1]	Storer et al.	976		
HSL10[1]	Storer et al.	1006		
genetic algorithms				
GA1[b600]	Yamada & Nakano	930	*3600000*	Sparc 2
GA3[1]	Nakano & Yamada	965		

schedule length in roman, computation time in seconds in italic

latter values were computed using the normalization coefficients of Dongarra [1993] and must be interpreted with care; their accuracy is at most up to two digits.

Figure 5.1 shows for each algorithm its mean relative error and its computer independent sum of computation times. Note that the time axis has a logarithmic scale.

The *shifting bottleneck* procedure SB1 of Adams, Balas and Zawack [1988] is fast but gives poor results. The variants SB3 and SB4 of Balas, Lenstra and Vazacopoulos [1995], which take the delayed precedences into account, are more effective. It is worthwhile to combine the straight shifting bottleneck procedure with some form of partial enumeration, as is clear from the results obtained by algorithm PE-SB of Adams, Balas and Zawack [1988] and by Bottle-5 and Bottle-6 of Applegate and Cook [1991]. (Note that the values for Bottle-t given here differ from those reported in the original paper: we used the enumeration scheme as described in the paper; they implemented a different scheme [Applegate and Cook, 1993].) Algorithm SB-GLS of Balas and Vazacopoulos [1994] is in an entirely different category. Apparently, the performance of the shifting bottleneck procedure is significantly enhanced if variable-depth search with a fine-grained neighborhood function is used to reoptimize partial schedules.

Among *threshold algorithms* the best results are obtained by the simulated annealing algorithm of Aarts, Van Laarhoven, Lenstra and Ulder [1994] and the iterative improvement algorithm Shuffle of Applegate and Cook [1991].

Regarding *iterative improvement*, Aarts, Van Laarhoven, Lenstra and Ulder [1994] report that their multi-start algorithm is inferior to threshold accepting and simulated annealing. Applegate and Cook's Shuffle algorithm works well, due to a neighborhood function that allows major changes in the schedule. We used our own outcomes of Bottle-5 as start solutions. The number t of machines to fix was chosen such that edge finder could rapidly fill in the remainder of

the schedule. We set t=1 for FT10, LA2 and LA19, t=2 for LA21, LA24 and LA24, and t=5 for the other instances. The results for these values of t are reported under Shuffle1. We also carried out more time consuming runs with t=1 for FT10 and LA2-LA24, t=3 for LA29, LA36 and LA37, and t=4 for LA27, LA38, LA39 and LA40. The outcomes, reported under Shuffle2, are good but expensive. Storer, Wu and Vaccari [1992] give very few computational results for their variants of iterative improvement. Their results for the instance FT10 are poor. It seems that their search strategy or their neighborhood function is not powerful enough.

The *threshold accepting* algorithm TA1 of Aarts, Van Laarhoven, Lenstra and Ulder [1994] competes with their simulated annealing algorithm in case simulated annealing finds an optimal schedule. Otherwise, threshold accepting is outperformed by simulated annealing. Almost all instances in our table belong to the latter category.

The *simulated annealing* algorithm SA of Van Laarhoven, Aarts and Lenstra [1992] produces reasonable results. Results of the same quality are obtained by the algorithms SA1 and SA2 of Aarts, Van Laarhoven, Lenstra and Ulder [1994] with a standard cooling schedule; an extremely slow cooling schedule ($SA1_{t\to\infty}$) gives very good results. To compute the mean and the standard deviation of the relative errors for the latter cooling schedule, we estimated the values for the missing entries. It is remarkable that the standard cooling schedule behaves similarly for the neighborhood functions N_1 (SA1) and N_{6a} (SA2). Good results are obtained by the bi-level variant SA-II of Matsuo, Suh and Sullivan [1988]. In comparison to other approximative approaches, simulated annealing may require large running times, but it yields consistently good solutions with a modest amount of human implementation effort and relatively little insight into the combinatorial structure of the problem type under consideration.

The advent of *taboo search* has changed the picture. Methods of this type produce excellent solutions in reasonable times, but these benefits come at the expense of a non-trivial amount of testing and tuning. Although few data are available, the algorithm TS1 of Taillard [1994] seems to perform extremely well. Also very good results are obtained by algorithm TS2 of Barnes and Chambers [1995]. Dell'Amico and Trubian's [1993] algorithm TS3 obtained even better results; apparently, their complicated neighborhood function is very effective. The algorithm TS-B of Nowicki and Smutnicki [1995], which applies taboo search and traces its way back to promising but rejected changes, is one of the current champions for job shop scheduling. For our thirteen instances it achieves a mean relative error of only 0.35% for the best result out of three runs.

Like TS-B, the *variable-depth search* algorithm GLS proposed by Balas and

Vazacopoulos [1994] combines conceptual elegance and computational excellence. GLS achieves a mean relative error of 0.43% for the best result out of four runs; it needs more time than TS-B, however. The iterated and reiterated variants IGLS and RGLS-k, which apply various reoptimization cycles to partial and complete solutions, perform still better. The best results reported so far have been obtained by RGLS-5: a single run achieves a mean relative error of only 0.28% and finds the optimum for eleven out of thirteen instances.

For many *genetic algorithms* no results for our instances are available. Sometimes only the result for FT10 is given. Yamada and Nakano [1992] found a schedule of length 930 four times among 600 trials. They also tested their algorithm GA1 on four 20-job 20-machine instances, but their outcomes are on average 6.5% above the best known upper bounds [Wennink, 1994; Vazacopoulos, 1995]. The results obtained by Aarts, Van Laarhoven, Lenstra and Ulder [1994] are not very strong. Their algorithm GA-II2 (using neighborhood function N_{6a}) performs slightly better than GA-II1 (using N_1).

As for genetic algorithms using string representations, the results obtained by Della Croce, Tadei and Volta's [1995] algorithm GA2 (published by Della Croce, Tadei and Rolando [1993]) and by Nakano and Yamada's [1991] algorithm GA3 are poor. The algorithm GA-P of Dorndorf and Pesch [1995] is even worse. Their algorithm GA-SB, which incorporates a shifting bottleneck variant, produces reasonable results. Values are reported for runs with population sizes of 40 and 60.

The *constraint satisfaction* algorithm of Nuijten and Aarts [1995] produces good results but needs a lot of time. For the *neural network* approaches no computational results are available that allow a proper comparison with other techniques.

5.8 Conclusion

5.8.1 Review

The local search algorithms discussed in this survey cover a broad range from straightforward to rather involved approaches. In general, the best results are obtained by taboo search and variable-depth search. The 'reiterated guided local search' algorithm of Balas and Vazacopoulos outperforms the other methods in terms of solution quality. The algorithm of Nowicki and Smutnicki, which combines taboo search with backtracking, is a close second and needs much less time. The recent shifting bottleneck algorithm of Balas and Vazacopoulos, which reoptimizes partial schedules by variable-depth search, is an effective and very fast alternative.

The other shifting bottleneck variants are not competitive anymore. For job

shop scheduling, simulated annealing does not seem to be attractive either. It can yield very good solutions, but only if time is of no concern.

Genetic algorithms perform poorly up to now. Often the neighborhood function applied in combination with the schedule representation chosen does not generate meaningful changes and it is hard to find improvements. Only when some kind of local search is embedded at a second level, the computational results are reasonable.

Constraint satisfaction is a promising technique and needs further investigation. It is too early to make an assessment of the use of neural networks for job shop scheduling.

A word of caution is in order regarding the validity of our conclusions. We have collected and compared the computational results reported on a set of benchmark instances. These problems are just on the borderline of being in reach of optimization algorithms. Further experiments on larger instances are required to improve our insights into the performance of the various breeds of the local search family.

5.8.2 Preview

There is still considerable room for improving local search approaches to the job shop scheduling problem. As shown in Figure 1, none of the existing algorithms achieves an average error of less than 2% within 100 seconds total computation time.

We have observed that many approaches operate at two levels, with, for instance, schedule construction, local search with big changes or partial enumeration at the top level, and local search with smaller changes at the bottom level. Such hybrid approaches are in need of a more systematic investigation. It might also be interesting to study three-level approaches with neighborhoods of smaller size towards the bottom.

The flexibility of local search and the results reported here provide a promising basis for the application of local search to more general scheduling problems. An example of practical interest is the multi-processor job shop, where each production stage has a set of parallel machines rather than a single one. Finding a schedule involves assignment as well as sequencing decisions. This is a difficult problem, for which no effective solution methods exist.

Applying local search to large instances of scheduling problems requires the design of data structures that allow fast incremental computations of, for example, longest paths. Johnson [1990] has shown that sophisticated data structures play an important role in the application of local search to large traveling salesman problems.

Our survey has been predominantly of a computational nature. There are several related theoretical questions about the complexity of local search. A central concept in this respect is PLS-completeness [Johnson, Papadimitriou and Yannakakis, 1988]. Many of the neighborhood functions defined in Section 5.3 define a PLS-problem, which may be PLS-complete. There are also complexity issues regarding the parallel execution of local search. For example, for some of the neighborhood functions it may be possible to verify local optimality in polylog parallel time.

6

Local search for the generalized job shop scheduling problem

In this chapter we discuss a representation and several neighborhood functions that can be used in local search methods for the generalized job shop scheduling problem. The chapter is structured as follows. Section 6.1 describes the representation we want to use and introduces several neighborhood functions. Section 6.2 discusses how neighbors and their values can be determined in an efficient way. Section 6.3 deals with the connectivity of the neighborhood functions introduced in Section 6.1.

6.1 Representation and neighborhood functions

In Chapter 2 we mentioned that in order to find an optimal schedule it is sufficient to consider schedules that have some special properties. Depending on these properties we distinguished several types of schedules: left-justified schedules, weakly active schedules, strongly active schedules, and left-optimal schedules. We also showed that there is a one-to-one correspondence between left-justified schedules and complete feasible orientations. Obviously, for each weakly active schedule there exists a unique complete feasible orientation, and the same holds for each strongly active and each left-optimal schedule. But the reverse statements are not true in general, since there may exist orientations that do not correspond to a weakly active schedule. For this reason we choose to consider

complete feasible orientations or, equivalently, left-justified schedules. So the representation of schedules considered in this chapter is that of complete feasible orientations.

A neighborhood function for the generalized job shop scheduling problem must be capable of modifying the chosen machine set for an operation, and it must be capable of changing for a given operation the order relative to other operations that have a machine in common with the given operation in their chosen machine set. For the most elementary neighborhood function that satisfies these conditions a neighbor is obtained as follows: first, choose an operation and delete it from all machine orderings; next, assign to this operation a (possibly different) machine set; finally, insert this operation in the machine orderings corresponding to the new machine set, such that the resulting orientation is feasible. We assume that a solution cannot be a neighbor of itself, so that at least the machine orderings of a schedule and each of its neighbors are different.

More formally, let $\mathcal{S}$ denote the solution space consisting of all possible combinations of machine set assignments and orientations that are feasible for this machine set assignment. Now the elementary neighborhood function $\mathcal{N}_1$ as introduced above is defined by:

$$\begin{aligned}\mathcal{N}_1((K,\Omega)) = \{(K',\Omega') \in \mathcal{S} \mid\ & \Omega' \text{ is feasible for } K',\ \Omega' \neq \Omega,\\ & \exists v \in V:\ K'|_{V\setminus\{v\}} = K|_{V\setminus\{v\}},\\ & \qquad \Omega'(E_{K|_{V\setminus\{v\}}}) = \Omega(E_{K|_{V\setminus\{v\}}})\},\end{aligned}$$

for each $(K,\Omega) \in \mathcal{S}$. Obviously, this neighborhood function satisfies the above conditions.

One can think of several variants of this elementary *reinsertion* neighborhood function. First, we introduce a variant $\mathcal{N}_2$, in which an operation is reinserted in the best way on any chosen machine set. Next, we introduce another variant $\mathcal{N}_3$, in which an operation is reinserted in the best way with the best machine set possible. Finally, for each of the three neighborhood functions introduced above, we define variants in which an operation may get another machine set and orientation only if it belongs to a longest path in the solution graph corresponding to the current solution.

The neighborhood function $\mathcal{N}_2$ is defined as follows. For each $(K,\Omega) \in \mathcal{S}$ the neighborhood $\mathcal{N}_2((K,\Omega))$ contains all solutions (K',Ω') from $\mathcal{N}_1((K,\Omega))$ for which no $(K',\Omega'') \in \mathcal{N}_1((K,\Omega))$ exists such that a longest path in the solution graph corresponding to (K',Ω'') is shorter than a longest path in the solution graph corresponding to (K',Ω'). So a neighbor is obtained by deleting an operation from all machine orderings, and then inserting it in one of its machine sets in a best possible way. Clearly, $\mathcal{N}_2((K,\Omega))$ is a subset of $\mathcal{N}_1((K,\Omega))$ for each $(K,\Omega) \in \mathcal{S}$.

The neighborhood function $\mathcal{N}_3$ is defined as follows. For each $(K, \Omega) \in \mathcal{S}$ the neighborhood $\mathcal{N}_3((K, \Omega))$ contains all solutions (K', Ω') from $\mathcal{N}_1((K, \Omega))$ for which no $(K'', \Omega'') \in \mathcal{N}_1((K, \Omega))$ exists with a shorter longest path in its corresponding solution graph than a longest path in the solution graph corresponding to (K', Ω'). So now, a neighbor is obtained by deleting an operation from all machine orderings, and then inserting it in a machine set in a best possible way, such that no other neighbor exists with a shorter longest path. Clearly, $\mathcal{N}_3((K, \Omega))$ is a subset of $\mathcal{N}_2((K, \Omega))$ for each $(K, \Omega) \in \mathcal{S}$.

However, a neighbor that is obtained by reinserting an operation v that does not belong to a longest path in the solution graph of the current schedule, does have a longest path in its solution graph which is at least as long as the longest path in the solution graph of the current schedule. The reason for this is that the longest path of the current schedule remains present in the solution graph of such a neighbor. So it may be profitable to consider only neighbors that are obtained by reinserting operations v that belong to a longest (or critical) path in the solution graph of the current schedule. We will denote the neighborhood functions corresponding to $\mathcal{N}_1, \mathcal{N}_2$, and $\mathcal{N}_3$ in which only critical operations may be reinserted by $\mathcal{N}_1^c, \mathcal{N}_2^c$, and $\mathcal{N}_3^c$, respectively.

More complicated neighborhood functions can be defined by generalizing the basic neighborhood functions described above. For instance, one can define a neighborhood function that simultaneously assigns to two operations a (possibly different) machine set and finds new machine orderings for the corresponding operations. However, we will restrict ourselves to the elementary neighborhood functions introduced above.

For the neighborhood functions described above it is not clear at first sight how one should check that a possible neighbor corresponds to a feasible schedule. Furthermore, it is not clear how a best possible neighbor in the sense of the neighborhood functions $\mathcal{N}_2$ and $\mathcal{N}_3$ can be determined efficiently. These questions are dealt with in the next section.

6.2 Determination of a neighbor

In this section we first study how we can identify whether a possible neighbor is feasible or not. Next, we study how for the neighborhood functions $\mathcal{N}_2$ and $\mathcal{N}_3$ a given operation can be reinserted in the best possible way. Since the methods for finding such a best reinsertion have the same behavior for the reinsertion of a critical operation as for a non-critical operation, they can also be used to find the best reinsertion of a given operation for the neighborhood functions $\mathcal{N}_2^c$ and $\mathcal{N}_3^c$.

Now suppose for the remainder of this section that we are given a feasible

schedule represented by (K, Ω) with a corresponding solution graph $\mathcal{G}$. For each of the neighborhood functions introduced in the previous section, let v denote the operation that is deleted from all machine orderings and is reinserted again on possibly a different machine set. The solution graph corresponding to the partial solution that is obtained from (K, Ω) by deleting v from all machine orderings is given by

$$\mathcal{G}_{(K|_{V\setminus\{v\}}, \Omega|_{\mathcal{P}_2(V\setminus\{v\})})}.$$

In the remainder of this section we use the following abbreviations:

$$\begin{aligned} V^- &= V\setminus\{v\}; \\ \mathcal{G}^- &= \mathcal{G}_{(K|_{V\setminus\{v\}}, \Omega|_{\mathcal{P}_2(V\setminus\{v\})})}; \\ \Omega^- &= \Omega|_{\mathcal{P}_2(V\setminus\{v\})}. \end{aligned}$$

Note that operation v is still a node in the graph $\mathcal{G}^-$. Clearly, $\mathcal{G}^-$ is acyclic since $\mathcal{G}$ is already acyclic. We denote the new machine set of v by $K'(v)$. Now the set

$$Q(v) = \{w \in V^- \mid K'(v) \cap K(w) \neq \emptyset\}$$

contains all operations whose machine set has a machine in common with the new machine set of v. For each of these operations we have to decide whether it will be scheduled before or after v. Each of these choices leads to a complete orientation. However, some of these orientations may be infeasible. The feasibility is subject of the next subsection.

6.2.1 Feasibility of neighbors

In this part we study how we can obtain feasible neighbors. By definition, a reinsertion of v leads to a feasible neighbor if and only if the solution graph that is obtained from $\mathcal{G}^-$ by reinserting v is acyclic. Clearly, if $Q^{\triangleleft}(v)$ is the set of operations $w \in Q(v)$ for which a path from w to v exists in the graph $\mathcal{G}^-$, then each $w \in Q^{\triangleleft}(v)$ must be scheduled before v. Similarly, if $Q^{\triangleright}(v)$ is the set of operations $w \in Q(v)$ for which a path from v to w exists in the graph $\mathcal{G}^-$, then each $w \in Q^{\triangleright}(v)$ must be scheduled after v. The set of operations w for which it has to be decided whether w will be scheduled before or after v is thereby reduced to

$$\begin{aligned} Q^{\diamond}(v) = \{w \in V^- \mid\ & K'(v) \cap K(w) \neq \emptyset, \\ & \text{no path exists in } \mathcal{G}^- \text{ from } w \text{ to } v, \\ & \text{no path exists in } \mathcal{G}^- \text{ from } v \text{ to } w\}. \end{aligned}$$

In the remainder of this chapter we just write Q, $Q^{\triangleleft}$, $Q^{\triangleright}$, and $Q^{\diamond}$ instead of $Q(v)$, $Q^{\triangleleft}(v)$, $Q^{\triangleright}(v)$, and $Q^{\diamond}(v)$, respectively, if it is clear which operation v is to be reinserted.

Now a neighbor is defined by selecting a subset L of $Q^{\diamond}$ of operations that are scheduled before v. Such a reinsertion will be denoted by (v, L). The cor-

responding solution graph $\mathcal{G}^L$ is obtained from $\mathcal{G}^-$ by adding arcs (u, v) for all $u \in Q^{\triangleleft} \cup L$, and arcs (v, w) for all $w \in Q \setminus (Q^{\triangleleft} \cup L)$.

The following theorem shows under which conditions on L the corresponding graph $\mathcal{G}^L$ is acyclic.

Theorem 6.1. *The graph $\mathcal{G}^L$ is acyclic if and only if $x \in L$ or $y \in Q^{\diamond} \setminus L$ for each pair $x, y \in Q^{\diamond}$ for which an x-y path in $\mathcal{G}^-$ exists.*

Proof. Suppose that there exist operations $x, y \in Q^{\diamond}$ for which an x-y path in $\mathcal{G}^-$ exists and for which we do not have that $x \in L$ or $y \in Q^{\diamond} \setminus L$. Then the only possibility is that $x \in Q^{\diamond} \setminus L$ and $y \in L$. But then the graph $\mathcal{G}^L$ contains the arcs (y, v) and (v, x). Since the x-y path in $\mathcal{G}^-$ remains present in $\mathcal{G}^L$, we clearly have a cycle.

Conversely, suppose that for each pair $x, y \in Q^{\diamond}$ with an x-y path in $\mathcal{G}^-$ we have $x \in L$ or $y \in Q^{\diamond} \setminus L$. Furthermore, assume that $\mathcal{G}^L$ contains a cycle. Since $\mathcal{G}^L$ is obtained from the acyclic graph $\mathcal{G}^-$ by adding arcs incident to v, this cycle must contain one of the added arcs, and hence it must contain v. There can exist three types of cycles, each corresponding to one of the following possibilities:

1. $\mathcal{G}^-$ contains a path from v to some $u \in L$. Therefore, $\mathcal{G}^L$ contains a cycle $(v, \ldots, u, v)$. But this is not possible since such an operation u cannot be an element of $Q^{\diamond}$, and hence not of L.
2. $\mathcal{G}^-$ contains a path from some $w \in Q^{\diamond} \setminus L$ to v. Therefore, $\mathcal{G}^L$ contains a cycle $(v, w, \ldots, v)$. Also this is not possible since such an operation w cannot be an element of $Q^{\diamond}$, and hence not of $Q^{\diamond} \setminus L$.
3. $\mathcal{G}^-$ contains a path from some $w \in Q^{\diamond} \setminus L$ to some $u \in L$, resulting in a cycle $(v, w, \ldots, u, v)$ in $\mathcal{G}^L$. But because of the existence of a w-u path in $\mathcal{G}^-$ for this $u \in L$ and $w \in Q^{\diamond} \setminus L$, we should have $w \in L$ or $u \in Q^{\diamond} \setminus L$. So also this situation does not occur.

Therefore, neither of the three types of cycles can exist, which proves that $\mathcal{G}^L$ is acyclic. □

Now for the neighborhood functions $\mathcal{N}_1$ and $\mathcal{N}_1^c$ it is clear how a feasible neighbor can be determined. First, choose the operation v that is to be reinserted and compute the graph $\mathcal{G}^-$. Next, choose the new machine set for this operation and determine the corresponding set $Q^{\diamond}$. Finally, choose a subset L of $Q^{\diamond}$ of operations that are to be scheduled before v, such that L satisfies the condition of Theorem 6.1.

The complexity of choosing a feasible reinsertion is determined as follows. Let $q \leq m$ be an upper bound on the cardinality of the machine sets H. Computing the graph $\mathcal{G}^-$ takes $\mathcal{O}(l)$ time. The time required to determine the set Q

depends on the data structure that is used, but it will not exceed $\mathcal{O}(ql)$, which is the time required to find for each machine in $K'(v)$ the operations on the corresponding machine path. When the transitive closure of the graph $\mathcal{G}^-$ is stored, determining the set $Q^\diamond$ from Q takes $\mathcal{O}(l)$ time. Choosing a set L that satisfies the condition of Theorem 6.1 takes also $\mathcal{O}(l)$ time. So in total, choosing a feasible neighbor takes $\mathcal{O}(ql)$ time.

6.2.2 Finding a best reinsertion on a given machine set

In this subsection we study how an operation can be reinserted in the best possible way. The criterion used here is the length of the longest s-t path in the resulting solution graph. First, we study the case in which the machine set on which this operation is to be inserted is given. From the previous subsection it follows that the number of possible reinsertions of a given operation v on a given machine set $K'(v)$ may be exponential in the size of $K'(v)$. In this subsection we will show that we can restrict ourselves to a set of $\mathcal{O}(l)$ reinsertions, which is guaranteed to contain an optimal one. Furthermore, we show that each of these reinsertions results in a feasible solution. Note that by restricting ourselves to this set of $\mathcal{O}(l)$ reinsertions we may discard some neighboring solutions of $\mathcal{N}_2((K, \Omega))$ for which v is scheduled on $K'(v)$. In fact, our method implicitly defines a neighborhood function that defines for each schedule a neighborhood that is contained in the neighborhood defined by $\mathcal{N}_2$.

Now let a subset L of $Q^\diamond$ be given and assume that the insertion (v, L) leads to a feasible orientation. $\mathcal{G}^L$ denotes the corresponding acyclic solution graph that arises from $\mathcal{G}^-$ by adding arcs (u, v) for all $u \in Q^\triangleleft \cup L$, and arcs (v, w) for all $w \in Q \setminus (Q^\triangleleft \cup L)$. Let for any $x, y \in \mathcal{V}$ the length of longest paths from node x to node y in $\mathcal{G}^-$ and $\mathcal{G}^L$ be denoted by $d^-[x, y]$ and $d^L[x, y]$, respectively, provided that such paths exist. Note that $d^-[s, u]$ and $d^-[u, t]$ are defined for all $u \in V$, since in $\mathcal{G}^-$ always paths exist from s to u and from u to t. The length of a longest path in the graph $\mathcal{G}^-$ is equal to $d^-[s, t]$, and the length of a longest path in the graph $\mathcal{G}^L$ is equal to $d^L[s, t]$.

Since $\mathcal{G}^L$ arises from $\mathcal{G}^-$ by only adding arcs incident to v, leaving the rest of $\mathcal{G}^-$ unchanged, we have

$$d^L[s, t] = \max\{d^-[s, t], d^L[s, v] + p(v, K'(v)) + d^L[v, t]\}. \tag{6.1}$$

Furthermore,

$$\begin{aligned} d^L[s, v] &= \max\{d^-[s, v], \max_{u \in Q^\triangleleft \cup L} (d^-[s, u] + p(u, K(u)))\} \\ &= d^-[s, v] + \max_{u \in Q^\triangleleft \cup L} (d^-[s, u] + p(u, K(u)) - d^-[s, v])^+, \end{aligned} \tag{6.2}$$

and

$$d^L[v,t] = \max\{d^-[v,t], \max_{w \in Q \setminus (Q^\triangleleft \cup L)} (d^-[w,t] + p(w, K(w)))\}$$
$$= d^-[v,t] + \max_{w \in Q \setminus (Q^\triangleleft \cup L)} (d^-[w,t] + p(w, K(w)) - d^-[v,t])^+, \tag{6.3}$$

where $x^+ = \max\{x, 0\}$. Substituting (6.2) and (6.3) in (6.1), we get

$$\begin{aligned} d^L[s,t] = \max\{&d^-[s,t], \\ &d^-[s,v] + p(v, K'(v)) + d^-[v,t] \\ &+ \max_{u \in Q^\triangleleft \cup L} (d^-[s,u] + p(u, K(u)) - d^-[s,v])^+ \\ &+ \max_{w \in Q \setminus (Q^\triangleleft \cup L)} (d^-[w,t] + p(w, K(w)) - d^-[v,t])^+\}. \end{aligned} \tag{6.4}$$

The problem is to find an $L \subseteq Q^\diamond$ for which $d^L[s,t]$ is minimal. Let for all $u \in Q$

$$a_u = d^-[s,u] + p(u, K(u)) - d^-[s,v], \tag{6.5}$$
$$b_u = d^-[u,t] + p(u, K(u)) - d^-[v,t]. \tag{6.6}$$

Then, a_u^+ can be viewed as the increase of the length of the longest path from s to v if v is positioned after u, and b_u^+ as the increase of the length of the longest path from v to t if v is positioned before u.

The following lemma shows that operations in $Q^\triangleleft$ do not influence the increase of the length of the longest path from s to v.

Lemma 6.2. *If $u \in Q^\triangleleft$, then $a_u \leq 0$.*

Proof. Since $u \in Q^\triangleleft$ there exists a u-v path in the graph $\mathcal{G}^-$. As a consequence $d^-[u,v] \geq 0$ and $d^-[s,v] \geq d^-[s,u] + p(u, K(u)) + d^-[u,v]$. But then

$$\begin{aligned} a_u &= d^-[s,u] + p(u, K(u)) - d^-[s,v] \\ &\leq -d^-[u,v] \\ &\leq 0, \end{aligned}$$

which completes the proof. □

Similarly, we have that $b_w \leq 0$ if $w \in Q^\triangleright$. So now

$$\Delta(L) = \max_{u \in L} a_u^+ + \max_{w \in Q^\diamond \setminus L} b_w^+ \tag{6.7}$$

is the increase in the length of the longest path from s to t through v when the reinsertion (v, L) is performed. Our goal is to find an $L^* \subseteq Q^\diamond$ such that

$$\Delta(L^*) = \min_{L \subseteq Q^\diamond} \Delta(L). \tag{6.8}$$

In the following we show that some sets L do not have to be considered in order to find such an L^*. First, we need the following definition.

Definition 6.3. Let $L \subseteq Q^\diamond$. Then $\Lambda(L)$ denotes the unique maximal subset of $Q^\diamond$ such that

$$\max_{u \in \Lambda(L)} a_u^+ = \max_{u \in L} a_u^+. \tag{6.9}$$

□

Now the following theorem shows that only the sets $\Lambda(L)$ have to be considered in order to find such an L^*.

Theorem 6.4. *For each $L \subseteq Q^\diamond$ we have $\Delta(\Lambda(L)) \leq \Delta(L)$.*

Proof. Let $L \subseteq Q^\diamond$. Since $L \subseteq \Lambda(L)$ we have

$$\begin{aligned}
\Delta(\Lambda(L)) &= \max_{u \in \Lambda(L)} a_u^+ + \max_{w \in Q^\diamond \setminus \Lambda(L)} b_w^+ \\
&\leq \max_{u \in \Lambda(L)} a_u^+ + \max_{w \in Q^\diamond \setminus L} b_w^+ \\
&= \max_{u \in L} a_u^+ + \max_{w \in Q^\diamond \setminus L} b_w^+ \\
&= \Delta(L).
\end{aligned}$$

□

It can easily be seen that for each $L \subseteq Q^\diamond$ we have that $L = \Lambda(L)$ if and only L satisfies

$$\max_{u \in L} a_u^+ < \min_{u \in Q^\diamond \setminus L} a_u^+. \tag{6.10}$$

Hence, in order to find an optimal set L^* we only have to consider sets L that satisfy equation (6.10). Now the following theorem shows that each such L does always lead to a feasible reinsertion.

Theorem 6.5. *Let $L \subseteq Q^\diamond$ satisfy equation (6.10). Then the reinsertion (v, L) is feasible.*

Proof. Let $x, y \in Q^\diamond$ be such that there exists an x-y path in $\mathcal{G}^-$. By Theorem 6.1 it is sufficient to prove that $x \in L$ or $y \in Q^\diamond \setminus L$. Since

$$\begin{aligned}
a_x &= d^-[s, x] + p(x, K(x)) - d^-[s, v] \\
&< d^-[s, x] + p(x, K(x)) + d^-[x, y] + p(y, K(y)) - d^-[s, v] \\
&\leq d^-[s, y] + p(y, K(y)) - d^-[s, v] \\
&= a_y,
\end{aligned}$$

we have $a_x^+ \leq a_y^+$, and therefore $x \in Q^\diamond \setminus L$ and $y \in L$ cannot hold simultaneously by the choice of L. □

So in order to find an optimal set L^* only sets L have to be considered that satisfy equation (6.10). Similarly, one can prove that it is also sufficient to consider sets L that satisfy

$$\max_{w \in Q^\diamond \setminus L} b_w^+ < \min_{w \in L} b_w^+. \tag{6.11}$$

in order to find an optimum, and that each such L results in a feasible reinsertion. For each of both possibilities at most $|Q^\diamond|+1$ sets L have to be considered, which number is obviously bounded from above by l. All sets L that satisfy equation (6.10) (or (6.11)) can be found efficiently in the following way. First, sort the operations of $Q^\diamond$ in order of increasing a_u (or decreasing b_u) and then find a set L for which $\Delta(L)$ is minimal. Since the operations on each machine path are already in the correct order, sorting can be done by merging the $\mathcal{O}(q)$ sorted lists corresponding to the machines of $K'(v)$. Hence, sorting takes $\mathcal{O}(l \log q)$ time. Now finding an optimal L requires $\mathcal{O}(l)$ time.

Sorting and selecting an optimal L can be done slightly faster, though the total time needed remains $\mathcal{O}(l \log q)$. The essential modification is that some operations are excluded from consideration already during the sorting. To be able to explain this modification we introduce the notion of *dominance*. Operation $y \in Q^\diamond$ is said to dominate $x \in Q^\diamond$ if $a_x \leq a_y$ and $b_x \leq b_y$; if $a_x = a_y$ and $b_x = b_y$, some tie-breaker is used to determine which operation dominates the other, for example the operation with lowest index. Now let two operations $x, y \in Q^\diamond$ be given and suppose that y dominates x. Then one can easily check that for each $L \subseteq Q^\diamond$ not containing x nor y the two following inequalities hold: $\Delta(L \cup \{x, y\}) \leq \Delta(L \cup \{y\})$ and $\Delta(L) \leq \Delta(L \cup \{x\})$. Hence, the sets L that contain only one of the operations x and y can be disregarded. This also holds for the special type of sets L that satisfy equation (6.10). Similarly, one can check that for each undominated operation x with $a_x \leq 0$ and for each set $L \subseteq Q^\diamond$ not containing x the inequality $\Delta(L \cup \{x\}) \leq \Delta(L)$ holds, and that for each undominated operation x with $b_x \leq 0$ and for each set $L \subseteq Q^\diamond$ not containing x the inequality $\Delta(L) \leq \Delta(L \cup \{x\})$ holds. Therefore, also operations x with $a_x \leq 0$ or $b_x \leq 0$ can be disregarded. Summarizing, the only interesting operations are those belonging to the set $Q^\cup$ of undominated operations x with $a_x > 0$ and $b_x > 0$.

Now suppose that we want to find a set L satisfying equation (6.10) that minimizes $\Delta(L)$. From the arguments given above it follows that this can by done by simultaneously determining the set $Q^\cup \subseteq Q^\diamond$ and sorting the operations $u \in Q^\cup$ in order of increasing a_u. Suppose we want to decide whether operation u belongs to $Q^\cup$, and if so, to insert it in the list of already ordered operations of $Q^\cup$. First, we search among the operations that have been sorted already for the operation w with lowest order for which $a_u \leq a_w$. If such a w

does not exist, then we insert u behind the last ordered operation. If such a w does exist and if it dominates u, then we discard u. Otherwise, we insert u just before w. Next, if u is inserted, then we remove all already sorted operations that are dominated by u. We finally obtain the set $Q^{\cup}$ sorted in order of increasing a_u. Let the function $\pi : \{1, \ldots, |Q^{\cup}|\} \rightarrow Q^{\cup}$ denote this order; hence, $a_{\pi(1)} < a_{\pi(2)} < \ldots < a_{\pi(|Q^{\cup}|)}$. Since $Q^{\cup}$ only contains undominated operations, we also have $b_{\pi(1)} > b_{\pi(2)} > \ldots > b_{\pi(|Q^{\cup}|)}$. Now the sets L^i, $0 \leq i \leq |Q^{\cup}|$, defined by

$$L^i = \begin{cases} \{u \in Q^{\diamond} \mid a_u \leq 0\} & \text{if } i = 0 \\ \{u \in Q^{\diamond} \mid a_u \leq a_{\pi(i)}\} & \text{if } i > 0 \end{cases} \tag{6.12}$$

are the only ones that we have to consider in order to find an optimal set L^*. Note that the value of $\Delta(L^i)$ can be easily determined since

$$\Delta(L^i) = \begin{cases} b_{\pi(1)} & \text{if } i = 0 \\ a_{\pi(i)} + b_{\pi(i+1)} & \text{if } 1 \leq i \leq |Q^{\cup}| \\ a_{\pi(|Q^{\cup}|)} & \text{if } i = |Q^{\cup}|. \end{cases} \tag{6.13}$$

Now it is trivial to find a set L^i that minimizes Δ. Similarly, by reversing the role of the a and b values, we could have considered the sets L_r^i, $0 \leq i \leq |Q^{\cup}|$, defined by

$$L_r^i = \begin{cases} \{u \in Q^{\diamond} \mid b_u > 0\} & \text{if } i = 0 \\ \{u \in Q^{\diamond} \mid b_u > b_{\pi(i)}\} & \text{if } i > 0. \end{cases} \tag{6.14}$$

The complexity of finding an optimal feasible reinsertion of an operation v on a given machine set $K'(v)$ is determined as follows. Computing the graph $\mathcal{G}^-$ takes $\mathcal{O}(l)$ time. The time required to determine the set Q depends on the data structure that is used, but it will not exceed $\mathcal{O}(ql)$, which is the time required to find for each machine in $K'(v)$ the operations on the corresponding machine path. Computing $d^-[s, u]$ and $d^-[u, t]$ for all $u \in Q \cup \{v\}$ takes $\mathcal{O}(|A|+ql)$ time. Since the solution graph is acyclic, a reaching algorithm can be applied to compute the longest paths from one node to all the other nodes in the graph. The time required by a reaching algorithm is linear in the number of arcs in the graph. The arc set in the reduced solution graph consists of $\mathcal{O}(l)$ dummy arcs, $\mathcal{O}(|A|)$ precedence arcs, and $\mathcal{O}(ql)$ machine arcs. The reaching algorithm is applied twice, once for computing $d^-[s, u]$ and once for computing $d^-[u, t]$. Computing a_u and b_u for all $u \in Q$ takes $\mathcal{O}(l)$ time. For both methods described sorting and finding an optimal L^* can be done in $\mathcal{O}(l \log q)$ time. Finally, computing $\mathcal{G}^{L^*}$ takes $\mathcal{O}(l)$ time.

Thus, the total time required to find an optimal reinsertion of an operation v on machine set $K'(v)$ is $\mathcal{O}(|A| + ql)$. In terms of the number of operations and

the number of machines, this is $\mathcal{O}(n^2 + ml)$. Note that for problems with precedence chains or precedence trees, and with a fixed upper bound on the size of the machine sets, the total running time is $\mathcal{O}(l)$. The gain in efficiency as compared to the method of trying out all possible reinsertions is highest for larger values of q, that is, for problems with large machine sets.

Now we briefly discuss how an operation v can be reinserted in the best possible way when we are free to choose the machine set $K'(v)$ on which the operation v will be processed. From the previous subsection it follows that we can find a best reinsertion in $\mathcal{O}(|\mathcal{H}(v)| \cdot (|A| + ql))$ time. However, when $p(v, K'(v))$ does not depend on $K'(v)$ we can do slightly better, since then $d^-[s, u]$ and $d^-[u, t]$ for all $u \in Q \cup \{v\}$ have to be computed only once. Then the total time needed is $\mathcal{O}(|A| + |\mathcal{H}(v)|ql)$.

6.3 Connectivity of neighborhood functions

An interesting property of neighborhood functions is *connectivity*. This property expresses to which extent solutions can be reached from an arbitrary given solution by making a sequence of transitions such that it starts at the given solution and each next solution is a neighbor of the previous one.

To be more specific we have to introduce several notions. Let a minimization problem be given and let the triple $(\mathcal{S}, \mathcal{X}, f)$ specify an instance of this problem, with $\mathcal{S}$ the solution space, $\mathcal{X}$ the cost space, and f the cost function. For a neighborhood function $\mathcal{N}: \mathcal{S} \to \mathcal{P}(\mathcal{S})$ we define the set $A_{\mathcal{N}}$ as

$$A_{\mathcal{N}} = \{(x, y) \mid x \in \mathcal{S}, y \in \mathcal{N}(x)\}.$$

So $A_{\mathcal{N}}$ contains pairs of solutions for which the second solution is a neighbor of the first solution. The directed graph $\mathcal{G}_{\mathcal{N}} = (\mathcal{S}, A_{\mathcal{N}})$ is called the *neighborhood graph* corresponding to $\mathcal{N}$.

Definition 6.6. A neighborhood function $\mathcal{N}$ is called *strongly connected* if the corresponding neighborhood graph is strongly connected. A neighborhood function $\mathcal{N}$ is called *optimum connected* if for each solution there exists a path to an optimal solution in the corresponding neighborhood graph. □

Note that each strongly connected neighborhood function is also optimum connected.

The extent to which a neighborhood function is connected has consequences for local search algorithms. For instance, if a given neighborhood function is not optimum connected, then there are solutions for which no sequence of transitions leads to an optimal solution. In this case any local search algorithm that starts

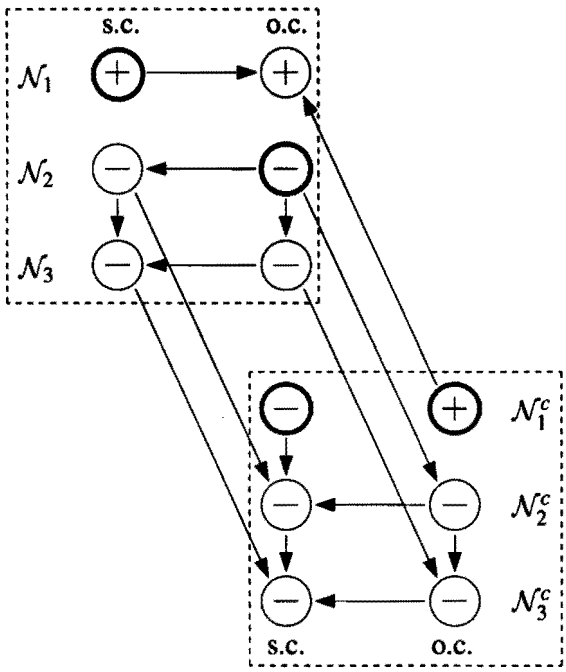


Figure 6.1: Connectivity of various neighborhood functions.

with such an initial solution and makes only transitions to neighboring solutions is unable to find an optimal solution. Under certain mild conditions the standard simulated annealing algorithm asymptotically converges to an optimal solution if the neighborhood function is optimum connected [Van Laarhoven, 1988].

In the following we discuss the connectivity of the neighborhood functions introduced in Section 6.1. Figure 6.1 summarizes to which extent the neighborhood functions $\mathcal{N}_1$, $\mathcal{N}_2$, and $\mathcal{N}_3$, in which each operation may be reinserted, and the neighborhood functions $\mathcal{N}_1^c$, $\mathcal{N}_2^c$, and $\mathcal{N}_3^c$, in which only operations may be reinserted that are on a longest s-t path in the current schedule, are connected. The sign + denotes that a neighborhood function is strongly or optimum connected and the sign − that it is not. Arrows denote that a result for the connectivity of one neighborhood function implies a result for the connectivity of a second neighborhood function. Bold circles denote results that are not implied by other results. Each of these results is the subject of one of the following theorems.

First, we consider the neighborhood functions in which all operations may be reinserted. The following theorem states that the neighborhood function $\mathcal{N}_1$ is strongly connected and therefore also optimum connected.

Theorem 6.7. *$\mathcal{N}_1$ is strongly connected.*

Proof. Let an initial schedule (K_0, Ω_0) and a final schedule (K_f, Ω_f) be given. We have to show that we can construct a finite sequence of schedules (K_i, Ω_i) leading from (K_0, Ω_0) to (K_f, Ω_f), such that $(K_{i+1}, \Omega_{i+1}) \in \mathcal{N}_1((K_i, \Omega_i))$ for all i. We construct this sequence in two stages. The first stage ends with a schedule (K_i, Ω_i) with $K_i = K_f$. In the second stage the machine set assignment remains the same and only the orientations are modified.

The schedules in the first stage are constructed as follows. As long as there exists an operation v in the current schedule (K_i, Ω_i) with $K_i(v) \neq K_f(v)$, reinsert such an operation v on machine set $K_f(v)$, resulting in the neighboring schedule (K_{i+1}, Ω_{i+1}). Here, each feasible reinsertion may be chosen.

The schedules in the second stage are constructed as follows. As long as for the current schedule (K_f, Ω_i) the set δ_{if} defined by

$$\delta_{if} = \{\{x, y\} \in \mathcal{P}_2(V) \mid K_f(x) \cap K_f(y) \neq \emptyset, \\ \Omega_i(\{x, y\}) \neq \Omega_f(\{x, y\})\}$$

is non-empty, there is a pair $\{v, w\}$ of operations such that $K_f(v) \cap K_f(w) \neq \emptyset$, $\Omega_i(\{v, w\}) = (v, w)$, $\Omega_f(\{v, w\}) = (w, v)$, and such that no $x \in V$ exists with (v, x) or $(x, w) \in A \cup \Omega_i(E_{K_f})$. Choose such a pair $\{v, w\}$ and reinsert v such that the order of v and w is reversed while the order for each other pair of operations remains the same. For the resulting orientation Ω_{i+1} we have $|\delta_{i+1,f}| = |\delta_{if}| - 1$. Since $|\delta_{if}|$ is strictly decreasing for increasing i, we finally reach an orientation Ω_i for which $\delta_{if} = \emptyset$. Clearly, in that case we have $\Omega_i = \Omega_f$.

What remains is to prove that each of the reinsertions in the second stage results in a feasible orientation. To achieve the proposed reinsertion the set L of operations of $Q^\circ(v)$ that have to be scheduled before v must be equal to

$$\{x \in Q^\circ(v) \mid \Omega_i(\{v, x\}) = (x, v) \text{ or } x = w\}.$$

In view of Theorem 6.1 it is sufficient to prove that $x \in L$ or $y \in Q^\circ(v) \setminus L$ for each pair $x, y \in Q^\circ(v)$ for which an x-y path in $\mathcal{G}^-_{(K_i, \Omega_i)}$ exists. Now let $x, y \in Q^\circ(v)$ and assume that there exists an x-y path in $\mathcal{G}^-_{(K_i, \Omega_i)}$ and that $y \in L$. Then we have to prove that also $x \in L$. Since $y \in L$ we have either $\Omega_i(\{v, y\}) = (y, v)$ or $y = w$. If $\Omega_i(\{v, y\}) = (y, v)$, then there exists an y-v path in $\mathcal{G}_{(K_i, \Omega_i)}$. Since there is also an x-y path in this graph, there exists an x-v path and therefore $\Omega_i(\{v, x\}) = (x, v)$ and thus $x \in L$. Now consider the case that $y = w$ and assume that $\Omega_i(\{v, x\}) = (v, x)$. Since there is an x-w path in $\mathcal{G}_{(K_i, \Omega_i)}$, there is also a v-w path in $\mathcal{G}_{(K_i, \Omega_i)}$ with at least the operation x between v and w. But by the choice of v and w such an x cannot exist. Hence, the assumption that $\Omega_i(\{v, x\}) = (v, x)$ cannot be true. So $\Omega_i(\{v, x\}) = (x, v)$ and thus $x \in L$. □

The following theorem states that in general the neighborhood function $\mathcal{N}_2$ is not optimum connected and thus not strongly connected either. This also implies that $\mathcal{N}_3$ is not optimum connected, nor strongly connected.

Theorem 6.8. *There are instances for which $\mathcal{N}_2$ is not optimum connected.*

Proof. We give an instance for which $\mathcal{N}_2$ is not optimum connected. Let

$$V = \{v_i \mid i \in \{1a, 1b, 2a, 2b, 3a, 3b, 4a, 4b\}\}$$

and let $M = \{\mu_1, \mu_2\}$. Let $\mathcal{H}$ be defined by

$$\mathcal{H}(v_i) = \begin{cases} \{\{\mu_1\}\} & \text{if } i \in \{1a, 1b, 3a, 3b\} \\ \{\{\mu_2\}\} & \text{if } i \in \{2a, 2b, 4a, 4b\}, \end{cases}$$

and let p be defined by

$$p(v_i, \{\mu(v_i)\}) = \begin{cases} 1 & \text{if } i = 3b \\ 2 & \text{otherwise.} \end{cases}$$

Let the precedence relation A be as depicted in Figure 6.2. Now consider the schedule of length 12 as depicted in Figure 6.3. It can be easily checked that the best reinsertion of v_{1a} or v_{1b} reverses the order of v_{1a} and v_{1b} and leads again to a schedule of length 12. A similar property holds for v_{2a} and v_{2b}, v_{3a} and v_{3b}, and v_{4a} and v_{4b}. So each best reinsertion gives a schedule which is different from the given schedule. For each schedule that is obtained by applying one or more of such reinsertions we have that on machine μ_1 the operations v_{1a} and v_{1b} are processed before v_{3a} and v_{3b}, and on machine μ_2 the operations v_{2a} and v_{2b} are processed before v_{4a} and v_{4b}. The length of each such schedule is 12. Again for each of these schedules a best reinsertion reverses two operations in a similar way as for the schedule of Figure 6.3.

Clearly, these schedules are not optimal, since each left-justified schedule in which the operations v_{3a} and v_{3b} are processed before v_{1a} and v_{1b} on machine μ_1, and v_{4a} and v_{4b} are processed before v_{2a} and v_{2b} on machine μ_2 has only length 11. An example of such a schedule is given in Figure 6.4. Obviously, such a schedule is optimal.

Hence, no optimal schedule is reachable from the schedule depicted in Figure 6.3. □

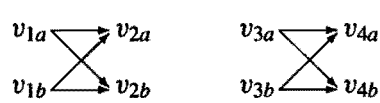


Figure 6.2: The precedence relation A on V.

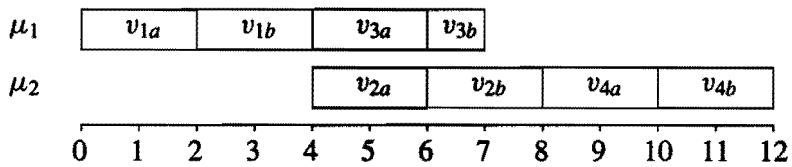


Figure 6.3: A feasible schedule of length 12.

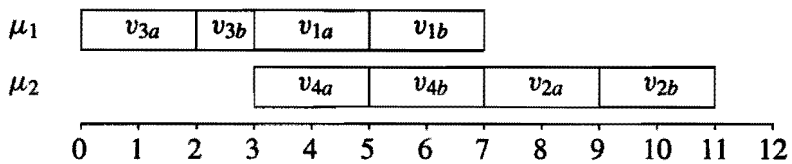


Figure 6.4: An optimal feasible schedule of length 11.

Now we consider the neighborhood functions in which only operations on a longest path may be reinserted. As a consequence of Theorem 6.8 the neighborhood functions $\mathcal{N}_2^c$ and $\mathcal{N}_3^c$ are in general not optimum connected nor strongly connected. So the only neighborhood function that remains to be studied is $\mathcal{N}_1^c$. First, we show that this neighborhood function is in general not strongly connected. Thereafter, we prove that it is optimum connected.

Theorem 6.9. *There are instances for which $\mathcal{N}_1^c$ is not strongly connected.*

Proof. We give an instance for which $\mathcal{N}_1^c$ is not strongly connected. Let $V = \{v_1, v_2, v_3\}$ and let $M = \{\mu_1, \mu_2\}$. Let $\mathcal{H}$ be defined by

$$\mathcal{H}(v_i) = \begin{cases} \{\{\mu_1\}\} & \text{if } i = 1, \\ \{\{\mu_2\}\} & \text{if } i \in \{2, 3\}, \end{cases}$$

and let p be defined by

$$p(v_i, \{\mu(v_i)\}) = \begin{cases} 3 & \text{if } i = 1 \\ 1 & \text{otherwise.} \end{cases}$$

There are no precedences ($A = \emptyset$). Now the only decision to be made is about the relative order of operations v_2 and v_3. The two left-justified schedules corresponding to each of these two orderings have length 3, the length of operation v_1. Neither v_2 nor v_3 are on a longest s-t path. The only operation on a longest path is v_1, but this operation is the only one on machine μ_1 and there is no decision to be made about its order. So none of the two solutions have a neighbor. Hence, the corresponding neighborhood graph is not strongly connected. □

The instance in the proof of the previous theorem may suggest that one can still find a sequence of solutions from an arbitrary solution to a solution for which the chosen machine sets and the relative orders of all operations on each longest s-t path are the same as in a given solution. However, the following instance, due to Van Laarhoven [1988], shows that also this is impossible.

Let $V = \{v_i \mid i \in \{1, \ldots, 6\}\}$ and let $M = \{\mu_1, \mu_2, \mu_3\}$. Let $\mathcal{H}$ be defined by

$$\mathcal{H}(v_i) = \begin{cases} \{\{\mu_1\}\} & \text{if } i \in \{1, 6\}, \\ \{\{\mu_2\}\} & \text{if } i \in \{2, 5\}, \\ \{\{\mu_3\}\} & \text{if } i \in \{3, 4\}, \end{cases}$$

and let $p(v_i, \{\mu(v_i)\}) = 1$ for all i. Let the precedence relation A be defined by

$$\tilde{A} = \{(v_1, v_2), (v_2, v_3), (v_4, v_5), (v_5, v_6)\}.$$

Since for each operation there is only one machine set it can be processed on, a solution is completely characterized by the orientation of the operations. There are four possible solutions, corresponding to the schedules depicted in Figure 6.5. Clearly, Ω_1 is reachable from Ω_2 and Ω_3 and Ω_2 is reachable from Ω_1 and Ω_4, but Ω_3 and Ω_4 are not reachable from any other solution. Since Ω_3 is the only solution with the longest s-t path $(s, v_1, v_2, v_3, v_4, v_5, v_6, t)$, no solution with this longest path can be reached from another solution.

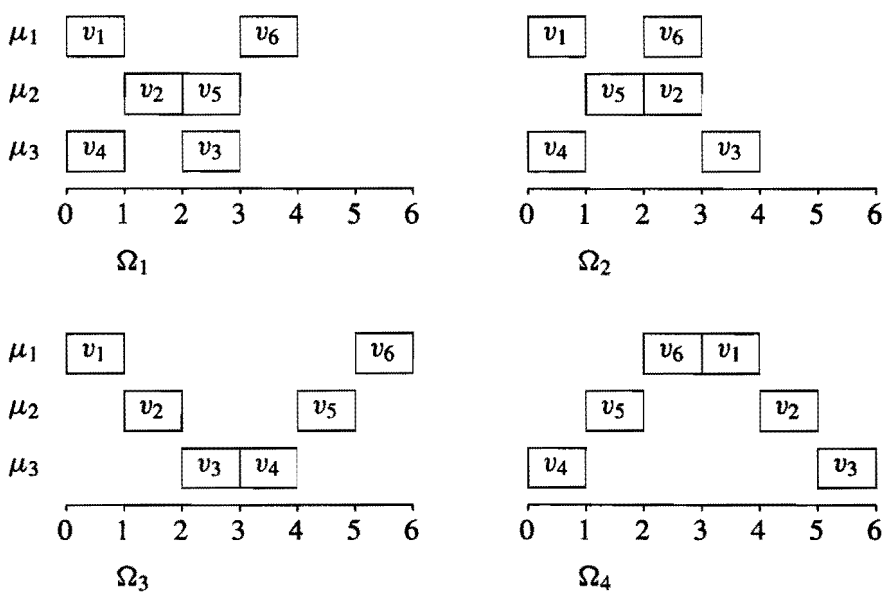


Figure 6.5: All possible left-justified schedules.

So the only possibility to prove that $\mathcal{N}_1^c$ is optimum connected is by directly proving that one can find a sequence of solutions from an arbitrary solution to an optimal solution. Theorem 6.11 shows that $\mathcal{N}_1^c$ is optimum connected. In order to prove this theorem we need the following lemma.

Lemma 6.10. *Let a schedule (K_i, Ω_i) and an optimal schedule (K_*, Ω_*) be given. Let*

$$\begin{aligned}
\delta_{i*}^1 &= \{x \in V \mid K_i(x) \neq K_*(x), x \text{ on a longest } s\text{-}t \text{ path in } \mathcal{G}_{(K_i,\Omega_i)}\},\\
\delta_{i*}^2 &= \{\{x, y\} \in \mathcal{P}_2(V) \mid \\
&\qquad K_i(x) = K_*(x), K_i(y) = K_*(y),\\
&\qquad \Omega_i(\{x, y\}) \neq \Omega_*(\{x, y\}),\\
&\qquad x \text{ and } y \text{ adjacent on a longest } s\text{-}t \text{ path in } \mathcal{G}_{(K_i,\Omega_i)}\}.
\end{aligned}$$

If $\delta_{i*}^1 = \emptyset$ and $\delta_{i*}^2 = \emptyset$, then the schedule (K_i, Ω_i) is optimal.

Proof. If $\delta_{i*}^1 = \emptyset$, then each operation on a longest s-t path in $\mathcal{G}_{(K_i,\Omega_i)}$ is processed by the same machine in (K_i, Ω_i) as in (K_*, Ω_*). Therefore, $\delta_{i*}^2 = \emptyset$ implies that each pair of adjacent operations x and y on a longest s-t path in $\mathcal{G}_{(K_i,\Omega_i)}$ occurs also in an s-t path in $\mathcal{G}_{(K_*,\Omega_*)}$. Since (K_*, Ω_*) is optimal, also (K_i, Ω_i) must be optimal. □

Now we are able to prove the following theorem.

Theorem 6.11. *$\mathcal{N}_1^c$ is optimum connected.*

Proof. Let an initial schedule (K_0, Ω_0) be given. We will show that we can construct a finite sequence of schedules (K_i, Ω_i) leading from (K_0, Ω_0) to an optimal schedule, such that $(K_{i+1}, \Omega_{i+1}) \in \mathcal{N}_1^c((K_i, \Omega_i))$ for all i. Let (K_*, Ω_*) be an optimal schedule.

To describe the construction of the sequence, we explain how for a schedule (K_i, Ω_i) in the sequence the next schedule (K_{i+1}, Ω_{i+1}) is obtained. We can distinguish three situations.

1. If $\delta_{i*}^1 = \emptyset$ and $\delta_{i*}^2 = \emptyset$ for the schedule (K_i, Ω_i), then, by Lemma 6.10 this schedule is optimal and we are done.
2. If $\delta_{i*}^1 \neq \emptyset$, then there is a v on a longest s-t path with $K_i(v) \neq K_*(v)$. Then reinsert v on the machine set $K_*(v)$ in an arbitrary way such that the resulting schedule (K_{i+1}, Ω_{i+1}) is feasible.
3. If $\delta_{i*}^1 = \emptyset$ and $\delta_{i*}^2 \neq \emptyset$, then each operation that occurs on a longest s-t path is processed on the same machine set as in (K_*, Ω_*). Furthermore, there exists at least one pair of adjacent operations v and w on a longest s-t path with $\Omega_i(\{v, w\}) = (v, w)$ and $\Omega_*(\{v, w\}) = (w, v)$. Then (K_{i+1}, Ω_{i+1}) is obtained by reinserting such a v on the same machine set, and such that the

order of v and w is reversed while the order for each other pair of operations remains the same. This reinsertion is feasible, since it is of the same type as the second stage reinsertions in the proof of Theorem 6.7.

What remains to show is that this method eventually finds an optimal schedule. Now let the sets ϕ^1_{i*} and ϕ^2_{i*} be defined by

$$\begin{aligned}\phi^1_{i*} &= \{x \in V \mid K_i(x) \neq K_*(x)\} \text{ and}\\ \phi^2_{i*} &= \{\{x, y\} \in \mathcal{P}_2(V) \mid K_i(x) = K_*(x),\ K_i(y) = K_*(y),\\ &\qquad \Omega_i(\{x, y\}) \neq \Omega_*(\{x, y\})\}.\end{aligned}$$

Clearly, we have $\delta^1_{i*} \subseteq \phi^1_{i*}$ and $\delta^2_{i*} \subseteq \phi^2_{i*}$. In the second situation, we obtain a new schedule with $|\phi^1_{i+1,*}| = |\phi^1_{i*}| - 1$, but for which $|\phi^2_{i+1,*}|$ may be larger than $|\phi^2_{i+1,*}|$. In the third situation, we obtain a schedule with $|\phi^1_{i+1,*}| = |\phi^1_{i*}|$ and $|\phi^2_{i+1,*}| = |\phi^2_{i*}| - 1$. So $(|\phi^1_{i*}|, |\phi^2_{i*}|)$ is strictly lexicographically decreasing in i. Therefore, we eventually reach a schedule for which both δ^1_{i*} and δ^2_{i*} are empty. By Lemma 6.10 this schedule must be optimal. □

Bibliography

E.H.L. AARTS, J.H.M. KORST (1989), *Simulated Annealing and Boltzmann Machines*, Wiley, Chichester.

E.H.L. AARTS, J.K. LENSTRA (eds.) (1995), *Local Search in Combinatorial Optimization*, Wiley, Chichester (in press).

E.H.L. AARTS, P.J.M. VAN LAARHOVEN, J.K. LENSTRA, N.L.J. ULDER (1994), A computational study of local search algorithms for job shop scheduling, *ORSA J. Comput. 6*, 118-125.

J. ADAMS, E. BALAS, D. ZAWACK (1988), The shifting bottleneck procedure for job shop scheduling, *Management Sci. 34*, 391-401.

S.B. AKERS (1956), A graphical approach to production scheduling problems, *Oper. Res. 4*, 244-245.

J. ALBERTON (1988), *NP-hardness of 3-job Scheduling Problems*, unpublished manuscript.

D. APPLEGATE, W. COOK (1991), A computational study of the job-shop scheduling problem, *ORSA J. Comput. 3*, 149-156.

D. APPLEGATE, W. COOK (1993), Personal communication.

K.R. BAKER, Z.-S. SU (1974), Sequencing with due-dates and early start times to minimize maximum tardiness, *Naval Res. Logist. Quart. 21*, 171-176.

E. BALAS (1969), Machine sequencing via disjunctive graphs: an implicit enumeration algorithm, *Oper. Res. 17*, 941-957.

E. BALAS (1985), On the facial structure of scheduling polyhedra, *Math. Programming Stud. 24*, 179-218.

E. BALAS, J.K. LENSTRA, A. VAZACOPOULOS (1995), The one-machine problem with delayed precedence constraints and its use in job shop scheduling, *Management Sci. 41*, 94-109.

E. BALAS, A. VAZACOPOULOS (1994), *Guided Local Search with Shifting Bottleneck for Job Shop Scheduling*, Management Science Research Report #MSRR-609, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

J.W. BARNES, J.B. CHAMBERS (1995), Solving the job shop scheduling problem with tabu search, *IIE Trans. 27*, 257-263.

J. BLAZEWICZ, P. DELL'OLMO, M. DROZDOWSKI, M.G. SPERANZA (1992), Scheduling multiprocessor tasks on three dedicated processors, *Inform. Process. Lett. 41*, 275-280.

J. BLAZEWICZ, M. DROZDOWSKI, J. WEGLARZ (1986), Scheduling multiprocessor tasks to minimize schedule length, *IEEE Trans. Comput. 35*, 389-393.

J. BLAZEWICZ, J.K. LENSTRA, A.H.G. RINNOOY KAN (1983), Scheduling subject to resource constraints: classification and complexity, *Discrete Appl. Math. 5*, 11-24.

F. BOCK (1958), *An Algorithm for Solving "Traveling Salesman" and Related Network Optimization Problems*, Internal report, Armour Research Foundation, Chicago, Illinois.

P. BRATLEY, M. FLORIAN, P. ROBILLARD (1973), On sequencing with earliest starts and due dates with application to computing bounds for the $(n/m/G/F_{\max})$ problem, *Naval Res. Logist. Quart. 20*, 57-67.

P. BRUCKER (1988), An efficient algorithm for the job-shop problem with two jobs, *Computing 40*, 353-359.

P. BRUCKER (1994), A polynomial algorithm for the two machine job-shop scheduling problem with a fixed number of jobs, *OR Spektrum 16*, 5-7.

P. BRUCKER, B. JURISCH, A. KRÄMER (1994), *Complexity of Scheduling Problems with Multi-purpose Machines*, Internal report, Fachbereich Mathematik/Informatik, Universität Osnabrück, Germany.

P. BRUCKER, B. JURISCH, B. SIEVERS (1994), A branch and bound algorithm for the job-shop scheduling problem, *Discrete Appl. Math. 49*, 107-127.

P. BRUCKER, R. SCHLIE (1990), Job-shop scheduling with multi-purpose machines, *Computing 45*, 369-375.

J. CARLIER (1982), The one-machine sequencing problem, *European J. Oper. Res. 11*, 42-47.

J. CARLIER, E. PINSON (1989), An algorithm for solving the job-shop problem, *Management Sci. 35*, 164-176.

J. CARLIER, E. PINSON (1990), A practical use of Jackson's preemptive schedule for solving the job-shop problem, *Ann. Oper. Res. 26*, 269-287.

J. CARLIER, E. PINSON (1994), Adjustments of heads and tails for the job-shop problem, *European J. Oper. Res. 78*, 146-161.

V. ČERNY (1985), Thermodynamical approach to the traveling salesman problem, *J. Optim. Theory Appl. 45*, 41-51.

G.A. CROES (1958), A method for solving traveling salesman problems, *Oper. Res. 6*, 791-812.

S. DAUZERE-PERES, J.-B. LASSERRE (1993), A modified shifting bottleneck procedure for job-shop scheduling, *Int. J. Prod. Res. 31*, 923-932.

L. DAVIS (1985), Job shop scheduling with genetic algorithms. J.J. GREFENSTETTE (ed.) (1985), *Proceedings of an International Conference on Genetic Algorithms and Their Applications*, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 136-140.

F. DELLA CROCE, R. TADEI, R. ROLANDO (1993), Solving a real world project scheduling problem with a genetic algorithm, *Belg. J. Oper. Res. Statist. Comput. Sci. 33*, 65-78.

F. DELLA CROCE, R. TADEI, G. VOLTA (1995), A genetic algorithm for the job shop problem, *Comput. Oper. Res. 22*, 15-24.

M. DELL'AMICO, M. TRUBIAN (1993), Applying tabu search to the job-shop scheduling problem, *Ann. Oper. Res. 41*, 231-252.

J.J. DONGARRA (1993), *Performance of Various Computers Using Standard Linear Equations Software*, Report CS-89-85, Computer Science Department, University of Tennessee, Knoxville, Tennessee.

U. DORNDORF, E. PESCH (1995), Evolution based learning in a job shop scheduling environment, *Comput. Oper. Res. 22*, 25-40.

J. DU, J.Y-T. LEUNG, G.H. YOUNG (1991), Scheduling chain-structured tasks to minimize makespan and mean flow time, *Information and Computation 92*, 219-236.

G. DUECK, T. SCHEUER (1990), Threshold accepting; a general purpose optimization algorithm, *J. Comput. Phys. 90*, 161-175.

E. FALKENAUER, S. BOUFFOUIX (1991), A genetic algorithm for job shop, *Proceedings of the 1991 IEEE Internationial Conference on Robotics and Automation*, IEEE Computer Society Press, Los Alamitos, California, 824-829.

M.L. FISHER, B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1983), Surrogate duality relaxation for job shop scheduling, *Discrete Appl. Math. 5*, 65-75.

H. FISHER, G.L. THOMPSON (1963), Probabilistic learning combinations of local job-shop scheduling rules. J.F. MUTH, G.L. THOMPSON (eds.) (1963), *Industrial Scheduling*, Prentice Hall, Englewood Cliffs, New Jersey, 225-251.

Y.P.S. FOO, Y. TAKEFUJI (1988a), Stochastic neural networks for solving job-shop scheduling: part 1. Problem representation, *IEEE International Conference on Neural Networks*, IEEE San Diego section & IEEE TAB Neural Network Committee, San Diego, California, 275-282.

Y.P.S. FOO, Y. TAKEFUJI (1988b), Stochastic neural networks for solving job-shop scheduling: part 2. Architecture and simulations, *IEEE International Conference on Neural Networks*, IEEE San Diego section & IEEE TAB Neural Network Committee, San Diego, California, 283-290.

M.R. GAREY, D.S. JOHNSON (1975), Complexity results for multiprocessor scheduling under resource constraints, *SIAM J. Comput. 4*, 397-411.

M.R. GAREY, D.S. JOHNSON (1979), *Computers and Intractability: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

M.R. GAREY, D.S. JOHNSON, R. SETHI (1976), The complexity of flowshop and jobshop scheduling, *Math. Oper. Res. 1*, 117-129.

B. GIFFLER, G.L. THOMPSON (1960), Algorithms for solving production scheduling problems, *Oper. Res. 8*, 487-503.

F. GLOVER (1989), Tabu search - Part I, *ORSA J. Comput. 1*, 190-206.

F. GLOVER (1990), Tabu Search - Part II, *ORSA J. Comput. 2*, 4-32.

F. GLOVER, E. TAILLARD, D. DE WERRA (1993), A user's guide to tabu search, *Ann. Oper. Res. 41*, 3-28.

D.E. GOLDBERG (1989), *Genetic Algorithms in Search, Optimization, and Machine Learning*, Addison-Wesley, Reading, Massachusetts.

R.L. GRAHAM, E.L. LAWLER, J.K. LENSTRA, A.H.G RINNOOY KAN (1979), Optimization and approximation in deterministic sequencing and scheduling, *Ann. Discrete Math. 5*, 287-326.

R. HAUPT (1989), A survey of priority rule-based scheduling, *OR Spektrum 11*, 3-16.

N. HEFETZ, I. ADIRI (1982), An efficient optimal algorithm for the two-machines unit-time jobshop schedule-length problem, *Math. Oper. Res. 7*, 354-360.

J.H. HOLLAND (1975), *Adaptation in Natural and Artificial Systems*, The University of Michigan Press, Ann Arbor, Michigan.

I. HOLYER (1981), The NP-completeness of edge-coloring, *SIAM J. Comput. 10*, 718-720.

J.A. HOOGEVEEN, S.L. VAN DE VELDE, B. VELTMAN (1994), Complexity of scheduling multiprocessor tasks with prespecified processor allocations, *Discrete Appl. Math. 55*, 259-272.

J.J. HOPFIELD, D.W. TANK (1985), Neural computation of decisions in optimization problems, *Biol. Cybernet. 55*, 141-152.

J.R. JACKSON (1956), An extension of Johnson's results on job lot scheduling, *Naval Res. Logist. Quart. 3*, 201-203.

D.S. JOHNSON (1990), Data structures for traveling salesmen, J.R. GILBERT, R. KARLSSON (eds.) (1990), *SWAT90, 2nd Scandinavian Workshop on Algorithm Theory*, Springer, Berlin, 287-305.

D.S. JOHNSON, C.H. PAPADIMITRIOU, M. YANNAKAKIS (1988), How easy is local search?, *J. Comput. System Sci. 37*, 79-100.

B. JURISCH (1992), *Scheduling Jobs in Shops with Multi-purpose Machines*, Ph.D. thesis, Fachbereich Mathematik/Informatik, Universität Osnabrück, Germany.

R.M. KARP (1972), Reducibility among combinatorial problems. R.E. MILLER, J.W. THATCHER (eds.) (1972), *Complexity of Computer Computations*, Plenum Press, New York, 85-103

S. KIRKPATRICK, C.D. GELATT JR., M.P. VECCHI (1983), Optimization by simulated annealing, *Science 220*, 671-680.

H. KRAWCZYK, M. KUBALE (1985), An approximation algorithm for diagnostic test scheduling in multicomputer systems, *IEEE Trans. Comput. 34*, 869-872.

M. KUBALE (1987), The complexity of scheduling independent two-processor tasks on dedicated processors, *Inform. Process. Lett. 24*, 141-147.

B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1976), Minimizing maximum lateness on one machine: computational experience and some applications, *Statist. Neerlandica 30*, 25-41.

B.J. LAGEWEG, J.K. LENSTRA, A.H.G. RINNOOY KAN (1977), Job-shop scheduling by implicit enumeration, *Management Sci. 24*, 441-450.

S. LAWRENCE (1984), *Resource Constrained Project Scheduling: an Experimental Investigation of Heuristic Scheduling Techniques (Supplement)*, Graduate School of Industrial Administration, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

J.K. LENSTRA, A.H.G. RINNOOY KAN (1979), Computational complexity of discrete optimization problems, *Ann. Discrete Math. 4*, 121-140.

J.K. LENSTRA, A.H.G. RINNOOY KAN, P. BRUCKER (1977), Complexity of machine scheduling problems, *Ann. Discrete Math. 1*, 343-362.

J.K. LENSTRA, D.B. SHMOYS (1995), Computing near-optimal schedules. PH. CHRÉTIENNE, E.G. COFFMAN, J.K. LENSTRA, Z. LIU (eds.) (1995), *Scheduling Theory and its Applications*, Wiley, Chichester.

S. LIN, B.W. KERNIGHAN (1973), An effective heuristic algorithm for the traveling-salesman problem, *Oper. Res. 21*, 498-516.

O. MARTIN, S.W. OTTO, E.W. FELTEN (1989), Large step Markov chains for the traveling salesman problem, *Complex Systems 5*, 299-326.

H. MATSUO, C.J. SUH, R.S. SULLIVAN (1988), *A Controlled Search Simulated Annealing Method for the General Jobshop Scheduling Problem*, Working paper 03-04-88, Graduate School of Business, University of Texas, Austin.

G.B. MCMAHON, M. FLORIAN (1975), On scheduling with ready times and due dates to minimize maximum lateness, *Oper. Res. 23*, 475-482.

W. MEYER (1992), *Geometrische Methoden zur Lösung von Job-Shop Problemen und deren Verallgemeinerungen*, Ph.D. thesis, Fachbereich Mathematik/Informatik, Universität Osnabrück, Germany.

R. NAKANO, T. YAMADA (1991), Conventional genetic algorithm for job shop problems, R.K. BELEW, L.B. BOOKER (eds.) (1991), *Proceedings of the Fourth International Conference on Genetic Algorithms*, San Diego, California, 474-479.

L. NÉMETI (1964), Das Reihenfolgeproblem in der Fertigungsprogrammierung und Linearplanung mit logischen Bedingungen, *Mathematica (Cluj) 6*, 87-99.

E. NOWICKI, C. SMUTNICKI (1995), A fast taboo search algorithm for the job shop problem, *Management Sci.*, to appear.

W.P.M. NUIJTEN, E.H.L. AARTS (1995), A computational study of constraint satisfaction for job shob scheduling, *European J. Oper. Res.*, to appear.

C.R. REEVES (ed.) (1993), *Modern Heuristic Techniques for Combinatorial Problems*, Blackwell, Oxford, England.

B. ROY, B. SUSSMANN (1964), *Les problèmes d'ordonnancement avec contraintes disjonctives*, Note DS No. 9 bis, SEMA, Montrouge, France.

N. SADEH (1991), *Look-ahead Techniques for Micro-opportunistic Job Shop Scheduling*, Ph.D. thesis, School of Computer Science, Carnegie-Mellon University, Pittsburgh, Pennsylvania.

S.V. SEVAST'JANOV (1994), On some geometric methods in scheduling theory: a survey, *Discrete Appl. Math. 55*, 59-82.

D.B. SHMOYS, C. STEIN, J. WEIN (1994), Improved approximation algorithms for shop scheduling problems, *SIAM J. Comput. 23*, 617-632.

Y.N. SOTSKOV (1991), The complexity of shop-scheduling problems with two or three jobs, *European J. Oper. Res. 53*, 326-336.

R.H. STORER, S.D. WU, R. VACCARI (1992), New search spaces for sequencing problems with application to job shop scheduling, *Management Sci. 38*, 1495-1509.

E. TAILLARD (1994), Parallel taboo seach technique for the jobshop scheduling problem, *ORSA J. Comput. 6*, 108-117.

F. TIOZZO (1988), *Building a Decision Support System for Operation Scheduling in a Large Industrial Department: a Preliminary Algorithmic Study*, Internal report, Department of Mathematics and Informatics, University of Udine, Italy.

N.L.J. ULDER, E.H.L. AARTS, H.-J. BANDELT, P.J.M. VAN LAARHOVEN, E. PESCH (1990), Improving TSP exchange heuristics by population genetics, *Proceedings International Workshop on Parallel Problem Solving from Nature*, Dortmund, Germany, October 1-3, 109-116.

R.J.M. VAESSENS, E.H.L. AARTS, J.K. LENSTRA (1995a), A local search template, *Evolutionary Comp.*, submitted.

R.J.M. VAESSENS, E.H.L. AARTS, J.K. LENSTRA (1995b), Job shop scheduling by local search, *Math. Programming Series B*, submitted.

P.J.M. VAN LAARHOVEN (1988), *Theoretical and Computational Aspects of Simulated Annealing*, Ph.D. thesis, Erasmus University Rotterdam, Rotterdam, The Netherlands.

P.J.M. VAN LAARHOVEN, E.H.L. AARTS, J.K. LENSTRA (1992), Job shop scheduling by simulated annealing, *Oper. Res. 40*, 113-125.

A. VAZACOPOULOS (1995), Personal communication.

M. WENNINK (1994), Personal communication.

M. WENNINK, R.J.M. VAESSENS (1995), *An Efficient Insertion Algorithm for a Scheduling Problem with Processor Sets*, unpublished manuscript.

D.P. WILLIAMSON, L.A. HALL, J.A. HOOGEVEEN, C.A.J. HURKENS, J.K. LENSTRA, S.V. SEVAST'JANOV, D.B. SHMOYS (1996), Short shop schedules, *Oper. Res.*, to appear.

T. YAMADA, R. NAKANO (1992), A genetic algorithm applicable to large-scale job-shop problems, R. MÄNNER, B. MANDERICK (eds.) (1992), *Parallel Problem Solving from Nature, 2*, North-Holland, Amsterdam, 281-290.

D.N. ZHOU, V. CHERKASSKY, T.R. BALDWIN, D.E. OLSON (1991), A neural network approach to job-shop scheduling, *IEEE Trans. Neural Networks 2*, 175-179.

Samenvatting

In dit proefschrift bestuderen we een bepaald type schedulingprobleem. In het algemeen treden schedulingproblemen op wanneer een verzameling activiteiten uitgevoerd dient te worden door een beperkte verzameling van hulpmiddelen. Gedurende de laatste decennia is er veel onderzoek gedaan op het gebied van deterministische schedulingproblemen, waarin alle gegevens die een instantie van het probleem beschrijven met zekerheid bekend zijn. Bijzonder veel aandacht is besteed aan machine schedulingproblemen, waarin machines de enig mogelijke hulpmiddelen zijn. De voornaamste restrictie is dat een machine slechts één activiteit tegelijkertijd kan uitvoeren.

De in de literatuur bestudeerde machine schedulingproblemen zijn in wezen wiskundige modellen voor schedulingproblemen die in de praktijk vaak aanzienlijk complexer zijn. Opdat een praktische probleem zo goed mogelijk in een model weerspiegeld wordt, is het gewenst dat de essentiële elementen van het praktische probleem in het model tot uitdrukking komen. Het probleem doet zich echter voor dat praktische problemen dikwijls zo lastig zijn dat eenvoudige modellen te ver afstaan van de realiteit en realistische modellen niet meer snel zijn op te lossen.

Een van de meest complexe machine schedulingproblemen die in de literatuur bestudeerd zijn is het job shop schedulingprobleem. Dit model is echter toch nog redelijk beperkt: voor elke activiteit is er precies één machine gegeven waarop de activiteit uitgevoerd kan worden. In dit proefschrift beschouwen we een uitbreiding van dit model door toe te staan dat iedere activiteit op een collectie van machines uitgevoerd kan worden en dat voor iedere activiteit verschillende van dergelijke collecties gegeven zijn, waaruit een collectie gekozen dient te worden die de activiteit daadwerkelijk uit zal voeren. Deze generalisatie van het job shop schedulingprobleem zullen we het gegeneraliseerde job shop schedulingprobleem (GJSSP) noemen.

Het GJSSP kan informeel als volgt worden beschreven. Gegeven is een verzameling operaties en een verzameling machines. De verzameling operaties representeert de verzameling van uit te voeren activiteiten. Een operatie dient zonder onderbreking uitgevoerd te worden door een collectie van machines. Voor elke operatie is een aantal toegestane collecties van machines gegeven die de

operatie kunnen uitvoeren. Voor elke operatie en elk van haar toegestane machinecollecties is een geheeltallige verwerkingstijd gegeven, die aangeeft hoe lang de uitvoering van de operatie op de machinecollectie duurt. Verder is er een precedentierelatie op de verzameling operaties gedefinieerd, die voor geordende paren van operaties aangeeft dat de uitvoering van de tweede operatie pas kan starten nadat de uitvoering van de eerste beëindigd is. Een schedule bestaat nu uit een toewijzing van een toegestane machinecollectie en een niet-negatieve starttijd aan elk van de operaties. Een toegestaan schedule is een schedule waarin geen enkele machine meer dan één operatie tegelijkertijd uitvoert en waarin voor elk paar operaties waarvoor een precedentie geldt de uitvoering van de tweede operatie niet start voordat de uitvoering van de eerste beëindigd is. De tijdsduur die voor een gegeven toegestaan schedule nodig is om alle operaties uit te voeren wordt de lengte van het schedule genoemd. Het probleem bestaat uit het vinden van een toegestaan schedule met een zo klein mogelijke lengte. Een dergelijk schedule wordt optimaal genoemd.

Het vinden van een toegestaan schedule met de kleinst mogelijke lengte is in zijn algemeenheid lastig, wat wil zeggen dat het probleem zeer waarschijnlijk niet in polynomiale tijd op te lossen is. Er zijn verscheidene mogelijkheden om deze complicatie te omzeilen. Enerzijds is het mogelijk om deelklassen van het GJSSP te bestuderen welke mogelijk wel in polynomiale tijd optimaal op te lossen zijn. Anderzijds is het mogelijk om met een schedule genoegen te nemen waarvan de lengte wellicht niet optimaal is, maar de optimale lengte wel voldoende dicht benadert. Algoritmen om dergelijke schedules te bepalen worden benaderingsalgoritmen genoemd. In dit proefschrift worden beide wegen bewandeld.

In Hoofdstuk 3 bestuderen we deelklassen van het GJSSP, waarbij we aangeven welke deelklassen nog gemakkelijk oplosbaar zijn en welke deelklassen lastig zijn. Hierbij pogen we de grens tussen gemakkelijk oplosbare en lastige problemen zo scherp mogelijk te markeren. We maken hierbij onderscheid tussen deelklassen die gekarakteriseerd worden door een vaste bovengrens op de lengte van het schedule en deelklassen waarbij een dergelijke bovengrens niet gegeven is.

Vervolgens bestuderen we benaderingsalgoritmen voor het GJSSP zonder verdere beperkingen. Hierbij hebben we ons toegespitst op de klasse van lokale zoekalgoritmen. Lokale zoekalgoritmen zijn algemeen toepasbaar en geven goede resultaten voor diverse lastige optimaliseringsproblemen. Zij zijn met succes toegepast voor het oplossen van diverse schedulingproblemen. In de literatuur zijn talrijke lokale zoekalgoritmen beschreven voor het oplossen van een deelklasse van het GJSSP, namelijk het job shop schedulingprobleem. Hierbij

lopen de resultaten uiteen van bevredigend tot uitstekend.

In principe begint een lokaal zoekalgoritme voor een gegeven probleem met een initiële oplossing en genereert het vervolgens een rij van nieuwe oplossingen, zodanig dat iedere oplossing in de rij verkregen is uit de voorgaande oplossing door een deel van deze oplossing te veranderen. Een buurruimtefunctie definieert welke veranderingen toegestaan zijn. Zij wijst impliciet aan iedere oplossing een verzameling van buuroplossingen (of buren) toe die door een toegestane verandering vanuit de gegeven oplossing in één stap bereikt kunnen worden. Een zoekstrategie specificeert op welke manier een buur uit de buurruimte van een oplossing in de rij gekozen wordt als de volgende oplossing in de rij. De meeste zoekstrategieën zijn zodanig dat buren van betere kwaliteit geprefereerd worden boven buren van minder goede kwaliteit. Op deze manier wordt een rij van oplossingen van steeds betere kwaliteit verkregen en de beste oplossing is dikwijls van goede kwaliteit. In de loop der tijd zijn er diverse soorten lokale zoekalgoritmen ontwikkeld. De meest voorkomende soorten worden in Hoofdstuk 4 beschreven aan de hand van een raamwerk waarbinnen vrijwel alle lokale zoekalgoritmen passen.

In Hoofdstuk 5 worden diverse lokale zoekalgoritmen uit de literatuur beschreven die ontwikkeld zijn voor het job shop schedulingprobleem. Hun kwaliteit wordt beoordeeld aan de hand van de resultaten die gevonden zijn voor dertien algemeen beschikbare probleeminstanties. Tevens worden in deze beoordeling de benodigde rekentijden voor het verkrijgen van deze resultaten meegenomen. Geconcludeerd mag worden dat algoritmen waarin verschillende combinaties van buurruimtefuncties en zoekstrategieën gebruikt worden en algoritmen waarin combinaties van lokaal zoeken en partiële aftelling gebruikt worden de beste resultaten geven.

In het laatste hoofdstuk geven we een aanzet die moet leiden tot toepassingen van lokale zoekalgoritmen voor het gegeneraliseerde job shop schedulingprobleem. We geven een aantal eisen waaraan een buurruimtefunctie minimaal moet voldoen, opdat een lokaal zoekalgoritme zich niet beperkt tot het zoeken in een te klein deelgebied van de oplossingsruimte en daardoor grote kans loopt oplossingen van goede kwaliteit niet te kunnen vinden. Vervolgens beschrijven we een aantal buurruimtefuncties die in voldoende mate aan deze eisen voldoen. In beginsel wordt bij elk van deze buurruimtefuncties een buur gedefinieerd door aan een enkele operatie een eventueel andere toegestane collectie van machines toe te wijzen en deze zodanig in te voegen tussen de overige operaties dat het geheel weer een toegestaan schedule oplevert. Hierbij mogen de starttijden van de overige operaties veranderen, maar blijven de onderlinge volgordes van deze operaties gelijk wanneer hun gekozen machinecollecties een machine gemeen

hebben. Het invoegen van een operatie kan op verschillende manieren gebeuren. We beschouwen de volgende drie mogelijkheden. Bij de eerste mogelijkheid mag iedere operatie op ieder van haar machinecollecties op een willekeurig toegestane plaats ingevoegd worden. Bij de tweede mogelijkheid mag iedere operatie op ieder van haar machinecollecties ingevoegd worden, maar nu zijn alleen die invoegingen toegestaan die, gegeven de keuze van de machinecollectie, een schedule opleveren met de kleinst mogelijke lengte. Bij de derde mogelijkheid mag iedere operatie alleen zodanig op een toegestane wijze ingevoegd worden, dat er geen andere invoegingen mogelijk zijn waarbij het resulterende schedule een kleinere lengte heeft. Voor de laatste twee buurruimtefuncties wordt een efficiënte methode gegeven om een best mogelijke invoeging te bepalen. Voor elk van de drie genoemde buurruimtefuncties is er een klasse van operaties aan te geven waarvan een invoeging nooit zal leiden tot een schedule met een kortere lengte. We beschouwen daarom bovendien varianten van deze buurruimtefuncties waarin invoegingen van dergelijke operaties niet toegestaan zijn.

Als laatste onderwerp bestuderen we de bereikbaarheid van deze buurruimtefuncties. Voor een buurruimtefunctie noemen we een oplossing bereikbaar vanuit een andere oplossing als er een reeks van stappen gemaakt kan worden beginnend bij de laatstgenoemde oplossing en eindigend bij de eerstgenoemde, zodanig dat in elke stap van een oplossing naar een buur wordt overgegaan. Van elk van de zes buurruimtefuncties voor het GJSSP geven we aan of vanuit elke willekeurige oplossing elke andere oplossing bereikt kan worden en of vanuit elke willekeurige oplossing een optimale oplossing bereikt kan worden. De eigenschap van bereikbaarheid geeft aan in welke mate een lokaal zoekalgoritme in staat is de oplossingsruimte te doorzoeken.

Curriculum vitae

Robert Johannes Maria Vaessens was born on March 10, 1966 in Cadier en Keer, the Netherlands. In 1984 he obtained his VWO-diploma at the Sint Maartenscollege in Maastricht. From September 1984 to April 1990 he studied mathematics at Eindhoven University of Technology and specialized in discrete mathematics. From February 1989 to May 1989 he visited the University of Århus in Denmark. In April 1990 he graduated cum laude under supervision of Prof.dr. J.H. van Lint and dr. E.H.L. Aarts with a master's thesis on the application of genetic algorithms to finding ternary codes with given length and distance. From May 1990 to July 1990 he was employed at Eindhoven University of Technology to write a paper on the subject of his master's thesis. In September 1990 he started as a Ph.D. student at Eindhoven University of Technology under the supervision of Prof.dr. J.K. Lenstra and Prof.dr. E.H.L. Aarts. The project he worked on was funded by the Landelijk Netwerk Mathematische Besliskunde (Dutch Graduate School in the Mathematics of Operations Research). During the first two years he attended the courses of this network, of which he obtained his diploma in 1993. From September 1994 to December 1994 he visited the Politecnico di Milano in Milan, Italy.

Stellingen

behorende bij het proefschrift

Generalized Job Shop Scheduling:

Complexity and Local Search

van

Robert Johannes Maria Vaessens

I

Om de kwaliteit en de efficiëntie van door verschillende onderzoekers ontworpen benaderingsalgoritmen voor een optimaliseringsprobleem goed te kunnen vergelijken is het noodzakelijk dat deze algoritmen getest worden op een algemeen beschikbare verzameling van probleeminstanties en dat de voor deze tests benodigde rekentijden uitgedrukt kunnen worden in een grootheid die niet afhangt van de computer waarop deze tests uitgevoerd zijn.

R.J.M. VAESSENS (1995), Dit proefschrift, hoofdstuk 5.

II

Van Laarhoven, Aarts en Lenstra [1992] beschouwen een buurruimtefunctie voor het job shop scheduling probleem. In Theorem 1 stellen zij dat het voor elke toegelaten oriëntatie mogelijk is om een rij van transities binnen de buurruimtegraaf te definiëren, welke eindigt in een toegelaten oriëntatie corresponderend met een optimaal schedule. Het bewijs van deze stelling is niet correct. De gemaakte fout is echter eenvoudig te herstellen.

P.J.M. VAN LAARHOVEN, E.H.L. AARTS, J.K. LENSTRA (1992), Job shop scheduling by simulated annealing, *Oper. Res. 40*, 113-125.

III

Aarts en Van Laarhoven [1985] stellen een koelschema voor hun simulated annealing algoritme voor, waarin het verschil tussen de koelparameters c_k en c_{k+1} behorende bij twee opeenvolgende homogene Markovketens groter is naar mate de variantie σ_k van de kosten van de oplossingen gevonden in de k-de Markovketen kleiner is. Het is te verwachten dat simulated annealing beter werkt als de correlatie tussen het decrement $c_k - c_{k+1}$ en de variantie σ_k positief in plaats van negatief is.

E.H.L. AARTS, P.J.M. VAN LAARHOVEN (1985), Statistical cooling: a general approach to combinatorial optimization problems, *Philips J. of Research 40*, 193-226.

IV

Zowel het twee-machine flow shop scheduling probleem als het twee-machine open shop scheduling probleem met transporttijden tussen de operaties van een opdracht is NP-lastig in de sterke zin. Dit is zelfs het geval als geëist wordt dat voor elke opdracht de bewerkingstijden van haar beide operaties aan elkaar gelijk zijn.

R.J.M. VAESSENS, M. DELL'AMICO (1995), *Flow and Open Shop Scheduling on Two Machines with Transportation Times and Machine-Independent Processing Times is NP-hard*, Ongepubliceerd manuscript.

V

Zij voor $n, r \in \mathbb{N}$ en $s \in \mathbb{Z}$ de grootheden $M_n^r(s)$ en $N_n^r(s)$ gedefinieerd door

$$M_n^r(s) = \left| \left\{ x \in \{0, \ldots, r-1\}^n \mid \sum_{i=1}^{n} x_i = s \right\} \right|$$

en

$$N_n^r(s) = \left| \left\{ x \in \{0, \ldots, r-1\}^n \mid \sum_{i=1}^{n} x_i \leq s \right\} \right|.$$

Dan geldt:

$$M_n^r(s) = \sum_{k=0}^{\lfloor s/r \rfloor} (-1)^k \binom{n}{k} \binom{n-1+s-kr}{n-1}$$

en

$$N_n^r(s) = \sum_{k=0}^{\lfloor s/r \rfloor} (-1)^k \binom{n}{k} \binom{n+s-kr}{n}.$$

VI

Beschouw het job shop scheduling probleem waarbij elke opdracht precies één operatie op elke machine heeft. De grootte van een instantie van dit probleem wordt dus gekarakteriseerd door het aantal opdrachten en het aantal machines. Random gegenereerde instanties van dit probleem zijn naar verwachting lastiger optimaal op te lossen naarmate de machinevolgordes van de opdrachten onderling een grotere gelijkenis vertonen. Random gegenereerde instanties van het flow shop scheduling probleem zijn derhalve het lastigst op te lossen.

E. TAILLARD (1993), Benchmarks for basic scheduling problems, *European J. Oper. Res. 64*, 278-285.

VII

Zij C een ternaire code van lengte 8 en minimum afstand 4 waarbij voor alle codewoorden $\mathbf{c} \in C$ de volgende voorwaarden gelden:

- $\mathbf{c} + (11102220) \in C$;
- $2 \cdot \mathbf{c} \in C$;
- $\mathbf{c}^{(15)(26)(37)(48)} \in C$;
- $\mathbf{c}^{(123)(567)} \in C$.

Hier eisen de derde en vierde voorwaarde dat voor elk codewoord $\mathbf{c}$ het woord verkregen door de aangegeven permutatie op $\mathbf{c}$ toe te passen ook weer een codewoord is. Een dergelijke code C bevat ten hoogste 99 codewoorden. De code die de woorden van Figuur 1 bevat en die aan bovenstaande voorwaarden voldoet bestaat uit 99 woorden. Deze code is in essentie uniek.

00000000
00012221
01200210
01201021
00110022
00122202
00101211

Figuur 1: Codewoorden behorende tot een ternaire (8,99,4)-code

R.J.M. VAESSENS, E.H.L. AARTS, J.H. VAN LINT (1993), Genetic algorithms in coding theory – a table for $A_3(n, d)$, *Discrete Appl. Math. 45*, 71-87.

VIII

Zij $p(d)$ de door de N.V. Nederlandse Spoorwegen gehanteerde prijs van een treinkaartje uit een bepaalde categorie voor een treinreis binnen Nederland over een afstand van d kilometers ($d \in \mathbb{N}$). Voor elk van de categorieën dient de functie p te voldoen aan:

$$\forall d_1, d_2 \in \mathbb{N} : p(d_1 + d_2) \leq p(d_1) + p(d_2).$$

Dit is echter niet het geval.

N.V. NEDERLANDSE SPOORWEGEN (1995), *NS-Reizigerstarief*, 1 januari 1995.

IX

Brouwer [1978] concludeert uit de tekst van een affiche van 30 oktober 1843 aangaande de openbare aanbestedingen voor het leveren en plaatsen van 388 gietijzeren en 356 hardstenen grenspalen, welke geplaatst dienden te worden op de Belgisch-Nederlandse grens, uit het feit dat de gietijzeren palen genummerd zijn, en uit het feit dat de paal met nummer 1 op het drielandenpunt van België, Duitsland en Nederland staat, dat de paal met het hoogste nummer, welke nabij de monding van het Zwin in de Noordzee tussen Knokke en Cadzand staat, het nummer 388 draagt. Er zijn ten minste twee redenen aan te geven waarom deze conclusie niet correct is. Bovendien is de bewering van Brouwer dat van de hardstenen palen geen voorbeelden meer over zijn onjuist.

T. BROUWER (1978), *Grenspalen in Nederland*, De Walburg Pers, Zutphen.

Procès-verbal descriptif de la délimitation entre les royaumes de Belgique et des Pays-Bas, annexé à la convention de limites conclue à Maestricht le 8 août 1843, *Moniteur Belge*, 15 april 1887, 84-114.

X

Men kan de gevaren van wildwaterkanoën aanzienlijk reduceren door de keuze van de te bevaren rivier aan te passen aan de kunde van de minst ervaren kanoër. Bovendien dient men op de hoogte te zijn van gevaarlijke passages en deze pas te bevaren nadat deze na verkenning bevaarbaar geacht worden.

XI

Het is een slechte zaak dat in de geschiedenislessen op Limburgse middelbare scholen nauwelijks aandacht wordt besteed aan de geschiedenis van het gebied dat thans door de provincie Limburg bestreken wordt.