EINDHOVEN
UNIVERSITY OF
TECHNOLOGY

Algorithmic support for automated planning boards

Citation for published version (APA):
Wennink, M. (1995). *Algorithmic support for automated planning boards*. [Phd Thesis 1 (Research TU/e / Graduation TU/e), Mathematics and Computer Science]. Technische Universiteit Eindhoven. https://doi.org/10.6100/IR444297

DOI:
10.6100/IR444297

Document status and date:
Published: 01/01/1995

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 23. Aug. 2022

Algorithmic Support for Automated Planning Boards

Marc Wennink

ALGORITHMIC SUPPORT

FOR

AUTOMATED PLANNING BOARDS

Dit onderzoek is gefinancierd door de Nederlandse Organisatie
voor Wetenschappelijk Onderzoek (NWO).

ALGORITHMIC SUPPORT

FOR

AUTOMATED PLANNING BOARDS

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 15 september 1995 om 14.00 uur

door

Marc Wennink

geboren te Doetinchem

Dit proefschrift is goedgekeurd
door de promotoren:

prof.dr. J.K. Lenstra
en
prof.dr. M.W.P. Savelsbergh

Preface

I remember vividly how, thirteen or fourteen years ago, my father had to make a schedule for the annual sports day in our village. Two hundred children had made a selection of a number of events that they could join in. My father's job was to find a way to let all children participate in the selected events and to take care that not too many children would be assigned to the same events at the same time. It took him weeks to find a schedule that he was satisfied with. At the end of this period, all four walls of his room were covered with sheets of paper on which he had written all kinds of information in the form of tables, lists, and figures. He had clearly enjoyed himself. Still, I had a feeling that the job could have been done in a more efficient way.

At that time I did not know that my father's task had been to make a *plan*, an assignment of *processes* to *resources* and *time intervals*, and that an *automated planning board* would have been a helpful tool. Even less did I know that ten years later I myself would spend four years in Eindhoven, conducting research in the area of automated planning boards.

This thesis reflects my experiences of the last four years. The goal of my research has been to design a planning board generator, a system that generates planning boards automatically on the basis of only limited information. The first two chapters, dealing with the planning board generator and planning problems in general, represent the first two years of my research. In the course of these two years I realized that the development of a planning board generator was a very ambitious project, certainly if it had to be carried out by one person. Therefore I decided to focus on the aspect of planning boards that I found the most interesting: the algorithmic support of manipulations.

After several discussions with my advisors, I decided to start looking for common properties of scheduling problems that can be used in the design of algorithms. I decided to use the disjunctive graph representation of job shop scheduling problems as a starting point, because I thought that it would be possible to

use similar models for more general problems. Occasionally I obtained a nice result, but progress was slow.

Then, one day my roommate Arjen Vestjens stepped into the room with a bath-tub. The bath-tub represents a kind of cost function that is used to describe several situations occurring in practical problem situations. It turned out that these bath-tubs as well as more general cost functions could be incorporated in the basic model in an elegant way. I was so content with this generalization that I decided, with a proper sense of proportion, to call the new problem class the *general scheduling problem*. The second part of this thesis is devoted to this general scheduling problem.

Two conflicting objectives arise in the design of a planning board generator: generality and efficiency. In my research, I have also struggled with these two objectives. I thank my supervisor, Jan Karel Lenstra, and my advisors, Martin Savelsbergh and Stan van Hoesel, for their efforts in helping me to find the right balance.

I would also like to thank Roberto Lioce and Carlo Martini for performing the computational experiments.

I am grateful to several other people for their support in the last four years, but there are better ways of thanking them than by writing down their names. I want to make an exception, however, for the members of INGRE (Interdisciplinary Network of Groningen Econometricians). Planning our half-yearly weekends made great demands on my talents in solving difficult scheduling problems.

Marc Wennink Eindhoven, July 1995

Contents

1 Introduction **1**

1.1 Planning boards 1
1.2 A planning board generator 3
1.3 Algorithms 7
1.4 Overview of the thesis 9

2 Problem specification **11**

2.1 Introduction 11
2.2 Problem instances 13
2.2.1 An overview of the specification method 13
2.2.2 Time 15
2.2.3 Resources and processes 16
2.2.4 Specifying feasible assignments 20
2.2.5 Other relations between objects 28
2.2.6 Extensions 30
2.2.7 Views of the instance graph 31
2.2.8 Examples of problem instances 32
2.3 Problem types 36
2.3.1 Restrictions on the instance graph 37
2.3.2 Specifying problem types 38
2.3.3 Examples of problem types 43

3 The general scheduling problem **53**

3.1 Introduction 53
3.2 The job shop scheduling problem 54
3.3 The general scheduling problem 58
3.3.1 Maximum cost flow 58

3.3.2 Constraints 59
3.3.3 Objective functions 62
3.3.4 Definition of the general scheduling problem 66
3.4 Solutions 68
3.4.1 The solution network 68
3.4.2 Feasibility 69
3.4.3 Obtaining selection-optimal schedules from flows . . . 71
3.4.4 The reduced solution network 73
3.5 Other objective functions and constraints 74
3.5.1 Regular cost functions 74
3.5.2 Non-regular cost functions 75
3.5.3 Multiple time windows 77

4 Solution methods **79**
4.1 Introduction 79
4.2 Complexity 80
4.2.1 Complexity of finding optimal selections 80
4.2.2 Complexity of finding feasible selections 81
4.3 Construction methods 84
4.3.1 Dispatching 84
4.3.2 Insertion 88
4.3.3 An efficient insertion algorithm 95
4.4 Local search methods 106
4.4.1 Local search 106
4.4.2 Two neighborhoods 107
4.5 Relaxation 120
4.5.1 Lagrangian relaxation 121
4.5.2 Penalty methods 124
4.5.3 Comparison of the two relaxation methods 125

5 Discussion **127**
5.1 Towards a planning board generator 127
5.1.1 Providing the PBG with the required information . . . 128
5.1.2 Procedures for supporting representations and manipulations 130
5.2 Machine scheduling 132
5.2.1 The general scheduling problem 132
5.2.2 Solution methods 134
5.2.3 Suggestions for further research 135

Appendix: The test set **137**

References **141**

Samenvatting **145**

1

Introduction

1.1 Planning boards

A picture tells more than a thousand words. Henry Laurence Gantt (1861-1919) may have had this in mind when he developed the representation mechanism that is currently known as the Gantt chart. An example of a Gantt chart is given in Figure 1.1.

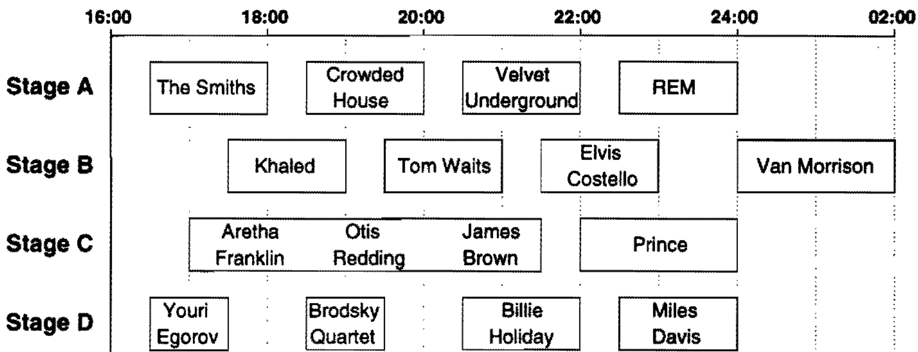

Figure 1.1: A Gantt chart.

This Gantt chart represents the program of a music festival. Sixteen artists and bands have been contracted by the organization, each of which will be performing on one of the four stages during a given time interval. For example, Tom Waits will perform on stage B from seven thirty until nine o'clock.

In general, the horizontal axis in a Gantt chart represents some *time period.* On the vertical axis, *resources* are set out, and the rectangles on the chart represent *processes.* In Figure 1.1, the resources are the stages, and the processes are the

performances of the artists. In other applications, the resources may be lecture rooms, machines, or vehicles, and processes may be courses that must be given, operations that must be performed, or goods that must be transported. The Gantt chart as a whole represents a *plan*, i.e., an assignment of processes to resources and time intervals.

Gantt developed his charts during the First World War at the Ordnance Bureau of the United States Army. Today, we use a Gantt chart mainly to represent a plan, but Gantt's original purpose was to provide methods for measuring the progress made in realizing a plan. He foresaw great possibilities for such methods (Gantt [1919]):

> ..., our record charts invariably indicate the capable men, and not only give us an indication of how to choose our leaders, but a continual measure of the effectiveness of their leadership after they are chosen.

In this thesis, I deal with planning problems. In these problems, we are given a set of processes, a set of resources, and a set of constraints. The processes must be assigned to the resources and time intervals in such a way that all constraints are satisfied. Planning problems arise in course scheduling, timetabling, production planning, vehicle dispatching, and many other application areas. In all these areas, *planning boards* are used to support the planner. A planning board is a tool that uses a Gantt chart to represent plans and provides the means to modify plans by manipulating the Gantt chart.

Planning boards can play a vital role in complex planning situations by integrating human insight and formal models (Anthonisse, Lenstra, and Savelsbergh [1988]). For example, when the quality of a plan has to be assessed, mathematical models can be used to test if all constraints are satisfied or to evaluate some criterion function, and the planner will use his own experience and knowledge of the problem situation to decide whether the plan as a whole is feasible.

Good representations of both the problem and the plan are crucial in order to get the highest benefit from this interaction. Although the Gantt chart is the primary representation in a planning board, other representations, such as data tables and inventory graphs, can also be incorporated. Using various representations of problem data and plans may lead to a better understanding of the problem being solved. Jones [1994] discusses the importance of representation and visualization in the context of optimization. In two wonderful books, Tufte ([1983, 1990]) treats the more general subject of envisioning information.

A planning board should also provide the means to manipulate the presented representations, so as to enable the planner to create and modify plans. The notion of manipulation should be interpreted broadly. In the terminology introduced by Anthonisse, Lenstra, and Savelsbergh [1988] with respect to interactive

planning systems, manipulations cover the entire spectrum from *assistant* functions to *advisor* functions. A planning board must provide the means to store and retrieve plans, to evaluate the quality of a plan, and to modify a given plan manually. On the other hand, the planning board must also be able to construct a plan by itself and to give suggestions for improving a given plan.

A planning board is generally used for more than one particular problem situation. Each such situation will be called a problem *instance*. A set of problem instances with well-specified common characteristics will be called a problem *type*. A planning board should be equipped to deal with all possible instances of some problem type. Consider, for example, the problem of assigning nurses to night shifts and day shifts in a hospital. Such a timetabling problem must be solved, say, every month. However, the number of available nurses and their desires with respect to vacation and days off are subject to change. Thus, every month a different instance of the nurse scheduling problem type must be solved, and a planning board must be capable to deal with each possible instance.

The combination of representations and manipulations results in a powerful tool for many different types of planning situations. Each problem type, however, requires its own sets of representations and manipulations. A single Gantt chart will suffice for certain machine scheduling problems, but in more complex production planning problems also inventory graphs are required. Similarly, the quality of a course schedule and the quality of a production plan are evaluated in completely different ways. One can say that each problem type requires its own planning board. Unfortunately, the development of an automated planning board is a highly time consuming process. The aim of my research has been to find out in what way the design of planning boards can be facilitated, and I have focused on the role of operations research.

1.2 A planning board generator

This thesis must be read in the light of the ultimate goal of my research: the development of a *planning board generator* (PBG). Given a specification of the problem type for which a planning board has to be developed and a specification of the desired representations and manipulations, a PBG should automatically create an initial version of the planning board. The context in which a PBG would be used is depicted in Figure 1.2.

A *planner* has to make plans for a number of instances of the same problem type and he thinks that a planning board may be a helpful tool. He therefore asks a planning board *designer* to build a planning board for that specific problem type. The designer asks for the characteristics of the problem type, the desired

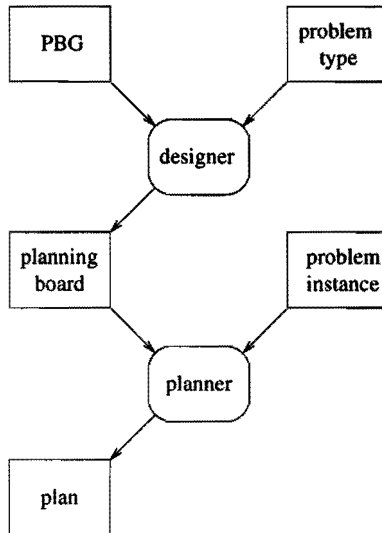

Figure 1.2: Context of a planning board generator.

representations, and the required manipulations. The designer activates the planning board generator to process this information and to produce an initial version of the planning board.

The idea that it should be possible to design a planning board generator has been inspired by the observation that planning boards that are used for entirely different purposes exhibit so many common elements. The same kinds of representations are provided and the same kinds of manipulations are performed. It seems a waste of time and energy to implement these common elements for each individual planning board from scratch. It must be possible to deal with them on a more general level and still obtain powerful planning boards.

Two conflicting objectives arise when the functionality of a planning board generator is discussed. On the one hand, a PBG should be as *general* as possible, supporting a braod variety of planning problem types; on the other hand, for a given problem type it should be able to generate a powerful planning board that is equipped to deal with the *specific* characteristics of that problem type in an efficient way. For a PBG to be of any use it is essential that the right balance between generality and efficiency is found.

There are two possible approaches in trying to obtain such a balance. In a 'top-down' approach, generally applicable methods are developed, which may

be adjusted to specific situations. In a 'bottom-up' approach, one develops efficient, problem-specific methods and tries to adjust these in order to deal with more general problem situations. The first approach leads to a guaranteed level of generality, possibly at the cost of efficiency. The main objective in the second approach is efficiency; if the desired level of efficiency cannot be maintained, then no further generalization is pursued.

We think that generality can be achieved for many aspects of a PBG without loss of efficiency, and the top-down approach seems most natural for these aspects. A *general problem class* will be identified, and methods will be developed that can handle all problems in this class. For aspects of a PBG in which it is not obvious how we can obtain generality while maintaining the desired level of efficiency, we must rely on a bottom-up approach. Ideally, this approach will result in sufficient generality, but we must take into consideration the possibility that methods will be developed that are only suited for specific subclasses of the general problem class. For each such subclass, a library of methods can be created. These methods must be reasonably efficient for all problem types in the considered class.

In this thesis, I discuss two topics that are important in the design of a PBG. First, I introduce a method for providing the PBG with information about the problem type for which a planning board is to be developed. A top-down approach is used in developing this method, since generality is essential. If we cannot give the desired information in a uniform way for all types of planning problems, then we must give up the hope of developing a system that automatically creates planning boards for all possible planning problems.

The second topic is the algorithmic support of manipulations. Powerful construction and improvement algorithms are very important for the advisor function of a planning board. Efficiency is essential in these algorithms. The option to let the planning board generate an initial plan will not be used too often if execution of the corresponding algorithm takes too much time. Generally applicable algorithms are bound to be too time-consuming or to give bad results for most individual problem instances. Therefore, we have decided to apply a bottom-up approach, and to study first the structure of a particular problem type, the job shop scheduling problem. The algorithms that have been applied successfully to this problem can be modified in such a way that they can deal with a more general class of problems. I believe that further generalization is still possible and that the discussed methods can be applied to more general machine scheduling problems as well. For other classes of problems, such as timetabling and resource constrained project scheduling problems, however, different methods will probably have to be developed.

Planning boards are *decision support systems.* They are designed to support decision making in practical planning situations. Much work has already been done in order to facilitate the development of decision support systems and, more specifically, interactive planning systems. For a survey of computer graphics standards, user interfaces, and user interface management systems, see Jones [1992].

In the literature, a few systems that are able to deal with more than one planning situation and that offer some possibilities for problem specification have been presented. Examples are the systems developed by Jackson et al. [1989], Jones and Maxwell [1986], and Woerlee [1991]. These systems differ from a PBG in that they are developed for a particular application area: material logistics, manufacturing scheduling, and production scheduling, respectively. The use of a PBG is not restricted to a particular application area. A very general problem class is considered: processes must be assigned to resources over time. This general structure allows the use of the same representation mechanism (e.g., Gantt charts) and similar manipulations for problems in completely different application areas. I think that the common aspects of the problems in this general problem class can be exploited effectively to facilitate the development and implementation of planning boards for all these application areas.

Although it is not concerned with planning tools that use the Gantt chart as the main representation mechanism, the *graph-based modeling system* (GBMS) of Jones [1990] is conceptually very similar to our PBG. Both the GBMS and the PBG aim at facilitating the development of user interfaces for interactive systems. In a GBMS, attributed graphs are used as representation mechanism. The created interface is called an *instance editor.* An instance editor enables a modeler to add and delete nodes and edges in order to construct a graph of the appropriate type. In a GBMS, the *designer* specifies a graph type and the manipulations that can be performed on graphs of that type. This specification is sufficient to automatically generate an instance editor for graphs of that type. Using a similar terminology, one can say that a planning board enables a planner to perform manipulations on (representations of) plans in order to construct a solution to a problem of the appropriate type. In our setting, the designer specifies a problem type and the desired representations and manipulations. From this specification, the PBG automatically constructs a planning board for instances of this type.

An important difference between a GBMS and a PBG is the representation mechanism that is used, attributed graphs versus Gantt charts. Another important difference relates to the way in which a graph type or problem type and the corresponding sets of manipulations are specified. Jones's GBMS allows the designer to create his own types of nodes and edges, each with its own set of attributes.

Also the manipulations can be developed by the designer. Although this bottom-up approach gives the designer a lot of freedom, it also requires knowledge of the relatively unfamiliar theory of graph-grammars. For our PBG, in contrast, we apply a top-down approach. There exists a general problem class, of which all problem types that the PBG can handle are special cases. The designer of a planning board can specify his particular problem type by defining the appropriate subclass of the general problem class. Similarly, a set of possible manipulations is given, from which a planning board designer chooses a subset. The designer in a GBMS, on the other hand, must specify the desired node types and edge types, as well as the desired manipulations, from scratch. The advantage of this top-down approach is that the implementation of the PBG can be specifically equipped to deal with the allowed problem types only, thus yielding a more effective and efficient system. Furthermore, it has enabled us to develop a specification method that exploits and highlights the common structural properties of the problem types, in contrast to the use of graph-grammars, which are more generally applicable. A disadvantage is that we cannot obtain full generality. Our PBG will only be able to deal with problems in the general problem class. However, we have tried to keep this class as general as possible indeed, only leaving problem types for which planning boards are not useful out of consideration.

1.3 Algorithms

The most obvious area in which operations research can contribute to the development of a planning board generator is the algorithmic support of manipulations. Many manipulations can be performed more efficiently if an appropriate mathematical model is used. Examples are the evaluation of the quality of a plan, suggestions for improvement of a given plan, and the construction of an initial plan. It must be possible to use the vast amount of knowledge, insight, and experience that has been obtained in optimization in order to develop appropriate models and algorithms. As discussed before, these models should demonstrate a balance between generality and efficiency. In my opinion, a PBG should support a limited number of fairly large problem types for which reasonably efficient algorithms can be developed. Together, these problem types should cover the area of planning problems as much as possible.

A class of planning problems that has received much attention is deterministic machine scheduling (Lawler et al. [1993]). In this class, each resource, or *machine*, can perform at most one process, or *operation*, at a time, and all information that defines a problem instance is known with certainty in advance. Research in machine scheduling is characterized by a huge number of problem

types, many of which are of only limited practical importance. Sometimes, the introduction of new models seems to be justified by the fact that they have not been studied before. As a consequence, the differences between problem types receive more attention than their common aspects.

In this thesis, I try to identify common structural properties that can be exploited in the design of efficient algorithms for larger problem classes. This leads to the introduction of the *general scheduling problem*. The general scheduling problem can be seen as a generalization of the well known job shop scheduling problem. In the job shop problem, there is a number of *jobs*, consisting of operations that must be performed in a prespecified order on prespecified machines. The problem is to determine a processing order of the operations in such a way that the makespan, i.e., the completion time of the last operation, is minimal. The job shop scheduling problem has earned a reputation for being one of the hardest problems in combinatorial optimization. Even small instances of this problem are very difficult to solve. In recent years, some progress has been made in developing efficient algorithms for the job shop scheduling problems by using so-called local search techniques. These algorithms exploit the fact that the quality of a solution can be evaluated efficiently by computing the longest path in some network.

The general scheduling problem generalizes the job shop scheduling problem in the area of both constraints and objective functions. It allows for release times, deadlines, minimum and maximum delay constraints, as well as non-regular performance measures and multi-criteria objectives. I will show that the quality of solutions of the general scheduling problem can be evaluated by solving a maximum cost flow problem, which can be seen as a generalization of the longest path problem. This result enables us to modify solution methods that have been applied successfully to the job shop scheduling problem in such a way that they can be applied to the more general problem as well.

In a sense, the research presented in this thesis is related to the work of Nuijten [1994]. He generalizes (other) results obtained for the job shop scheduling problem and incorporates these results in a constraint satisfaction framework for a broad class of scheduling problems. Both constraint satisfaction and local search are not always considered as worthy members of the optimization techniques establishment. This is not the right attitude. Local search and constraint satisfaction can give good results *only if* information about the structure of the problem at hand is exploited. In this sense, they are not much different from, for example, branch-and-bound.

1.4 Overview of the thesis

The remainder of this thesis is organized as follows.

In Chapter 2, the basic objects that occur in planning problems, processes and resources, are discussed in more detail. I show how attributed graphs can be used to represent properties of objects and relations between objects and introduce a specification method, based on attributed graphs, for problem instances. This method serves as a basis for the problem type specification method. A number of examples of problem instances and types demonstrate the power of the introduced methods. This chapter is based on a paper by Martin Savelsbergh and myself [1994].

In Chapter 3, I discuss the job shop scheduling problem and in particular the role that is played by longest paths in solution methods for this problem. Then, the maximum cost flow problem, which can be seen as a generalization of the longest path problem, is discussed, and it is shown that the maximum cost flow problem is useful in the context of generalizations of the job shop scheduling problem. More general constraints and objective functions are presented, culminating in the introduction of the general scheduling problem. Solutions of general scheduling problems can be represented by solution networks, which are shown to be useful for feasibility testing and quality evaluation.

Solution methods for the general scheduling problem are presented in Chapter 4. First, construction methods, using dispatching (list scheduling) and insertion techniques, are discussed. An efficient implementation of the insertion algorithm is not easy to obtain, but I show that it is possible to develop such an algorithm when the objective is similar to makespan minimization. Computational experiments show that the insertion algorithm outperforms dispatching algorithms. Next, local search methods are treated. Properties of two neighborhoods are studied, and computational experiments are performed for problems with makespan minimization as objective. I show that local search can be readily applied to a large number of instances of the general scheduling problem. Furthermore, relaxation techniques that make it possible to deal with the other instances as well are discussed. More research is required, however, in order to obtain efficient local search algorithms for the entire class of general scheduling problems. The discussion of the insertion algorithm is based on a paper by Rob Vaessens and myself [1995]; the computational experiments have been conducted in cooperation with Roberto Lioce and Carlo Martini [1995].

Finally, in Chapter 5, I discuss to what extent the presented results contribute to the development of a planning board generator and to the theory of machine scheduling. Moreover, I give some suggestions for further research.

2

Problem specification

2.1 Introduction

A PBG must be capable of generating planning boards for a broad variety of problem types, including timetabling, course scheduling, and production scheduling problems. At first sight, these problem types have only little in common, but on closer examination we find several common aspects, both in the structure of the problems and in the way that planning boards can be used in solving these problems. These common aspects make it possible to generate automatically at least some elements of a planning board on the basis of relatively little information about the problem type and expressed desires with respect to the representations and manipulations.

Any planning board contains a number of procedures to support various representations and manipulations. Although the functionality of these procedures is almost the same for all planning boards, the actual implementation must be tailored to the specific problem situation for which a planning board is to be designed.

Consider, for example, the procedure for drawing Gantt charts. In a Gantt chart, resources are represented on the vertical axis, the horizontal axis represents some time interval, and rectangles represent processes that are assigned to resources and time intervals. These properties hold for all planning problems. Consider now a particular problem type in which two kinds of resources, machines and employees, appear, time is measured in hours, with ten hours in a day and 5 days in a week, and processes are divided into groups. In a planning board for this problem type, the two kinds of resources, machines and employees, appear separatedly on the vertical axis of the Gantt chart, the appropriate time system is set out on the horizontal axis, and processes that belong to the same group are represented by rectangles of identical color.

Another example is the 'move' manipulation, which enables a planner to determine a new assignment for some process. In all situations, and for all planning problems, the planner must specify which process is to be moved and where it is to be positioned, and it must be checked if the new assignment is feasible. Consider a specific problem type, in which each process must be assigned to a combination of a machine and an employee, and certain precedence constraints must be satisfied. A planning board for this particular problem type will offer the possibility to specify the process that is to be moved, then ask to which machine-employee-combination it is to be assigned and at what time the process must be started. It checks whether the proposed assignment is feasible, the capacity limitations of the resources are not violated, and the precedence constraints are satisfied

For each of the various representations and manipulations that may be used in planning boards, a PBG will contain a basic implementation. On the basis of the specification of the problem type and the desired representations and manipulations, the PBG will tailor these basic implementations to provide customized implementations that are appropriate for the considered problem type and that match the expressed desires as closely as possible. This tailoring may be as simple as setting certain parameter values and incorporating predefined subroutines in the basic implementations, but it may also involve generating pieces of new code.

The examples above demonstrate that a precise specification of the problem type for which a planning board is to be generated is essential. It is especially important that the designer of a planning board can specify those characteristics of a problem type that affect the implementation of the representations and manipulations. In this chapter, I discuss how problem instances can be specified, and how the proposed method can serve as a basis for a specification method for problem types.

In the specification of problem instances we can use the general problem structure as a starting point: processes must be assigned to resources and time intervals in such a way that certain constraints are satisfied. A problem instance can thus be described by specifying the processes and resources and their properties, the time system, and the constraints that must be satisfied. In the proposed specification method, we make use of attributed graphs. Processes and resources are represented by nodes and their properties by attributes. Examples of attributes are the size of a process, and the capacity of a resource. Various edges, arcs, and auxiliary nodes are introduced to describe relations between objects. In this way, we are able to formulate to which resources we can assign a process and which constraints must be satisfied.

The class of instances that can be specified with the proposed method will be called the general problem class. Problem types are then defined by imposing restrictions on the general problem class. A problem type consists of all instances in the general problem class that satisfy certain restrictions. The general problem class as well as individual instances are considered as special kinds of problem types.

In Section 2.2, we describe the instance specification method in detail, demonstrating how the most important aspects of problem instances can be formulated in terms of attributed graphs. The main goal is, however, not to discuss how individual properties can be specified, but to illustrate a generally applicable approach: problem instances are described in terms of objects and different kinds of relations between objects. Objects are represented by nodes, properties of objects are described in terms of attributes, and each kind of relation is described by using a specific graph construct. This approach makes it possible to modify, if necessary, the proposed specification method in a natural way. If it turns out that a particular property of an object or some relation cannot be described properly, one may consider introducing a new attribute or an appropriate graph construct.

In Section 2.3, we show how problem types can be specified by formulating restrictions on the general problem class.

We have included several examples of problem instances and types in order to demonstrate the power and wide applicability of the introduced methods.

2.2 Problem instances

2.2.1 An overview of the specification method

The instance specification method has been developed with the following objectives in mind:

1. It should be possible to specify instances of all problems for which we feel that a planning board provides a useful tool. In fact, the method will implicitly define a general problem class. Note that this class should be fairly large for a PBG to be of any interest.

2. The instance specification method must form the basis of a type specification method. It should therefore be possible to specify subclasses of the general problem class by identifying the typical properties of problem instances within such subclasses.

3. It should be possible to use the specification of an instance to efficiently perform some of the tasks of a planning board, such as verifying feasibility of

assignments and providing information on various aspects of the problem in hand.

We have tried to achieve these objectives by using a widely applicable and well-studied paradigm, namely attributed graphs, and by stressing the common structural properties of the problems.

Attributed graphs are widely used as a mechanism for describing many kinds of models in different fields of research, from sociology to chemistry. It is therefore not only a mechanism that many people are familiar with, but also one that has proven to be very flexible. Furthermore, there exists a massive theory on problems related to graphs, which we can use.

All problems for which a planning board is useful deal with assignments of processes to resources and time intervals. Therefore it must be possible to specify the characteristics of the processes and the resources that occur in the instance as well as the time system. Furthermore, it must be possible to specify relations between processes, between resources, and between processes and resources. The most important relation that has to be specified is which assignments of processes to resources are feasible and which are not.

The attributed graph associated with an instance will be called an *instance graph*. The instance graph must be able to represent a variety of situations, e.g., a process that can be performed on any resource out of a set of resources, a process that requires a specific combination of resources, and a process that can be performed on any combination of resources out of a set of possible combinations of resources.

The core structure of the instance graph specifies the feasible assignments of processes to resources. Each process and each resource are represented by a node. For each possible choice of resources and each possible combination of resources an auxiliary node is introduced that is connected to the objects involved and thus forms a $K_{1,n}$ where n is the number of objects involved.

We can specify several other relations between objects by using graph constructs as well. Binary relations, i.e., relations involving exactly two objects, can be modeled by an edge or an arc. An example of a binary relation is the precedence relation indicating that one process must be performed before an other one. n-Ary relations, i.e., relations involving a set of n objects ($n > 2$), can be modeled by a $K_{1,n}$.

Not all aspects of problem instances can be represented in terms of nodes, arcs, and edges. These aspects can be divided into two categories: properties of individual objects, and properties of assignments. Examples of the first category are the sizes of the processes, the capacities of the resources, and the length of the planning period. These properties will be specified as attributes of the corre-

sponding nodes, or as attributes of the time system. An example of the second category is the processing time of a process when it is assigned to a particular combination of resources. In order to specify these kinds of properties, attributes of appropriate auxiliary nodes are introduced.

Before we discuss the various elements of the instance graph, we will spend some time on the key elements of plans: time, processes, and resources.

2.2.2 Time

Time plays an important role in planning boards. The way in which time occurs is different for different problems. For a timetabling problem for schools, the planning period may cover five days consisting of eight hours, with an hour being the smallest time unit, whereas for a machine scheduling problem, the planning period may be one working day of 12 hours, with a minute being the smallest time unit. For flexibility purposes we allow the introduction of a time system for each problem instance. This time system consists of time units on different levels. The smallest time unit is specified in the first level, the second smallest time unit in the second level, etc. For the jth time unit also the conversion factor with respect to the $(j-1)$st time unit is given. For example, a planning period covering one day, with one second being the smallest time unit, is specified in the following way.

number of levels: 4

level 1

unit name: second

level 2

unit name: minute

conversion factor: 60

level 3

unit name: hour

conversion factor: 60

level 4

unit name: day

conversion factor: 24

planning period: 1 day

2.2.3 Resources and processes

Resources and their attributes

Depending on the application, a resource can be almost anything. In a production planning application, personnel, money, raw materials, and machines may all be resources. In a time-tabling application for a school, the resources may be classrooms and teachers.

An important concept related to resources is that of a *capability*. A resource may possess several capabilities, i.e., it may be able to perform various tasks. A mechanic may be qualified to change oil and to repair brakes. A teacher may be qualified to teach mathematics as well as physics.

Although a resource may be able to perform different tasks, some set-up or change-over may be necessary before a particular task can be performed. Such a set-up brings the resource in the required *mode*. The corresponding set-up time can be either sequence dependent or sequence independent. In the first case, the set-up time is completely determined by the required mode. In the second case, the set-up time depends on the current mode of the resource and the mode that is required. The concept of mode is different from the concept of capability. A sawing machine may possess only one capability, the capability *sawing*, but several modes, one for each possible size of an object that can be sawn.

Most resources are not free commodities. Their utilization by some process will affect the possibilities for utilization by other processes. Quantities related to the utilization of a resource are *usage* and *consumption*.

The *usage* of resource R by process P at time t is a quantity that indicates to what extent R is occupied by P. The *total usage* of R at time t, i.e. the sum of the usage of all processes assigned to R at time t, is limited by the *capacity* of R at time t.

The *consumption* of resource R by process P during a time interval $[t_1, t_2]$ is a quantity that indicates to what extent R is consumed by P during that interval. The *total consumption* of R up to time t^*, i.e. the sum of the consumption of R up to t^* over all processes assigned to R, is limited by the *supply* of R up to t^*.

Based on the different kinds of utilization the following distinction between resources can be made (see also Błaźewicz et al. [1986]):

- *Renewable resources*: Resources for which only their total usage at every moment is constrained. An example of such a resource is a painting machine, which can paint all day but no more than one object at the same time.
- *Non-renewable resources*: Resources for which only their total consumption up to any given moment is constrained. An example of such a resource is finances.

- *Doubly constrained resources*: Resources for which both total usage and total consumption are constrained. An example is personnel. An employee can perform only a limited number of tasks at the same time and can perform these tasks only during a limited number of man hours.

Usage and consumption can both occur in either continuous or discrete quantities. A painting machine may be able to paint up to ten objects at the same time. Its capacity then is ten units, and the assignment of a painting job to this machine implies a usage of one unit. It is impossible to paint half objects, and therefore usage, in this case, is a discrete quantity. The storage capacity of a truck, in contrast, can be used in continuous quantities. With respect to consumption, fuel will be consumed in continuous quantities, but in assembling a car steering wheels will be consumed in discrete quantities.

Mostly, usage and consumption will be modeled as one-dimensional quantities. Multi-dimensionality, however, is possible. For an employee there may exist a limit on the number of hours per day he may work (e.g., 10 hours) as well as a limit on the number of hours per week (e.g., 40 hours). This cannot be modeled by supplies that replenish the 'hours-inventory' to 10 at the beginning of each day, because the week-limit may then be exceeded. By modeling consumption and supply as two-dimensional quantities we can solve this problem. Another example of two-dimensional usage is found in transportation problems, where the usage of a truck is limited with respect to volume as well as weight.

The capacity and supply of a resource may change during the planning period as a result of instance-dependent and plan-dependent factors. For example, during a certain period the capacity of some machine may be smaller than normal because of maintenance, or the supply of some half-product will increase at the moment that a process that represents the production of that half-product is completed. Instance-dependent changes in capacity and supply will be modeled in terms of attributes of the resources. Plan-dependent changes in the supply of a resource will be modeled as (possibly negative) consumption of that resource. Similarly, the available capacity of a resource is reduced when it is used by a process, and it is increased again as soon as that process is completed.

A resource that possesses several capabilities may have different performance levels with respect to these different capabilities. The aforementioned mechanic may be very fast in changing oil, but he may be very slow when it comes to repairing brakes. In order to handle such differences, resource characteristics related to speed, usage, and consumption have to be specified for each of its capabilities.

A resource node has the following attributes:

name: A name that uniquely identifies the resource.

availability periods: A set of time intervals specifying the time periods during which the resource is available.

category: An indicator that specifies the type of resource, i.e., renewable, non-renewable, or doubly constrained.

number of modes: A natural number. For each mode the following must be specified:

- name: A name that uniquely identifies the mode.
- set-up time: A function of the status of the resource, i.e., the active mode, that computes the set-up time that is needed to bring the resource into this mode.

usage: This attribute consists of three sub-attributes.

- divisibility: An indicator that specifies whether usage of the considered resource is modeled as a continuous or as a discrete quantity. In case usage is continuous the value is 0 and in case usage is discrete the value is the discretization unit.
- dimension: A natural number representing the dimension of usage of the considered resource.
- capacity: A function of time, describing the plan-independent capacity.

consumption: This attribute consists of three sub-attributes.

- divisibility: An indicator that specifies whether consumption of the considered resource is modeled as a continuous or as a discrete quantity. If consumption is continuous the value is 0 and if consumption is discrete the value is the discretization unit.
- dimension: A natural number representing the dimension of consumption.
- supply: A function of time, describing the plan-independent supply.

number of capabilities: A natural number. For each capability the following four sub-attributes must be specified:

- name: A name that uniquely identifies the capability. If different resources possess the same capability, the same name must be used when referring to this capability.
- speed: A real number that is used to determine the duration of a process when that process is assigned to this resource. It is not necessarily equal to the actual speed of the resource, as will be illustrated in Section 2.2.4.

usage factor: A real number that is used to determine the usage of the resource when a process is assigned to this resource. If this resource is used in combination with other resources, then this attribute is also used to determine the usage of those resources. In Section 2.2.4 we will discuss the use of the usage factor extensively.

consumption factor: A real number that is used to determine the consumption of the resource when a process is assigned to this resource. If this resource is used in combination with other resources, then this attribute is also used to determine the consumption of those resources. We will discuss the use of the consumption factor extensively in Section 2.2.4.

The usage attribute is specified only for renewable and doubly constrained resources, the consumption attribute for non-renewable and doubly constrained resources. These attributes represent plan-independent properties of a resource. Usage and consumption occur only during the time intervals in which processes are assigned to that resource, and the used and consumed amounts will be different for different processes. It is therefore not possible to model usage and consumption in terms of resource attributes. Furthermore, actual usage and consumption may occur only during part of the total processing time. For example, a production run may require a machine for the entire period, whereas it may require an employee to start the machine only for the first five minutes of the period. Similarly, the entire required amount of raw materials may be consumed at the beginning of the process, whereas fuel may be consumed at a constant rate during the processing period. The specification of these usage and consumption patterns will be discussed in Section 2.2.4.

Processes and their attributes

A process can be anything that must be assigned to a set of resources and a time interval in order to obtain a feasible plan. In a timetabling problem for a school, the processes may be the lessons that must be assigned to teachers, class-rooms, and time intervals. In a machine scheduling problem, the processes are the tasks that must be assigned to machines and time intervals.

There are two kinds of processes: *non-repetitive* and *repetitive* processes. A non-repetitive process is performed only once. A repetitive process can be performed an arbitrary number of times; the number of repetitions is determined by the planner by specification of the processing interval.

A process node has the following attributes:

name: A name that uniquely identifies the process.

type: An indicator that specifies whether the process is non-repetitive or repetitive.

mode: An indicator that specifies the mode in which the resource(s) to which the process is assigned have to be.

size: A real number that is used to determine the duration of the process in case of a non-repetitive process, or to determine the length of one repetition in case of a repetitive process.

usage intensity: A vector that is used to determine the usage of the resource(s) that the process is assigned to.

consumption intensity: A vector that is used to determine the consumption of the resource(s) that the process is assigned to.

release time: A point in time before which the process cannot be performed.

deadline: A point in time after which the process cannot be performed.

due date: A point in time by which the process preferably should be completed.

split: An indicator that specifies whether the process may be preempted or not. It has value 0 if the process, once started, cannot be interrupted, 1 if the process may be temporarily stopped and later resumed using the same resources, and 2 if the process can also be resumed using other resources.

2.2.4 Specifying feasible assignments

In this section, we discuss *requirement relations*. Requirement relations are relations between processes and resources indicating which resources can be used to perform a process. We use an incremental approach with respect to requirement relations. We start with simple requirements, i.e., processes requiring a single capability, and continue with more complicated requirements, i.e., processes requiring combinations of capabilities and processes requiring one of several combinations of capabilities. At the end of this section we will discuss how aspects related to feasible assignments such as duration, set-up times, usage, and consumption are modeled.

Processes requiring a single capability

A process that requires a specific capability can be assigned to any resource that possesses that capability. In fact, a choice has to be made between all resources that possess the required capability. Such a choice is modeled by a $K_{1,n}$, in which

the auxiliary node, the capability node, is connected to all nodes representing resources that possess the considered capability. For each capability that is required by some process, a capability node is introduced, and the associated $K_{1,n}$ is formed. The requirement relation is modeled by an edge connecting the process node and the capability node that represents the required capability.

A capability node has only one attribute:

name: The name uniquely identifies the capability. It must be the same as the name, given as a resource attribute, of the capability it represents.

An example is given in Figure 2.1. Processes P_1 and P_2 require capability C_1, which is possessed by resources R_1 and R_2. Process P_3 requires capability C_2, which is possessed by resources R_2 and R_3.

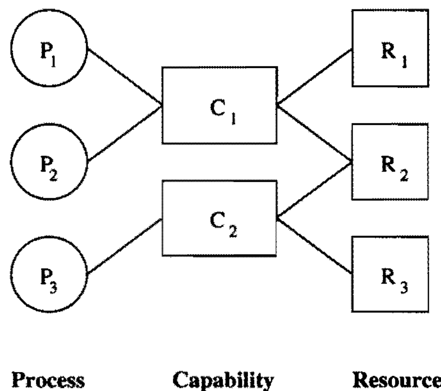

Figure 2.1: Processes requiring a single capability.

Processes requiring a combination of capabilities

In many problems, performing a process requires the use of a combination of resources, or, better, a combination of capabilities. Both paint and a painter are needed to paint an object. Lessons cannot be taught unless a teacher and a classroom are available. Such a combination of capabilities is again modeled by a $K_{1,n}$. The auxiliary node, the *capability set* node, is connected to all capability nodes representing the required capabilities. If a process requires a particular combination of capabilities, this is modeled by introducing an edge connecting the process node and the corresponding capability set node. The process can be performed by any combination of resources that possesses all capabilities in the capability set.

Note that, although any combination of capabilities may be considered as a capability set, only those combinations that are actually required by the processes are of interest.

A capability set node has several attributes. However, most of them will be introduced when we discuss duration, consumption, and usage in more detail. Here, we only introduce one attribute:

name: A name that uniquely identifies the capability set.

An example of requirement relations dealing with capability sets is given in Figure 2.2.

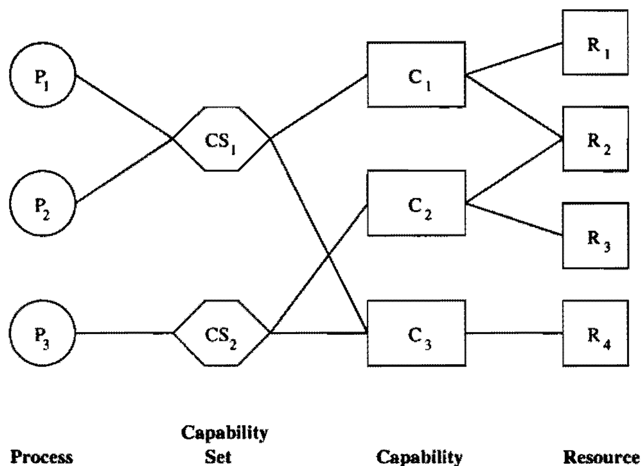

Figure 2.2: Processes requiring a combination of capabilities.

Processes P_1 and P_2 both require a resource possessing the capability C_1 *and* a resource possessing capability C_3. The resource combination $\{R_1, R_4\}$ would satisfy these requirements. Process P_3 requires a resource with the capability C_2 and a resource with the capability C_3. The combination $\{R_3, R_4\}$ is feasible. The combination $\{R_2, R_4\}$ is feasible for all three processes.

Processes requiring one of several combinations of capabilities

Sometimes, a process does not require a specific combination of capabilities, but one of several alternative capability sets. In such a case, we say that the process requires a particular *function*. This is modeled by a $K_{1,n}$, in which the auxiliary node, the function node, is connected to all capability set nodes between which

a choice can be made. If a process requires a function, a choice between the associated capability sets is to be made. The process then can be performed by any set of resources that possesses all capabilities in the chosen capability set.

Functions may be useful for production problems in which different production modes occur, which imply the possible use of totally different resources, e.g., production by hand or by machine.

A function node has only one attribute:

name: A name that uniquely identifies the function.

An example of requirement relations with functions is given in Figure 2.3. Processes P_1 and P_2 both require function F_1, implying that they must be performed by resources possessing the capabilities in capability set CS_1 *or* by resources possessing the capabilities in CS_2.

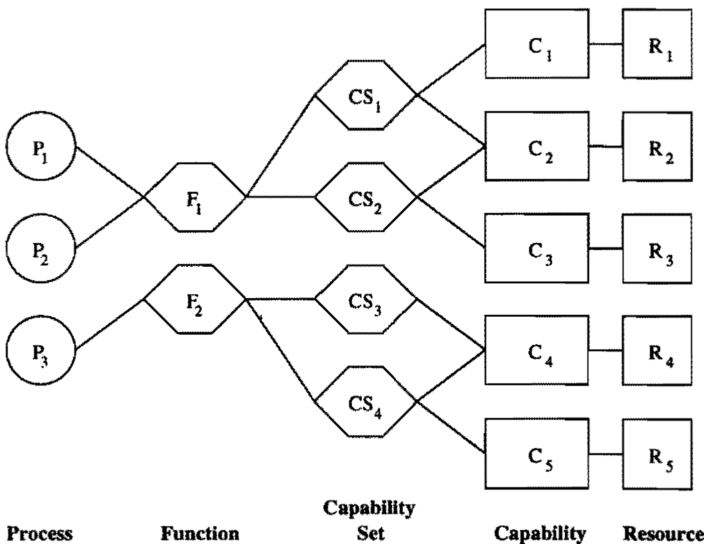

Figure 2.3: Processes requiring one of several combinations of capabilities.

In summary, three different types of $K_{1,n}$'s may occur in the subgraph representing the requirement relations. The auxiliary nodes are the capability nodes, the capability set nodes, and the function nodes, respectively. The first $K_{1,n}$ represents a *choice* relation between resources possessing the same capability. The second states that a particular *combination* of capabilities is required. The third again represents a possibility for *choice*, now between several capability sets.

Each process node is connected, via a *requirement edge*, to either a capability node, or a capability set node, or a function node.

Duration, set-up times, usage, consumption

The subgraph representing the requirement relations enables us to identify for each process the resources to which it can be assigned. Given such an assignment, we want to determine the following (assuming that the considered process is non-repetitive):

- The *set-up times* required to bring the resources in the correct modes.
- The *duration* of the process: the time to complete the process. Together with the starting time of the process, the duration determines the *processing interval*.
- The *usage pattern* of each resource that is used: the time interval within the processing interval during which the process occupies the resource and the amount of the resource that is occupied by the process during this time interval.
- The *consumption pattern* of each resource that is consumed: the time interval within the processing interval during which the process consumes the resource and the amount of the resource that is consumed by the process during this time interval. We will assume that consumption takes place at a constant rate. Instantaneous consumption can be modeled by specifying an infinitesimal time interval.

In case of repetitive processes, a set-up is only performed at the beginning of the first repetition, and the duration corresponds to the length of one repetition. We will assume that the usage and consumption patterns will be the same for every repetition. If, for example, at the beginning of a repetition a certain amount of raw materials is consumed, the same amount will be consumed at the beginning of all other repetitions.

The set-ups are much easier to deal with than duration, usage and consumption, because the set-up time of a particular resource does not depend on the other resources that are used. Given an assignment of a process to a combination of resources, for each of those resources the set-up time can be computed using the mode attribute of the process and the set-up attribute of the considered resource. The duration of the process, as well as the usage and the consumption of a specific resource, may depend on all the resources that occur in the assignment. For example, when a process requires a machine and fuel, the consumption of fuel will not only depend on the process but also on the machine that is actually used.

When a process requires a single capability, duration, consumption, and usage can easily be determined. The duration of the process is computed as the quotient of the size attribute of that process and the speed attribute of the resource it is assigned to. The interval during which any consumption or usage occurs is equal to the processing interval. The consumed amount is computed as the product of the consumption intensity attribute of the process and the consumption factor attribute of the resource, and the used amount is computed as the product of the usage intensity and the usage factor attributes.

The situation is more complicated when a process requires a combination of capabilities. In order to deal with these complications, we introduce several attributes for the capability set nodes. The first deals with the duration of the process.

duration: A function of the size attribute of the process and the speed attributes of the resources that computes the duration, i.e.,

$$d(s_0, s_1, \dots, s_n),$$

where s_0 is the value of the size attribute of the process, n is the number of capabilities in the capability set, and s_i $(i = 1, \dots, n)$ is the value of the speed attribute of the resource possessing capability i.

Suppose the capability set consists of the capabilities *machine, employee*, and *material*. Let s_0 be the size attribute of the process, and let s_1, s_2, and s_3 be the speed attributes of the machine, the employee, and the raw material, respectively. If the speed is completely determined by the speed of the machine, the following duration function is used:

$$d(s_0, s_1, s_2, s_3) = s_0/s_1.$$

If the ability of the employee to work with that machine plays a role, the duration function may be something like

$$d(s_0, s_1, s_2, s_3) = s_0/v_{s_1,s_2},$$

where $V = \{v_{ij}\}$ is some predefined matrix, v_{ij} being the speed at which a process is performed when the value of the speed attribute of the used machine is i and the value of the speed attribute of the used employee is j. Note that in this case the speed attributes do not reflect the actual speed of the resources, but are merely indices used to extract the correct values from a matrix.

For each capability in the capability set, the following two attributes are specified in order to describe the usage pattern.

usage interval: A function of the starting time and the duration of the process that computes the time interval during which usage of the resource possessing the considered capability takes place, i.e.,

$$[t_1, t_2] = t_u(start, duration).$$

usage volume: A function of the usage factor attribute of the process and the usage intensity attributes of the resources that computes the volume used, i.e.,

$$v_u(i^u, f_1^u, \ldots, f_n^u),$$

where i^u is the value of the usage intensity attribute of the process, n is the number of capabilities in the capability set, and f_i^u $(i = 1, \ldots, n)$ is the value of the usage factor attribute of the resource possessing capability i.

Let us again consider the example with the capability set consisting of the capabilities *machine, employee*, and *material.* Some machines require an employee during the first five minutes for starting it up, whereas others do not. This can be modeled by assigning a value of 1 to the usage factor attributes of the machines that do require an employee ($f_1^u = 1$), and a value of 0 to those that do not ($f_1^u = 0$), and by applying the following usage pattern:

$$t_u(start, duration) = [start, start + 5],$$

$$v_u(i^u, f_1^u, f_2^u, f_3^u) = f_1^u.$$

For each capability in the capability set, the following two attributes are specified in order to describe the consumption pattern.

consumption interval: A function of the starting time and the duration of the process that computes the time interval during which consumption of the resource possessing the considered capability takes place, i.e.,

$$[t_1, t_2] = t_c(start, duration),$$

where *start* is the starting time of the process, and *duration* is the value computed by the function given in the duration attribute described above.

consumption volume: A function of the consumption factor attribute of the process and the consumption intensity attributes of the resources that computes the volume consumed, i.e.,

$$v_c(i^c, f_1^c, \ldots, f_n^c),$$

where i^c is the value of the consumption intensity attribute of the process, n is the number of capabilities in the capability set, and f_i^c $(i = 1, \dots, n)$ is the value of the consumption factor attribute of the resource possessing capability i.

Consider again the capability set again consisting of the capabilities *machine*, *employee*, and *material*, and let the *material* capability be the only one that is possessed by non-renewable resources. If a process consumes a fixed amount of the material at a constant rate during the entire interval, the consumption pattern for the *material* capability is as follows:

$$t_c(start, duration) = [start, start + duration],$$

$$v_c(i^c, f_1^c, f_2^c, f_3^c) = i^c,$$

where the value of the consumption intensity attribute (i^c) equals the fixed consumed amount.

The attributes of the capability sets allow us to specify complex consumption and usage patterns for a large variety of problems. Obviously, assigning the correct values to the different process and resource attributes is important. Sometimes it may be necessary to assign to, for example, the speed attribute of a resource, a value that has little to do with the actual speed of that resource.

Example 2.1 We consider the problem of making coffee, more specifically the capability set *MakeCoffee* consisting of the capabilities *CoffeeMachine, Filter, GroundBeans, Water,* and *Coffee.*

There are two resources that possess the capability *CoffeeMachine.* Machine A can make one liter of coffee in 15 minutes, machine B does it in 10 minutes. Making one liter of coffee requires 0.10 units of *GroundBeans*, consumed at the beginning of the processing interval, and 1 liter of water, consumed gradually during the entire processing interval. Independently of the amount of coffee to be made, one filter is required, consumed at the start.

If we want to perform the process *0.8Coffee*, making 0.8 liter of coffee, we model this by assigning the value 0.8 to the consumption intensity attribute as well as to the size attribute of the process. Furthermore, we assign the values $4/60$ and $6/60$ to the speed attributes of resource A and B respectively, and use one minute as time unit.

The attributes of the capability set *MakeCoffee* then are:

duration: *size / CoffeeMachine.speed*

(CoffeeMachine)

usage interval: $[start, start + duration]$

usage volume: 1

(Filter)

consumption interval: $[start, start]$

consumption volume: 1

(GroundBeans)

consumption interval: $[start, start]$

consumption volume: *consumption intensity* $*0.10$

(Water)

consumption interval: $[start, start + duration]$

consumption volume: *consumption intensity*

(Coffee)

consumption interval: $[start, start + duration]$

consumption volume: - *consumption intensity*

□

2.2.5 Other relations between objects

Specifying the requirement structure, i.e., specifying for each process the combinations of resources it can be assigned to, is only part of the specification of a problem instance. Usually various other relations exist between processes and resources. Many of these can easily be specified in the instance graph.

Precedence relations

A precedence relation indicates that the set of allowed start and completion times of some process depends on the start or completion time of some other process. Precedence relations are binary relations that can be represented by arcs between process nodes in the instance graph. We distinguish four types:

- A *finish-to-start* relation indicates that process A must be completed before some process B is started.
- A *start-to-start* relation indicates that process A must be started before process B is started.
- A *start-to-finish* relation indicates that process A must be started before process B is completed.
- A *finish-to-finish* relation indicates that process A must be completed before process B is completed.

A *time-lag* may be associated with each of the four types of precedence relations. For example, a finish-to-start relation can be stated as 'process A must be completed at least 10 minutes before process B starts.'

The attributes of precedence arcs are:

type: Either finish-to-start, or start-to-start, or start-to-finish, or finish-to-finish.

time lag: The minimum and maximum allowed time lag.

Common resource relations

A set of processes may be related because they have to be processed on the same set of resources. Such an n-ary relation is specified as a $K_{1,n}$; the auxiliary node is the common resource set node.

The only attribute of a common resource set node is:

name: A name that uniquely identifies the common resource set.

In case of a common resource relation between several processes, there is no need for individual requirement edges. Instead the requirement edge will connect the common resource set node to a capability node, a capability set node, or a function node. In this light, the common resource relation can be seen as an *obligation* relation, in contrast to the *choices* represented by functions and capabilities. As soon as one of the processes in a common resource set is assigned to a particular resource combination, all other processes in that set have to be assigned to the same resource combination.

Process group relations

A set of processes may be related for some other reason than common resources. Such an n-ary relation is represented by a $K_{1,n}$; the auxiliary node is the process group node. For instance, the notion of a job in machine scheduling problems can be modeled as a process group.

The attributes of the process group node are:

name: A name that uniquely identifies the process group.

release time: No process in the process group can be performed before its release time.

deadline: No process in the process group may be completed after its deadline.

due date: All processes should preferably be completed by their due date.

exclusion: If this attribute has the value *true*, then no two processes in the process group may be performed at the same time.

Note that release time, deadline, and due date can be specified for all processes in a process group separately, but even when the exclusion attribute has the value *false*, the introduction of a process group can be useful for emphasizing characteristic structures in problem instances.

Resource group relations
Similarly to process groups, resource groups can be used to emphasize certain relations between resources. The introduction of resource groups imposes no additional restrictions on the set of feasible plans.

The attribute of the resource group node is:

name: A name that uniquely identifies the resource group.

2.2.6 Extensions

In the previous sections, we have shown that it is possible to specify many different aspects of the problem instances that we are interested in by using attributed graphs. The combined use of graph elements and attributes has enabled us to deal with such diverse problem characteristics as precedence constraints, resources possessing the same capabilities, and complex consumption patterns. Our general problem class, i.e., the class of problem types of which instances can be specified with our method, includes a large variety of problems that are of theoretical or practical interest. However, there still exist many problem instances that currently cannot be specified. This is not a consequence of the lack of expressive power of attributed graphs, but it is a consequence of the selection criteria that we have applied with respect to the problem characteristics that we wanted to be able to specify. These selection criteria have been chosen somewhat arbitrarily. The main goal of this chapter is to show that it is possible to develop a method that allows us to specify a fairly large class of interesting problem types. Both the notions 'fairly large' and 'interesting' are subjective, but we think that they do apply to the general problem class that is induced by our specification method.

2.2.7 Views of the instance graph

In the sections above, we have discussed the various components that can be used to specify a problem instance in terms of an instance graph. The specification of the nodes, arcs, edges, and $K_{1,n}$'s with their associated attributes results in an instance graph that represents all information about the problem instance under consideration. During the planning process, a planner may want to view parts of this information. If he wants to make an assignment for a particular process, he may be interested in the set of resources to which the considered process can be assigned, or he may want to know which other processes have a precedence relation with the considered process. This kind of information can easily be obtained because it corresponds to relatively small subgraphs of the instance graph. Such a subgraph will be called a *view* of the instance graph. Many views can be defined. A few examples follow below.

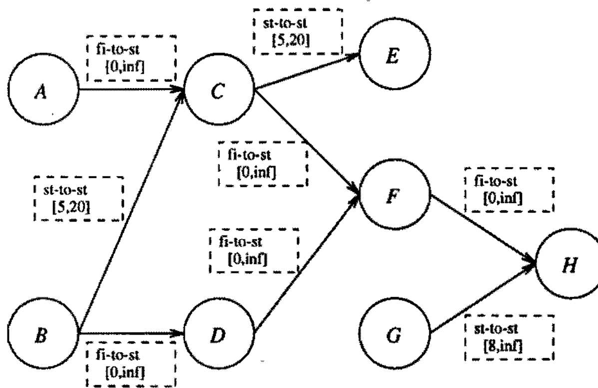

Figure 2.4: A precedence view.

Assignment view: This view presents for a set of processes the resources to which it can be assigned, i.e., all the paths originating from one of the processes and ending at a resource. Examples of this view have already been given in Figures 2.1 to 2.3.

Process view: This view presents for a set of processes the process groups and common resource sets to which they belong.

Resource view: This view presents for a set of resources the resource groups and capabilities to which they belong.

Precedence view: This view presents for a set of processes the precedence relations between them. An example is given in Figure 2.4. The arc from node B to node C implies that process B must be started at least 5 time units and at most 20 time units before process C is started.

A view is a convenient mechanism to present parts of the information embedded in the instance graph. Note that it is possible that a view contains some redundancy. For example, when a particular capability is possessed by only one resource, the corresponding capability node can be considered as redundant. By connecting the process, common resource set, function, and capability set nodes that are connected to that capability node, directly to the corresponding resource node, a view can be *reduced* to a smaller, possibly more insightful one.

2.2.8 Examples of problem instances

In this section we describe the instance graphs, or parts of it, of three problem instances. The first example shows what kinds of connections between the different types of nodes are possible. In the second example, we consider a timetabling problem for a school. The third example deals with a factory scheduling problem with a non-trivial consumption pattern.

Example 2.2 Consider the following production planning problem. Four products of two different types are to be made. Products *1* and *2* are products of the first type, products *3* and *4* are of the second type. Processes *P1, P2, P3,* and *P4* represent the production of products *1, 2, 3,* and *4*. There is a fifth process, *M*, which represents some maintenance activities. Performing *P1* can be done in two modes, the normal mode or the special mode, *P2* requires the special mode. In both modes an employee and a machine of the type *MT1* are used, but some machines of that type can only be used in the special mode and others only in the normal mode. The processes *P3* and *P4* require a machine of the type *MT2* and an employee. The combination that is used for *P3* must also be used for *P4*. The maintenance process is to be performed by one of the employees. The instance graph for this problem is given in Figure 2.5.

This example illustrates that all different kinds of resource requirements discussed in Section 2.2.4 can occur in the same problem instance. Process *P1* requires a function, process *P2* requires a capability set, and process *M* requires a capability. Furthermore, since processes *P3* and *P4* must be performed on the same combination of resources, there is an associated requirement edge connecting the corresponding common resource set node with, in this case, a capability set node. □

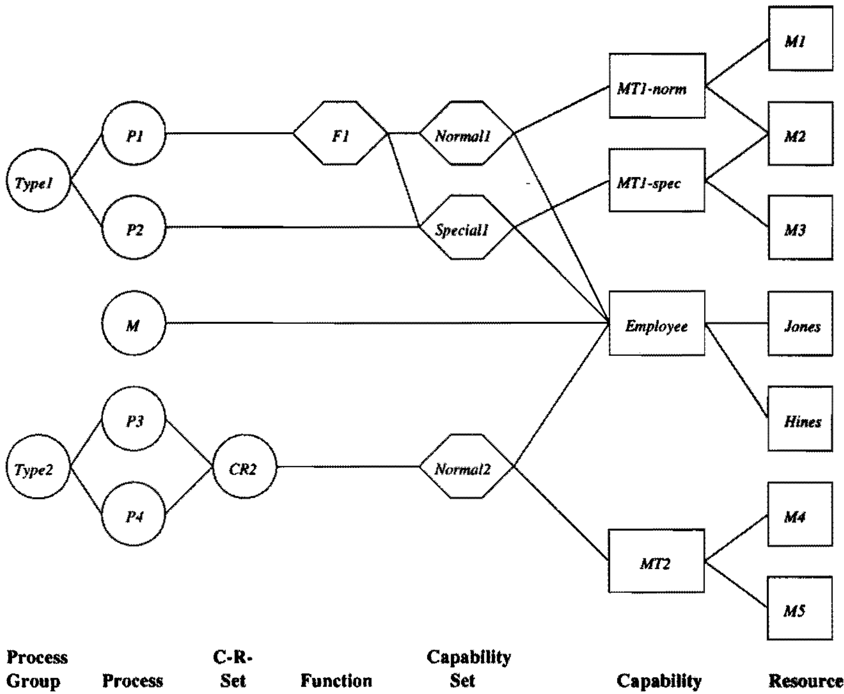

Figure 2.5: A production planning problem instance.

Example 2.3 We consider the following timetabling problem. Each week, each group of pupils has to get lessons in different subjects during a specified number of hours. Some lessons require teachers with full qualification, others require teachers with partial qualification. Most lessons can be given in normal classrooms, but for some subjects the lessons require special rooms. Chemistry lessons, for example, must be given in rooms in which experiments with chemicals can be performed.

We can describe this situation in the following way. The processes are the lessons given to the different groups. They are non-repetitive. An example is the chemistry lesson for group 3. There are two kinds of resources: teachers and classrooms. A teacher can have several capabilities, representing the subjects he is qualified to teach and the levels of qualification. Since a teacher may be fully qualified for one subject and partially qualified for the other, it is necessary to introduce capability nodes for each possible combination of subject and qualification. A classroom also may have several capabilities. A particular classroom

may be suited for chemistry lessons, as well as normal (for example, English) lessons.

Each lesson requires a particular kind of classroom and a particular kind of teacher. These requirements are represented by requirement edges connecting the processes and capability sets. Each capability set consists of two capabilities, one representing the required kind of classroom, the other representing the required kind of teacher. For example, the lesson English for group 4 requires the capability set *E-F*, representing a normal classroom and a teacher with full qualification for the subject English.

A part of an instance is depicted in Figure 2.6. Only the lessons English and chemistry for groups 3 and 4 and not all teachers and classrooms are considered.

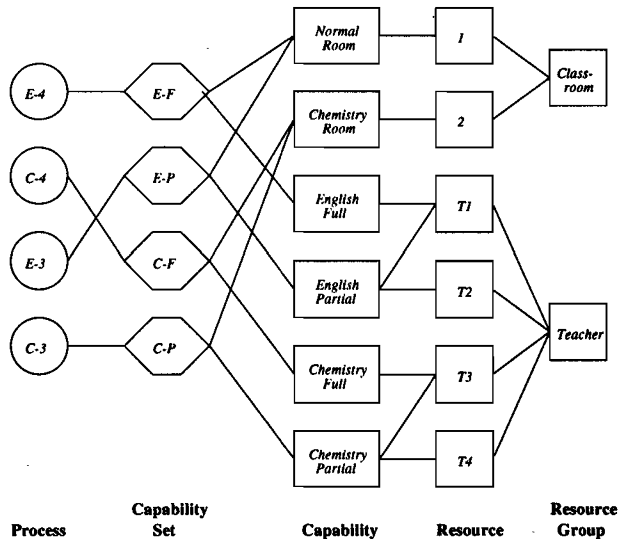

Figure 2.6: Part of a timetabling instance.

The planning period will consist of 5 days of say 8 hours. The resources are renewable resources with capacity 1. A process uses both kinds of resources fully during the entire processing interval, which is always one hour. □

Example 2.4 Jones and Maxwell [1986] give an example of a factory scheduling problem. There are three processes, *1MSUB, 2MSUB*, and *ASSEM*. Process *1MSUB* uses a machine of the type *A-MILL* and materials *MAT1* and *MAT2* to create half-products *SUB1*. Process *2MSUB* uses the machine *MILL* and the half-product *SUB3* to create another half-product, *SUB2*. The process *ASSEM* is per-

formed by a worker, who assembles half-products *SUB1* and *SUB2* to create the end-product *WIDGET*. The reduced assignment view of this instance is given in Figure 2.7.

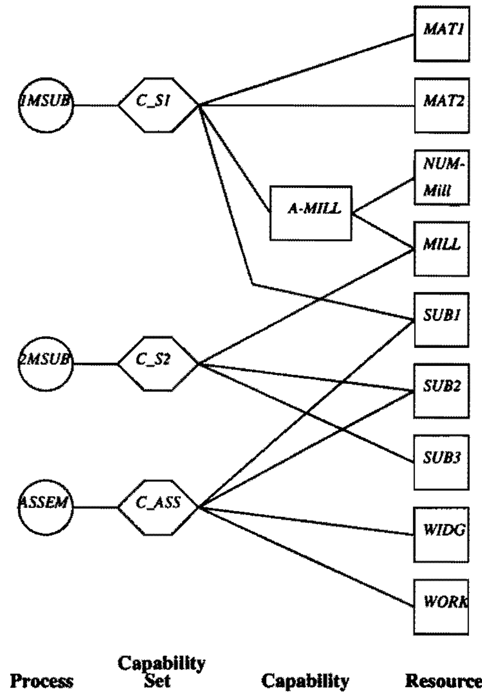

Figure 2.7: A reduced assignment view of a factory scheduling instance.

All processes are repetitive. The machines are renewable resources; the materials, half-products, and end-products are non-renewable resources. The machines have capacity 1 and are always used up to capacity. The materials are consumed at a constant rate during the processing period, and the half products and end products are consumed (produced) at the beginning (end) of each repetition.

We consider the capability set *CS1*. The duration solely depends on the *MILL* that is used, and the consumed volumes are completely determined by the consumption intensity (i^c) of the process and the consumption factor of the considered resource ($f^c(.)$). The attributes of capability set *CS1* then are the following:

name: *CS1*

duration: = *size / A-MILL.speed*

(MAT1)

consumption interval: $[start, start + duration]$

consumption volume: *consumption intensity / MAT1.consumptionfactor*

(MAT2)

consumption interval: $[start, start + duration]$

consumption volume: *consumption intensity / MAT2.consumptionfactor*

(A-MILL)

usage interval: $[start, start + duration]$

usage volume: 1

(SUB1)

consumption interval: $[start + duration, start + duration]$

consumption volume: - *consumption intensity / SUB1.consumptionfactor*

□

2.3 Problem types

In general, planning boards are not developed as a tool for solving one particular problem instance, but as a tool for solving a set of problem instances with similar characteristics. We use the term *problem type* to refer to a set of problem instances satisfying certain conditions. The general problem class, i.e., the set of all problem instances that can be specified with the aid of the instance specification method, is a problem type itself.

A PBG requires information about the characteristics of the problem type for which a planning board has to be generated and about the desired representations and manipulations. Information about the desired manipulations could be something of the following sort: the planning board must support the reassignment of a process. If at a given moment process A is assigned to resource R_1 with start time t_1, the user should be able to reassign process A to resource R_2 with start time t_2. It should be obvious that the implementation of such a manipulation will depend on the problem type. For a problem type in which all resources are available during the whole planning period, checking the feasibility of a reassignment is much easier than for a problem type in which the resources are only available during certain time intervals.

In fact, the main purpose of the specification of a problem type is to provide the necessary information to implement the desired manipulations and representations as efficiently and effectively as possible. Consequently, the specification of a problem type should provide information on those common characteristics of the various problem instances that are important for the development of manipulations and representations. Therefore, care should be taken to ensure that the specification of the problem type is accurate. It should contain all problem instances for which the planning board is to be used, but as few others as possible because they may lead to a planning board that is less efficient and effective than possible.

Problem types will be specified by formulating restrictions on the instance graph. A problem type is then the set of all instances that satisfy the given restrictions.

2.3.1 Restrictions on the instance graph

The characteristics of a problem instance are defined by the structure of the associated instance graph and the values of the attributes of the various graph elements. We can therefore distinguish two types of restrictions: on the graph structure and on the attribute values.

Restrictions on the graph structure

Characteristics of a problem type that can be described in terms of restrictions on the graph structure relate to the presence or absence of the various graph elements and to the way these graph elements are interconnected.

Consider the well-known job shop scheduling problem. The restrictions that have to be imposed on the graph structure are the following:

- The only node types that appear are process, resource, capability, and process group.
- Each capability node is connected to exactly one resource node.
- Each requirement edge connects a process to a capability.
- Precedence relations occur in chains and only involve processes in the same process group.

If the problem type becomes more complex, it is not always possible to specify restrictions that are valid for all graph elements of the same type. For example, consider the extension of the job shop scheduling problem in which each task requires both a machine and some raw material. In an instance graph, this extension would be modeled with a capability set node that specifies that a machine

and some raw material are required for a task. However, in a type specification it is insufficient to specify that capability set nodes exist and that each capability set node is connected to exactly two capability nodes, because also instances in which a task is assigned to two machines would satisfy this restriction.

In order to deal with such problems, we need to be able to distinguish nodes of the same type. The notion of a *node group* is introduced precisely for that reason. A node group is a group of nodes of the same type that are subject to the same restrictions. In the above extension of the job shop scheduling problem, we would introduce two capability node groups, *Machine* and *Inventory*, and impose the following restrictions:

- The only node types that appear are process, resource, capability, capability set, and process group.
- Each capability node is connected to exactly one resource node.
- Each capability set node is connected to exactly one capability node in the node group *Machine* and to exactly one capability node in the node group *Inventory*.
- Each requirement edge connects a process to a capability set.
- Precedence relations occur in chains and only involve processes in the same process group.

Besides restrictions on the presence or absence of the various graph elements and on the the way they are interconnected, it is often necessary to also impose restrictions on the number of them. For example, if the raw material in our extension of the job shop scheduling problem is the same for all tasks, one should be able to specify that the number of nodes in the node group *Inventory* is precisely one.

Restrictions on the attributes

Characteristics of a problem type that can be specified in terms of restrictions on the attributes of the graph elements relate to their domains. The domain of an attribute is its set of admissible values. In a problem type specification it is possible to reduce a domain by specifying restrictions on the set of admissible values. Attribute restrictions are specified for node groups.

2.3.2 Specifying problem types

In this subsection, we present a syntax which can be used to specify the restrictions discussed in Section 2.3.1 more formally. The syntax can be modified or

extended in order to deal with other kinds of restrictions. In Section 2.3.3 several problem types are specified using this syntax.

Node groups

The first part of the specification of the problem type concerns the node groups that appear in an instance. For each node type the associated node groups and a specification of the number of nodes in these node groups have to be given. For example,

process:	Task1	$\{1,\ldots,\infty\}$
	Task2	$\{1,\ldots,\infty\}$
resource:	Machine	$\{1,\ldots,4\}$
capability:	Type	$\{1\}$

indicates that there are three node types and four node groups. The node groups Task1 and Task2 may have an arbitrary number of nodes, the node group Machine may have one up to four nodes, and the node group Type has precisely one node.

Instances with five or more resources or more than one capability do not belong to this problem type. Also instances with common resource set nodes, function nodes, capability set nodes, process group nodes, or resource group nodes do not belong to this problem type.

In the example above, the number of nodes in a node group is restricted to be in a specified set, e.g., $\{1\}$, $\{1,\ldots,4\}$, and $\{1,\ldots,\infty\}$. In addition to specifying a set, it is also possible to relate the cardinality of a node group to the number of nodes in another node group. For example,

resource:	Machine	$\{2,\ldots,100\}$
capability:	Type	#(Machine)

would indicate that the number of nodes in the node group Type is equal to the number of nodes in the group Machine.

Graph structure

The second part of the specification of a problem type concerns the structure of the instance graphs. It deals with n-ary relations, requirement edges, and precedence arcs.

For each of the $K_{1,n}$'s associated with the node groups, the node groups to which they can be connected and a specification of the cardinality of these node groups have to be given. For example,

capability:	Type	Machine	#(Machine)
capability set:	Transform	Type	{1}
		InputInv	{1}
		OutputInv	{1}

indicates that each node in the node group Type is connected to all nodes in the node group Machine, and that each node in the node group Transform is connected to one node in the node group Type, one in the node group InputInv, and one node in the node group OutputInv.

Requirement edges connect process nodes and common resource set nodes to function nodes, capability set nodes, and capability nodes. Restrictions on the possible positions of requirement edges are specified for all relevant node groups. For example,

requirement edges:	Task	Type
	CRSet1	Mode1
	CRSet2	Mode2

indicates that (process) nodes in the node group Task are connected by a requirement edge to (capability) nodes in the node group Type, and that (common resource set) nodes in the node groups CRSet1 and CRSet2 are connected by a requirement edge to (function) nodes in the groups Mode1 and Mode2, respectively.

Precedence arcs usually occur between processes in the same node group or in the same process group. Furthermore, the graphs describing precedence relations often have a special structure, such as chains or trees. The structure of the precedence graph is specified for all relevant node groups. For example, suppose Group1,. . .,Group4 are process node groups, and Job is a node group of process groups. Then,

precedence arcs:	Group1	intree
	Group2	outtree
	Group3 ∪ Group4	general
	Job	chain

indicates, that the subgraph induced by precedence arcs of (process) nodes in the node group Group1 is an intree, that the subgraph induced by precedence arcs of (process) nodes in the node group Group2 is an outtree, and that the subgraph induced by precedence arcs of (process) nodes in the node groups Group3 and Group4 does not have a special structure. Furthermore, for each process group in the node group Job the following must hold: the subgraphs induced by precedence arcs of nodes representing processes in that process group are chains,

Attributes

The third part of the specification of a problem type concerns the attributes of the different objects. For each node group as well as for the precedence arcs, a domain is specified for each attribute. For example,

process:	Task	type	non-repetitive
		size	{1,2}
resource:	Machine	category	non-renewable
precedence:	Group1	type	{finish-to-start,start-to-start}
		time lag	[0, 50]

indicates that processes in the node group Task are non-repetitive and have size 1 or 2, that resources in the node group Machine are non-renewable, and that the precedence relations between nodes in the node group Group1 are either finish-to-start or start-to-start with a minimum time lag of 0 and a maximum time lag of 50 time units.

It is also possible to enforce two attributes to take on the same values. For example,

process:	Task	size	$\{1,\ldots,\infty\}$
		consumption intensity	size

indicates that the size attribute of processes in the node group Task can take on any positive integer value, and that the value of the consumption intensity attribute is equal to the value of the size attribute.

The attributes of capability sets are functions of attributes of other objects. In the problem type specification, either these functions are completely specified, or restrictions on their shape are imposed. In referring to attributes of other objects we apply a two-field notation: *object.attribute*, where *object* is a node group. When misinterpretation is impossible, it suffices to only state *attribute*. For example,

capability set

ResourceSet	duration	size/MacCap.speed
	(MacCap)	
	usage interval	$[start, start + duration]$
	usage volume	1
	(InvCap)	
	consumption interval	$[start, start]$
	consumption volume	$a * consumptionintensity$
		$a \in (0, \infty)$

indicates that for a capability set in the node group ResourceSet the duration function is defined as the quotient of the size of the process and the speed of the

resource possessing a capability in the node group MacCap. Furthermore, that resource is used during the entire processing interval, the resource possessing a capability in the node group InvCap is consumed at the beginning of the processing interval, and the function that is used for computing the consumed volume is a linear function of the consumption intensity attribute of the process.

When certain attributes of objects are irrelevant for the problem type we want to specify, for example the due date attribute in case no due dates occur, this can be indicated by 'does not apply'. If no domain is specified for an attribute of an object, we assume that the default domain specifications apply. The default domain specifications for the different attributes are the following:

ATTRIBUTES		
process:	type	non-repetitive
	mode	does not apply
	size	$(0,\infty)$
	usage intensity	1
	consumption intensity	does not apply
	release time	does not apply
	deadline	does not apply
	due date	does not apply
	split	0
resource:	availability periods	does not apply
	category	renewable
	number of modes	0
	usage	
	divisibility	1
	dimension	1
	capacity	$c(t) = 1$
	consumption	
	divisibility	1
	dimension	1
	supply	$s(t) \in \{1, \ldots, \infty\}, t = 0$ $s(t) = 0, t \neq 0$
	number of capabilities	*no default*
	speed	1
	usage factor	1
	consumption factor	does not apply

capability set:	duration	*no default*
	usage interval	*no default*
	usage volume	*no default*
	consumption interval	*no default*
	consumption volume	*no default*
process group:	release time	does not apply
	deadline	does not apply
	due date	does not apply
precedences:	type	finish-to-start
	time lag	$[0,\infty]$

Time occurs in the problem type specification in a similar way as attributes. In the default time system, there is only one level and there is no restriction on the length of the planning period:

TIME

number of levels	1
planning period	$\{1,\ldots,\infty\}$

2.3.3 Examples of problem types

Example 2.5 The simple job shop scheduling problem type discussed in Section 2.3.1, in which each task requires one specific machine, can be specified as follows:

NODE GROUPS

process:	Task	$\{1,\ldots,\infty\}$
resource:	Machine	$\{1,\ldots,\infty\}$
capability:	MacCap	#(Machine)
process group:	Job	$\{1,\ldots,\infty\}$

GRAPH STRUCTURE

capability:	MacCap	Machine	$\{1\}$
process group:	Job	Task	$\{1\}$
requirement edge:	Task	MacCap	
precedence arcs:	Job	chain	

ATTRIBUTES

process:	Task	type	non-repetitive
resource:	Machine	category	renewable
		number of capabilities	1
		(MacCap, 1)	

Note that although non-repetitive and renewable are the default values for the

type and category attributes, we have explicitly mentioned them in the problem type specification for clarity. □

Example 2.6 In Section 2.2.8, an example of a timetabling problem has been discussed. Lessons must be assigned to teachers and classrooms, and the feasibility of an assignment depends on the qualification of the teacher (what subject, full or partial qualification) and the properties of the classroom (chemistry lessons require specific facilities for performing experiments). The example can be seen as an instance of the problem type in which each lesson (process) requires a combination (capability set) of a type of classroom and a specific qualification (both capabilities).

In the terminology of our specification method, these restrictions can be formulated in the following way. We only give the node groups and the graph structure restrictions that are related to the processes, resources, capabilities, and capability sets.

NODE GROUPS

process:	Lesson	$\{1,\ldots,\infty\}$
resource:	Teacher	$\{1,\ldots,\infty\}$
	Classroom	$\{1,\ldots,\infty\}$
capability:	Qualification	$\{1,\ldots,\infty\}$
	ClassroomType	$\{1,\ldots,\infty\}$
capability set:	Room&Qual	$\{1,\ldots,\infty\}$

GRAPH STRUCTURE

capability:	Qualification	Teacher	$\{1,\ldots,\infty\}$
	ClassroomType	Classroom	$\{1,\ldots,\infty\}$
capability set:	Room&Qual	ClassroomType	$\{1\}$
		Qualification	$\{1\}$
requirement edges:	Lesson	Room&Qual	

□

Example 2.7 Anthonisse, Van Hee, and Lenstra [1988] describe the resource-constrained project scheduling (RCPS) problem as follows.

> A set of tasks is to be processed by a set of resources. For each task, there is a release time and a deadline, which define a time interval in which the task must be processed. Once a task is started it must be completed without interruption.
>
> For any two tasks, there are a lower bound and an upper bound on the length of the time period between the completion of one task and the start of the other. (...)

> A function may be performed by various combinations of resources, each with its own speed. In general, each function has a class of feasible resource sets, and the processing time of a task depends on the feasible resource set that is chosen to perform the function it requires. The processing time is the amount of work (i.e., the number of units of the function) required by the task divided by the speed of the feasible resource set.
>
> No resource can be allocated to two tasks at the same time. If a set of resources is allocated to a task, then each of its constituent resources is occupied by that task from its starting time until its completion time. For each resource there is a set of time intervals during which the resource is available.

We will now specify this problem type with our specification method.

The time system is the default system. There is only one time unit, and the planning period can be as small or as large as one wants.

Four kinds of objects are distinguished in the description above: tasks, resources, resource sets, and functions. In our terminology, they are processes, resources, capability sets, and functions, respectively. In fact, a fifth kind of object, the *project*, is considered, which is equal to the process group. Beside these five objects, our method requires capability nodes. Each resource has a unique capability. Therefore, the number of capability nodes is equal to the number of resource nodes. There are no restrictions on the number of nodes of any of the other types.

NODE GROUPS

process:	Task	$\{1,\ldots,\infty\}$
resource:	Resource	$\{1,\ldots,\infty\}$
capability:	ResCap	#(Resource)
capability set:	ResourceSet	$\{1,\ldots,\infty\}$
function:	Function	$\{1,\ldots,\infty\}$
process group:	Project	$\{1,\ldots,\infty\}$

Each resource possesses a unique capability. Therefore, each capability node (belonging to the node group ResCap) is connected to one resource node. The capability $K_{1,n}$'s are in fact $K_{1,1}$'s. The ResourceSets may consist of any number of resources, or better, of any number of capabilities. Each function node may be connected to any number of ResourceSet nodes (possibly one), and each process requires a function. For each Project, the number of tasks in it is completely instance-dependent. Precedence relations are only possible between tasks in the same Project.

GRAPH STRUCTURE

capability:	ResCap	Resource	$\{1\}$
capability set:	ResourceSet	ResCap	$\{1,\ldots,\infty\}$
function:	Function	ResourceSet	$\{1,\ldots,\infty\}$
process group:	Project	Task	$\{1,\ldots,\infty\}$
requirement edges:	Task	Function	
precedence arcs:	Project	general	

All processes are non-repetitive. The usage intensity attribute does not apply, because it is not required in order to specify the attributes of a capability set related to the usage. The consumption intensity attribute does not apply because all resources are renewable. The size attribute of processes, and the release time, deadline, and due date attributes of Projects and processes may take on any value. The tasks cannot be preempted.

Only renewable resources occur. A resource cannot be used for different tasks at the same time. Hence, the capacity is constantly 1, and the usage discretization unit is also 1. The speed is not determined by individual resources, but only by the resource set as a whole. Therefore, the speed attribute does not apply. Also, the usage factor and the consumption factor do not apply.

Since no consumption occurs, the consumption attributes do not apply. The duration is determined as the quotient of the size attribute of the process and a number representing the speed of a resource combination that is different for each capability set. Each resource that a process is assigned to is used during the entire processing interval. The used volume is equal to one for each resource.

The precedence relations are always of the finish-to-start type. There are no restrictions on the corresponding time lags.

ATTRIBUTES

process:	Task	type	non-repetitive
		usage intensity	does not apply
		consumption intensity	does not apply
		release time	$[0,\infty)$
		deadline	$[0,\infty)$
		due date	$[0,\infty)$
resource:	Resource	category	renewable
		number of capabilities	1
		(ResCap, 1)	
		speed	does not apply
		usage factor	does not apply
		consumption factor	does not apply

capability set:	ResourceSet	duration	$a * size$
			$a \in (0, \infty)$
		(ResCap)	
		usage interval	$[start, start + duration]$
		usage volume	1
process group:	Project	release time	$[0,\infty)$
		deadline	$[0,\infty)$
		due date	$[0,\infty)$
precedences:	Project	time lag	$[[0,\infty),[0,\infty)]$

Anthonisse, Van Hee, and Lenstra [1988] give an example of an RCPS instance. In Figure 2.8, the corresponding instance graph is given.

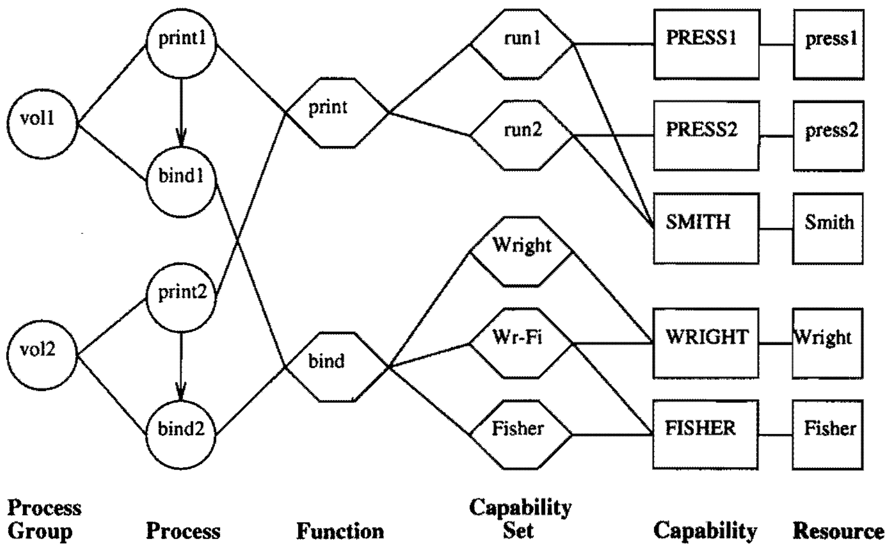

Figure 2.8: The RCPS instance.

□

Example 2.8 Jones and Maxwell [1986] discuss factory scheduling problems. In these problems three kinds of objects occur: processes, inventories, and machines. Processes are performed on one machine and consume and produce the contents of inventories.

In their article, Jones and Maxwell introduce a specification method that is based on networks. We can specify this problem type with our method as well.

We introduce three resource node groups. One node group consists of nodes that represent machines, the other two represent continuous inventories and discrete inventories, respectively. The reason for introducing two inventory node groups is that different restrictions on the attribute values can be specified for continuous and discrete inventories.

Since different consumption patterns apply when an inventory serves as input or occurs as output of a process, for each of both types of inventory two capability node groups are introduced. There are no functions, common resource sets, process groups, and resource groups.

NODE GROUPS

process:	Process	$\{1,\ldots,\infty\}$
resource:	Machine	$\{1,\ldots,\infty\}$
	ContInventory	$\{1,\ldots,\infty\}$
	DiscInventory	$\{1,\ldots,\infty\}$
capability:	MachineType	$\{1,\ldots,\infty\}$
	ContInInv	$\{1,\ldots,\infty\}$
	ContOutInv	$\{1,\ldots,\infty\}$
	DiscInInv	$\{1,\ldots,\infty\}$
	DiscOutInv	$\{1,\ldots,\infty\}$
capability set:	ResourceSet	#(Process)

There may be several machines of a particular type. For each process, the actually required inventories are completely specified. Hence, the corresponding $K_{1,n}$'s that connect capabilities and resources are simple edges. Each ResourceSet consists of one machine type and an arbitrary number of inventories, which can be discrete or continuous, and can serve as input or occur as output. There are no precedence relations.

GRAPH STRUCTURE

capability:	MachineType	Machine	$\{1,\ldots,\infty\}$
	ContInInv	ContInventory	$\{1\}$
	ContOutInv	ContInventory	$\{1\}$
	DiscInInv	DiscInventory	$\{1\}$
	DiscOutInv	DiscInventory	$\{1\}$
capability set:	ResourceSet	MachineType	$\{1\}$
		ContInInv	$\{0,\ldots,\infty\}$
		ContOutInv	$\{0,\ldots,\infty\}$
		DiscInInv	$\{0,\ldots,\infty\}$
		DiscOutInv	$\{0,\ldots,\infty\}$
requirement edges:	Process	ResourceSet	

The time system is again simple. There is only one time unit, and the planning period can be as small or as large as one wants.

All processes are repetitive. The size attribute can take on any value. Jones and Maxwell introduce a process attribute (Time/Lot) that deals with both the duration of one repetition and the consumption. In our approach, we enforce the size attribute and the consumption intensity to take on the same value. The usage intensity does not apply. We assume that a process may not be interrupted during a repetition, hence the split attributes have value 0. There are no release times, deadlines, or due dates.

Machines are renewable resources, with capacity equal to 1. They may have several capabilities, all representing a particular MachineType. Since the duration of a process is uniquely determined by the size of that process, the speed attribute for machines does not apply. Similarly, usage factor and consumption factor do not apply.

The inventories are non-renewable resources. The resources of the ContInventory type have continuous divisibility, the resources of the DiscInventory type have discrete divisibility. Again, the speed, usage factor, and consumption factor attributes do not apply. The supply attribute can take on any value.

Consumption patterns for the different inventories depend on whether they are discrete or continuous and on whether they are used as input or as output. The consumed volumes are determined by dividing a number that is different for each ResourceSet by the consumption intensity of the process. This number is positive if the inventory is used as input, and negative if the inventory is used as output. Consumption of the continuous inventories occurs during the entire processing period; consumption of discrete inventories occurs at the beginning (when used as input) or at the end (when used as output) of each repetition.

process:
    Process
        type repetitive
        usage intensity does not apply
        consumption intensity size
resource:
    Machine
        category renewable
        number of capabilities $\{1, \ldots, \infty\}$
        (MachineType, $\{1, \ldots, \infty\}$*)*
        speed does not apply
        usage factor does not apply
        consumption factor does not apply

ContInventory	
category	non-renewable
consumption	
divisibility	0
supply	$s(t) \in [0, \infty)$
number of capabilities	2
(ContInInv, 1)	
speed	does not apply
usage factor	does not apply
consumption factor	does not apply
(ContOutInv, 1)	
speed	does not apply
usage factor	does not apply
consumption factor	does not apply
DiscInventory	
category	non-renewable
consumption	
supply	$s(t) \in \{0, \ldots, \infty\}$
number of capabilities	2
(DiscInInv, 1)	
speed	does not apply
usage factor	does not apply
consumption factor	does not apply
(DiscOutInv, 1)	
speed	does not apply
usage factor	does not apply
consumption factor	does not apply
capability set:	
ResourceSet	
duration	size
(Machine)	
usage interval	$[start, start + duration]$
usage volume	1
(ContInInv)	
consumption interval	$[start, start + duration]$
consumption volume	$a / consumption\ intensity$ $a \in (0, \infty)$

(ContOutInv)

consumption interval	$[start, start + duration]$
consumption volume	$a/consumption\ intensity$
	$a \in (-\infty, 0)$

(DiscInInv)

consumption interval	$[start, start]$
consumption volume	$a/consumption\ intensity$
	$a \in (0, \infty)$

(ContOutInv)

consumption interval	$[start + duration, start + duration]$
consumption volume	$a/consumption\ intensity$
	$a \in (-\infty, 0)$

In Section 2.2.8 we have already discussed an instance of this problem type. The graph given in Figure 2.7 does not fit in the specification of the problem type above since it represents a reduced view. In the complete graph the connections between capability sets and resources are always established via capabilities. The resource *SUB2* would be connected to two capability nodes, one representing the use of *SUB2* as input and one representing the use of *SUB2* as output. *MAT2* on the other hand, although also occurring in two capability sets, would be connected to only one capability node, since it is only used as input.

□

3

The general scheduling problem

3.1 Introduction

Practical relevance and theoretical importance are the criteria that are used for the justification of research in machine scheduling or any other research area. Theoretical importance in machine scheduling is often closely related to practical relevance: methods and techniques developed for a purely theoretical problem as well as the obtained insight may be useful in developing solution methods for more practical problems.

In recent years, much research in machine scheduling has failed to meet these criteria. Many very specific problem types are studied, most of which are of only limited practical relevance, and the methods that are developed for these theoretical problems use only occasionally interesting new ideas or concepts. Furthermore, theoretical exercises that have been performed for one type of problem are repeated again and again for slightly different problem types.

A general treatment of manipulations for a large class of problems, as is required in a planning board generator, does not benefit from research in which attention is focused on the distinguishing characteristics of problem types rather than on their common aspects. Methods are required that can be applied to a broad range of problem types but still use problem-specific information in such a way that efficient application to individual problem types is possible.

To support the advisor function of a planning board, we need two kinds of methods, *construction* methods and *improvement* or *local search* methods. Construction methods are required for the creation of an initial plan, for the completion of partial plans, and for the efficient insertion of newly arriving processes. Local search methods help to create plans of sufficient quality, can be used to give suggestions for improvement to the planner, and are essential for efficiently adjusting a plan when infeasibility arises due to machine break-downs or other

unexpected circumstances.

Both solution methods require an appropriate mathematical representation of solutions. A good representation enables efficient evaluation of the quality of solutions and highlights their structural properties. An example of such a representation is the directed graph that is used for the job shop scheduling problem. This directed graph, which is closely related to the disjunctive graph of Roy and Sussmann [1964], has proved its usefulness in several solution methods. The quality of a solution is equivalent to the length of the longest path in the graph.

In this chapter, I discuss first the job shop scheduling problem and the associated graph representation of solutions. Then I show that we can use a similar representation for solutions of scheduling problems with more general constraints, such as deadlines and delay constraints, and more general objective functions, including a large class of non-regular objective functions. These generalizations result in the introduction of the *general scheduling problem* and the associated *solution network*. The quality of a solution of the general scheduling problem is obtained by solving a maximum cost flow problem in this solution network. Finally, I discuss some properties of solution networks, that will be used in Chapter 4 in order to develop efficient construction and local search methods.

3.2 The job shop scheduling problem

The job shop scheduling problem can be described as follows. We are given a set $\mathcal{M}$ of machines and a set $\mathcal{O}$ of operations. Associated with each operation $v \in \mathcal{O}$ is a processing time $p_v \in \mathbb{N}$ and a machine $M_v \in \mathcal{M}$. Operation v must be performed on machine M_v during p_v consecutive time units. Preemption is not allowed: the processing of an operation may not be interrupted and resumed at a later time. There is a precedence relation Π on $\mathcal{O}$, where $(v, w) \in \Pi$ indicates that operation w cannot be started before operation v is completed. In the job shop problem, the precedence relation is closely related to the concept of *job*. A job consists of a set of operations that must be performed in a prespecified order. If v and w are two consecutive operations of the same job, then $(v, w) \in \Pi$.

A schedule is characterized by the starting times S_v for all operations v. A schedule is feasible if at any given moment no two operations are performed on the same machine and the precedence constraints are satisfied. The objective is to find a feasible schedule with minimal makespan. The makespan, $C_{\max}$, of a schedule is equal to the maximum completion time over all operations.

Let Δ be the collection of pairs of operations that require the same machine, i.e., $\Delta = \{\{v, w\} \mid M_v = M_w\}$. The job shop problem can then be formulated as a disjunctive programming problem in the following way.

$$
(JP) \quad
\begin{array}{lll r}
\min & C_{\max} & & \\
\text{s.t.} & C_{\max} \geq S_v + p_v, & \forall v \in \mathcal{O} & (3.1) \\
& S_w \geq S_v + p_v, & \forall (v,w) \in \Pi & (3.2) \\
& S_w \geq S_v + p_v \vee S_v \geq S_w + p_w, & \forall \{v,w\} \in \Delta & (3.3) \\
& S_v \geq 0, & \forall v \in \mathcal{O} & (3.4)
\end{array}
$$

The important scheduling decision is to determine for each pair of operations that require the same machine, whether one is performed before the other, or the other way around. This decision is often referred to as a *selection*: for each of the disjunctive constraints (3.3), either $S_w \geq S_v + p_v$ or $S_v \geq S_w + p_w$ is selected. A selection, denoted by σ, transforms (JP) into a linear programming problem (JP^σ). Let Δ^σ be the collection of ordered pairs of operations (v, w) that require the same machine, and for which it has been decided that v must be performed before w. The linear program is then as follows.

$$
(JP^\sigma) \quad
\begin{array}{lll r}
\min & C_{\max} & & \\
\text{s.t.} & C_{\max} \geq S_v + p_v, & \forall v \in \mathcal{O} & (3.1) \\
& S_w \geq S_v + p_v, & \forall (v,w) \in \Pi & (3.2) \\
& S_w \geq S_v + p_v, & \forall (v,w) \in \Delta^\sigma & (3.3') \\
& S_v \geq 0, & \forall v \in \mathcal{O} & (3.4)
\end{array}
$$

This linear program has a very special structure, which is highlighted when the dual problem is considered. With each constraint of the form $S_w \geq S_v + p_v$ (by rewriting $C_{\max}$ as S_z, also the constraints (3.1) are of this form) we associate a dual variable x_{vw}. Let $\bar{A}$ be the set of constraints of the aforementioned form, i.e., $\bar{A}$ is the set of constraints (3.1), (3.2), and (3.3′). For the sake of convenience, each such constraint will be denoted by a pair (v, w). The dual problem can then be formulated as follows.

$$
(JD^\sigma) \quad
\begin{array}{lll r}
\max & \sum_{(v,w)\in\bar{A}} p_v x_{vw} & & \\
\text{s.t.} & \sum_{(v,z)\in\bar{A}} x_{vz} = 1 & & (3.5) \\
& \sum_{(u,v)\in\bar{A}} x_{uv} - \sum_{(v,w)\in\bar{A}} x_{vw} \leq 0\,, & \forall v \in \mathcal{O} & (3.6) \\
& x_{vw} \geq 0\,, & \forall (v,w) \in \bar{A} & (3.7)
\end{array}
$$

(JD^σ) is a maximum cost flow problem. By introducing slack variables x_{sv} for all $v \in \mathcal{O}$, the constraints (3.6) are transformed into equalities. Constraints

(3.5) and (3.6) then represent flow balance constraints for a network $N = (V, A)$, with $V = \mathcal{O} \cup \{s, z\}$, and $A = \bar{A} \cup \{(s, w) \mid w \in V \setminus \{s, z\}\}$.

That is, for each constraint of the form $S_w \geq S_v + p_v$ there is an arc $(v, w) \in A$, and for each constraint of the form $S_v \geq 0$ there is an arc $(s, v) \in A$. This corresponds to rewriting the nonnegativity constraints as $S_v - S_s \geq 0$, where S_s has a fixed value of 0. The cost of an arc (v, w) in the network is equal to p_v, with $p_s = 0$, and each arc has infinite capacity. The demand of node z is 1, all other demands are 0. Node s has indefinite supply (there is no flow balance constraint for node s), but we can transform the problem into a balanced problem by taking the supply of s equal to 1, i.e., $\sum_{(s,w)\in\bar{A}} x_{sw} = 1$. (JD^σ) thus represents the problem of finding a maximum cost flow of size 1 from node s to node z in the network N, or equivalently, the problem of finding the longest path from node s to node z. The longest path from s to z is often referred to as the *critical path.*

From duality theory we know that, if one of (JP^σ) and (JD^σ) is feasible and has a finite optimal value, the other is also feasible and has identical optimal value. Thus, for a given selection of the disjunctive constraints in a job shop problem, the makespan can be computed by solving a longest path problem. In fact, also the starting times of all operations can be found in that way. By taking as the starting time of operation v the length of the longest path from node s to node v in N, an optimal solution of (JP^σ) is obtained. This solution corresponds to the so-called *left-justified* schedule. In a left-justified schedule, no operation can be started earlier without changing for at least one machine the order in which the operations are performed on that machine. Often, there are several other optimal solutions of (JP^σ). The *right-justified* schedule, for example, is the one in which all operations are started as late as possible without increasing the makespan. By studying the relation between the problems (JP^σ) and (JD^σ), we can describe the set of all optimal solutions. I will come back to this in Section 3.4.3.

Example 3.1 Consider the following instance of the job shop problem. There are three machines, A, B, and C, and three jobs. The first job consists of operations 1, 2, and 3, which have to be performed in that order, the second job consists of operations 4, 5, and 6, and the third job of operations 7, 8, and 9.

The required machines and processing times of the operations are given in the following table.

v	1	2	3	4	5	6	7	8	9
M_v	A	C	B	B	A	C	A	B	C
p_v	3	4	2	3	3	2	3	2	3

Let (1, 7, 5) be the order in which the operations are performed on machine A, (4, 8, 3) on machine B, and (2, 6, 9) on machine C. The network and the left-justified schedule corresponding to this selection are given in Figure 3.1. Note that, for reasons of clarity, not all the arcs (s, v) and (v, z) are given.

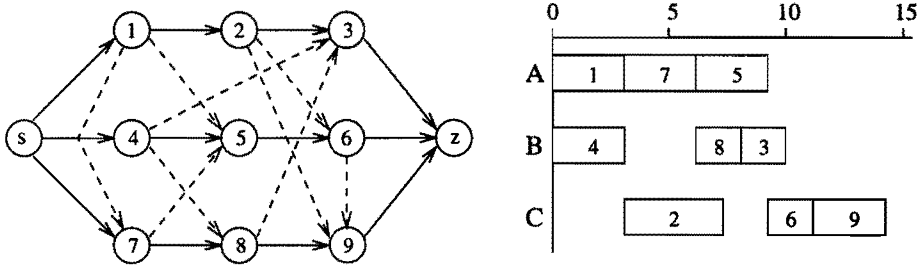

Figure 3.1: A job shop example; network and left-justified schedule.

The critical path is the path (1,7,5,6,9), and the makespan is 14. □

The fact that the makespan of a solution to the job shop scheduling problem can be computed by solving a longest path problem is well known. Roy and Sussmann [1964] have introduced the disjunctive graph representation for scheduling problems. In a disjunctive graph, nodes represent operations, (directed) arcs represent precedence constraints, and (undirected) edges represent disjunctive constraints. The scheduling decision results in orienting the edges one way or the other. The graph that then arises is identical to the network N, except for the nodes s and z, and the arcs incident to these nodes.

Many solution procedures for the job shop scheduling problems make use of the disjunctive programming formulation and the disjunctive graph representation. In Chapter 4, I will show that similar solution procedures may also be applied to other scheduling problems, with more general constraints and more general objective functions. An important step in this direction is made in the next section. There I will discuss the main theorem, which states that also for more general scheduling problems optimal starting times for a given processing order of the operations can be found by solving a maximum cost flow problem.

3.3 The general scheduling problem

3.3.1 Maximum cost flow

Consider the general maximum cost flow problem on a network $N = (V, A)$.

$$
(\mathcal{D}) \quad
\begin{array}{lll}
\max & \sum_{(v,w)\in A} c_{vw} x_{vw} & \\
\text{s.t.} & \sum_{(u,v)\in A} x_{uv} - \sum_{(v,w)\in A} x_{vw} = b_v\,, & \forall v \in V \qquad (3.8) \\
 & 0 \le x_{vw} \le h_{vw}\,, & \forall (v,w) \in A \qquad (3.9)
\end{array}
$$

Associated with each arc $(v, w) \in A$ are a *cost* c_{vw} and a *capacity* h_{vw}. Associated with each node $v \in V$ is a *demand* b_v. Negative demands are sometimes referred to as *supplies*. Note that in most literature about network flow problems, the flow balance constraints are of the form *outflow − inflow = supply*, whereas the constraints in $(\mathcal{D})$ are of the form *inflow − outflow = demand*. Accordingly, the signs of the b_v in the formulation that I use are different from those in other formulations.

The dual of $(\mathcal{D})$ is

$$
(\mathcal{P}) \quad
\begin{array}{lll}
\min & \sum_{v\in V} b_v S_v + \sum_{(v,w)\in A} h_{vw}\alpha_{vw} & \\
\text{s.t.} & S_w - S_v + \alpha_{vw} \ge c_{vw}\,, & \forall (v,w) \in A \qquad (3.10) \\
 & S_v \text{ unrestricted}\,, & \forall v \in V \qquad (3.11) \\
 & \alpha_{vw} \ge 0\,, & \forall (v,w) \in A \qquad (3.12)
\end{array}
$$

I will now discuss the main results in maximum cost flow duality. For a more extensive treatment of this subject, see, e.g., Ahuja et al. [1993]. Note first that constraints (3.10) can be rewritten as

$$\alpha_{vw} \ge c_{vw} + S_v - S_w.$$

Since α_{vw} occurs with a nonnegative coefficient h_{vw} in the objective function, we have

$$\alpha_{vw} = \max\{0, c_{vw} + S_v - S_w\}.$$

Hence, for given S_v and S_w, α_{vw} is completely determined. Therefore, a solution of the problem $(\mathcal{P})$ is characterized by the values of S_v for all $v \in V$.

Lemma 3.1 (*Weak duality*)
Let $z_{\mathcal{P}}(S)$ denote the objective function value of some feasible solution S of $(\mathcal{P})$ and let $z_{\mathcal{P}}(x)$ denote the objective function value of some feasible solution x of $(\mathcal{D})$. Then $z_{\mathcal{P}}(S) \geq z_{\mathcal{P}}(x)$.

From Lemma 3.1 it follows that any feasible solution of $(\mathcal{D})$ gives a lower bound on the optimal value of $(\mathcal{P})$, and any feasible solution of $(\mathcal{P})$ gives an upper bound on the optimal value of $(\mathcal{D})$.

Lemma 3.2 (*Strong duality*)
If any of the problems $(\mathcal{P})$ and $(\mathcal{D})$ has a finite optimal solution, then there exist solutions S^* of $(\mathcal{P})$ and x^* of $(\mathcal{D})$ such that $z_{\mathcal{P}}(S^*) = z_{\mathcal{P}}(x^*)$.

Problem $(\mathcal{P})$ is a generalization of (JP^{σ}), formulated in Section 3.2. Just as it is possible to solve (JP^{σ}) by considering its dual, problem $(\mathcal{D})$ can be used to solve $(\mathcal{P})$. Obviously, the complexity of solving a general maximum cost flow problem is higher than the complexity of solving a longest path problem (see Section 3.4.3), but, as a compensation, the formulation of problem $(\mathcal{P})$ offers many modeling options that can be exploited in order to handle more general scheduling problems. I will now discuss the possibilities for modeling constraints and objective functions.

3.3.2 Constraints

The constraints $S_w - S_v + \alpha_{vw} \geq c_{vw}$ as they occur in the formulation of $(\mathcal{P})$ can be used to model different kinds of scheduling constraints.

Start-start constraints
Similarly as in the job shop scheduling problem, we introduce a dummy operation s with processing time $p_s = 0$ and fixed starting time $S_s = 0$. Although $S_s = 0$ does not fit in the formulation of $(\mathcal{P})$, this is no real problem. Adding the constraint $S_s = 0$ to $(\mathcal{P})$ corresponds to replacing the constraint (3.8) for s in $(\mathcal{D})$ by

$$\sum_{u:(u,s)\in A} x_{us} - \sum_{w:(s,w)\in A} x_{sw} + \delta_s = b_s,$$

where δ_s is an unrestricted variable. The variable δ_s does not occur in any other constraint, and therefore this constraint is never binding or, equivalently, there is no flow balance constraint for node s. However, since total demand must be equal to total supply, and all other demands and supplies are known, the supply of

node s must be equal to the total excess demand of all other nodes in the network $N = (V, A)$. That is,

$$\sum_{u:(u,s)\in A} x_{us} - \sum_{w:(s,w)\in A} x_{sw} = - \sum_{v\in V\setminus\{s\}} b_v.$$

Since $S_s = 0$, the value of b_s has no effect on the value of the objective function of $(\mathcal{P})$.

We can now handle several kinds of constraints that occur in scheduling problems. The *nonnegativity* constraints $S_v - S_s \geq 0$ and the *precedence* constraints $S_w - S_v \geq p_v$ have already been discussed for the job shop scheduling problem. We can rewrite such constraints as $S_w - S_v + \alpha_{vw} \geq c_{vw}$, which is of the form of constraints (3.10), and force α_{vw} to take on the value 0 by using the coefficient $h_{vw} = +\infty$. For the corresponding (dual) maximum cost flow problem, this would imply that the arc (v, w) has infinite capacity.

Other constraints that occur frequently in scheduling problems are *release time* constraints ($S_v \geq r_v$) and *deadline* constraints ($S_v + p_v \leq d_v$). For these constraints, we make use of the dummy operation s again, and formulate them as $S_v - S_s \geq r_v$ and $S_s - S_v \geq -d_v + p_v$, respectively. The associated α's are again penalized with an infinite h-coefficient.

Minimum and maximum delay constraints are generalizations of the precedence constraints. A minimum delay constraint ($S_w - S_v \geq p_v + \delta_{vw}^{\min}$) indicates that an operation w cannot be started earlier than $\delta_{vw}^{\min}$ time units after the completion of operation v. Similarly, a maximum delay constraint ($S_v - S_w \geq -p_v - \delta_{vw}^{\max}$) indicates that operation w must be started before $\delta_{vw}^{\max}$ time units after the completion of operation v.

All these constraints are of the form $S_w - S_v \geq c_{vw}$, relating the starting time of one operation to that of another operation. I will refer to them as *start-start* constraints. They comprise all four types of precedence relations (finish-to-start, start-to-start, start-to-finish, and finish-to-finish) discussed in Chapter 2 (see also Bartusch et al. [1988]). Associated with each start-start constraint $S_w - S_v \geq c_{vw}$ there is an arc (v, w) with cost c_{vw} and infinite capacity in the network N.

The start-start constraints can also be applied to represent other kinds of relations and constraints. As shown before, the correct value of the makespan is obtained by introducing start-start constraints $C_{\max} - S_v \geq p_v$ for all $v \in V$. It may also be useful to drop the interpretation of the S_v's as starting times altogether. For example, let i_v be the amount of some resource required by operation v, and let I_v be the size of the inventory of that resource after operation v is performed. Suppose that operation w is performed immediately after operation v. Then, the 'start-start' constraint $I_w - I_v \geq i_w$ must hold. In this thesis, however, I will not deal with this kind of constraint.

Disjunctive constraints

In the job shop problem, the disjunctive constraints indicate that for each pair of operations that require the same machine (vehicle, workman, or any renewable resource of unit capacity) it must be determined whether one is performed before or after the other. When these decisions have been made, the disjunctive constraints are transformed into ordinary start-start constraints. It is possible to formulate more general disjunctive constraints which still have the property that, for any selection, they reduce to start-start constraints.

One generalization involves the introduction of machine sets. Let $M_v \subseteq \mathcal{M}$ be the machine set associated with operation v. Then v must be performed on all machines in M_v simultaneously during p_v consecutive time units. Machine sets are also useful for modeling other constraints than those reflecting limited machine capacity. Consider, for example, a situation in which two operations, v and w, do not require a common machine but are still not allowed to be performed at the same time. This situation occurs for example in open shop problems and is modeled by introducing a dummy machine which then is included in both M_v and M_w.

Sequence-dependent set-up times arise frequently in machine scheduling problems. If operation w is performed immediately after operation v on some machine, then a set-up is required to bring that machine in the required mode. The corresponding nonnegative set-up time is denoted by δ_{vw}. The set-up times are assumed to satisfy the triangle inequality: $\delta_{uv}+\delta_{vw} \geq \delta_{uw}$. The set-up times will often be incorporated in sequence-dependent processing times $p_{vw} = p_v + \delta_{vw}$.

The general disjunctive constraints can then be formulated as

$$S_w \geq S_v + p_{vw} \vee S_v \geq S_w + p_{wv} \quad \forall \{v, w\} : M_v \cap M_w \neq \emptyset.$$

Much care must be taken in modeling a problem when sequence-dependent processing times occur. For example, if set-ups must be performed for several machines in a machine set M_v, then the largest set-up time determines the starting time of operation v.

Adjusted start-start constraints

From now on, I will refer to the constraints $S_w - S_v + \alpha_{vw} \geq c_{vw}$ as *adjusted* start-start constraints. The terms α_{vw} that occur in the adjusted start-start constraints all appear in only one constraint, and can therefore not be related to other variables than the S_v and S_w in that constraint. A natural interpretation of the adjusted start-start constraints is to regard them as 'soft' start-start constraints. Consider the constraint $S_w - S_v + \alpha_{vw} \geq c_{vw}$. If $S_w - S_v \geq c_{vw}$, then α_{vw}, if it has a positive coefficient h_{vw} in the objective function, will take on the value 0. If $S_w - S_v < c_{vw}$, then α_{vw} will be equal to $c_{vw} + S_v - S_w$. Thus, α_{vw} can be

seen as the amount of violation of a constraint $S_w - S_v \geq c_{vw}$, and h_{vw} can be considered as the penalty that is to be paid for each unit by which that constraint is violated.

In fact, the adjusted start-start constraints are no real constraints. Their main use is in modeling various optimization criteria, as will be seen shortly.

3.3.3 Objective functions

In Section 3.2, we have seen how we can model an objective like makespan minimization by introducing start-start constraints $C_{\max} - S_v \geq p_v$ for all operations v, and including $C_{\max}$ in the objective function. In a similar way, we can model the maximum lateness, which is defined as $\max_{v \in \mathcal{O}} S_v + p_v - d_v$, with d_v the due date of operation v. We introduce start-start constraints $L_{\max} - S_v \geq p_v - d_v$ and include $L_{\max}$ in the objective function.

More generally, let

$$S_{z^i_{\max}} = \max_{v \in \mathcal{O}_i} \{S_v + \eta_v\},$$

with $\mathcal{O}_i$ some subset of the set of operations, and η_v, $\forall v \in \mathcal{O}_i$, given constants. Let the objective be to minimize $S_{z^i_{\max}}$. This situation can be modeled by introducing start-start constraints $S_{z^i_{\max}} - S_v \geq \eta_v$, for all $v \in \mathcal{O}_i$, and include $S_{z^i_{\max}}$ in the objective function with coefficient $b_{z^i_{\max}} = 1$. Analogously, let

$$S_{z^j_{\min}} = \min_{v \in \mathcal{O}_j} \{S_v + \varepsilon_v\}.$$

If we want to maximize $S_{z^j_{\min}}$, we introduce constraints $S_v - S_{z^j_{\min}} \geq -\varepsilon_v$, and include $S_{z^j_{\min}}$ with coefficient $b_{z^j_{\min}} = -1$ in the objective function.

Note that we are not able to handle the maximization of $S_{z^i_{\max}}$ or the minimization of $S_{z^j_{\min}}$. The variables $S_{z^i_{\max}}$ and $S_{z^j_{\min}}$ would take on infinite values rather than the actually desired values, because the start-start constraints would not be restrictive.

The most straightforward use of the first term of the objective function of $(\mathcal{P})$, $\sum b_v S_v$, is in minimizing the weighted sum of starting times or, equivalently since preemption is not allowed, the weighted sum of completion times. In that case, b_v is set to be equal to the weight associated with operation v.

Often, however, the aim is not to complete the operations as *early* as possible, but as *close* as possible to their due dates. The objective then is, for example, to minimize the total sum of earlinesses and tardinesses:

$$\min \sum_{v \in \mathcal{O}} E_v + \sum_{v \in \mathcal{O}} T_v.$$

The earliness E_v of operation v is defined as $\max\{d_v - (S_v + p_v), 0\}$, and the tardiness T_v as $\max\{(S_v + p_v) - d_v, 0\}$. The start-start constraints do not apply if we want to model the earliness of an operation, but here the adjusted start-start constraints come to the rescue:

$$S_v - S_s + E_v \geq d_v - p_v,$$

$$S_s - S_v + T_v \geq p_v - d_v,$$

$$E_v \geq 0, T_v \geq 0.$$

E_v is the adjustment term α_{sv} in the adjusted start-start constraint $S_v - S_s + \alpha_{sv} \geq d_v - p_v$. In the objective function, the coefficient h_{sv} has value 1. Similarly, T_v is equivalent to the adjustment term α_{vs} and appears in the objective function with coefficient $h_{vs} = 1$.

Note that it is also possible to model the tardiness by introducing a start-start constraint $S_{T_v} - S_v \geq p_v - d_v$ and $S_{T_v} \geq 0$, and taking $b_{T_v} = 1$. Both approaches have their advantages and disadvantages. An advantage of using an adjusted start-start constraint is that only one constraint is introduced, corresponding to one arc (v, s) with capacity 1 in the dual network. When start-start constraints are applied, one node T_v and two arcs of infinite capacity, (s, T_v) and (v, T_v), are introduced. Using adjusted start-start constraints results in a smaller network than using start-start constraints.

As the following lemma shows, it is possible to model any cost function that is bounded from below and convex piecewise linear in the starting time of an operation, or in the difference between two starting times, $S_w - S_v$.

Lemma 3.3 (*Convex piecewise linear functions*)
Let $f : \mathbb{R} \rightarrow \mathbb{R}$ be a convex piecewise linear function, with m breakpoints $s_1, \ldots, s_m$, and let $l \in \{1, \ldots, m\}$ be such that $f(y) \geq f(s_l), \forall y \in \mathbb{R}$. That is, f is minimal in s_l. Then, for each possible value of y, $f(y)$ is the outcome of a linear program of the following form:

$$\begin{aligned} f(y) = \min \quad & \beta + \sum_{k=1}^{m} \gamma_k \alpha_k \\ \text{s.t.} \quad & y + \alpha_k \geq s_k, && \forall k \in \{1, \ldots, l\} \\ & -y + \alpha_k \geq -s_k, && \forall k \in \{l+1, \ldots, m\} \\ & \alpha_k \geq 0, && \forall k \in \{1, \ldots, m\}, \end{aligned}$$

with $\gamma_k \geq 0$, for all $k \in \{1, \ldots, m\}$.

Proof. Any convex piecewise linear function f with breakpoints $s_1, \ldots, s_m$ and a minimum in s_l can be written as

$$f(y) = \beta + \sum_{k=1}^{l} \gamma_k (s_k - y)^+ + \sum_{k=l+1}^{m} \gamma_k (y - s_{k-1})^+,$$

with $\gamma_k \geq 0$, for all k. Here, $z^+ = \max\{0, z\}$.

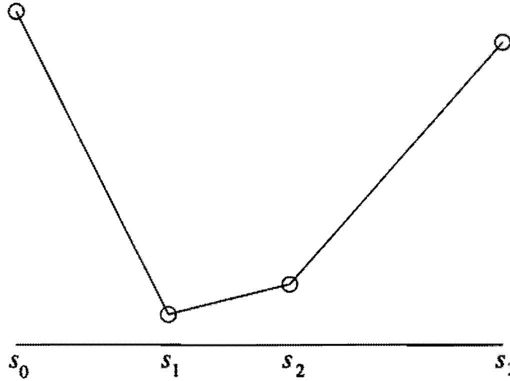

Figure 3.2: A convex piecewise linear function.

Consider, for example, the function f depicted in Figure 3.2. Its slope is θ_1 in the interval $(-\infty, s_1)$, θ_2 in (s_1, s_2), and θ_3 in $(s_2, +\infty)$, and f is minimal for $y = s_1$ ($f(s_1) = \beta$). We write f as

$$f(y) = \beta + \gamma_1 (s_1 - y)^+ + \gamma_2 (y - s_1)^+ + \gamma_3 (y - s_2)^+,$$

with $\gamma_1 = -\theta_1$, $\gamma_2 = \theta_2$, and $\gamma_3 = \theta_3 - \theta_2$.

Furthermore, we have,

$$(s_k - y)^+ = \min\{\alpha_k \mid y + \alpha_k \geq s_k, \alpha_k \geq 0\},$$

and

$$(y - s_k)^+ = \min\{\alpha_k \mid -y + \alpha_k \geq -s_k, \alpha_k \geq 0\}.$$

Thus, since $\gamma_k \geq 0$, for all k, the value of the optimal solution of the linear program formulated in the lemma is equal to $f(y)$. □

By taking $y = S_w - S_v$ (and $-y = S_v - S_w$), the constraints in the linear program in Lemma 3.3 assume the form of the adjusted start-start constraints.

Also the objective function fits in the formulation of $(\mathcal{P})$. This shows that it is possible to model cost functions that are convex piecewise linear in $S_w - S_v$, for any pair (v, w). Since $S_v = S_v - S_s$, also cost functions that are convex piecewise linear in S_v, for any v, can be handled.

Now suppose that we want to model a cost function f that is strictly decreasing in S_v, i.e., there is no breakpoint s_l for which f takes on its minimal value. If it is possible to give an upper bound s_{m+1} on the value of S_v, the function f can be transformed into an equivalent function

$$f'(S_v) = \begin{cases} f(S_v) & , S_v \leq s_{m+1} \\ f(s_{m+1}) & , S_v > s_{m+1} \end{cases},$$

which is convex piecewise linear and has a minimum in the breakpoint s_{m+1}. We then apply Lemma 3.3 to the new function f'.

Similarly, if f is strictly increasing in S_v and s_0 is a lower bound on the value of S_v, we can use

$$f'(S_v) = \begin{cases} f(S_v) & , S_v \geq s_0 \\ f(s_0) & , S_v < s_0 \end{cases}.$$

Summarizing, we can model all cost functions that are convex piecewise linear in S_v and in $S_v - S_w$ by using adjusted start-start constraints, provided that they have a minimum. This minimum may occur either in one of the breakpoints or in some extreme point of the domain.

An example of a cost function that is formulated in terms of the difference of two starting times is the *job handling time*. Consider a job consisting of several operations that have to be performed in a prespecified order. The handling time of a job is defined as the time between the start of the first operation and the completion of the last operation of a job. Let v be the first operation of job j and let w be its last operation. Suppose that we want to minimize the handling time $t_j = S_w + p_w - S_v$ of job j. This can be done by adding the adjusted start-start constraint

$$S_v - S_w + t_j \geq p_w,$$

and including t_j in the objective function with coefficient $h_{t_j} = 1$.

3.3.4 Definition of the general scheduling problem

An instance of the general scheduling problem is described in the following way.

Let $\mathcal{O}$ be the set of operations, and let $\mathcal{M}$ be the set of machines. Associated with each operation $v \in \mathcal{O}$ is a machine set $M_v \subseteq \mathcal{M}$, a processing time $p_v > 0$, a release time r_v, and a deadline d_v. Operation v must be performed on all machines in M_v simultaneously during p_v consecutive time units. Processing operation v cannot be started before its release time and must be completed before its deadline.

Each machine can process at most one operation at a time. If operation w is performed immediately after operation v, then a set-up is required which takes δ_{vw} time units. The set-up times satisfy the triangle inequality, i.e., $\delta_{uv} + \delta_{vw} \geq \delta_{uw}$, for all operations u, v, and w such that $M_u \cap M_v \neq \emptyset$ and $M_v \cap M_w \neq \emptyset$.

Let $\Pi_{\min}$ be the set of minimum delay constraints, and let $\Pi_{\max}$ be the set of maximum delay constraints. A minimum delay (v, w) in $\Pi_{\min}$ indicates that operation w cannot be started earlier than $\delta_{vw}^{\min}$ time units after the completion of operation v. Similarly, a maximum delay constraint (v, w) in $\Pi_{\max}$ indicates that operation w must be started before $\delta_{vw}^{\max}$ time units after the completion of operation v.

The objective is to find a schedule, characterized by the starting times S_v of the operations v, that satisfies all the constraints and that minimizes the cost function

$$C(S) = \sum_{i=1}^{l_1} \lambda_i S_{z^i_{\max}} - \sum_{i=1}^{l_2} \mu_i S_{z^i_{\min}} + \sum_{v \in \mathcal{O}} f_v(S_v) + \sum_{v,w \in \mathcal{O}} f_{vw}(S_w - S_v).$$

In this formulation, $l_1, l_2 \in \mathbb{N}$, $\lambda_i > 0$, and $\mu_i > 0$. $S_{z^i_{\max}} = \max_{v \in \mathcal{O}_{1i}} \{S_v + \eta_{vi}\}$, $i = 1, \ldots, l_1$, for given subsets $\mathcal{O}_{1i} \subseteq \mathcal{O}$. $S_{z^i_{\min}} = \min_{v \in \mathcal{O}_{2i}} \{S_v + \varepsilon_{vi}\}$, $i = 1, \ldots, l_2$, for given subsets $\mathcal{O}_{2i} \subseteq \mathcal{O}$. Here, η_{vi} and ε_{vi} are given constants. The functions f_v and f_{vw} are convex piecewise linear and bounded from below.

A cost function of this form is referred to as a *tractable* cost function. Examples of tractable cost functions are the makespan, the maximum lateness, the weighted sum of completion times, the weighted sum of earlinesses and tardinesses, and the weighted sum of job handling times. There are, however, many more tractable cost functions, including weighted sums of the cost functions just mentioned.

As we have seen in the previous sections, an instance (GP) of the general scheduling problem can be formulated as a disjunctive programming problem.

$$
(GP)\quad
\begin{array}{lll}
\min & \beta + \sum\limits_{v\in V} b_v S_v + \sum\limits_{(v,w)\in A_2} h_{vw}\alpha_{vw} & \\
\text{s.t.} & S_w - S_v \ge c_{vw}\,, & \forall (v,w)\in A_1, \\
& S_w - S_v + \alpha_{vw} \ge c_{vw}\,, & \forall (v,w)\in A_2, \\
& S_w \ge S_v + p_{vw} \vee S_v \ge S_w + p_{wv}\,, & \forall \{v,w\}\in A_3, \\
& S_s = 0, & \\
& \alpha_{vw} \ge 0\,, & \forall (v,w)\in A_2.
\end{array}
$$

Here, $V = \mathcal{O} \cup \{s, z^1_{\max}, \ldots, z^{l_1}_{\max}, z^1_{\min}, \ldots, z^{l_2}_{\min}\}$. A_1 is the set of start-start constraints, comprising the release time and deadline constraints as well as the minimum and maximum delay constraints. A_3 is the set of pairs $\{v, w\}$ of operations that require processing by at least one common machine, i.e., $M_v \cap M_w \neq \emptyset$. The set-up times are incorporated in the processing times: $p_{vw} = p_v + \delta_{vw}$.

Now the main theorem can be formulated.

Theorem 3.4 (*Finding a selection-optimal schedule*)
Consider an instance (GP) of the general scheduling problem. For any selection of the disjunctive constraints, optimal feasible values of S_v, if they exist, can be obtained by solving a maximum cost flow problem.

Proof. A selection σ transforms (GP) into the linear program

$$
(GP^\sigma)\quad
\begin{array}{lll}
\min & \beta + \sum\limits_{v\in V} b_v S_v + \sum\limits_{(v,w)\in A_2} h_{vw}\alpha_{vw} & \\
\text{s.t.} & S_w - S_v \ge c_{vw}\,, & \forall (v,w)\in A_1, \\
& S_w - S_v + \alpha_{vw} \ge c_{vw}\,, & \forall (v,w)\in A_2, \\
& S_w \ge S_v + p_{vw}\,, & \forall (v,w)\in A_3^\sigma, \\
& S_s = 0, & \\
& \alpha_{vw} \ge 0\,, & \forall (v,w)\in A_2.
\end{array}
$$

The dual of (GP^σ), leaving the constant term β out of consideration, is

$$
(GD^\sigma)\quad
\begin{array}{lll}
\max & \sum\limits_{(v,w)\in A^\sigma} c_{vw} x_{vw} & \\
\text{s.t.} & \sum\limits_{(u,v)\in A^\sigma} x_{uv} - \sum\limits_{(v,w)\in A^\sigma} x_{vw} = b_v\,, & \forall v\in V, \\
& 0 \le x_{vw} \le h_{vw}\,, & \forall (v,w)\in A^\sigma,
\end{array}
$$

where $A^\sigma = A_1 \cup A_2 \cup A_3^\sigma$ and $h_{vw} = +\infty$ for all $(v, w) \in A_1 \cup A_3^\sigma$, and $b_s = -\sum_{v\in V\setminus\{s\}} b_v$. It is a maximum cost flow problem on a network $N^\sigma = (V, A^\sigma)$.

If we use a maximum cost flow algorithm that explicitly uses the dual problem (there are several algorithms that do so, see Ahuja et al. [1993]) in order to solve

(GD^σ), then an optimal flow as well as optimal values of S_v and α_{vw}, if they exist, are obtained.

As will be discussed in Section 3.4.3, it is also possible to obtain optimal values of S_v and α_{vw} from the optimal flow irrespective of how this flow is obtained (Lemma 3.12). □

The problem of minimizing the weighted sum of earlinesses and tardinesses of operations on a single machine, where the order in which the operations are performed is fixed, can be considered as a special case of the problem that is discussed here. The polynomial solvability of this problem is one of the major results in scheduling with non-regular objective functions (see, e.g., Baker and Scudder [1990]).

The careful reader will have noticed that A^σ represents both the set of constraints in (GP^σ) and the set of arcs in the network N^σ. Since the relation between the two sets is so strong, I expect that this notation will not cause any confusion.

3.4 Solutions

3.4.1 The solution network

Consider an instance (GP) of the general scheduling problem. For each feasible schedule there is a unique selection of the disjunctive constraints, since for any pair of starting times S_v and S_w only one of the constraints $S_w - S_v \geq p_{vw}$ and $S_v - S_w \geq p_{wv}$ can hold (for p_{vw} and $p_{wv} > 0$). For each selection, however, there are in general many feasible schedules. In fact, there may be several schedules that are optimal for a given selection. Theorem 3.4 states that it is possible to obtain one such *selection-optimal* schedule in polynomial time by solving a maximum cost flow problem. Finding an optimal schedule for the general scheduling problem therefore reduces to finding an optimal selection, i.e., a selection for which the corresponding selection-optimal schedule has lowest cost.

Let Σ be the set of possible selections, and let A_3^σ be the set of selected disjunctive constraints characterizing the selection $\sigma \in \Sigma$. Then (GP) can be reformulated as

$$\min_{\sigma \in \Sigma} g(\sigma),$$

with $g(\sigma)$ the value of the optimal solution of the problem (GP^σ).

The solution space in the reformulated problem consists of the set of possible selections. The properties of a solution can be studied by looking at the sub-

problem that must be solved in order to obtain the corresponding solution value $g(\sigma)$. If (GP^σ) is infeasible, we say that $g(\sigma) = +\infty$. The dual of (GP^σ) is a maximum cost flow problem (GD^σ) on a network $N^\sigma = (V, A_1 \cup A_2 \cup A_3^\sigma) = (V, A^\sigma)$. The network N^σ will be referred to as the *solution network* corresponding to the selection σ.

For the general scheduling problem, the following corollary of Lemma 3.2 holds.

Corollary 3.5 (*Strong duality*)
If $g(\sigma)$, the cost of a selection-optimal schedule corresponding to the selection σ, is finite, then there exists a flow x^* with cost $g(\sigma)$ in the solution network N^σ. If the maximum cost flow in the solution network N^σ has finite cost z^σ, then there exists a selection-optimal schedule with cost z^σ.

3.4.2 Feasibility

From Theorem 3.4 we know that we can find a selection-optimal schedule if one exists. Since all individual cost functions f_v and f_{vw} as well as $S_{z^i_{min}}$ and $S_{z^i_{max}}$ are assumed to be bounded from below, and the number of operations is finite, (GP) is not unbounded, and thus (GP^σ) is not unbounded for any selection σ. It is, however, very well possible that no feasible solution of (GP^σ) exists. The following well known lemma from duality theory deals with the situation in which no finite optimal solution exists.

Lemma 3.6 (*Infeasibility and unboundedness*)
Let $(\mathcal{P})$ be an arbitrary linear programming problem, and let $(\mathcal{D})$ be its dual. If $(\mathcal{P})$ is unbounded, then $(\mathcal{D})$ is infeasible. If $(\mathcal{P})$ is infeasible, then $(\mathcal{D})$ is infeasible or unbounded.

Consider a linear program $\min\{cx \mid Ax = b, x \geq 0\}$. The *homogeneous form* of this problem is $\min\{cx \mid Ax = 0, x \geq 0\}$. The last statement of Lemma 3.6 can be strengthened in the following way.

Lemma 3.7 (*Infeasibility and unboundedness*)
If $(\mathcal{P})$ is infeasible, then $(\mathcal{D})$ is unbounded in homogeneous form.

Proof. See Bazaraa et al. [1990], p.252. □

Now we can formulate a necessary and sufficient condition for a selection to be feasible. Let the capacity of a cycle in a network be defined as the smallest capacity of any of the arcs in the cycle, and let the cost of a cycle be defined as the sum of the costs of all arcs in the cycle.

Theorem 3.8 (*Feasibility*)
Consider an instance (GP) of the general scheduling problem. Let σ be an arbitrary selection of the disjunctive constraints. A feasible schedule corresponding to this selection exists if and only if the network N^σ does not contain a cycle of infinite capacity with positive cost.

Proof. Suppose (GP^σ) is infeasible. The homogeneous form of (GD^σ) is

$$
(GD_0^\sigma) \quad
\begin{array}{lll}
\max & \sum_{(v,w)\in A^\sigma} c_{vw}x_{vw} & \\
\text{s.t.} & \sum_{(u,v)\in A^\sigma} x_{uv} - \sum_{w:(v,w)\in A^\sigma} x_{vw} = 0\,, & \forall v \in V \\
 & 0 \le x_{vw} \le h_{vw}\,, & \forall (v,w) \in A^\sigma
\end{array}
$$

(GD_0^σ) is again a maximum cost flow problem on N^σ. From Lemma 3.7 it follows that (GD_0^σ) is unbounded. Because all demands are zero, positive flow can only occur in cycles in N^σ. The number of cycles is finite. Therefore, a cycle with positive cost and infinite capacity must exist.

Now suppose that there exists a cycle in N^σ of infinite capacity and with positive cost. Then, (GD^σ) is unbounded, which implies that (GP^σ) is infeasible (Lemma 3.6). □

Note that in the case of the job shop problem all arcs in a solution network have infinite capacity and positive cost, except for the arcs leaving s.

Corollary 3.9 (*Job shop problem*)
A selection of the disjunctive constraints in a standard job shop problem is infeasible if and only if the corresponding solution network contains a cycle.

Since all arcs in A_3^σ have infinite capacity and positive cost, also the following corollary of Theorem 3.8 holds.

Corollary 3.10 (*Total ordering on machines*)
Consider an instance (GP) of the general scheduling problem. Let σ be an arbitrary selection of the disjunctive constraints. If the corresponding ordering imposed on the operations that require a common machine is not transitive, then the selection σ is infeasible.

In the following, $N_\infty = (V, A_\infty)$ denotes the network obtained from a solution network N by deleting all arcs with finite capacity (thus, $h_{vw} = \infty$, for $(v, w) \in A_\infty$). A selection σ is feasible if and only if N_∞^σ does not contain a cycle with positive cost.

3.4.3 Obtaining selection-optimal schedules from flows

We have seen that the value of a selection-optimal schedule can be obtained by computing the maximum cost flow in the solution network corresponding to that selection. We are, however, not just interested in the value of the schedule, but also in the schedule itself, i.e., the starting times of the various operations. In general, there are several schedules that are optimal for a given selection. We can use the well known complementary slackness relations to obtain a description of the set of selection-optimal schedules.

Lemma 3.11 (*Complementary slackness relations*)
Let $(\mathcal{D})$ be a maximum cost flow problem on a network $N = (V, A)$, and let $(\mathcal{P})$ be its dual. Let $c^S_{vw} = c_{vw} + S_v - S_w$ denote the reduced cost *of an arc $(v, w) \in A$ with respect to the values of S_v and S_w. S^* is an optimal solution of $(\mathcal{P})$ and x^* is an optimal solution of $(\mathcal{D})$ if and only if for every $(v, w) \in A$:*

If $c^{S^*}_{vw} > 0$, then $x^*_{vw} = h_{vw}$.
If $0 < x^*_{vw} < h_{vw}$, then $c^{S^*}_{vw} = 0$.
If $c^{S^*}_{vw} < 0$, then $x^*_{vw} = 0$.

Corollary 3.12 (*Relating optimal solutions of $(\mathcal{P})$ and $(\mathcal{D})$*)
Let x^* be an optimal solution of $(\mathcal{D})$. Let S be a feasible solution of the following system of equalities and inequalities:

$$S_w - S_v \geq c_{vw}, \forall (v, w) : x^*_{vw} = 0$$

$$S_v - S_w \geq -c_{vw}, \forall (v, w) : x^*_{vw} = h_{vw}$$

$$S_w - S_v = c_{vw}, \forall (v, w) : 0 < x^*_{vw} < h_{vw}$$

Then S is an optimal solution of $(\mathcal{P})$.

A solution of the system mentioned in Corollary 3.12 can be obtained by solving a longest path problem on the so-called *residual network* corresponding to the flow x^*. For each (v, w) such that $x^*_{vw} = 0$ there is an arc (v, w) with cost c_{vw} in the residual network; for each (v, w) such that $x^*_{vw} = h_{vw}$ we introduce an arc (w, v) with cost $-c_{wv}$; finally, for each (v, w) such that $0 < x^*_{vw} < h_{vw}$, an arc (v, w) with cost c_{vw} and an arc (w, v) with cost $-c_{wv}$ are introduced. By taking S_v equal to the length of the longest path from s to v in the residual network, an optimal solution of $(\mathcal{P})$ is obtained.

The S thus obtained could be called the left-justified schedule corresponding to x^*, because the S_v are as small as possible while still satisfying the system in

Corollary 3.12. Similarly, by adding a node t and arcs (v, t) with zero cost for all $v \in V$ to the residual network, computing the length d_{vt} of the longest path from v to t, and taking $S_v = d_{st} - d_{vt}$ for all $v \in V$, also the right-justified schedule corresponding to an optimal flow in the solution network can be found. Note, however, that $(\mathcal{D})$ may have several optimal solutions, each of which has different left-justified and right-justified schedules.

In order to obtain a schedule that is optimal for a given feasible selection σ of the disjunctive constraints, one must construct the solution network N^σ, compute the maximum cost flow x^* in N^σ, and extract starting times S_v from the optimal solution of the maximum cost flow problem. The time required to create a network is negligible compared to the time required to solve a maximum cost flow problem on that network. Furthermore, since starting times can be obtained from an optimal flow by solving a longest path problem (Corollary 3.12), which is a special case of the maximum cost flow problem, the time complexity of finding a selection-optimal schedule is completely determined by the complexity of the maximum cost flow problem.

The maximum cost flow problem and the equivalent minimization variant have received much attention throughout the last decennia, and many algorithms have been proposed. Ahuja et al. [1993] discuss five basic pseudo-polynomial algorithms and six polynomial algorithms, and they give references to many more publications in this field. The underlying ideas in all these algorithms are quite similar: either one maintains a feasible flow and gradually reduces dual infeasibility, or one tries to achieve primal feasibility while maintaining dual feasibility. Still, the performance of the different algorithms varies considerably. It is therefore not obvious which type of algorithm will be most efficient in solving maximum cost flow problems that arise from the general scheduling problem, and it is worthwhile to study the relation between problem structure (cost function, precedence structure, etcetera) and the performance of network flow algorithms. The best available time bound for solving the maximum cost flow problem is $\mathcal{O}(\min\{nm \log(n^2/m) \log(nC), nm(\log\log U) \log(nC), (m \log n)(m+n \log n)\})$, where n is the number of nodes, m is the number of arcs, C is the largest arc cost, and U is the largest arc capacity (see Ahuja et al. [1993]).

The longest path problem can be solved in $\mathcal{O}(\min\{nm, n^{1/2}m \log(nC)\})$ time. In the same time, also the presence of a cycle with positive cost can be detected. If the network is acyclic, as in the case of a feasible selection in the job shop problem (Corollary 3.9), longest paths can be found in $\mathcal{O}(m)$ time.

The given time bounds show that the number of nodes and the number of arcs determine to a large extent the amount of time required to solve maximum cost flow and longest path problems. This may be important when modeling issues

have to be settled. For example, as discussed in Section 3.3.3, one can model the tardiness of an operation in two ways, either by using start-start constraints or by using an adjusted start-start constraint. The first method results in the introduction of a node and two arcs in the solution network, whereas the second method requires the introduction of only one arc.

Note also, that for each breakpoint in a piecewise linear cost function an arc is introduced. It is possible to get close approximations of non-linear convex functions by means of piecewise linear functions, but this requires a large number of breakpoints, and therefore a large number of arcs in the solution networks.

3.4.4 The reduced solution network

Corollary 3.10 offers a possibility for reducing the size of a solution network in order to speed up the maximum cost flow algorithm. Let A_3^σ be the set of selected disjunctive constraints, corresponding to a selection σ. From Corollary 3.10 we know that, if σ is a feasible selection, (V, A_3^σ) does not contain a cycle, which implies that for each machine a total ordering on all operations that require that machine must exist. The following lemma states that only arcs (u, v) such that u and v are consecutive operations in such an ordering have to be included in the solution network.

Lemma 3.13 (*Reduced solution network*)
Let σ be a feasible selection, and let $O_\mu = (v^1, v^2, \ldots, v^k)$ be the corresponding processing order of operations on machine μ. Any maximum cost flow $\bar{x}$ in the solution network N^σ is such that $\bar{x}_{v^i, v^j} = 0$, for all i, j, with $j \neq i + 1$.

Proof. Let x be a maximum cost flow in the solution network N^σ. Suppose that $x_{v^i, v^j} = \zeta > 0$, for some i, j, with $j \neq i + 1$. All arcs $(v^k, v^l), l > k$, have positive cost p_{v^k, v^l} and infinite capacity. Furthermore, since $p_{uw} = p_u + \delta_{uw}$, with $p_u > 0$ and the set-up times δ_{uv} satisfying the triangle inequality, we have $p_{v^k, v^l} + p_{v^l, v^m} > p_{v^k, v^m}$, $\forall k, l, m$. Therefore, the flow $\bar{x}$ with $\bar{x}_{v^i, v^j} = 0$, and $\bar{x}_{v^k, v^{k+1}} = x_{v^k, v^{k+1}} + \zeta$, for $k = i, \ldots, j - 1$, is feasible and has higher cost, thus contradicting the optimality of the flow x. □

Thus, no arc (v^i, v^j), with $j \neq i + 1$, is ever used in an optimal flow. Any solution network can therefore be reduced by removing these redundant arcs. Note that also the arcs (s, v^i), with $i > 1$, corresponding to constraints $S_{v^i} \geq 0$, can be removed as long as the arc (s, v^1) remains. The same procedure can be applied to all other machines. The resulting network will be called the *reduced solution network*. The number of arcs in a reduced solution network can be substantially

smaller than the number of arcs in the original *complete* solution network. Let ν be the number of operations that require a specific machine. The number of arcs for this machine in the complete solution graph is $\mathcal{O}(\nu^2)$, whereas in the reduced solution graph only $\mathcal{O}(\nu)$ arcs remain. When the reduced solution graph is used, optimal starting times for a feasible selection of a job shop problem can be obtained in $\mathcal{O}(n)$ time, where n is equal to the number of operations.

3.5 Other objective functions and constraints

In the previous sections, we have established that the main decision that has to be made in order to find a solution of the general scheduling problem consists of determining the order in which operations are performed on the various machines. Given this order, corresponding optimal starting times of the operations can be obtained by solving a maximum cost flow problem. In Chapter 4, this result will be used in the development of solution methods.

In this section, some types of constraints and cost functions are discussed that do not fit in the description of the general scheduling problem as given in Section 3.3.4. I will not give an extensive treatment of all possible complications. First, I will show that in case of a regular objective function, even if it does not fit in the formulation of (GP), the subproblem still reduces to a longest path problem. Then, I will discuss a type of objective function that makes this subproblem $\mathcal{NP}$-hard. Finally, I will discuss a class of more general constraints and formulate a conjecture on the $\mathcal{NP}$-hardness of the corresponding subproblem.

3.5.1 Regular cost functions

A cost function is called regular if it is non-decreasing in the starting times of the operations. Consider now the problem GSPR, a variant of the general scheduling problem in which the cost function is regular but not necessarily convex.

Lemma 3.14 (*Left-justified schedules*)
Let N^σ be the solution network corresponding to a feasible selection σ for an instance of GSPR. The left-justified schedule, in which S_v is equal to the length of the longest path from s to v in N^σ, is optimal for σ.

Proof. Given some processing order, no operation can be started earlier than at its starting time in the left-justified schedule. Furthermore, no cost reduction can be obtained by processing an operation later than at its earliest possible starting time. Thus, the left-justified schedule is optimal. □

Note that it is possible that the first operation on a given machine, for which there are no operations on other machines that have to be performed before, still cannot start at its release time. For example, if there exists a maximum delay relation with some other operation that is to be performed later, this may force the operation to be started later than one would expect to be necessary at first glance.

Lemma 3.14 holds for all regular cost-functions, convex or non-convex. Let $U_v = 1$ if $S_v > d_v$, and $U_v = 0$ otherwise. Then, $\sum_{v \in O} U_v$, the number of late jobs, is a regular cost function, and, although U_v is certainly not convex in S_v, the left-justified schedule is optimal.

If the considered cost function is non-increasing in the starting times of the operations, the right-justified schedule, in which each operation is started as late as possible while still satisfying all constraints, is optimal. Also the right-justified schedule can be obtained by solving a longest path problem.

3.5.2 Non-regular cost functions

If the cost function is regular or tractable, a selection-optimal schedule can be obtained in polynomial time (Lemma 3.14 and Theorem 3.4). In case of general non-regular cost functions this is no longer true, as the following theorem shows.

Theorem 3.15 (*Minimizing the number of violated constraints*)
Let $A^* \subseteq A_1$ be an arbitrary subset of the set of start-start constraints. The problem of finding starting times S_v for all $v \in V$, corresponding to a selection σ, such that all constraints in $A_1 \setminus A^*$ are satisfied and the number of violated constraints in A^* is minimal is $\mathcal{NP}$-hard.

Proof. I will show that the problem is $\mathcal{NP}$-hard even if there are no disjunctive constraints. The proof uses a transformation from the vertex cover problem. Consider an undirected graph $G = (W, E)$, describing an instance of the vertex cover problem. From this graph, we obtain an instance of the scheduling problem in the following way. For each $w \in W$, we introduce two operations w_a and w_b, and start-start constraints $S_{w_b} - S_{w_a} \geq 1$ and $S_{w_a} - S_{w_b} \geq -2$. For each edge $\{v, w\} \in E$, we introduce start-start constraints $S_{w_b} - S_{v_a} \geq 3$ and $S_{v_b} - S_{w_a} \geq 3$. Each operation has a machine for itself, so there are no disjunctive constraints.

In Figure 3.3, a graph G and the network $N_1 = (V, A_1)$ of the corresponding scheduling problem are given.

Note that all arcs in N_1 are from a-operations to b-operations, except for those corresponding to start-start constraints of the form $S_{w_a} - S_{w_b} \geq -2$. The set of constraints of this form will be denoted by A^*. I claim that the graph G contains a

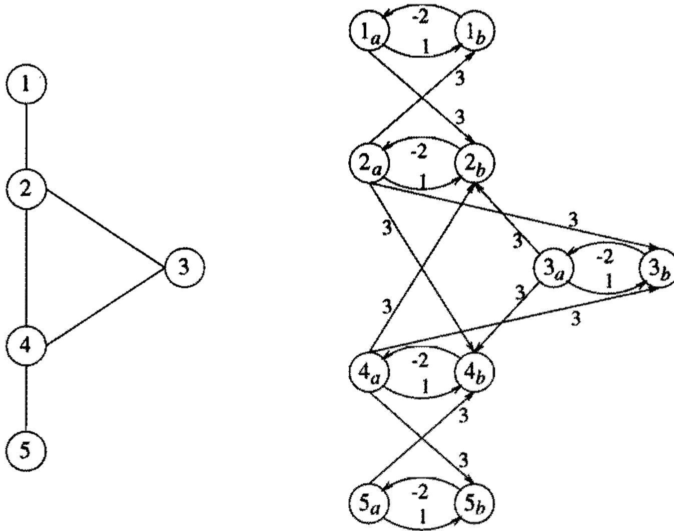

Figure 3.3: A graph G and the corresponding network N_1.

vertex cover of size smaller than or equal to k, with k some given positive integer, if and only if there exists a schedule for the corresponding scheduling problem in which at most k constraints in A^* are violated.

Suppose that a schedule with k violated constraints, $(w_b^1, w_a^1), \ldots, (w_b^k, w_a^k)$, exists. Then $C = \{w^1, \ldots, w^k\} \subset V$ is a vertex cover for G. If C is not a vertex cover, then there is an edge $\{u, v\} \in E$ with $u \neq w^i$, and $v \neq w^i$, for all $i \in \{1, \ldots, k\}$. In N_1, this corresponds to a cycle $(u_a, v_b, v_a, u_b, u_a)$ of positive length, which indicates that in any schedule at least one of the associated constraints must be violated. This contradicts the assumption that only the k given constraints were violated.

Now suppose that $C = \{w^1, \ldots, w^k\} \subset V$ is a vertex cover for G. Let $A_C = \bigcup_{i=1}^k (w_b^i, w_a^i)$. Then the network $N_C = (V, A_1 \setminus A_C)$ does not contain a cycle with positive length, and thus starting times S_v for all $v \in V$ which do not violate any of the constraints in $(A_1 \setminus A_C)$ can be obtained. Any cycle in N_C would alternately visit a-operations and b-operations. A cycle containing only two operations, (v_a, v_b, v_a) has length -1. Any larger cycle would contain an arc of the form (u_a, v_b) which indicates that in the original graph an edge $\{u, v\}$ exists. This implies that either $u = w^i$ or $v = w^i$, for some $i \in \{1, \ldots, k\}$. Then, not both (u_b, u_a) and (v_b, v_a) occur in N_C, which makes it impossible for

a cycle to contain (u_a, v_b). □

The cost function of the problem described in Theorem 3.15 is

$$\sum_{(v_b, v_a) \in A^*} g(S_{v_a} - S_{v_b}),$$

with $g(y) = 1$ if $y > 2$, and $g(y) = 0$ otherwise. This function is clearly not convex.

3.5.3 Multiple time windows

The start-start constraints $S_w - S_v \geq c_{vw}$ allow us to impose minimum and maximum values on the time lags between any two operations as well as on the starting times of individual operations. In some practical situations, however, other kinds of restrictions on the starting times of operations are encountered. The machines may, for example, not be available throughout the entire planning period. As long as for each machine μ a single availability interval $[t_1^\mu, t_2^\mu]$ is given, such a situation can be modeled by introducing two start-start constraints $S_v \geq T_1^{M_v}$ and $S_v \leq T_2^{M_v}$, with $T_1^{M_v} = \max_{\mu \in M_v} t_1^\mu$ and $T_2^{M_v} = \min_{\mu \in M_v} t_2^\mu$.

I conjecture that the problem of finding optimal starting times for a given processing order of the operations becomes $\mathcal{NP}$-hard when multiple time windows occur.

Conjecture 3.16 *(Multiple time windows)*
Consider GSPT, a variant of the general scheduling problem in which each operation must be processed in one of at most k given time intervals. For this problem, finding a selection-optimal schedule is $\mathcal{NP}$-hard.

If Conjecture 3.16 holds, then it is not sufficient to determine the processing order of the operations; one must also determine in which time intervals the operations must be performed. This decision is of a different nature than the ordering decisions and therefore disjunctive constraints cannot be used in modeling multiple time windows.

4

Solution methods

4.1 Introduction

In Chapter 3, I have shown that the main difficulty in the general scheduling problem arises from the disjunctive constraints. As soon as a feasible selection of the disjunctive constraints has been obtained, the corresponding optimal starting times can be found in polynomial time by solving a maximum cost flow problem. Furthermore, the situation in which no feasible starting times exist for a given selection can be detected easily; a selection is infeasible if and only if there exists a cycle of infinite capacity with positive cost in the corresponding solution network.

The problem of finding a good schedule has thus been reduced to the problem of finding a good selection of the disjunctive constraints. Although this is a substantial reduction when the size of the solution space is considered (there are infinitely many possible schedules, but there is only a finite number of possible selections), it does not help us too much from a complexity viewpoint. In fact, the general scheduling problem is $\mathcal{NP}$-hard, and therefore we may not hope to find an optimal selection in polynomial time. Of course, it is always possible to evaluate all possible selections or to develop branch-and-bound methods or other partial enumeration schemes. These methods, however, will require too much time, especially for application in an interactive planning system, and therefore other approaches must be considered.

In this chapter, I discuss two kinds of solution techniques, construction methods and local search algorithms. These two techniques can be combined in an algorithm that finds selections of reasonable quality in reasonable time. Initially, a construction algorithm is applied to obtain a feasible selection, and this selection is then used as a starting point for a local search algorithm. Unfortunately, even the problem of finding a feasible selection is already $\mathcal{NP}$-hard for the general

scheduling problem. However, in Section 4.2 I will identify a class of instances for which feasible selections can be obtained in polynomial time. In my discussion of construction and local search methods (Sections 4.3 and 4.4), I will focus on this class of problems. In Section 4.5, I will discuss two relaxation techniques for solving also the more difficult problems.

Many of the results presented in this chapter involve generalizations of well known concepts. The presentation is in terms of selections and solution networks rather than in terms of schedules. I hope to make clear that the concept of solution networks does not simply provide a *different* way of looking at solutions of scheduling problems: it gives also *more insight* in the structure of these solutions. This insight can be used in the design of better solution methods.

4.2 Complexity

4.2.1 Complexity of finding optimal selections

The general scheduling problem is $\mathcal{NP}$-hard. I could leave it at this observation, but I want to emphasize the hardness of the problem by mentioning a number of very special cases that are already $\mathcal{NP}$-hard. Lawler et al. [1993] give an extensive treatment of the subject. This section is largely based on their work, and I use their three-field classification scheme.

One subclass of the general scheduling problem is the class of single-machine scheduling problems. The following problems are already $\mathcal{NP}$-hard: $1|r_j|L_{\max}$, $1|prec|\sum C_j$, $1||\sum T_j$. A few results for single-machine scheduling are worth mentioning. $1||L_{\max}$ can be solved by applying the earliest due date (EDD) rule, which is due to Jackson [1955]. Lawler [1973] generalized this result to solve $1|prec|f_{\max}$ in $\mathcal{O}(n^2)$ time. Furthermore, Smith [1956] showed that $1||\sum w_j C_j$ is solved by putting the operations in order of non-decreasing ratios p_j/w_j.

In the area of single-machine scheduling with non-regular objective functions, there are hardly any positive results (see Baker and Scudder [1990]). The problem of minimizing the weighted sum of earlinesses and tardinesses is $\mathcal{NP}$-hard even if all due dates are assumed to be identical.

Another well studied subclass of the general scheduling problem is the class of shop scheduling problems. The following open shop problems are known to be $\mathcal{NP}$-hard: $O3||C_{\max}$, $O2||L_{\max}$, $O2||\sum C_j$. Solvable in polynomial time is only $O2||C_{\max}$ (Gonzales and Sahni [1976]). $\mathcal{NP}$-hard flow shop problems are $F3||C_{\max}$, $F2||L_{\max}$, and $F2||\sum C_j$. $F2||C_{\max}$ can be solved in $\mathcal{O}(n \log n)$ time by Johnson's algorithm [1954]. For job shop problems, the situation is even worse. $J3|n = 3|C_{\max}$ (number of jobs is at most 3) and $J2||C_{\max}$ are already

$\mathcal{NP}$-hard. Solvable in polynomial time are $J|n = 2|C_{max}$ (Akers [1956]) and $J2|n = k|C_{max}$ (Brucker [1994]). I am not aware of any work on shop scheduling with non-regular criteria.

4.2.2 Complexity of finding feasible selections

Consider the job shop scheduling problem in which all jobs must be completed before a given common deadline d^*. This problem has a feasible solution if and only if the corresponding standard job shop scheduling problem (without deadlines) has a solution with makespan at most d^*. Thus, since the standard job shop scheduling problem is $\mathcal{NP}$-hard, finding a feasible solution for the problem with deadlines is $\mathcal{NP}$-hard. This implies that finding a feasible solution for the general scheduling problem is $\mathcal{NP}$-hard. In contrast to the results presented in Section 4.2.1 it is possible to identify reasonably large subclasses of the general problem for which the feasibility problem can be solved in polynomial time. In the discussion of complexity issues, I will focus on the structure of the start-start constraints; the number of operations, the number of machines, the machine requirements, and the processing times are all assumed to be part of the problem instance.

Theorem 3.8 states that a selection σ is feasible if and only if the network $N_\infty^\sigma = (V, A_1 \cup A_3^\sigma)$ does not contain a cycle with positive cost. (Recall that N_∞ is the network obtained from N by removing all arcs with finite capacity.) This leads to the following observation.

Lemma 4.1 (*Infeasible problem*)
Consider an instance of the general scheduling problem. If the corresponding network $N_1 = (V, A_1)$ contains a cycle with positive cost, then the instance is infeasible.

Proof. Suppose that N_1 contains a cycle with positive cost. Then, for any selection σ, this cycle will also be in $N_\infty^\sigma = (V, A_1 \cup A_3^\sigma)$, and thus no feasible selection exists. □

A cycle with positive cost in N_1 occurs, for example, when there are conflicting precedence constraints (e.g., u before v, v before w, and w before u) or when some deadline cannot be reached (e.g., u must be processed before w and $p_u + p_w$ is larger than d_w, the deadline of w). The existence of a cycle with positive cost in a network can be checked in polynomial time by applying a longest path algorithm with an incorporated cycle detection mechanism.

If N_1 does not contain a cycle with positive cost, then a feasible selection may exist or not. If N_1 is acyclic, then the feasibility problem turns out to be easy. A

topological ordering of the nodes in a network $N = (V, A)$ is a function $\pi : V \to \{1, \ldots, |V|\}$ such that $\pi(v) \neq \pi(w)$ for all $v \neq w$, and $\pi(v) < \pi(w)$ for all $(v, w) \in A$. A network is acyclic if and only if it possesses a topological ordering of its nodes.

Lemma 4.2 (*Maintaining acyclicity*)
Let $N = (V, A)$ be an acyclic network, and let π be a topological ordering of the nodes in N. Any network N' that arises from N by adding arcs (u, v) such that $\pi(u) < \pi(v)$ is acyclic.

Proof. The proof is by induction. Define $A_\pi = \{(x, y) \mid \pi(x) < \pi(y)\}$. Let $\mathcal{N}^k, k \in \mathbb{Z}^+$, be the set of networks that can be formed by adding k arcs (u, v) from A_π to N.

Suppose that all networks in $\mathcal{N}^k$ are acyclic. Let $N^k = (V, A^k) \in \mathcal{N}^k$. If $N^{k+1} = (V, A^k \cup \{(x, y)\})$, with $(x, y) \in A_\pi$, contains a cycle, then this cycle must contain the arc (x, y), since N^k is acyclic. Let $C = (x, y, w_1, \ldots, w_l, x)$ be such a cycle. Then N^k must contain the path $O = (y, w_1, \ldots, w_l, x)$. All arcs (u, v) in A^k are such that $\pi(u) < \pi(v)$, and thus $\pi(x) > \pi(w_l) > \ldots > \pi(w_1) > \pi(y)$, which contradicts the assumption that $(x, y) \in A_\pi$. Thus N^{k+1} is acyclic. This holds for any $N^k \in \mathcal{N}^k$ and for any $(x, y) \in A_\pi$. Hence all networks in $\mathcal{N}^{k+1}$ are acyclic.

The induction assumption holds for $k = 0$. Hence all networks that are formed by adding arcs from A_π to N are acyclic. □

Lemma 4.2 is used in the proof of the following theorem.

Theorem 4.3 (*Feasible selections*)
Consider an instance of the general scheduling problem. If the corresponding network $N_1 = (V, A_1)$ is acyclic, then a feasible selection exists and one such feasible selection can be found in polynomial time.

Proof. A topological ordering π of the nodes in N_1 can be obtained in $\mathcal{O}(m_1)$ time, where $m_1 = |A_1|$. The arc set $A_3^\sigma = \{(v, w) \mid \{v, w\} \in A_3, \pi(v) < \pi(w)\}$ can be obtained in $\mathcal{O}(m_3)$ time, where $m_3 = |A_3|$. The network $N_\infty^\sigma = (V, A_1 \cup A_3^\sigma)$ is acyclic (Lemma 4.2). Thus, the selection σ is feasible and can be found in $\mathcal{O}(m_1 + m_3)$ time. □

Theorem 4.3 gives only a sufficient condition for an instance of the general scheduling problem to have a feasible selection. Problems that do not satisfy this condition do not necessarily have to be infeasible, but, as has been shown

for the job shop scheduling problem with deadlines, finding a feasible selection for such problems is in general $\mathcal{NP}$-hard. There are, however, also problems for which N_1 does contain a cycle (with non-positive cost) and for which feasible selections can be found in polynomial time. Consider, for example, the job shop problem with maximum delays. This problem is identical to the standard job shop scheduling problem, except for the maximum bounds on the delays between two consecutive operations of a job. The network N_1 for a typical instance of this problem is depicted in Figure 4.1. The maximum delay constraints correspond to the dotted arcs.

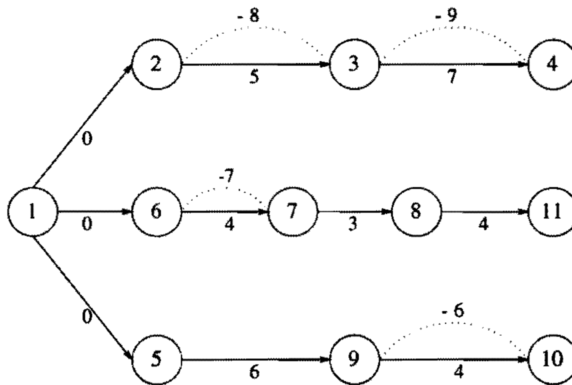

Figure 4.1: N_1 for a job shop problem with maximum delays.

Consider an arbitrary instance of the job shop problem with maximum delays. Let π be an ordering of the nodes in N_1 such that $\pi(v) < \pi(w)$ if (v, w) represents a normal precedence constraint, and $\pi(w) = \pi(v) + 1$ if (w, v) represents a maximum delay constraint. The numbering of the nodes in Figure 4.1 corresponds to such an ordering. The selection σ such that $(x, y) \in A_3^\sigma$ for all $\{x, y\} \in A_3$ with $\pi(x) < \pi(y)$ is feasible.

This example shows that there are also problems with cyclic N_1 for which the feasibility problem is easy. In the remainder of this thesis, however, I will maintain the distinction between problems with acyclic N_1, for which feasible selections can be found in polynomial time, and problems with cyclic N_1, for which the problem of finding a feasible selection is $\mathcal{NP}$-hard.

Let A_1 be the set of start-start constraints in an arbitrary instance of the general scheduling problem. Suppose that all constraints $(v, w) \in A_1$ are such that $c_{vw} \geq 0$. If the network (V, A_1) is acyclic, a feasible selection can be found in polynomial time. If the network contains a cycle, the instance is trivially in-

feasible. Thus, we can say that the constraints $(v, w) \in A_1$ such that $c_{vw} \geq 0$ (nonnegativity, precedence, and minimum delay constraints) are 'easy' from a feasibility point of view. Correspondingly, the constraints $(v, w) \in A_1$ such that $c_{vw} < 0$ (maximum delays and deadlines) are considered to be 'hard'. Such a distinction between easy and hard constraints is used in Section 4.5 in order to apply relaxation techniques to problems with cyclic N_1.

4.3 Construction methods

A straightforward way to obtain a selection would be to consider the disjunctive constraints $\{u, v\} \in A_3$ one after the other and decide for each constraint whether u is to be performed before v or vice versa. In this way, a selection (and the corresponding solution network) is constructed gradually.

In this section, I discuss two special kinds of construction methods that have received considerable attention in the scheduling literature. Both *dispatching* (or *list scheduling*) and *insertion* algorithms do not handle one disjunctive constraint after the other, but consider in each iteration a set of constraints associated with a particular operation. I show that both methods can be implemented in such a way that feasible selections are obtained for instances of the general scheduling problem that have acyclic N_1. Furthermore, I describe an efficient implementation of the insertion algorithm for a specific subclass of the general scheduling problem.

4.3.1 Dispatching

Dispatching is one of the earliest proposed solution methods for scheduling problems. It is very simple and very fast, and, probably for this reason, it is the technique that is most widely applied. In the original setting, a dispatching algorithm creates a left-justified schedule by selecting in each iteration an operation v that has not yet been scheduled and positioning it as early as possible after the scheduled operations on the machines in the machine set M_v. Here, I will discuss dispatching in terms of processing orders and solution networks.

Because of the general nature of the constraints and the objective functions, the selection-optimal starting time of an operation cannot be determined if only its predecessors are known. This does not imply, however, that it is impossible to use the idea of dispatching for the general scheduling problem. The only difference is that in each iteration not the starting time of an operation is determined but only its position in the processing order of the operations that require the same machines. At the beginning of each iteration, we have a set U of already

positioned operations and a set $\mathcal{O} \setminus U$ of operations that still have to be positioned. Then an operation $v \in \mathcal{O} \setminus U$ is selected, and it is positioned after the operations in U that require a machine in M_v. In the solution network, this corresponds to adding arcs (v, w) for all operations $w \in \mathcal{O} \setminus (U \cup \{v\})$ such that $\{v, w\} \in A_3$. Note that the arcs (u, v) for operations $u \in U$ such that $\{u, v\} \in A_3$ must have been added in previous iterations.

The general form of a dispatching algorithm is as follows.

```
procedure general dispatching
begin
   A_3^σ := ∅;
   U := ∅;
   repeat
      select an operation v from O \ U;
      A_3^σ := A_3^σ ∪ {(v, w) | w ∈ O \ (U ∪ {v}), {v, w} ∈ A_3};
      U := U ∪ {v};
   until U = O;
end.
```

Figure 4.2: The general dispatching algorithm.

For each pair of operations $\{v, w\} \in A_3$, it is decided whether v is performed before w (the arc (v, w) is in A_3^σ) or the other way around (the arc (w, v) is in A_3^σ). Thus, a dispatching algorithm of the above form will result in an arc set A_3^σ that corresponds to a selection. This selection σ, however, does not have to be feasible.

The most important implementation issue for dispatching algorithms concerns the selection of an operation in each iteration. After an operation has been selected, its position is uniquely determined. A dispatching algorithm can thus be characterized by the order in which the operations are selected. Many different criteria can be (and have been) applied in order to determine this *dispatching order*. The following theorem states that an optimal selection can always be obtained by some dispatching algorithm.

Theorem 4.4 (*Optimal dispatching*)
Let τ be an optimal selection of the disjunctive constraints of an instance of the general scheduling problem, and let A_3^τ be the corresponding arc set. Let π be an ordering of the set of operations such that $\pi_v < \pi_w$ if $(v, w) \in A_3^\tau$. When the operations are selected in order of increasing π-value, the general dispatching algorithm yields the selection τ.

Proof. As discussed before, the general dispatching algorithm always results in a selection. Let σ be this selection. We just have to show that $A_3^\sigma = A_3^\xi$. Suppose that there is a pair of operations $\{v, w\}$ such that $(v, w) \in A_3^\sigma$ and $(w, v) \in A_3^\xi$. Since $(w, v) \in A_3^\xi$, w is selected before v in the dispatching algorithm. Since v and w require a common machine, this implies that the arc (w, v) is added to A_3^σ, which is a contradiction. □

Theorem 4.4 does not help us much. We know that an optimal dispatching order exists, but it is by no means easy to find such an order. It is, however, possible to develop a polynomial-time dispatching algorithm that finds feasible selections for instances for which N_1 is acyclic. This result is a direct corollary of Theorem 4.3.

Corollary 4.5 (*Feasible dispatching*)
Consider an instance of the general scheduling problem for which N_1 is acyclic. Let π be a topological ordering of the nodes in N_1. Dispatching the operations in order of increasing π-value results in a feasible selection.

Recall the notion of reduced solution network introduced in Section 3.4.4. The general dispatching algorithm described above creates a complete solution network, and the reduced network can be obtained by deleting redundant arcs. It is, however, more efficient to immediately create the reduced solution network. The algorithm displayed in Figure 4.3 creates a reduced solution network corresponding to a feasible selection for instances of the general scheduling problem for which N_1 is acyclic.

Theorem 4.6 (*Dispatching*)
Consider an instance of the general scheduling problem for which N_1 is acyclic. The reduced solution network corresponding to a feasible selection can be obtained by applying the acyclic dispatching algorithm in $\mathcal{O}(m_1 + nq)$ time, where $m_1 = |A_1|$, $n = |\mathcal{O}|$, and $q = \max_{v \in \mathcal{O}} |M_v|$.

Proof. At any moment in the course of the algorithm, the set C of candidates for selection consists of operations for which all predecessor operations have already been scheduled. Thus, the dispatching order corresponds to a topological ordering, which guarantees for acyclic N_1 that a feasible selection is obtained.

The time complexity of the algorithm is $\mathcal{O}(m_1 + nq)$. The initialization of the δ-values requires $\mathcal{O}(m_1)$ time, and for each arc $(v, w) \in A_1$, with both v and $w \in \mathcal{O}$, one δ-value is updated once in the course of the algorithm, which again takes $\mathcal{O}(m_1)$ time in total. Furthermore, in each of the n iterations, an operation can be selected in constant time, and then $\mathcal{O}(q)$ arcs are added to A_3^σ. □

```
procedure acyclic dispatching
begin
  A_3^σ := ∅;
  U := ∅;
  for all μ ∈ M do
    l_μ := null;          (last operation on machine μ)
  for all v ∈ O do
    δ_v := number of arcs (u, v) ∈ A_1 such that u ∈ O;
  C := {v | δ_v = 0};  (candidates for dispatching)
  repeat
    select an operation v from C;
    for all μ ∈ M_v do
      begin
        if l_μ ≠ null then A_3^σ := A_3^σ ∪ {(l_μ, v)};
        l_μ := v;
      end;
    U := U ∪ {v};
    for all w ∈ O \ U do
      if (v, w) ∈ A_1 then
        begin
          δ_w := δ_w − 1;
          if δ_w = 0 then C := C ∪ {w};
        end;
  until U = O;
end.
```

Figure 4.3: The acyclic dispatching algorithm.

Note that also in the acyclic dispatching algorithm in each iteration an operation must be selected from a given set. This set C, however, is now such that feasibility is guaranteed no matter which operation is selected. Different selection criteria (priority rules) can be applied to obtain feasible solutions for different kinds of problems. Obviously, the choice of the priority rule may affect the time complexity of the algorithm.

Both the objective function and the structure of the constraints are of importance in deciding which priority rule can be expected to give selections of reason-

able quality. In general it is, however, not possible to decide beforehand which priority rule will perform best for a certain problem type. Extensive empirical studies have been performed for different kinds of scheduling problems (see, e.g., Panwalkar et al. [1977], and Haupt [1989]), but the main conclusion seems to be that it is very difficult to develop a priority rule that performs well for all instances of a given problem type. Considering the fact that the objective function of the general scheduling problem is much more general than any objective function considered in these studies on dispatching algorithms, the hope of finding a priority rule that performs well for all instances of the general problem must be very small.

4.3.2 Insertion

The outcome of a dispatching algorithm is completely determined by the dispatching order. In each iteration, the selected operation is positioned after the previously selected operations even if positioning that operation somewhere else might be much more favorable. This lack of flexibility is absent in insertion algorithms. In an insertion algorithm, again in each iteration an operation is selected, but its position relative to the previously selected operation is not predetermined.

Insertion algorithms have been proposed for several problem types. Nawaz et al. [1983] describe an insertion method for flow shop problems. Bräsel et al. [1993] and Werner and Winkler [1995] report good results for construction algorithms for open shop and job shop problems based on insertion techniques.

I will use the term *partial selection* to indicate a situation in which only those disjunctive constraints $\{u, v\} \in A_3$ have been settled for which both u and v are in some subset $U \subset \mathcal{O}$. The solution network in which only the corresponding arc set $A_3^\sigma(U)$ is included will be referred to as the *partial solution network* $N^\sigma(U)$. A partial selection is feasible if the corresponding partial solution network does not contain a cycle of infinite capacity with positive cost.

Let v be an operation that is to be inserted in a partial solution network $N^\sigma(U)$. Let $Q \subseteq U$ be the set of previously selected operations that require at least one of the machines in M_v. Inserting operation v involves determining its position relative to the operations in Q. For each operation in Q it must be determined whether it will be performed before v or after v. An insertion can thus be characterized by the subset $L \subseteq Q$ of operations that are to be performed before v. In the solution network, arcs (u, v) for all $u \in L$ and (v, w) for all $w \in Q \setminus L$ are added.

An insertion will be denoted by the pair (v, L). If $L = Q$, then insertion is identical to dispatching. Thus, insertion is a generalization of dispatching.

Example 4.1 Consider an instance of the general scheduling problem with $\mathcal{O} = \{1, \ldots, 9\}$, $V = \mathcal{O} \cup \{s, z_1, z_2\}$, and $\mathcal{M} = \{A,B,C,D\}$. In Figure 4.4, a partial solution network $N^{\sigma}(U)$ for this instance is depicted. The solid arcs represent arcs in A_1, the dotted arcs represent arcs in A_2, and the dashed arcs represent arcs in $A_3^{\sigma}(U)$. Here, $U = \{1, 2, 3, 5, 6, 7, 8\}$. The letters associated with the dashed arcs correspond to the machines for which the arcs indicate a processing order. For example, on machine B, the processing order is $(1, 2, 8)$. Operations 4 and 9 still have to be inserted. Observe that the partial selection σ for operations in U is feasible; there is no cycle in $N^{\sigma}(U)$ containing only arcs of infinite capacity. All cycles in $N^{\sigma}(U)$ contain at least one of the arcs in A_2.

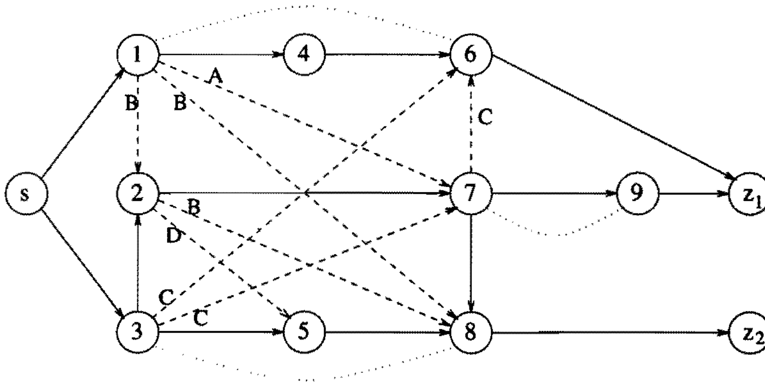

Figure 4.4: A partial solution network.

Suppose that operation 4, which requires the machine set {B,C}, is to be inserted. Then, $Q = \{1, 2, 3, 6, 7, 8\}$. A possible insertion is $(4, L)$, with $L = \{1, 2, 7\}$. Note that this insertion is infeasible, since the arc $(3, 7)$ together with the added arcs $(7, 4)$ and $(4, 3)$ forms a cycle of infinite capacity. The insertion $(4, L')$ with $L' = \{1, 2, 3, 7\}$ is feasible. □

The general form of an insertion algorithm is given in Figure 4.5.

When an insertion algorithm is developed, two major implementation issues have to be dealt with: which selection rule is used to determine the insertion order, and how must the set of operations be determined after which the selected operation is positioned?

The advantage of insertion as compared to dispatching is clear. After an operation has been selected, it is possible to look for a good position for that operation. In dispatching algorithms, if operation v is selected before operation w,

```
procedure general insertion
begin
  A_3^σ := ∅;
  U := ∅;
  repeat
    select an operation v from O \ U;
    Q := {u ∈ U | {u, v} ∈ A_3};
    determine a subset L ⊆ Q;
    A_3^σ := A_3^σ ∪ {(u, v) | u ∈ L} ∪ {(v, w) | w ∈ Q \ L};
    U := U ∪ {v};
  until U = O;
end.
```

Figure 4.5: The general insertion algorithm.

then v will be performed before w. If insertion is used, we can decide to perform w before or after v, whatever seems best at the moment that w is inserted. The insertion order is therefore less important for the quality of the selection obtained with an insertion algorithm than the dispatching order for the quality of a dispatching algorithm.

One would expect an insertion algorithm to give better selections than a dispatching algorithm, and in general this is true, although at some cost. A good insertion algorithm is more time consuming both in implementation and in running time. I will show that it is possible to develop an insertion algorithm that is guaranteed to give feasible selections for the same class of problems for which a feasible dispatching algorithm can be applied, and that still offers a lot of freedom for choosing the position. How this freedom can be used strongly depends on the specific structure of the problem and the objective function. In Section 4.3.3, an insertion algorithm for a problem with makespan (or maximum lateness) minimization as objective is described. This algorithm is shown to perform better than any dispatching algorithm, while still reasonably little time is required.

In this section, I will only discuss feasibility aspects of insertion algorithms. Suppose that all operations in some subset $U \subset \mathcal{O}$ have already been selected and inserted. Let $A_3^\sigma(U)$ be the corresponding set of added arcs. I assume that the network $N_\infty^\sigma(U) = (V, A_1 \cup A_3^\sigma(U))$ is acyclic. Suppose that operation v is now selected and has to be inserted. I will show that a feasible insertion of v, i.e., an insertion such that $N_\infty^\sigma(U \cup \{v\})$ is acyclic, exists and can be found in

polynomial time. As a consequence, we can develop a polynomial-time insertion algorithm for instances with acyclic N_1 that gives a feasible selection for any insertion order.

Lemma 4.7 (*Feasible insertion*)
Let $N = (V, A)$ be an acyclic network. For any $v \in V$ and any $Q \subset V$, there exists a subset $L \subseteq Q$ such that the network N^L that arises from N by adding arcs (u, v) for all $u \in L$ and arcs (v, w) for all $w \in Q \setminus L$ is acyclic.

Proof. Let v be an arbitrary element of V, and let Q be an arbitrary subset of V. Suppose that the network N^L corresponding to a particular $L \subseteq Q$ contains a cycle. Since N is acyclic, this cycle must contain at least one of the added arcs (u, v) for some $u \in L$, or (v, w) for some $w \in Q \setminus L$. There are three possible situations:

1. N contains a path from v to some $u \in L$.
2. N contains a path from some $w \in Q \setminus L$ to v.
3. N contains a path from some $w \in Q \setminus L$ to some $u \in L$.

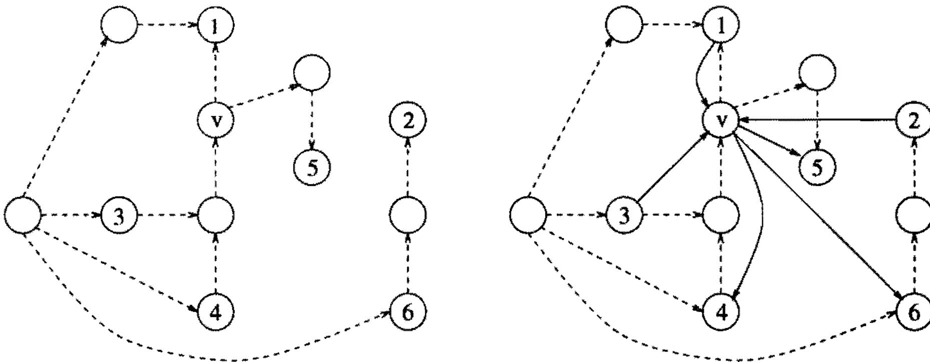

Figure 4.6: An acyclic network and three cycles after insertion.

In Figure 4.6, the relevant part of an acyclic network is depicted. Operation v is to be inserted, and $Q = \{1, 2, 3, 4, 5, 6\}$. The insertion (v, L) with $L = \{1, 2, 3\}$ results in a network with three cycles corresponding to the three situations described above (situation 1: $u = 1$; situation 2: $w = 4$; situation 3: $u = 2, w = 6$).

If we can show that it is possible to choose L in such a way that none of the three situations occurs, then the proof is complete.

Let $Q_1 \subseteq Q$ be the set of nodes u for which a path from u to v in N exists. Let $Q_2 \subseteq Q$ be the set of nodes x for which there is no path from x to v and no path from v to x in N. Let Q_3 be the set of nodes w for which a path from v to w in N exists. In the example of Figure 4.6, $Q_1 = \{3, 4\}$, $Q_2 = \{2, 6\}$, and $Q_3 = \{1, 5\}$. Observe that Q_1, Q_2, and Q_3 are pairwise disjoint and form a partition of Q.

Any L for which none of the three situations holds must be such that $Q_1 \subseteq L$ and $Q_3 \subseteq Q \setminus L$. Therefore, we only have to assign the elements in Q_2 to L and $Q \setminus L$ in such a way that no cycle is created. Situations 1 and 2 cannot occur in any assignment of Q_2 to L and $Q \setminus L$ because of the definition of Q_2. Suppose that situation 3 occurs. That is, there are a $u \in L$ and a $w \in Q \setminus L$ such that a path from w to u in N exists. Note that u is not in Q_3 and w is not in Q_1. There are again three possibilities:

1. $u \in Q_1$, $w \in Q_2 \cup Q_3$: the existence of a path from w to u in N together with the fact that $u \in Q_1$ implies that w must be in Q_1, which is a contradiction.
2. $u \in Q_2$, $w \in Q_3$: the existence of a path from w to u in N together with the fact that $w \in Q_3$ implies that u must be in Q_3, which is a contradiction.
3. $u \in Q_2$, $w \in Q_2$: this situation may occur.

Thus, a cycle in N^L can only occur if a path in N from $w \in Q_2$ to $u \in Q_2$ exists, and u is assigned to L and w to $Q \setminus L$.

Since N is acyclic, it is possible to obtain an ordering ρ of the operations in Q_2 such that $\rho(w) < \rho(u)$ if a path from w to u in N exists. For any $r \in \mathbb{N}$, the assignment of the elements in Q_2 to L and $Q \setminus L$ such that $u \in L$ if $\rho(u) \leq r$ and $u \in Q \setminus L$ if $\rho(u) > r$ results in a feasible insertion (v, L). □

Theorem 3.8 states that only arcs with infinite capacity are relevant in determining the feasibility of a selection. Let σ be a partial selection for the operations in $U \subset \mathcal{O}$. From Lemma 4.7 it follows that if $N^\sigma(U)$ does not contain a cycle of infinite capacity, then any operation can be inserted in such a way that the resulting solution network again does not contain a cycle of infinite capacity. Since N_1 is assumed to be acyclic, it follows that, independent of the order in which operations are inserted, a feasible selection can always be obtained by inserting the operations one after another.

The insertion algorithm given in Figure 4.7 creates a reduced solution network corresponding to a feasible selection for instances of the general scheduling problem for which N_1 is acyclic. Recall from Section 3.4.4 that the machine

```
procedure acyclic insertion
begin
  A_3^σ := ∅;
  U := ∅;
  for all μ ∈ M do
    O_μ := (s_μ, t_μ);
  repeat
    select an operation v from O \ U;
    for all μ ∈ M_v do
      begin
        x̄ := last x in O_μ such that ∃ path from x to v in (V, A_1 ∪ A_3^σ);
        ȳ := first y in O_μ such that ∃ path from v to y in (V, A_1 ∪ A_3^σ);
        select two consecutive operations (u, w) from (x̄, ..., ȳ) ⊂ O_μ;
        A_3^σ := A_3^σ ∪ {(u, v), (v, w)} \ {(u, w)};
        insert v between u and w in O_μ;
      end;
    U := U ∪ {v};
  until U = O;
end.
```

Figure 4.7: The acyclic insertion algorithm.

ordering $O_\mu = (v^1, \ldots, v^l)$ represents the order in which operations are performed on machine μ. Such a machine ordering corresponds to a path in the reduced solution network. In the described algorithm dummy operations s_μ and t_μ are used to mark the beginning and end of a machine ordering. Initially, O_μ is set to (s_μ, t_μ) for all $\mu \in \mathcal{M}$.

Theorem 4.8 (*Insertion*)
Consider an instance of the general scheduling problem for which N_1 is acyclic. By applying the acyclic insertion algorithm, the reduced solution network corresponding to a feasible selection can be obtained in $\mathcal{O}(nq(m_1 + nq))$ time, where $n = |\mathcal{O}|$, $q = \max_{v \in \mathcal{O}} |M_v|$, and $m_1 = |A_1|$.

Proof. In each iteration, an operation v is selected and inserted. First, a feasible position for v on the first machine of its machine set is determined and the corresponding arcs are added to the solution network. From the proof of Lemma 4.7 it follows that any position between x and y on this machine is feasible. The section between x and y in the machine ordering O_μ corresponds to the set Q_2 and

the ordering ρ of elements in Q_2 is obtained directly from O_μ. The procedure is then repeated for the other machines in M_v.

The time complexity of the acyclic insertion algorithm is $\mathcal{O}(nq(m_1 + nq))$. There are n iterations. In each iteration, for all $\mathcal{O}(q)$ machines in the machine set M_v the operations between which an operation v can be inserted must be determined. For this purpose, the nodes w for which no path from w to v exists and the nodes u for which no path from v to u in $(V, A_1 \cup A_3^\sigma)$ exists must be identified. This can be done by applying any longest (or shortest) path algorithm. Such an algorithm requires $\mathcal{O}(m_1 + nq)$ time, since the considered networks are acyclic, $|A_1| = m_1$, and $|A_3^\sigma|$ is at most nq in a reduced solution network. □

Example 4.2 Consider again the situation in Example 4.1. Instead of dealing with the solution network depicted in Figure 4.4, we will use the reduced solution network. In the reduced network, the arcs (1,8) and (3,6) do not occur.

Operation 4 is to be inserted on machines B and C. $O_B = (1, 2, 8)$. There is a path from 1 to 4, and there is no path from 2 to 4 that does not use an arc in A_2. Hence, $x = 1$. Furthermore, since there is no infinite capacity path from 4 to any node in O_B, we have $y = t_B$. This corresponds to three feasible insertion positions for operation 4 on machine B: between 1 and 2, between 2 and 8, and after 8. Suppose that we decide to insert 4 between 2 and 8. Then the arc (2,8) is removed, and arcs (2,4) and (4,8) are added. Now, consider machine C. We have $O_C = (3, 7, 6)$. There is a path from 3 to 4, since the arc (2, 4) has been added, and a path from 4 to 6. Hence, $x = 3$, $y = 6$. The corresponding possible insertion positions are between 3 and 7 and between 7 and 6. The solution network that arises when the first possibility is selected is given in Figure 4.8. □

As the example shows, there may be many feasible positions for an operation to be inserted in. Let ν be an upper bound on the number of operations that are already positioned on a machine. In general, there are $\mathcal{O}(\nu)$ possible insertion positions for an operation v on a machine μ. The number of machines on which v is to be inserted is $\mathcal{O}(q)$. Thus, the total number of possible insertion positions is $\mathcal{O}(\nu^q)$. Finding the best insertion by evaluating all possible insertions requires $\mathcal{O}(\nu^q T)$ time, where T is the time required to evaluate a (partial) selection. In case of large machine sets, this time requirement will be prohibitively large, and it may be wise to use some heuristic to determine good insertion positions.

In the next section, I will show for a specific subclass of the general scheduling problem that it is possible to find the best insertion position in time polynomial in q.

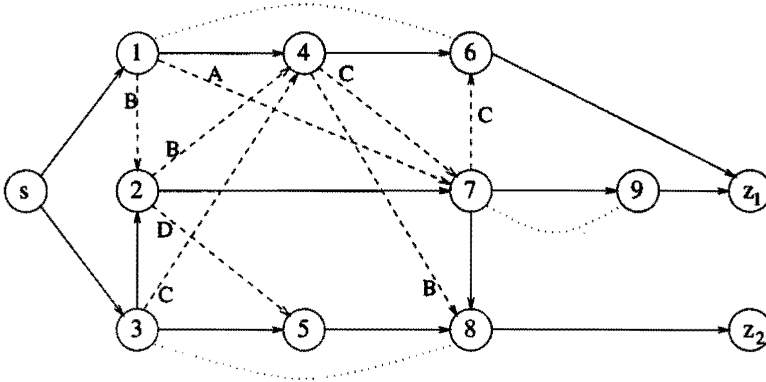

Figure 4.8: The reduced solution network after insertion of operation 4.

4.3.3 An efficient insertion algorithm

In this section, I consider the following special case of the general scheduling problem:

$$
\begin{array}{lll}
\min & S_t & \\
\text{s.t.} & S_w - S_v \geq c_{vw}\,, & \forall (v,w) \in A_1 \\
& S_w \geq S_v + p_v \vee S_v \geq S_w + p_w\,, & \forall \{v,w\} \in A_3 \\
& S_s = 0, &
\end{array}
$$

with $c_{vw} \geq p_v > 0, \forall (v, w) \in A_1, w \neq t$. A_1 does not contain any arcs (t, v), and the network $N_1 = (V, A_1)$ is acyclic, with $V = \mathcal{O} \cup \{s, t\}$.

All well known shop scheduling problems in which the objective is to minimize the makespan or the maximum lateness can be formulated in this way, as well as variants of these problems in which each operation requires a machine set rather than a single machine.

For this special case, an optimal flow in a solution network N^σ corresponds to a longest path from s to t in N^σ (see Section 3.2). I assume that A_1 is such that there is a path from s to u and a path from u to t in $N_1 = (V, A_1)$, for each $u \in \mathcal{O}$. This is not a strong assumption, since constraints $S_u - S_s \geq -K$ and $S_t - S_u \geq -K$, for some large K, can be added.

Note that all arcs have infinite capacity and, except for possibly the arcs (v, t), positive cost. Any cycle in a solution network therefore indicates that the corresponding selection is infeasible.

When the general insertion algorithm was introduced, it was mentioned that the most important implementation issue was concerned with determining the set of operations after which v should be inserted. In the algorithm this corresponded to the step 'determine a subset $L \subseteq Q$'. In this section, I will present a method to find the best subset $L \subseteq Q$, or, equivalently, the best insertion of a given operation $v \in \mathcal{O} \setminus U$ in a given partial solution network $N^\sigma(U)$. Here, an insertion is considered the best if it results in an acyclic solution network $N^\sigma(U \cup \{v\})$ in which the longest path from s to t is as short as possible.

As mentioned in Section 4.3.2, the number of possible insertions is exponential in the size of the machine set M_v. I will show that we only need to consider a set of $\mathcal{O}(n)$ insertions, which is guaranteed to contain an optimal one. In the presented approach, the position in which v is to be inserted is not determined for each machine separately, as in the acyclic insertion algorithm. Instead, all operations that use at least one of the machines in M_v are considered simultaneously, as in the general insertion algorithm.

First, I will describe a procedure for finding an $L^* \subseteq Q$ such that the insertion (v, L^*) results in a solution network in which the longest path from s to t is as short as possible. Afterwards, I will show that (v, L^*) is a feasible insertion, by showing that the resulting solution network is acyclic.

Let $N = N^\sigma(U)$ be the acyclic partial solution network corresponding to the selection σ for the operations in U, and let N^L be the network that arises from N by adding arcs (u, v) for all $u \in L$, and arcs (v, w) for all $w \in Q \setminus L$, for some $L \subseteq Q$. For the time being, I assume that N^L is acyclic, i.e., the insertion (v, L) is feasible. Let d_{uw} and d^L_{uw} be the lengths of the longest paths from node u to node w in N and N^L respectively. Note that d_{su} and d_{ut} are well defined for all $u \in \mathcal{O}$, even if u has not yet been inserted, since N_1 is assumed to contain paths from s to u and from u to t for all $u \in \mathcal{O}$.

Since N^L arises from N by only adding arcs incident to v, leaving the rest of N unchanged, we have

$$d^L_{st} = \max\{d_{st}, d^L_{sv} + d^L_{vt}\}. \tag{4.1}$$

Furthermore,

$$d^L_{sv} = \max\{d_{sv}, \max_{u \in L}(d_{su} + p_u)\} = d_{sv} + \max_{u \in L}(d_{su} + p_u - d_{sv})^+, \tag{4.2}$$

and

$$d^L_{vt} = \max\{d_{vt}, \max_{w \in Q \setminus L}(p_v + d_{wt})\} = d_{vt} + \max_{w \in Q \setminus L}(p_v + d_{wt} - d_{vt})^+, \tag{4.3}$$

where $x^+ = \max\{x, 0\}$. Substituting (4.2) and (4.3) in (4.1), we get

$$d^L_{st} = \max\{d_{st}, d_{sv} + \max_{u \in L}(d_{su} + p_u - d_{sv})^+ + d_{vt} + \max_{w \in Q \setminus L}(p_v + d_{wt} - d_{vt})^+\}$$

The problem is to find an $L \subseteq Q$ for which d_{st}^L is minimal. Let $a_u = d_{su} + p_u - d_{sv}$, $b_u = p_v + d_{ut} - d_{vt}$. Then a_u^+ is the increase of the length of the longest path from s to v if v is positioned after u, and b_u^+ is the increase of the length of the longest path from v to t if v is positioned before u. Let $C(L)$ be the increase in the length of the longest path from s to t through v when the insertion (v, L) is performed. That is, $C(L) = d_{st}^L - (d_{sv} + d_{vt})$. We are interested in finding L^* such that

$$C(L^*) = \min_{L \subseteq Q} C(L) = \min_{L \subseteq Q}(\max_{u \in L} a_u^+ + \max_{w \in Q \setminus L} b_w^+).$$

Let u be an arbitrary operation in Q. We must assign u to either L or $Q \setminus L$. Three situations can be distinguished:

- $a_u \leq 0$. For any $L, u \notin L$, we have $C(L \cup \{u\}) \leq C(L)$, and u can be assigned to L^* immediately.
- $a_u > 0, b_u \leq 0$. For any $L, u \notin L$, we have $C(L \cup \{u\}) \geq C(L)$, and u can be assigned to $Q \setminus L^*$ immediately.
- $a_u > 0$ and $b_u > 0$. Now we cannot conclude beforehand for a given L with $u \notin L$ whether $C(L \cup \{u\})$ or $C(L)$ has the smallest value.

Thus, in looking for an optimal L, only the operations u with $a_u > 0$ and $b_u > 0$ still have to be assigned. The following lemma states that an optimal assignment can be obtained efficiently.

Lemma 4.9 (*Optimal partition*)
Given a set W, and associated with each $w \in W$ two positive numbers a_w, b_w. Consider the problem of finding a partition (W_1, W_2) of W such that $C(W_1) = \max_{w \in W_1} a_w + \max_{w \in W_2} b_w$ is minimal. There exists an optimal partition of the form $W_1 = \{w \in W \mid a_w < k\}$, $W_2 = \{w \in W \mid a_w \geq k\}$, for some $k \in \mathbb{N}$, and it can be found in $\mathcal{O}(n \log n)$ time, where $n = |W|$.

Proof. Let $u, w \in W$ be such that $a_u \leq a_w$ and $b_u \leq b_w$; w is said to *dominate* u. If $a_u = a_w$ and $b_u = b_w$, some tie-breaker is used to determine which element dominates the other, for example the one with lowest index. Suppose that $w \in W_1$ and $u \in W_2$. Then $C(W_1 \cup \{u\}) \leq C(W_1)$. Similarly, if $w \in W_2$ and $u \in W_1$, then $C(W_1 \setminus \{u\}) \leq C(W_1)$. Thus, we only have to consider those partitions (W_1, W_2) of W for which both u and w are in W_1 or in W_2. After having decided whether the dominating element is assigned to W_1 or to W_2, we assign the dominated element to the same set.

Now consider $W' \subseteq W$, the set of undominated elements. For any $X \subseteq W'$, let $\bar{X} \subseteq W$ be the set consisting of elements in X and elements that are dominated by some element in X. Note that $C(\bar{X}) = \max_{w \in X} a_w + \max_{w \in W' \setminus X} b_w$.

If we sort the elements of W' in order of increasing a_w, we get $a_{\pi(1)} < a_{\pi(2)} < \ldots < a_{\pi(|W'|)}$ and $b_{\pi(1)} > b_{\pi(2)} > \ldots > b_{\pi(|W'|)}$. Let X be an arbitrary subset of W', and let $i \in \{0, \ldots, |W'|\}$ be the largest index such that $\pi(i) \in X$, with $i = 0$ if $X = \emptyset$. Let $T^i = \{\pi(j) \mid j \leq i\} \subseteq W'$. Since

$$\max_{w \in T^i} a_w = \max_{w \in X} a_w, \text{ and } \max_{w \in W' \setminus T^i} b_w \leq \max_{w \in W' \setminus X} b_w,$$

we have

$$C(\bar{T}^i) \leq C(\bar{X}). \tag{4.4}$$

It follows from (4.4) that there exists an i such that $C(\bar{T}^i) \leq C(W_1)$, for any $W_1 \subseteq W$. Hence, there exists an optimal partition of the form $(\bar{T}^i, W \setminus \bar{T}^i)$. Note that

$$C(\bar{T}^i) = a_{\pi(i)} + b_{\pi(i+1)},$$

with $a_{\pi(0)} = b_{\pi(|W'|+1)} = 0$. Let $i_{\text{opt}} \in \operatorname{argmin}_{i \in \{0,\ldots,|W'|\}} a_{\pi(i)} + b_{\pi(i+1)}$. Then $(\bar{T}^{i_{\text{opt}}}, W \setminus \bar{T}^{i_{\text{opt}}})$ is an optimal partition.

The total time required to obtain this optimal partition is $\mathcal{O}(n \log n)$. Instead of first removing the dominated elements from W and then sorting the remaining elements, we first sort the elements of W in order of non-decreasing a-value, which can be done in $\mathcal{O}(n \log n)$ time, and then remove the dominated elements in the following way.

Let $a_{\rho(1)} \leq a_{\rho(2)} \leq \ldots \leq a_{\rho(|W|)}$. In the first iteration of the procedure we set $\pi(1) = \rho(1)$, and $l = 1$. At the beginning of each following iteration, we have a list of operations with $a_{\pi(i)} < a_{\pi(i+1)}$ and $b_{\pi(i)} > b_{\pi(i+1)}$, for $i = 1, \ldots, l-1$, where $l < |W|$ is the number of operations in the list, and $a_{\pi(0)} = 0$, $b_{\pi(l+1)} = 0$. In the jth iteration, we consider the operation $\rho(j)$.

If $b_{\rho(j)} \geq b_{\pi(l)}$, then we remove all operations that are dominated by $\rho(j)$: let i be the smallest index such that $b_{\pi(i)} \leq b_{\rho(j)}$; we set $l = i$ and $\pi(l) = \rho(j)$.

If $b_{\rho(j)} < b_{\pi(l)}$ and $a_{\rho(j)} > a_{\pi(l)}$, then we add $\rho(j)$ to the list by setting $\pi(l+1) = \rho(j)$ and $l = l + 1$.

If $b_{\rho(j)} < b_{\pi(l)}$ and $a_{\rho(j)} = a_{\pi(l)}$, then $\rho(j)$ is dominated by $\pi(l)$ and is therefore not added to the list.

After $|W|$ iterations, the procedure terminates with the list of undominated nodes sorted in order of increasing a-value. Each operation is added at most once to the list and deleted at most once from the list, and therefore the procedure takes $\mathcal{O}(n)$ time.

Finding an i_{opt} for which $a_{\pi(i_{\text{opt}})} + b_{\pi(i_{\text{opt}}+1)}$ is minimal and adding the dominated operations to obtain $\bar{T}^{i_{\text{opt}}}$ both take $\mathcal{O}(n)$ time. Hence, the optimal partition can be found in $\mathcal{O}(n \log n)$ time. □

Example 4.3 Consider the following instance of the partition problem. The elements are already sorted in order of non-decreasing a-value.

w	1	2	3	4	5	6	7	8	9	10
a_w	3	4	6	7	8	9	9	11	12	16
b_w	16	11	12	10	10	6	8	5	2	4

Note that 2, 4, 6, and 9 are dominated by, respectively, 3, 5, 7, and 10.

i	0	1	2	3	4	5	6	7
$\pi(i)$		1	3	5	7	8	10	
$a_{\pi(i)}$	0	3	6	8	9	11	16	
$b_{\pi(i)}$		16	12	10	8	5	4	0
$a_{\pi(i)} + b_{\pi(i+1)}$	16	15	16	16	14	15	16	

$C(\bar{T}^i)$ is minimal for $i = 4$, and $(\{1,2,3,4,5,6,7\}, \{8,9,10\})$ is the corresponding optimal partition with value 14. □

For any element $v \in \mathcal{O} \setminus U$, an insertion (v, L^*) in the partial solution network $N^\sigma(U)$ can be found such that the length of the longest path from s to t in the resulting solution network is as small as possible. It remains to be shown that the insertion (v, L^*) is feasible.

Lemma 4.10 (*Feasible insertion*)
Let $N^\sigma(U)$ be an acyclic partial solution network, and let Q be the set of operations u in U such that $\{u, v\} \in A_3$. Let $L^k = \{w \in Q \mid a_w \leq 0 \vee (a_w < k, b_w > 0)\}$. The insertion (v, L^k) is feasible, for any $k \in \mathbb{N}$.

Proof. Recall, from the proof of Lemma 4.7, the partition (Q_1, Q_2, Q_3) of Q. Q_1 is the set of operations u in Q for which a path from u to v in $N^\sigma(U)$ exists, Q_2 is the set of operations u for which no paths from u to v and from v to u in $N^\sigma(U)$ exist, and Q_3 is the set of operations u for which a path from v to u in $N^\sigma(U)$ exists. It has been shown that the insertion (v, L) is feasible if $L = Q_1 \cup \{u \in Q_2 \mid \rho(u) < r\}$, for some $r \in \mathbb{N}$ and some ordering ρ of the operations in Q_2 such that $\rho(w) < \rho(u)$ if a path from w to u in $N^\sigma(U)$ exists. In other words, L is feasible if $Q_1 \subseteq L$, $Q_3 \cap L = \emptyset$, and there are no $u, w \in Q_2$, such that a path from w to u in $N^\sigma(U)$ exists and $u \in L$, $w \in Q \setminus L$.

Let $u \in Q_1$. Since there exists a path from u to v, we have $d_{sv} \geq d_{su} + p_u$, and thus $a_u = d_{su} + p_u - d_{sv} \leq 0$. Therefore, $Q_1 \subseteq L^k$.

Let $w \in Q_3$. Since there exists a path from v to w, we have $d_{sw} \geq d_{sv} + d_{vu}$, and thus $a_w = d_{sw} + p_w - d_{sv} \geq d_{vu} + p_w > 0$. Furthermore, $d_{vt} \geq d_{vw} + d_{wt}$, and thus $b_w = p_v + d_{wt} - d_{vt} \leq p_v - d_{vw} \leq 0$. Therefore, $Q_3 \cap L^k = \emptyset$.

Let $u, w \in Q_2$ be such that a path from w to u in $N^\sigma(U)$ exists. Then, $a_u = d_{su}+p_u-d_{sv} \geq d_{sw}+d_{wu}+p_u-d_{sv} \geq d_{sw}+p_w+p_u-d_{sv} > a_w$. Furthermore, $b_w = p_v + d_{wt} - d_{vt} \geq p_v + d_{wu} + d_{ut} - d_{vt} > b_u$. Suppose that $u \in L^k$. Then, $a_u \leq 0$ or $a_u < k, b_u > 0$. In the first case, also $a_w \leq a_u \leq 0$, and $w \in L^k$. In the second case, $a_w < a_u < k$, and $b_w > b_u > 0$, and $w \in L^k$.

This completes the proof. □

In Figure 4.9, the algorithm for finding an optimal insertion of an operation v in an acyclic partial solution network $N^\sigma(U)$ is given.

```
procedure best insertion
begin
  Q := {u ∈ U | {u, v} ∈ A_3};                                   (1)
  for all u ∈ Q ∪ {v} do                                         (2)
    compute d_su and d_ut;
  for all u ∈ Q do                                               (3)
    begin
      compute a_u = d_su + p_u − d_sv and b_u = d_ut + p_u − d_vt;
      if a_u ≤ 0 then L := L ∪ {u}; Q := Q \ {u};
      if (a_u > 0, b_u ≤ 0) then Q := Q \ {u};
    end;
  sort the operations in Q in order of non-decreasing a_u;      (4)
  Q' := {u ∈ Q | u not dominated by any w ∈ Q};                 (5)
              (a_π(1) ≤ ... ≤ a_π(|Q'|))
  a_π(0) := 0; b_π(|Q'|+1) := 0;
  i_opt :=argmin_{i∈{0,...,|Q'|}} a_π(i) + b_π(i+1);            (6)
  for all u ∈ Q do                                               (7)
    if a_u ≤ a_π(i_opt) then L := L ∪ {u};
end;
```

Figure 4.9: The best insertion algorithm.

Theorem 4.11 (*Best insertion*)
The best insertion algorithm finds an optimal insertion (v, L) of operation v in a reduced partial solution network $N^{\sigma}(U)$ in $\mathcal{O}(m_1+nq)$ time, where $m_1 = |A_1|$, $n = |\mathcal{O}|$, and $q = \max_{v\in\mathcal{O}} |M_v|$.

Proof. The obtained insertion is such that $w \in L$ if $a_w \leq 0$, and $w \in Q \setminus L$ if $a_w > 0$ and $b_w \leq 0$ (step 3). The remaining operations are assigned to L and $Q \setminus L$ in an optimal way (Lemma 4.9) in steps 4 to 7. From Lemma 4.10 follows that (v, L) is feasible.

The time required to determine the set Q in step 1 depends on the data structure that is used, but it will not exceed $\mathcal{O}(qn)$, which is the time required to find for each machine in M_v the operations that are performed on that machine.

Step 2 of the algorithm requires $\mathcal{O}(m_1+nq)$ time. Since the solution network is acyclic, a so-called reaching algorithm can be used to compute the longest paths from one node to all the other nodes in the network (see, e.g., Ahuja et al. [1993]). The time required by a reaching algorithm is linear in the number of arcs in the network. The arc set in the reduced solution network consists of m_1 arcs in A_1, and $\mathcal{O}(nq)$ arcs in $A_3^{\sigma}(U)$. The reaching algorithm is applied twice, once for computing d_{su} and once for computing d_{ut}.

In step 3, for each operation $u \in Q$ some elementary operations are performed. These take $\mathcal{O}(n)$ time.

Sorting the operations of Q in step 4 requires $\mathcal{O}(n \log q)$ time. On each machine, the operations are already in the correct order. All that is required is to merge these $\mathcal{O}(q)$ sorted lists. If the information about the processing order on the various machines would not be used, then step 4 would require $\mathcal{O}(n \log n)$ time.

Removing the dominated operations from Q in step 5 requires $\mathcal{O}(n)$ time, as has been shown in the proof of Lemma 4.9.

Finding i_{opt} in step 6 and adding the dominated operations in step 7 both require $\mathcal{O}(n)$ time.

Thus, the total time required is $\mathcal{O}(m_1 + nq)$. □

Suppose that A_1 represents tree-like precedence constraints, and that there is a fixed upper bound on the size of the machine sets. This is true for job shop problems ($q = 1$) and open shop problems ($q = 2$). Then $m_1 = \mathcal{O}(n)$, and also the running time of the best insertion algorithm is $\mathcal{O}(n)$.

Example 4.4 Consider the partial solution network given in Figure 4.10. The number associated with a node v represents the processing time p_v of operation v. The length of an arc (u, v) is equal to p_u. The objective is to minimize the makespan S_t.

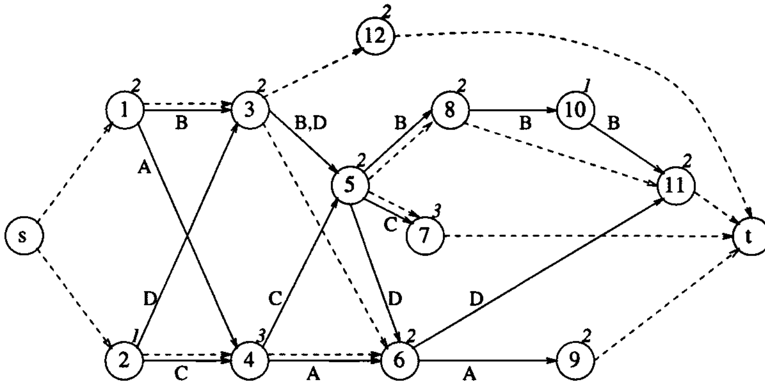

Figure 4.10: The partial solution network.

We can associate a (partial) left-justified schedule with this solution network. The makespan of this schedule is $d_{st} = 12$. The schedule is depicted in Figure 4.11.

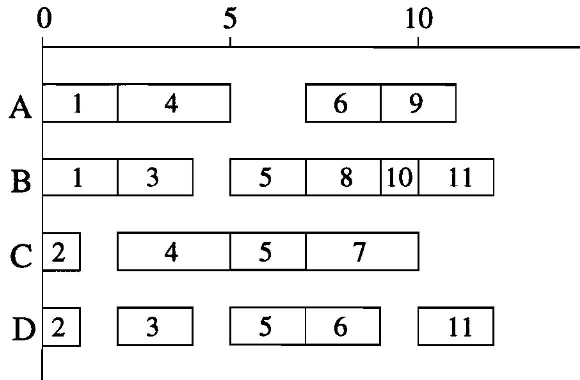

Figure 4.11: The (partial) left-justified schedule.

We want to find the best insertion of operation 12 on the machine set {A,D}. Applying the best insertion algorithm, we find:

(1) $Q = \{1, 2, 3, 4, 5, 6, 9, 11\}$

(2)

u	1	2	3	4	5	6	9	11	12
d_{su}	0	0	2	2	5	7	9	10	4
d_{ut}	12	11	9	10	7	4	2	2	2

(3)

u	1	2	3	4	5	6	9	11
a_u	-2	-3	0	1	3	5	7	8
b_u	12	11	9	10	7	4	2	2

Note that $a_u \leq 0$ for $u = 1, 2, 3$. Thus operations 1,2, and 3 are in L.

(4) and (5) $Q' = \{4, 5, 6, 11\}$ (operation 9 is dominated by operation 11).

j	1	2	3	4
$\pi(j)$	4	5	6	11
$a_{\pi(j)}$	1	3	5	8
$b_{\pi(j)}$	10	7	4	2

(6)

$$a_{\pi(0)} + b_{\pi(1)} = 0 + 10 = 10$$
$$a_{\pi(1)} + b_{\pi(2)} = 1 + 7 = 8$$
$$a_{\pi(2)} + b_{\pi(3)} = 3 + 4 = 7$$
$$a_{\pi(3)} + b_{\pi(4)} = 5 + 2 = 7$$
$$a_{\pi(4)} + b_{\pi(5)} = 8 + 0 = 8$$

The minimum is obtained for $i_{\text{opt}} = 3$ (or $i_{\text{opt}} = 2$).

(7) The optimal subset obtained is $L^* = \{1, 2, 3, 4, 5, 6\}$, with $C(L^*) = 5 + 2 = 7$. Thus, operation 12 is inserted between operations 6 and 9 on machine A and between operations 6 and 11 on machine D, and the makespan of the new schedule is $d_{st}^{L^*} = \max\{d_{st}, d_{sv} + d_{vt} + C(L^*)\} = \max\{12, 4+2+7\} = 13$.

The new schedule is depicted in Figure 4.12. □

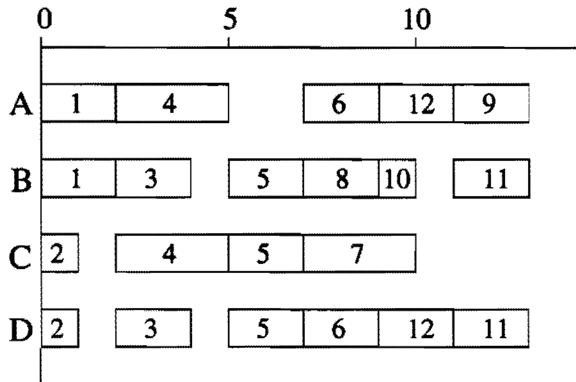

Figure 4.12: The new schedule.

Computational experiments

The best insertion algorithm can be incorporated in an $\mathcal{O}(n(m_1+nq))$ construction algorithm. In each iteration, an operation is selected and inserted in the partial solution network in the best possible position. As in dispatching algorithms, various priority rules can be applied in the selection of the operation that is to be inserted.

Lioce and Martini [1995] report on computational experiments with our constructive insertion algorithm. They consider nine different problem types, including job shop (JS), flow shop (FS), and open shop (OS) problems, as well as variants of these problems in which operations require machine sets (JS2, JS3, FS2, FS3) or more general precedence relations occur (OSh, OSp). In all these problem types, the objective is to minimize the makespan. The test set is described in the appendix.

A summary of the results is given in Table 4.1. For each of the nine problem types, results are reported for the insertion algorithm with two different priority rules and for the dispatching algorithm with two different priority rules. In the insertion algorithm, rule I1 selects operations randomly, and rule I2 selects operations in order of decreasing processing time. Among the considered priority rules, rule I2 gives the best results on average for the problem instances in the test set. In the dispatching algorithm, priority rule D1 selects operations randomly from the set of dispatchable operations (C in the acyclic dispatching algorithm). Rule D2 selects the operation v that has minimal d_{sv} (*earliest possible starting time*), and in case of a tie, the operation with maximal $p_v + d_{vt}$ (*most*

work remaining) is selected. We use the relative distance from the best known solution value as a measure for the quality of a solution obtained by one of the algorithms. Let z^A be the value of a solution obtained by algorithm A, and let z^* be the best known solution value. The relative distance then is equal to

$$\frac{z^A - z^*}{z^*} * 100.$$

In the appendix, it is described how the best known solution values are obtained.

problem type	**average relative distance from best known value** D1	D2	I1	I2
JS	110.09	42.47	22.39	16.15
JS2	148.17	71.40	25.37	22.10
JS3	151.03	70.36	29.21	19.64
FS	120.66	60.93	17.60	9.95
FS2	138.92	88.07	36.22	25.85
FS3	138.44	88.62	38.86	20.13
OS	167.61	100.78	14.91	1.95
OSh	151.88	100.81	23.18	8.84
OSp	140.05	87.18	29.26	18.02
all problems	139.90	75.45	26.20	16.54
cpu (sec.)	(0.13)	(0.82)	(1.22)	(1.21)

Table 4.1: Comparison of dispatching and insertion algorithms.

From Table 4.1, we can conclude that insertion gives better results than dispatching for the considered problem types. For all problem types, the insertion algorithm with priority rule I2 finds solutions that are on average within 26% of the best known solution values. The 'good' dispatching algorithm does not obtain a score of less than 40% for any problem type. Furthermore, we see that the choice of the priority rule has a substantial effect on the quality of the obtained solutions. Both the dispatching and the insertion algorithm give better results when the good priority rule is used than when the random rule is used. However, the random insertion rule still gives fairly good results, and clearly outperforms the good dispatching rule. Insertion requires more time than dispatching ($\mathcal{O}(n^2)$ versus $\mathcal{O}(n)$ for these problem types, when the random priority rule is

applied), but this larger time requirement seems to be outweighed by the much higher quality of the obtained solutions.

For various problem types, it is possible to develop good construction methods by exploiting the specific structural properties of the considered problem. The major advantage of dispatching and insertion, as compared to such other construction methods, is their wide applicability. Furthermore, with respect to insertion we add that solutions of reasonable quality are obtained for a large class of problems.

No experiments have been performed for problems with other objectives than makespan minimization. Dispatching is likely to perform even worse for the more general problems than for the problems considered in the reported experiments. Before efficient insertion methods can be developed, the structure of optimal flows must be studied more extensively. Finding the best insertion position for a single operation, which can be done efficiently when the makespan is to be minimized, is probably much more difficult for arbitrary objective functions.

Insertion offers more flexibility than dispatching, and the described insertion algorithm is shown to outperform dispatching algorithms. I am not aware of any existing purely constructive methods in which the disjunctive constraints are dealt with individually, one after the other. Such methods would provide even more flexibility than insertion methods. It would be interesting to find out if this flexibility can be exploited in order to obtain better selections.

4.4 Local search methods

4.4.1 Local search

The solution space of an instance of the general scheduling problem consists of the set of all possible selections. Since this set is finite and well defined, it is possible to generate and evaluate all solutions in order to find the optimum. This procedure, however, is often too time consuming. The solution procedures discussed in the previous section, dispatching and insertion, require very little time but the obtained solutions are often of poor quality. Local search provides an alternative way to find good solutions.

The idea behind local search is very simple. One starts with a feasible solution and tries to obtain a new, improved solution by modifying it slightly. This procedure is repeated over and over again.

The most important concept in local search is the *neighborhood function* H. Let Σ be the solution space. Associated with each solution $\sigma \in \Sigma$ there is a neighborhood $H(\sigma) \subset \Sigma$. Each solution in $H(\sigma)$ is called a neighbor of σ. A

local search algorithm searches the solution space by moving from one solution to another (neighboring) solution, over and over again. A move from σ to τ is denoted by $[\sigma, \tau]$.

A more general neighborhood concept is used in 'genetic algorithms'. Here, at each stage a *population* $\Sigma_1 \subset \Sigma$ is maintained, and a *hyperneighborhood* $H(\Sigma_1) \subset \Sigma$ is defined for such a population. A new population is then obtained from the current one by selecting a set of solutions in this hyperneighborhood.

A straightforward implementation of local search is 'iterative improvement'. Given an initial solution, the neighborhood of this solution is searched for solutions of higher quality. When no such neighbor exists, the current solution is a *local optimum*. If there is a better solution in the neighborhood, then either the best neighbor ('best improvement') or the first neighbor of higher quality that is found ('first improvement') is selected as the next solution.

Iterative improvement always ends up with a local optimum. The quality of a local optimum, however, may be poor. Several methods have been proposed that improve on iterative improvement algorithms by offering the possibility to get out of a local optimum. These methods do not only have fancy names such as 'variable depth search', 'simulated annealing', or 'taboo search', but they may also work very well. Vaessens et al. [1994] survey local search algorithms for the standard job shop scheduling problem, and find that taboo search and variable depth search perform best for this problem.

A neighborhood function H is said to be *connected* if for each solution σ_1 there is an optimal solution σ_k that can be reached from σ_1 by performing a sequence of moves $[\sigma_1, \sigma_2], [\sigma_2, \sigma_3], \ldots, [\sigma_{k-1}, \sigma_k]$, where $\sigma_i \in H(\sigma_{i-1})$. Connectivity is a property of mainly theoretical importance. The fact that an optimal solution can be reached from any solution does by no means guarantee that such an optimal solution will actually be found. If a neighborhood function is connected, then simulated annealing can be proved to converge almost surely to an optimal solution. Simulated annealing as an optimization algorithm does require, however, infinite running time. Any optimization algorithm, even complete enumeration, is more efficient. Still, connected neighborhood functions are preferred to those that are not connected. When a neighborhood function that is not connected is used, the probability of getting trapped in an uninteresting part of the solution space is generally higher than when a connected neighborhood function is used.

4.4.2 Two neighborhoods

In this section, I restrict my attention to the study of neighborhoods. I discuss two neighborhoods that can be applied in iterative improvement, simulated anneal-

ing, variable depth search, or taboo search algorithms for the general scheduling problem. In all these algorithms, it is important that the neighborhood can be searched efficiently. Two aspects of neighborhoods are important in this context: *infeasibility detection* and *neighborhood reduction.* Time spent in considering infeasible neighbors is wasted time. If conditions can be formulated under which infeasibility (or feasibility) is guaranteed, these conditions can be used to discard infeasible neighbors in an early stage and to focus on feasible neighbors. Neighborhood reduction can be applied when it is possible to distinguish 'promising' and 'unpromising' neighbors beforehand. The unpromising neighbors are discarded, and attention is restricted to the promising neighbors. When the reduced neighborhood is much smaller than the original one, the time spent in searching a neighborhood is reduced substantially. I will show for the two neighborhoods that are discussed in this section that they can be reduced by discarding a large number of neighbors that are guaranteed to be non-improving.

The swap neighborhood

The swap neighborhood is the simplest neighborhood for the general scheduling problem that one can think of. For a given selection σ, the swap neighborhood $H^s(\sigma)$ consists of all selections τ that can be obtained from σ by reversing the processing order of one pair of operations $\{u, v\}$. That is, τ is a neighbor of σ if $A_3^\tau = A_3^\sigma \setminus (u, v) \cup (v, u)$, for some $(u, v) \in A_3^\sigma$. We say that τ is obtained from σ by performing the swap (u, v).

Suppose that operation x is performed between operations u and v on some machine. Then (u, x), (x, v), and (u, v) are in A_3^σ. If (u, v) is replaced by (v, u), then there is a cycle consisting of the arcs (u, x), (x, v), and (v, u), and the resulting selection is infeasible. In the remainder, I will therefore consider only swaps (u, v) such that u and v are performed consecutively on some machine. Note that this corresponds to the arc (u, v) appearing in the reduced solution network. The following lemma gives necessary and sufficient conditions for such a swap to be feasible.

Lemma 4.12 (*Feasible swap*)
Let σ be a feasible selection. The selection τ obtained from σ by performing the swap (u, v) is feasible if and only if the longest path from u to v in N_∞^σ, not containing the arc (u, v), is shorter than $-p_{vu}$.

Proof. The selection τ is feasible if and only if there is no cycle with positive length in N_∞^τ. Since N_∞^σ does not contain a cycle with positive length and N_∞^τ arises from N_∞^σ by reversing the arc (u, v), any positive length cycle in N_∞^τ must contain the arc (v, u) which has length p_{vu}. Therefore, a cycle with positive

length in N^{τ}_{∞} will occur if and only if there is a path from u to v in N^{σ}_{∞}, not using the (reversed) arc (u, v), that is longer than $-p_{vu}$. □

Lemma 4.12 states that it is possible to check if a swap is feasible in the time that is required to compute a longest path in N^{σ}_{∞}. Especially when such a longest path can be computed efficiently, for example when N^{σ}_{∞} is acyclic, or when the number of infeasible neighbors is expected to be very high, it may be worthwhile to check if a neighbor is feasible before spending a lot of time in evaluating its quality.

The following lemma states that, if we are only interested in improving neighbors, the swap neighborhood can be reduced substantially. Later on, I will use Lemma 4.12 to show that for specific problem types, any selection in this reduced swap neighborhood is feasible.

Recall that $g(\sigma)$, the cost of a selection σ, is equal to the value of a maximum cost flow in the solution network N^{σ}.

Lemma 4.13 (*Critical swap neighborhood*)
Let σ be a feasible selection, and let x be a feasible flow in the solution network $N^{\sigma} = (V, A^{\sigma})$ with cost $g(\sigma)$. Let $(u, v) \in A^{\sigma}$ be such that $x_{uv} = 0$. Let τ be the selection obtained from σ by performing the swap (u, v). Then $g(\tau) \geq g(\sigma)$.

Proof. The solution network N^{τ} contains all arcs in A^{σ}, except the arc (u, v) which is replaced by an arc (v, u). Since $x_{uv} = 0$, the flow x', with

$$x'_{wz} = \begin{cases} 0, & (w, z) = (v, u), \\ x_{wz}, & \text{otherwise}, \end{cases}$$

is feasible for the network N^{τ}. The cost of this flow is equal to $g(\sigma)$, the cost of the flow x in N^{σ}. Hence, $g(\tau) \geq g(\sigma)$. □

Thus, if there exists a selection τ such that $g(\tau) < g(\sigma)$ in the swap neighborhood of σ, then τ is obtained from σ by performing a swap (u, v) such that $x_{uv} > 0$, where x is a maximum cost flow in N^{σ}. The flow x will sometimes be referred to as a *critical flow*, and the arcs (u, v) such that $x_{uv} > 0$ will be called *critical arcs*. From now on, I will only consider swaps of critical arcs. The corresponding neighborhood is called the critical swap neighborhood.

Note that the number of arcs $(u, v) \in A^{\sigma}_{3}$ such that $x_{uv} > 0$ can be as high as the number of arcs in the reduced solution network. Therefore, Lemma 4.13 does not help us in reducing the time complexity of finding, for example, the best neighbor. In most cases, however, the critical swap neighborhood is much smaller than the original neighborhood, and the neighborhood search will require much less time.

The following lemma identifies a subclass of the general scheduling problem for which all neighbors of a feasible selection in the critical swap neighborhood are guaranteed to be feasible.

Lemma 4.14 (*Feasible critical swaps*)
Consider an instance of the general scheduling problem with

$$c_{uv} \geq p_{uw}, \textit{for all } u,v,w \textit{ such that } (u,v) \in A_1 \textit{ and } \{u,w\} \in A_3.$$

Let σ be a feasible selection. All selections in the critical swap neighborhood $H^{cs}(\sigma)$ are feasible.

Proof. Let $g(\sigma)$ be the cost of a maximum cost flow x in N^σ, and let $(u, v) \in A_3^\sigma$ be such that $x_{uv} > 0$.

First, I show that the longest path from u to v in N_∞^σ has length p_{uv}.

Suppose that there is a path $O = (u, w_1, \ldots, w_k, v)$ from u to v in N_∞^σ with length $d_O > p_{uv}$. Then, the flow x', with

$$x'_{yz} = \begin{cases} 0, & (y,z) = (u,v), \\ x_{yz} + x_{uv}, & (y,z) \text{ in } O, \\ x_{yz}, & \text{otherwise}, \end{cases}$$

satisfies all flow balance and flow bound constraints and has cost $(d_O - p_{uv})x_{uv} + g(\sigma) > g(\sigma)$, which contradicts the optimality of the flow x. Hence, the longest path from u to v in N_∞^σ has length p_{uv}.

Now I show that there cannot be any path from u to v in N_∞^σ, not using the arc (u, v) itself. From Lemma 4.12 it then follows that the selection τ obtained from σ by swapping the critical arc (u, v) is feasible, thus completing the proof.

Recall form Section 3.3 that $V = \mathcal{O} \cup \{s, z_{\max}^1, \ldots, z_{\max}^{l_1}, z_{\min}^1, \ldots, z_{\min}^{l_2}\}$. The nodes $z_{\min}^i$, $i = 1, \ldots, l_2$, do not have any outgoing arcs in the solution network since they correspond to variables $S_{z_{\min}^i}$ that occur only in constraints of the form $S_v - S_{z_{\min}^i} \geq -\varepsilon_v$. Similarly, the nodes $z_{\max}^i$, $i = 1, \ldots, l_1$, do not have any outgoing arcs. Hence, a path from u to v in N_∞^σ does not contain any of the z-nodes.

The node s is also not contained in any path from u to v. Suppose that there would be such a path containing s. Then there would exist a path from u to s in N_∞^σ with positive length, which contradicts the feasibility of σ.

Thus, any path from u to v in N_∞^σ is of the form $O = (u, w_1, \ldots, w_k, v)$, with $w_i \in \mathcal{O}$. For the length d_O of such a path we have

$$
\begin{aligned}
d_O &\overset{(*)}{\geq} p_{uw_1} + p_{w_1w_2} + \ldots + p_{w_k,v} \\
&= p_u + p_{w_1} + \ldots + p_{w_k} + \delta_{uw_1} + \delta_{w_1w_2} + \ldots + \delta_{w_kv} \\
&\overset{(**)}{>} p_u + \delta_{uv} = p_{uv},
\end{aligned}
$$

which is in contradiction with the length of the longest path from u to v having length p_{uv}. Inequality $(*)$ holds because of the condition formulated in the lemma, and $(**)$ holds since $p_{w_i} > 0$ and the set-up times δ_{xy} satisfy the triangle inequality. □

All problems with sequence-independent processing times, in which the constraints in A_1 are nonnegativity or ordinary precedence constraints, satisfy the condition formulated in Lemma 4.14. Examples are the job shop and the open shop scheduling problem. An example of a different problem that satisfies this is the open shop problem with sequence-dependent set-up times, where A_1 consists of nonnegativity arcs (s, v) only.

If processing times are sequence-independent, but c_{uv} is not guaranteed to be larger than the processing time of u, then a critical swap may result in an infeasible selection, even if all c_{uv} are nonnegative. For example, let $(x, y) \in A_3^\sigma$ be a critical arc with length $p_{xy} = p_x = 6$, and let (x, w) and (w, y) be arcs in A_1 with $c_{xw} = c_{wy} = 2$. Swapping the arc (x, y) would result in a cycle of length $4 + p_y$, consisting of the arcs (x, w), (w, y), and (y, x).

Similarly, if the processing times are sequence-dependent, critical swaps may be infeasible, even if $c_{uv} > p_u$ for all $(u, v) \in A_1$. For example, let $(x, y) \in A_3^\sigma$ be a critical arc with length $p_{xy} = p_x + \delta_{xy} = 2 + 4 = 6$, and let (x, w) and (w, y) be arcs in A_1 with $c_{xw} = p_x = 2$, and $c_{wy} = p_w = 2$. Then again, swapping (x, y) results in a positive length cycle.

When the restrictions on the possible values of p_{uv} and c_{uv} as formulated in Lemma 4.14 are satisfied, the critical swap neighborhood is connected.

Theorem 4.15 (*Connectivity*)
Consider an instance of the general scheduling problem that satisfies the condition formulated in Lemma 4.14. For each feasible selection σ_0 there exists a finite sequence of selections $(\sigma_0, \sigma_1, \sigma_2, \ldots, \sigma_l)$, where σ_l is an optimal selection, and σ_i is obtained from σ_{i-1} by performing a critical swap.

Proof. The proof is analogous to the proof by Van Laarhoven et al. [1992] of the connectivity of the swap neighborhood function for the job shop scheduling problem.

Let σ^* be an optimal selection. Associated with each feasible selection τ, there is a critical flow x^τ in N^τ. Let $\Delta(\tau)$ be the number of arcs $(u, v) \in A_3^\tau$ such that $(v, u) \in A_3^{\sigma^*}$, and let $\Delta^c(\tau)$ be the number of critical arcs $(u, v) \in A_3^\tau$ such that $(v, u) \in A_3^{\sigma^*}$. Obviously, $\Delta^c(\tau) \leq \Delta(\tau)$, for all τ.

Suppose that $\Delta^c(\tau) = 0$. Then the flow x^τ is feasible in N^{σ^*}, and thus $g(\sigma^*) \geq g(\tau)$. Since σ^* is optimal, equality must hold and τ is optimal.

Suppose that $\Delta^c(\tau) > 0$. Then there is a critical arc $(u, v) \in A_3^\tau$ such that $(v, u) \in A_3^{\sigma^*}$. From Lemma 4.14 it follows that the swap (u, v) results in a feasible selection τ', and $\Delta(\tau') = \Delta(\tau) - 1$.

Thus, for an arbitrary feasible selection σ_i, either optimality can be demonstrated, or a critical arc (u, v) such that $(v, u) \in A_3^{\sigma^*}$ can be identified. In the second case, a feasible selection σ_{i+1} is formed by performing the swap (u, v). Starting from σ_0, a series of feasible selections with decreasing $\Delta(\sigma_i)$ can be obtained. After at most $\Delta(\sigma_0)$ steps, a selection σ_l is found with $\Delta^c(\sigma_l) = 0$. □

Theorem 4.15 is of only limited importance. It can be compared to Theorem 4.4, which states that an optimal selection can be obtained by a dispatching algorithm. In the proofs of both theorems, extensive use is made of information about an optimal selection. This information is generally not available, and therefore the sequence of selections described in Theorem 4.15 cannot be constructed, just as it is not possible to construct the optimal dispatching order.

For the problem in its most general form, no connectivity results for the swap neighborhood can be given, even if also non-critical swaps are considered. Consider the example given in Figure 4.13.

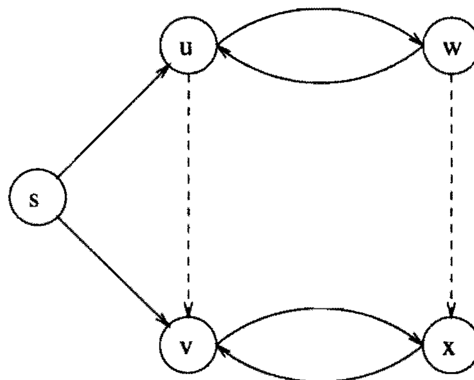

Figure 4.13: Only infeasible swaps.

The solid arcs are in A_1, the dashed arcs are in A_3^σ. The arcs in A_2 are not drawn here. All arcs in A_1 have length 0. Suppose that there is a unique optimal selection σ^* with $A_3^{\sigma^*} = \{(v, u), (x, w)\}$. Any swap of an arc in A_3^σ results in an infeasible selection. Thus, it is not possible to obtain an optimal solution by performing swaps only.

The swap neighborhood has been thoroughly studied for the job shop scheduling problem. Several authors have proposed modifications of the critical swap neighborhood in order to improve its performance (Van Laarhoven et al. [1992], Matsuo et al. [1988], Dell'Amico and Trubian [1993]). These modifications may also be applied to the swap neighborhood for the general scheduling problem.

The jump neighborhood

The jump neighborhood (or reinsertion neighborhood) $H^j(\sigma)$ of a selection σ consists of all selections that can be obtained by changing the position of one operation in the processing order. In other words, a selection τ is in the jump neighborhood of σ if it can be obtained from σ by reversing only arcs (u, v) and (v, w) in A_3^σ, for some $v \in \mathcal{O}$. Performing the move from σ to τ boils down to 'removing' v from σ, i.e., removing all arcs (u, v) and (v, w) in A_3^σ, and then 'reinserting' v in the thus formed partial solution network $N^\sigma(V \setminus \{v\})$. Analogous to insertion as described in Section 4.3.2, a jump can be denoted by a pair (v, L), where $L \subset \mathcal{O}$ is the set of operations w such that $\{v, w\} \in A_3$ and $(w, v) \in A_3^\tau$.

Feasibility aspects of insertion have already been discussed in Section 4.3.2. The following lemma characterizes feasible jumps.

Lemma 4.16 (*Feasible jump*)
Let σ be a feasible selection. Let $N^\sigma(V \setminus \{v\})$ be the partial solution network that arises when all arcs (u, v) and (v, w), for some $v \in \mathcal{O}$, are removed from A_3^σ. Let d'_{xy} be the length of the longest path from x to y in $N_\infty^\sigma(V \setminus \{v\})$. Let $Q := \{u \in \mathcal{O} \mid \{u, v\} \in A_3\}$, and let $L \subseteq Q$.

The selection τ obtained from σ by performing the jump (v, L) is feasible if and only if

- $d'_{vu} \leq -p_{uv}$, $\forall u \in L$,
- $d'_{wv} \leq -p_{vw}$, $\forall w \in Q \setminus L$,
- $d'_{wu} \leq -(p_{uv} + p_{vw})$, $\forall u \in L, w \in Q \setminus L$.

Proof. The three conditions correspond to the three cycles in Figure 4.6; a cycle with positive length of one of the three types arises from the insertion (v, L) if and only if the corresponding condition is violated. □

Note that the swap neighborhood is contained in the jump neighborhood. Any swap (u, v) can be seen as a reinsertion of operation u (or v). Since the jump neighborhood is much larger, one may expect that it contains better selections, and therefore that a local search algorithm that uses this neighborhood will perform better than an algorithm that uses the swap neighborhood. This is only partially true. Especially when one is interested in the time-quality ratio of an algorithm, smaller neighborhoods may be preferred. A smaller neighborhood can be searched more efficiently, and the higher quality of neighbors in the larger neighborhood may be outweighed by the amount of time that is required to find them. Powerful neighborhood reduction is therefore required.

Lemma 4.17 (*Critical jump neighborhood*)
Let σ be a feasible selection, and let x be a critical flow in the solution network $N^\sigma = (V, A^\sigma)$ with cost $g(\sigma)$. Let $v \in \mathcal{O}$ be such that $x_{uv} = x_{vw} = 0$, for all $(u, v), (v, w) \in A_3^\sigma$. Let τ be a selection obtained from σ by performing a jump of operation v. Then $g(\tau) \geq g(\sigma)$.

Proof. The solution network N^τ contains all arcs in A^σ, except for arcs which have zero flow. The flow x', with

$$x'_{wz} = \begin{cases} 0, & w = v \text{ or } z = v, \\ x_{wz}, & \text{otherwise}, \end{cases}$$

is feasible for the network N^τ. The cost of this flow is equal to $g(\sigma)$, the cost of the flow x in N^σ. Hence, $g(\tau) \geq g(\sigma)$. □

Reinsertion of an operation v that does not occur in the critical flow will never result in an improved selection. The size of the jump neighborhood is, however, not only determined by the number of operations that can be reinserted, but also by the number of possible reinsertion positions for each operation. As discussed in Section 4.3.2, this number is exponential in the size of the machine sets. Therefore, a complete search of the critical jump neighborhood will be too time consuming, even if the machine sets are of moderate size. Other search strategies must be considered. If the problem is of the form as described in Section 4.3.3, the best reinsertion position of a given operation v can be computed in $\mathcal{O}(n(m_1 + nq))$ time, and finding the best neighbor in the critical jump neighborhood takes $\mathcal{O}(n(m_1 + nq))$ time. If good insertion positions cannot be determined efficiently, it may be necessary to use heuristic arguments in order to obtain a reduced jump neighborhood of acceptable size.

Since a swap can be seen as a special case of a jump, the positive connectivity results that have been given for the critical swap neighborhood also hold for

the critical jump neighborhood. If the jump neighborhood is reduced even further, for example by considering only 'optimal' reinsertions, then the connectivity properties are lost. Although the jump neighborhood is much larger than the swap neighborhood, also the jump neighborhood function is not connected for the problem in its most general form. In the example given in Figure 4.13, there are no feasible swaps and no feasible reinsertions, except for reinsertions that do not modify the current selection.

Computational experiments

Based on the results presented above, local search methods that use the swap and the jump neighborhood for the general scheduling problem can be developed. Much research, however, has still to be performed before such methods can be made time efficient. The jump neighborhood is too large to make a complete neighborhood search possible, and it is not clear beforehand in which way a sufficient reduction of the size of this neighborhood can be obtained. Furthermore, although a selection can be evaluated in polynomial time by solving a maximum cost flow problem, the time requirement is still so high that a straightforward implementation would spend too much time in evaluating each of the many selections that must be considered. Efficient updating of maximum cost flows when small modifications of solution networks occur is required in order to obtain a more efficient algorithm.

I will therefore discuss only some experiments with the two neighborhoods performed on the subclass of the general scheduling problem that is described in Section 4.3.3. For this subclass, evaluation of a selection requires only longest path computations, and the best neighbor in the jump neighborhood can be found efficiently.

The goal of the first experiment is to compare the quality of local minima obtained by the iterative improvement algorithm with the jump neighborhood and the swap neighborhood. The goal of the second experiment is to find out whether the better performance of the jump neighborhood, as observed in the first experiment, still holds when a time-quality comparison is made. For this purpose, both neighborhoods are incorporated in a taboo search algorithm, and then both algorithms are given the same amount of time. In both experiments, the test set as described in the appendix is used.

Experiment 1.

For all 90 problem instances in the test set, 1000 selections are generated by applying the dispatching algorithm with the random priority rule. Starting from each of these 90,000 selections, four iterative improvement algorithms are ap-

plied in order to find local minima. The four algorithms use either the swap or the jump neighborhood, and apply either first improvement or best improvement.

Since only improving neighbors are of interest, only swaps or reinsertions on the critical flow (or, in this case, critical path) need to be considered. The algorithms that use the jump neighborhood consider each operation on the critical path, and check if it can be reinserted in a profitable way by applying the best insertion algorithm described in Section 4.3.3. When first improvement is used, the first profitable reinsertion is actually performed, and the procedure is repeated for the new selection. When best improvement is applied, all operations on the critical path are considered, and the reinsertion that is most profitable is performed.

Let σ be the current selection, and let d_{st} be the makespan of the corresponding schedule. Let τ be a neighboring selection with makespan d'_{st}. In the algorithms used in both experiments, the value of d'_{st} is not computed exactly, but an estimate e'_{st} is computed that has the following properties (see, e.g., Dell'Amico and Trubian [1993]):

- If $d'_{st} < d_{st}$, then $d'_{st} \leq e'_{st} \leq d_{st}$.
- If $d'_{st} \geq d_{st}$, then $e'_{st} = d'_{st}$.

The advantage of using the estimate is that it can be computed more efficiently than the actual value, while improving neighbors are still identified.

In the iterative improvement algorithm that uses the swap neighborhood, the estimate for d'_{st} is computed for each possible swap on the critical path. When first improvement is used, the first swap for which the estimate is smaller than d_{st} is actually performed. When best improvement is used, the swap with the smallest estimate (if smaller than d_{st}) is performed.

For each of the four algorithms, a sample of 1000 local minima for each of the 90 instances is obtained. The empirical distributions of the quality of the four kinds of local minima, based on these four times 90,000 observations, are depicted in Figure 4.14. On the horizontal axis, the relative distance (in percents) from the best known value is set out. The vertical axis represents the observed fraction of local minima with smaller relative distance.

The jump neighborhood clearly outperforms the swap neighborhood. More often solutions with small relative distance from the best known are obtained. With first improvement, about 90% of the observed local minima obtained with the jump neighborhood have a relative distance of 40% or less, against 20% of the local minima obtained with best improvement. For best improvement, these figures are 56% and 12%, respectively. The average relative distance for a local minimum obtained with the jump neighborhood using first improvement is about

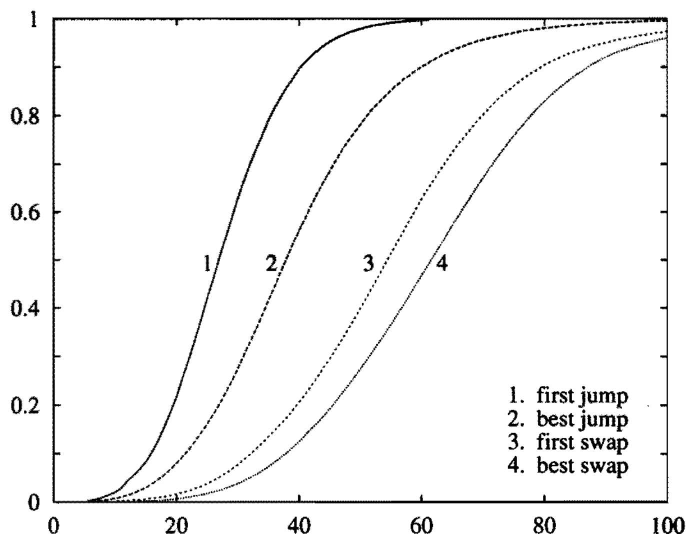

Figure 4.14: Distribution of local minima.

28%, whereas the same value for the swap neighborhood is about 56%.

Surprisingly, first improvement gives better results than best improvement. For both neighborhoods, the local optima obtained with first improvement are on average closer to the best known solution value than the local minima obtained with best improvement. The average distance is about 40% for best improvement with the jump neighborhood and about 62% with the swap neighborhood against, respectively, 28% and 56% for first improvement. Spending more time in order to find the best neighbor brings, apparently, an increased risk of getting stuck early in a local minimum.

Figure 4.14 gives information aggregated over all nine considered problem types. There are, however, some variations in the results obtained for different problem types. Consider the empirical density functions depicted in Figures 4.15 and 4.16. The vertical axis now represents the fraction of observations that have a particular relative distance from the best known value.

The OSh problem is a variant of the open shop problem in which each job consists of two sets of operations. The operations of the first set must be performed before the operations of the second set, but there is no restriction on the processing order of operations within the same set. The four density functions for the job shop problem have a similar shape, and in particular the density functions

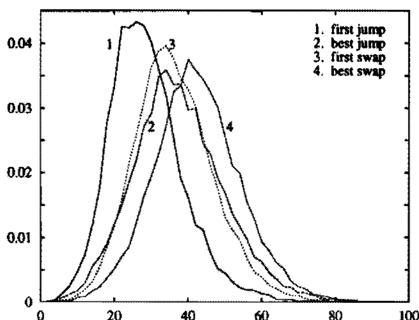

Figure 4.15: Density functions for job shop instances.

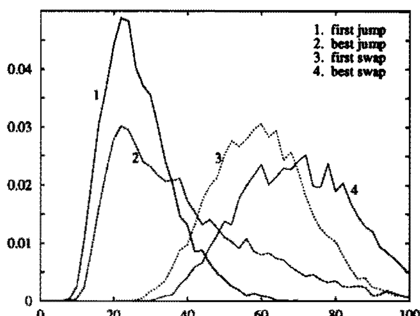

Figure 4.16: Density functions for OSh instances.

of best improvement with the jump neighborhood and first improvement with the swap neighborhood are almost identical. For the OSh instances, on the other hand, the differences between the swap and the jump neighborhood are much more pronounced.

Experiment 2

The first experiment has shown that the jump neighborhood is more powerful than the swap neighborhood. The question that remains is whether jumps are also competitive when the higher time requirement is taken into account. Lioce and Martini [1995] have implemented a taboo search algorithm in which they have incorporated the swap and the jump neighborhood. The taboo search implementation is similar to the implementation of Dell'Amico and Trubian [1993]. Several experiments have been conducted in which a fixed number of iterations was performed. For all problem types, the jump neighborhood gave better results than the swap neighborhood, thus confirming the results of the experiment described above. I will only discuss the experiments in which the same amount of time has been given to both algorithms, and refer to the report of Lioce and Martini for a discussion of other experiments.

For each of the 90 problem instances, five different selections were obtained by the constructive insertion algorithm, in which the first 60% of the operations were inserted in order of decreasing processing times, and the remaining 40% (with smallest processing times) were inserted in random order. Starting from each selection 5000 iterations of the taboo search algorithm with the jump neighborhood were performed. After 1000 iterations, the value of the best solution found so far as well as the required time was recorded. The same was done after

problem	**average relative distance from best known value**			
type	*swap*		*jump*	
	$t1000$	$t5000$	1000 *iter*	5000 *iter*
JS	2.7	2.1	4.9	2.9
JS2	7.8	7.1	6.0	3.2
JS3	7.1	6.4	5.5	2.7
FS	2.6	2.0	3.1	1.6
FS2	13.6	12.6	6.0	3.8
FS3	11.6	10.7	5.3	3.3
OS	2.7	2.4	0.2	0.1
OSh	7.4	6.7	2.6	1.3
OSp	9.8	7.6	4.1	1.7
All problems	7.0	6.2	4.4	2.4
cpu (sec.)	(50.3)	(248.3)	(50.3)	(248.3)

Table 4.2: Time-quality comparison of the two neighborhoods in a taboo search algorithm.

the complete run of 5000 iterations. Then, the algorithm with the swap neighborhood was given the same amount of time. The best solution found within the time required for 1000 jump iterations as well as the best solution found during the entire run were recorded. In Table 4.2, a summary of the results is given.

For each of the nine problem types, the last two columns give the average relative distance from the best known value for selections obtained by the taboo search algorithm with the jump neighborhood after 1000 and 5000 iterations. The two other columns, $t1000$ and $t5000$ give the average relative distance obtained by the algorithm with the swap neighborhood when it was given the same amount of time as required by 1000 and 5000 jump iterations. The average time requirement for 1000 jump iterations was 50.3 seconds on a SUN SPARC 5. In the same time, on average about 20,000 swap iterations could be performed.

In Table 4.2, we see that the jump neighborhood outperforms the swap neighborhood also in a time-quality comparison, except for the job shop (JS) and the flow shop (FS) problem. For the job shop problem, the swap neighborhood gives better results than the jump neighborhood when the same amount of time is given to both algorithms. For the considered instances of the flow shop problem, this

is true for the time required by 1000 jump iterations (2.6% versus 3.1%), but the situation is reversed when 5000 jump iterations are considered (2.0% versus 1.6%).

Very good results are obtained for the open shop problems. After 1000 jump iterations, the average relative distance from the optimal value is only 0.21%. Insertion seems to be very powerful for open shop problems, because also the constructive insertion algorithm gave very good results for this problem. We must keep in mind, however, that the results depend strongly on the difficulty of the considered instances. The job shop instances are known to be hard. Many researchers have tried to solve them with only limited success. Perhaps more difficult open shop problems must be generated before we can make any justified statements about the power of the insertion technique when applied to this kind of problem.

4.5 Relaxation

Consider an arbitrary instance of the general scheduling problem:

$$(GP)\quad \begin{array}{lll} \min & \beta + \sum_{v \in V} b_v S_v + \sum_{(v,w) \in A_2} h_{vw} \alpha_{vw} & \\ \text{s.t.} & S_w - S_v \geq c_{vw}\,, & \forall (v,w) \in A_1, \\ & S_w - S_v + \alpha_{vw} \geq c_{vw}\,, & \forall (v,w) \in A_2, \\ & S_w \geq S_v + p_{vw} \vee S_v \geq S_w + p_{wv}\,, & \forall \{v,w\} \in A_3, \\ & S_s = 0 & \\ & \alpha_{vw} \geq 0\,, & \forall (v,w) \in A_2, \end{array}$$

A straightforward approach for solving this problem consists of two phases:

1. Find a feasible selection of the disjunctive constraints using one of the construction methods discussed in Section 4.3.
2. Use this selection as a starting point for a local search algorithm that uses one of the neighborhoods discussed in Section 4.4.

With respect to the second phase, I will focus on the critical swap neighborhood. If we are not able to apply this simple neighborhood, then there is little hope that the more complicated jump neighborhood can be applied successfully.

As discussed in Section 4.3, a feasible selection for problem (GP) can be obtained by a dispatching or insertion algorithm if the network $N_1 = (V, A_1)$ is acyclic. Furthermore, in Section 4.4, I have given a condition on the lengths of

the arcs in N_1 under which critical swaps are guaranteed to be feasible and the critical swap neighborhood function is connected. If these conditions are satisfied, it is possible to apply straightforwardly any local search algorithm that uses the critical swap neighborhood. For example, taboo search and simulated annealing algorithms that have been applied successfully to the job shop problem require only modification of the selection evaluation procedure, and some finetuning of some of the parameters. Of course, more research is required to develop efficient algorithms, but the basic ideas behind existing algorithms can also be applied to instances of the general scheduling problem that satisfy the abovementioned conditions.

There are, however, many problems that do not satisfy these conditions. For example, N_1 is cyclic for job shop problems with deadlines, and if the processing times are sequence-dependent feasibility of critical swaps cannot be guaranteed. Even if a feasible selection is found for such a problem, the probability that a local search algorithm gets stuck in a selection of poor quality is very high. In Figure 4.13, I gave an example of a feasible selection which could only be left by performing an infeasible swap.

In this section, I discuss two different relaxation techniques that may be useful in developing solution methods for such 'hard' problems.

4.5.1 Lagrangian relaxation

Let $A_1^* \subseteq A_1$ be a set of constraints such that the network $N_1^* = (V, A_1^*)$ is acyclic and $c_{uv} \geq p_{uw}$, for all u, v, and w such that $(u, v) \in A_1^*$ and $\{u, w\} \in A_3$. A_1^* can then be considered as a set of 'easy' constraints and $A_1 \setminus A_1^*$ as a set of 'difficult' constraints. Let λ be a nonnegative vector of *Lagrangian multipliers*. The Lagrangian relaxation $LR(\lambda)$ of problem (GP) is

$$
\begin{array}{lll}
\min & \beta + \sum\limits_{v \in V} b_v S_v + \sum\limits_{(v,w) \in A_2} h_{vw} \alpha_{vw} + \sum\limits_{(v,w) \in A_1 \setminus A_1^*} \lambda_{vw}(c_{vw} + S_v - S_w) & \\
\text{s.t.} & S_w - S_v \geq c_{vw}\,, & \forall (v, w) \in A_1^*, \\
(LR(\lambda)) & S_w - S_v + \alpha_{vw} \geq c_{vw}\,, & \forall (v, w) \in A_2, \\
& S_w \geq S_v + p_{vw} \vee S_v \geq S_w + p_{wv}\,, & \forall \{v, w\} \in A_3, \\
& S_s = 0 & \\
& \alpha_{vw} \geq 0\,, & \forall (v, w) \in A_2.
\end{array}
$$

Note that the Lagrangian relaxation is again an instance of the general scheduling problem. The hard constraints are removed, and the new objective function

can be rewritten as

$$\beta^\lambda + \sum_{v \in V} b_v^\lambda S_v + \sum_{(v,w) \in A_2} h_{vw}\alpha_{vw},$$

with

$$\beta^\lambda = \beta + \sum_{(v,w) \in A_1 \setminus A_1^*} \lambda_{vw} c_{vw},$$

and

$$b_v^\lambda = b_v + \sum_{w:(v,w) \in A_1 \setminus A_1^*} \lambda_{vw} + \sum_{u:(u,v) \in A_1 \setminus A_1^*} \lambda_{uv}.$$

The problem structure, however, has changed in such a way that a feasible selection can be obtained by one of the described construction methods (the network (V, A_1^*) is acyclic), and the critical swap neighborhood of a feasible selection contains only feasible selections. The construction and local search methods discussed before can thus be applied to find good solutions for the Lagrangian relaxation. In the following lemma, a relation between the values of the optimal solutions of the original problem and the Lagrangian relaxation is established.

Lemma 4.18 (*Lower bound*)
Let (GP) be an instance of the general scheduling problem, and let $LR(\lambda)$ be the Lagrangian relaxation. For any nonnegative vector λ of Lagrangian multipliers, the value $z_{LR(\lambda)}$ of the Lagrangian relaxation is a lower bound on z_{GP}, the optimal value of (GP).

Proof. Let S^* be an optimal solution of (GP), with value z_{GP}, and let λ be a nonnegative vector of Lagrangian multipliers. S^* is feasible for problem $LR(\lambda)$, and it has value $z_{GP} + \sum_{(v,w) \in A_1 \setminus A_1^*} \lambda_{vw}(c_{vw} + S_v^* - S_w^*) \leq z_{GP}$, since $c_{vw} + S_v^* - S_w^* \leq 0$ (feasibility) and $\lambda_{vw} \geq 0$. The value of any feasible solution of $LR(\lambda)$ is an upper bound on the optimal value $z_{LR(\lambda)}$. Thus, $z_{GP} \geq z_{LR(\lambda)}$. □

The optimization problem

$$\max_{\lambda \geq 0} z_{LR(\lambda)}$$

is referred to as the *Lagrangian multiplier problem* or the *Lagrangian dual*. Let z_{LD} be the optimal value of this problem. From Lemma 4.18, we have for any nonnegative vector λ of Lagrangian multipliers,

$$z_{LR(\lambda)} \leq z_{LD} \leq z_{GP}. \tag{4.5}$$

The optimal solution of the Lagrangian dual gives the strongest lower bound on the value of the optimal solution of (GP) that can be obtained by solving a Langrangian relaxation. The following theorem identifies situations in which the optimal solution of a Lagrangian relaxation can be shown to be optimal for the original problem as well.

Theorem 4.19 (*Optimal solutions*)
Let (GP) be an instance of the general scheduling problem. Let S^λ be an optimal solution of the Lagrangian relaxation $LR(\lambda)$. If S^λ is feasible for problem (GP) and $\lambda_{vw}(c_{vw}+S_v^\lambda-S_w^\lambda)=0$ for all $(v,w)\in A_1\setminus A_1^*$, then $z_{LR(\lambda)}=z_{LD}=z_{GP}$, and S^λ is an optimal solution of (GP).

Proof. Since S^λ is an optimal solution of $LR(\lambda)$, we have

$$z_{LR(\lambda)}=\beta+\sum_{v\in V}b_vS_v^\lambda+\sum_{(v,w)\in A_2}h_{vw}\alpha_{vw}^\lambda+\sum_{(v,w)\in A_1\setminus A_1^*}\lambda_{vw}(c_{vw}+S_v^\lambda-S_w^\lambda),$$

with $\alpha_{vw}^\lambda=\max\{0,c_{vw}+S_v^\lambda-S_w^\lambda\}$. If $\lambda_{vw}(c_{vw}+S_v^\lambda-S_w^\lambda)=0$ for all $(v,w)\in A_1\setminus A_1^*$, and S^λ is feasible for (GP), then

$$z_{LR(\lambda)}=\sum_{v\in V}b_vS_v^\lambda+\sum_{(v,w)\in A_2}h_{vw}\alpha_{vw}^\lambda\geq z_{GP}.$$

From (4.5) it follows that $z_{GP}=z_{LR(\lambda)}$, and that S^λ is an optimal solution of (GP). □

In general, however, there is a positive *duality gap* $z_{GP}-z_{LD}$. It may still be worthwhile to try to solve the Lagrangian dual. The optimal solution of the Lagrangian dual often gives a good lower bound on the optimal value of the original problem. Furthermore, one may find feasible or near-feasible solutions of the original problem while looking for optimal solutions of the Lagrangian dual. It may sometimes be possible to modify such near-feasible solutions in order to obtain feasible solutions of good quality, for example by using them as a starting point for a penalty method as will be described in Section 4.5.2.

Several strategies for solving the Lagrangian dual problem can be thought of. A natural approach would be to start with some vector of Lagrangian multipliers, solve the corresponding Lagrangian relaxation, and use information about the solution of this relaxation to update the multipliers. In almost all successful applications of Lagrangian relaxation, problem specific multiplier adjustment strategies are used, and such strategy will probably also be required when Lagrangian relaxation is applied to the general scheduling problem.

4.5.2 Penalty methods

An alternative approach to the Lagrangian relaxation method discussed above is to apply a penalty method. We can associate a penalty cost π_{uv} with each hard constraint $(u, v) \in A_1 \setminus A_1^*$. For each unit by which the constraint (u, v) is violated, a cost π_{uv} is incurred. As has been discussed shortly in Section 3.3.2, adjusted start-start constraints are useful for modeling such violations.

Let $S_w - S_v \geq c_{vw}$ be a hard constraint. Replacing this constraint by the constraints $S_w - S_v + \alpha_{vw} \geq c_{vw}$ and $\alpha_{vw} \geq 0$ corresponds to transforming the original constraint into a soft constraint, with α_{vw} representing the amount of violation of the original constraint. Such a violation can then be penalized by introducing a penalty cost h_{vw} for each unit of violation.

In this way, the following problem is obtained:

$$
\begin{array}{lll}
\min & \beta + \sum_{v \in V} b_v S_v + \sum_{(v,w) \in A_2} h_{vw}\alpha_{vw} + \sum_{(v,w) \in A_1 \setminus A_1^*} \pi_{vw}\alpha_{vw} & \\
\text{s.t.} & S_w - S_v \geq c_{vw}\,, & \forall (v, w) \in A_1^*, \\
(GP(\pi)) & S_w - S_v + \alpha_{vw} \geq c_{vw}\,, & \forall (v, w) \in A_2 \cup (A_1 \setminus A_1^*), \\
& S_w \geq S_v + p_{vw} \vee S_v \geq S_w + p_{wv}\,, & \forall \{v, w\} \in A_3, \\
& S_s = 0 & \\
& \alpha_{vw} \geq 0\,, & \forall (v, w) \in A_2 \cup (A_1 \setminus A_1^*).
\end{array}
$$

$(GP(\pi))$ is again an 'easy' instance of the general scheduling problem, and the discussed construction and local search methods can be used to find good solutions.

If the penalty coefficients are sufficiently large, any solution of $(GP(\pi))$ in which one of the constraints in $A_1 \setminus A_1^*$ is violated (or, better, in which α_{vw} is positive for some $(v, w) \in A_1 \setminus A_1^*$), has higher cost than any selection in which none of these constraints is violated. Therefore, for π sufficiently large, the optimal solution of $(GP(\pi))$ is also an optimal solution for (GP).

Hence, a natural approach for solving an instance of the general scheduling problem would be (1) to identify hard constraints, (2) to incorporate these constraints in the objective function with high penalty coefficients, (3) to find a feasible selection for the modified problem, and (4) to apply local search (with the critical swap neighborhood) to find good selections for the modified problem.

There is, however, one important disadvantage of large penalties combined with local search. When penalties become larger, it becomes increasingly difficult to get out of local minima. For a given selection, a number of constraints is violated and a number of constraints is satisfied. Any swap that does not change the number of violated constraints is much preferred to a swap that causes an increase of the number of violated constraints. Performing such a 'bad' move may,

however, be necessary to obtain any substantial improvement in later stages of the algorithm.

If the algorithm gets trapped in a local minimum of this kind, it will be difficult to get out of it. Even if more sophisticated search strategies such as taboo search or simulated annealing are applied, it is difficult to prevent the search to be limited to a small part of the solution space around some local minimum. The quality of these solutions can be poor, for example when the number of violated constraints is positive, and local search is likely to result in selections of poor quality.

One way of overcoming this problem is to start with low penalties and gradually increase the penalty coefficients of constraints that are violated. Starting with low penalties π_0, a good selection is obtained for the problem $(GP(\pi_0))$ by applying local search. For this selection, some of the constraints in $A_1 \setminus A_1^*$ are violated and some are not. The penalty coefficients of the violated constraints are then increased whereas the other penalty coefficients remain unchanged (or are increased less strongly, or even decreased). A new problem $(GP(\pi_1))$ is obtained, and the procedure is repeated.

In this way, a sequence of problems $(GP(\pi_0)), (GP(\pi_1)), (GP(\pi_2)), \ldots$ is formed. The first problems are used to search for promising areas of the solution space, both with respect to feasibility and quality, and to identify hard constraints. In later stages, attention is focused on these promising areas and because of the increasing penalties the search is guided more and more in the direction of feasible selections.

Obviously, the proposed approach is not guaranteed to give feasible selections. However, because of the low penalties, local search methods such as taboo search and simulated annealing are given the opportunity to search the solution space more or less thoroughly in order to identify promising areas. The probability of getting trapped in a small area around some local minimum is smaller than when large penalties are used.

4.5.3 Comparison of the two relaxation methods

There are two important differences between the penalty method and Lagrangian relaxation.

First, the penalty method is guaranteed to give optimal solutions of (GP) if the subproblem $(GP(\pi))$ can be solved to optimality. If the Lagrangian subproblem $LR(\lambda)$ can be solved to optimality, however, only the optimal solution of the Lagrangian dual can be obtained, which gives only a lower bound on the optimal solution of (GP).

Secondly, there are substantial differences between the subproblems $(GP(\pi))$

and $LR(\lambda)$. In $(GP(\pi))$, the hard start-start constraints are replaced by adjusted start-start constraints. The solution networks of (GP) and $(GP(\pi))$ are identical, except for the capacities of the arcs associated with the hard start-start constraints. In the solution networks associated with $LR(\lambda)$, the arcs corresponding to hard start-start constraints are removed entirely, and some of the demands b_v are changed. Possibly, more efficient maximum cost flow problems can be applied to networks from which specific arcs are removed. If this is the case, local search methods applied to $LR(\lambda)$ will be more efficient than similar methods applied to $(GP(\pi))$.

Anyhow, more research is required before any of the two proposed relaxation methods can be implemented successfully. It is important to find good methods for identifying promising areas of the solution space and to develop good strategies for updating the Lagrangian multipliers or the penalty coefficients.

5

Discussion

The main topic of this thesis has been the identification of common aspects of planning problems that can be exploited in the design of a planning board generator (PBG). I have used a top-down approach to develop specification methods for problem instances and types, and used a bottom-up approach to define the general scheduling problem and to develop solution methods for it in order to provide algorithmic support for the manipulations offered by a planning board.

In this chapter, I summarize the results obtained and describe, in an informal way, some steps that have to be taken before a PBG can actually be developed. Furthermore, I discuss how the results obtained for the general scheduling problem contribute to the theory of machine scheduling and present some further generalizations and suggestions for future research.

5.1 Towards a planning board generator

In Chapter 1, the concept of a planning board generator (PBG) was introduced. Given a specification of the problem type for which a planning board has to be developed and a specification of the desired representations and manipulations, a PBG should automatically create an initial version of the planning board. In the discussion that followed, a conflict between two objectives of a PBG was pointed out. On the one hand, a PBG should be as general as possible, supporting a broad variety of planning problem types; on the other hand, for a given problem type it should be able to generate a planning board that can deal with the specific characteristics of that problem type in an efficient way.

In order to deal with both objectives, a PBG will contain a basic implementation of each of the procedures that support a representation or manipulation. On the basis of information about the problem, these basic implementations can be tailored to the specific properties of the problem type for which a planning

board is to be developed. This tailoring may occur in several ways. Sometimes one formulates a general procedure and tunes it to the specific properties of a problem type by setting some parameter values. In other cases, the requirements for some representation or manipulation are structurally different for different problem types and the tailoring of a basic procedure may involve incorporating appropriate predefined subroutines.

The distinction between the two tailoring methods discussed is related to the distinction between a top-down and a bottom-up approach. In a top-down approach, generally applicable methods are developed, which may be adjusted to deal with specific situations. Often when such an approach is applied successfully, a general procedure that requires tailoring in the form of the setting of parameter values can be formulated. In a bottom-up approach, one develops an efficient, problem-specific method and tries to adjust this method in order to deal with more general problem situations. No further generalization is pursued if the desired level of efficiency cannot be maintained. The bottom-up approach will then result in a number of structurally different methods for different classes of problems. In such a situation, tailoring of a basic implementation of a procedure will generally involve incorporating methods that are most appropriate for a considered problem type.

Obviously, a lot of work has still to be done before a PBG can actually be created. In the following, I present some of my thoughts about two important elements of a PBG, methods for providing it with the required information, and the above mentioned procedures for supporting various representations and manipulations. In the discussion, I pay special attention to the possibilities and impossibilities of top-down and bottom-up approaches in trying to obtain the desired results.

5.1.1 Providing the PBG with the required information

Any planning problem is characterized by the processes and resources, the time system, and the constraints that must be satisfied when processes are assigned to resources and time intervals. The constraints are expressed in terms of properties of the processes and resources or in terms of relations between processes and resources.

This problem structure has enabled us to develop a general method, based on attributed graphs, for the specification of problem instances. Nodes represent processes and resources. Attributes of nodes represent properties of the corresponding objects. Graph constructs like arcs, edges, and $K_{1,n}$'s are introduced to represent relations between objects. In Chapter 2, a detailed specification of properties of processes and resources and the various kinds of relations is given.

The instance specification method, however, can easily be extended or modified in order to deal with elements of planning problems that have not been discussed.

The instance specification method has served as a basis for the problem type specification method. A problem type is a set of instances that satisfy certain restrictions. In the type specification method, these restrictions are formulated as restrictions on the attributes and the structure of the instance graph.

In this thesis, I have not discussed methods for providing the PBG with information about desired representations and manipulations. A top-down approach seems appropriate in designing such methods.

There is only a limited number of representations that will occur on a planning board: Gantt charts, inventory graphs, data tables, the 'views' of the instance graph as introduced in Section 2.2.7, and perhaps some novel representation mechanisms like Jones's [1988] three-dimensional Gantt chart. The desires of a user of a planning board with respect to representations will be formulated in terms of the representation mechanism and the elements of an instance or a plan that must be represented. In the case of a timetabling problem for a school, the planner may want two different Gantt charts, one with teachers and one with classrooms along the vertical axis. In a production scheduling problem, the planner may desire a data table in which for each process the name, the size, the release time, and the due date are given.

With respect to manipulations the situation is similar. There are only a few essentially different kinds of manipulations. Any planning board must support manual plan construction. Therefore, a planner must be enabled to *insert* a process on the Gantt chart, to *move* a process from one position on the Gantt chart to another, to *shift* a process in time, and to *remove* a process from the Gantt chart. Furthermore, any planning board should be able to automatically *construct* a plan, to *complete* a partial plan, and to *evaluate* and *improve* a given plan. Beside the manipulations that are required for creating and modifying plans, a planning board must, of course, support standard graphical manipulations in order to enable a planner to get the best views of the problem and the plan. All planning boards, irrespective of the problem type for which they are designed, should offer the mentioned manipulations. However, a further specification of the desired manipulations may be necessary, especially when they are related to the advisor functions of a planning board. In the case of the timetabling problem, the planner may want to ask the planning board to remove all lessons given by a particular teacher from the plan, and to complete the resulting partial plan without reassigning any of the lessons to that teacher. In a machine scheduling problem, the planner may want to know if a given schedule can be improved without changing the processing order of the operations that are scheduled for the next

four hours.

Knowledge of the specific characteristics of problem types is not required for the development of a language for transferring information about representations and manipulations to the PBG. The specification method introduced in Chapter 2 highlights those characteristics of problem types that are essential for the implementation of representations and manipulations. It would therefore be sufficient if the developed language allows for incorporating elements from the problem specification.

For example, in the type specification of the timetabling problem, we can identify two resource groups, *Teacher* and *Classroom*. A simple statement like

Gantt(*Teacher*)
Gantt(*Classroom*)

would then suffice to indicate that two Gantt charts are desired, one with teachers and one with classrooms along the vertical axis.

5.1.2 Procedures for supporting representations and manipulations

After the designer of a planning board has provided the PBG with information about the problem type and the desired representations and manipulations, this information must be processed and an initial version of a planning board must be produced. To a large extent, this comes down to generating customized implementations of procedures by tailoring the basic implementations of these procedures to the formulated desires.

The most important element in all procedures that occur in a planning board is the *plan*. The implementation of a procedure determines how a plan is represented or how a plan can be modified. If we want to develop powerful basic implementations, we must be able to refer to elements of plans in a uniform way for different problem types.

A general method for representing problem instances has already been discussed: the instance graph of Chapter 2. This instance graph can be extended in order to represent plans as well. One possible way is to introduce an assignment attribute for processes. This attribute contains the resources and the time intervals that a process is assigned to. If the assignment attribute contains the predecessors and successors of a process on the various resources as well, then the solution network as introduced in Chapter 3 can be constructed from information represented in the extended instance graph.

As soon as we have decided about a uniform way of storing information about plans, it is possible to develop general procedures for representing these plans. Such a procedure can be designed, for example, for drawing a Gantt chart. We have already discussed the possibilities of providing a PBG with the necessary

information about the problem type and the desires formulated by the planner. This information can be used to set certain parameters in the considered procedure, ensuring that the time system and the resources are represented properly on the axes. The information stored in the assignment attribute can then immediately be translated into rectangles drawn at the right position. Similar procedures can be formulated for data tables, inventory graphs, and other representation mechanisms.

The assignment attribute will also play an important role in procedures that support manipulations. Consider, for example, the 'move' manipulation, which involves indicating which process is to be moved, specifying its new position, and checking if the new assignment is feasible. Processes can be indicated in a uniform way for all planning problems, for example by mouse clicking. The same holds for the specification of their new positions, although several clicks may be required, one for each resource that the process will be assigned to, and at least one for the starting time of the process. When the process and its new position have been determined, the assignment attribute is updated correspondingly, and immediately the representations are updated as well.

Note, however, that it may not be possible to develop one subroutine for checking the feasibility of a plan that can be used in all possible problem situations. In case of the general scheduling problem, feasibility is tested by determining whether the solution network contains a cycle of positive length with infinite capacity. In timetabling problems, other methods will be much more effective. While the previously mentioned elements of the 'move' procedure can be dealt with by means of a proper handling of input parameters, the feasibility testing may require the incorporation of entire subroutines. Different subroutines must be developed for all problem classes that require a different treatment of feasibility testing.

The same holds when a manipulation invokes the use of a construction or improvement algorithm. Algorithms developed for the entire general problem class are bound to be very time consuming or to give bad results when applied to specific subclasses. It does not make sense to create a library with the most efficient algorithms for all possible problem types and to include this library in a PBG. The number of possible planning problems is so large that it will be impossible to fill the library to any reasonable extent. A bottom-up approach may be successful: we identify algorithms that are efficient for certain problem types, and adjust them in such a way that they can handle wider problem classes as well. In the end, this approach should lead to a limited number of fairly large problem classes. For each such problem class, a library of algorithms can be created. These algorithms must be sufficiently efficient for all problem types in the con-

sidered class.

In Chapters 3 and 4, I have shown how such a bottom-up approach works for a class of machine scheduling problems. A relatively straightforward generalization of the mathematical model that is used in many solution methods for the job shop problem makes it possible to handle much more general objective functions and constraints. Although the result of this approach is that generally applicable methods are obtained that may not be very efficient when applied to a specific problem type, the loss in efficiency can be reduced substantially by exploiting additional information about that problem type. If we know, for example, that the objective in a particular problem type is to minimize the makespan, then the general method can be customized for this problem by replacing the incorporated maximum cost flow procedure by a longest path procedure, and by using the efficient insertion procedure of Section 4.3.3 instead of a generally applicable insertion algorithm. Tailoring can thus be applied within subroutines, even if the subroutines cannot be applied to all planning problems.

5.2 Machine scheduling

5.2.1 The general scheduling problem

Research in machine scheduling is characterized by a huge number of very specific problem types, many of which are of only limited practical importance. In my research, I have studied a relatively large class of scheduling problems, and I have paid special attention to generalizations of existing models that allow for more realistic types of constraints and objective functions.

The general scheduling problem introduced in Chapter 3 includes several well studied models, such as single-machine, flow shop and job shop scheduling problems. One important common property of these problems is that solutions are characterized by the order in which operations are processed on the various machines. As soon as this processing order is determined, the starting times of the operations and the quality of the schedule, i.e., the makespan, can be computed efficiently. Because of this property local search methods, in which new solutions are obtained by making small modifications in the processing order, are very attractive for these problems. The general scheduling problem preserves this property, but at the same time it allows for several new kinds of constraints and objective functions.

The most fundamental generalization that we have obtained concerns the objective functions. In the scheduling literature, regular objective functions have received by far the most attention. Regular cost functions are non-decreasing in

the completion times of the operations. As a consequence, left-justified schedules in which each operation is started as early as possible are preferred. Left-justified schedules can be analyzed easily, which is one of the main reasons for the popularity of regular cost functions. In the general scheduling problem it is possible to formulate a large variety of non-regular objective functions. For example, any cost function that is convex piecewise linear in the starting time of an operation can be dealt with in the general scheduling problem.

The general scheduling problem allows for more general constraints as well. Release times and minimum delays can be considered as straightforward generalizations of the nonnegativity and precedence constraints in the job shop scheduling problem, but also deadlines and maximum delays occur in the general problem.

A final generalization concerns the machine requirements. In the job shop scheduling problem, each operation requires one particular machine. In the general scheduling problem, each operation must be performed on a given machine *set*. This property makes it possible to model problems like the open shop scheduling problem, in which certain operations are not allowed to be performed at the same time.

The mentioned generalizations are useful in several applications. In just-in-time management, for example, the preferred completion times are not as *early* as possible, but as *close* as possible to some given due date. Nonregular cost functions are used to model the penalty associated with a discrepancy between completion time and due date. Nonregular criteria may also apply when a given schedule must be adjusted, for example because new orders have arrived or because a machine has broken down. The quality of the new schedule is not only expressed in terms of the actual objective function, but also in terms of the difference between the original schedule and the new schedule. When there is a large deviation from the original schedule, the new schedule is valued less than when the deviation is small. Taking the deviation into account results in a non-regular objective function even if the originally formulated cost function is regular.

I think, however, that the main contribution is not simply the formulation of a general model that can be applied in practical situations. More important is the framework that is offered for studying problems with difficult but realistic characteristics, such as maximum delays, deadlines, and non-regular objective functions. These characteristics have received too little attention in the machine scheduling community.

5.2.2 Solution methods

A relatively small price is to be paid for all the generalizations. For a given processing order of the operations in a job shop scheduling problem, the starting times can be obtained by solving a *longest path* problem; for the general scheduling problem a *maximum cost flow problem* in a (reduced) solution network must be found.

I have discussed two kinds of solution methods for instances of the general scheduling problem, construction methods and local search methods.

In the first type of construction method, dispatching, solutions are obtained by selecting the operations one after the other and positioning them after all previously selected operations. In insertion methods, operations are also selected one after the other, but they can be positioned anywhere. For both construction methods, I have formulated conditions under which feasible solutions are guaranteed to be obtained. Although insertion methods offer more freedom, and therefore are expected to give better solutions, their time requirement may make them less interesting. For a special subclass of the general scheduling problem, in which no deadlines or maximum delays occur and the objective is simply to minimize the makespan, an efficient implementation is presented. This implementation is shown to give substantially better results than any dispatching algorithm. I have mentioned also a third type of construction method, in which the disjunctive constraints are considered one after the other and for each constraint it is decided which of the two concerned operations is performed first. This method has not been studied in detail.

With respect to local search methods, most attention has been paid to studying the swap neighborhood, which allows for reversing the processing order of two consecutive operations or, equivalently, reorienting an arc in the reduced solution network. Generalizing a well known result for the job shop scheduling problem, it is shown that swapping an arc that does not occur in the maximum cost flow in the solution network cannot result in a solution of higher quality. For this reason, the maximum cost flow is referred to as the critical flow. Furthermore I have formulated conditions under which swapping a critical arc is guaranteed to result in a feasible solution. Another neighborhood, the jump neighborhood, has been introduced. This neighborhood allows for removing one operation from the processing order, and reinserting it in a completely different position. Computational experiments for the problem with makespan minimization as objective suggest that the jump neighborhood can be applied successfully. For job shop problems with machine sets and problems with open shop characteristics, better results are obtained with the jump neighborhood than with the swap neighborhood.

Although the general scheduling problem allows for deadlines and maximum delay constraints, these constraints cause a sizeable complication when a local search method is used. Normally only feasible solutions are allowed in local search methods. When deadlines or maximum delay constraints occur, the existence of feasible neighbors is not guaranteed. One possible way of circumventing this problem is to relax the hard constraints and include them in the objective function. I have discussed two relaxation methods, using Lagrangian relaxation and penalty costs, that may be applied in a local search context.

5.2.3 Suggestions for further research

Although local search methods seem to be particularly suitable for the general scheduling problem, other techniques can be applied as well. For the job shop problem, for example, several solution methods have been developed that use the disjunctive programming formulation that was at the basis of our discussion. Branch-and-bound methods (see, e.g., Applegate and Cook [1991], or Brucker et al. [1994]) have been applied more or less successfully to the job shop problem. The relative success of these algorithms is to some extent due to the lower bounding procedures that are applied. For a branch-and-bound algorithm for the general scheduling problem to be succesful, fast and good lower bounding procedures are required. Much effort has been put in obtaining such procedures for the job shop problem, but these cannot readily be applied to the general scheduling problem, because they rely heavily on the one-dimensional character of the makespan as objective function. For a similar reason, the well known shifting bottleneck heuristic of Adams et al. [1988], cannot be applied directly to the general scheduling problem. Nevertheless, it may still be worthwhile to investigate the possibilities of such other solution methods.

The key result of Chapter 3 is that as soon as a decision about the processing order has been made, the corresponding starting times of the operations can be computed efficiently. This result may be useful for even more general problems as well. One important generalization would involve allowing an operation to be performed on any of a given collection of machines or machine sets. These kinds of machine requirements occurs in, e.g., multiprocessor job shop scheduling and vehicle routing problems. In these problems, it is not sufficient to determine the processing order of the operations; it must also be decided on which machine sets the various operations are performed. As soon as both kinds of decisions have been made, again a solution network can be created, and the corresponding maximum cost flow problem can be solved in order to obtain the starting times of the operations. Problems in which multiple time windows occur can be tackled in a similar way. Besides determining the processing order, one must also de-

cide about the time intervals in which the various operations are to be performed before the starting times can be obtained.

Before considering such further generalizations, we must first try to develop truly efficient local search methods for instances of the general scheduling problem itself. Interesting research topics are related to finding the most efficient algorithms for solving maximum cost flow problems in solution networks, to designing efficient methods for updating a maximum cost flows after a small modification of the network, and to the early identification of interesting swaps. Furthermore, more research must be conducted in the area of relaxation techniques before we can apply local search successfully to problems with hard constraints like maximum delays and deadlines.

Appendix: The test set

For the computational experiments of Chapter 4, we have constructed a test set of 90 problem instances of nine different types. In all problem types, the objective is to minimize the makespan, and the only start-start constraints that occur are nonnegativity and ordinary precedence constraints.

JS instances are standard job shop problems.
JS2 instances are modified job shop problems in which each operation requires a machine set consisting of two machines.
JS3 instances are modified job shop problems in which each operation requires a machine set consisting of three machines.
FS instances are standard flow shop problems.
FS2 instances are modified flow shop problems in which each operation requires a machine set consisting of two machines.
FS3 instances are modified flow shop problems in which each operation requires a machine set consisting of three machines.
OS instances are standard open shop instances.
OSh instances are modified open shop instances in which each job consists of two groups of operations. All operations in the first group must be performed before all operations in the second group.
OSp instances are modified open shop instances in which there are two special machines, a 'preprocessor' and a 'postprocessor'. For each job, the operation on the preprocessor must be performed first, and the operation on the postprocessor must be performed last. The other operations can be performed in any desired order.

In the test set, 14 JS instances are included: the notorious 10×10 (10 jobs, 10 machines) instance of Fisher and Thompson [1963], the instances LA02 (10×5), LA19 (10×10), LA21, LA24, LA25 (15×10), LA27, LA29 (20×10), LA36,

LA37, LA38, LA39, and LA40 (15×15) of Lawrence [1984], and the (20×15) instance Tail17 of Taillard [1993].

For each JS instance, a JS2 instance is created in the following way. Let M be the number of machines in the JS instance. The number of machines in the JS2 instance is then $2M$. Each operation v has a machine set $\{\mu_1^v, \mu_2^v\}$, with $\mu_1^v \in \{1, \dots, M\}$ the machine that must process v in the original JS instance, and $\mu_2^v \in \{M+1, \dots, 2M\}$. The second machine in the machine set is selected according to a simple rule. This rule is best explained by means of an example. In Table .1, the machines $\mu_2^v \in \{6, \dots, 10\}$ are given for all operations v in a 7×5 JS2 instance.

The JS3 instances are obtained in a similar way. Let N be the number of jobs and M be the number of machines in a JS instance. The number of machines in the corresponding JS3 instance is then $3M + 2N$. Each operation v has a machine set $\{\mu_1^v, \mu_2^v, \mu_3^v\}$, with $\mu_1^v \in \{1, \dots, M\}$ the machine that must process v in the original JS instance, and $\mu_2^v \in \{M + 1, \dots, M + 2N\}$ and $\mu_3^v \in \{M + 2N + 1, \dots, 3M + 2N\}$ selected according to a simple rule. Again, this rule is illustrated best by the 7×5 example in Table .1.

2nd machine in JS2					2nd and 3rd machine in JS3				
6	7	8	9	10	6, 20	13, 21	12, 22	19, 23	11, 24
10	6	7	8	9	7, 25	14, 26	6, 27	13, 28	12, 29
9	10	6	7	8	8, 24	15, 20	7, 21	14, 22	6, 23
8	9	10	6	7	9, 29	16, 25	8, 26	15, 27	7, 28
7	8	9	10	6	10, 23	17, 24	9, 20	16, 21	8, 22
6	7	8	9	10	11, 28	18, 29	10, 25	17, 26	9, 27
10	6	7	8	9	12, 22	19, 23	11, 24	18, 20	10, 21

Figure .1: Machine requirements in the JS2 and JS3 instances obtained from a JS instance with 7 jobs and 5 machines.

The eight FS instances in the test set are the relatively small CAR1 up to CAR8 of Carlier [1978]. The FS2 and FS3 instances are obtained from these flow shop problems in the same way as the JS2 and JS3 instances were obtained from job shop instances. Note that the FS2 and FS3 instances cannot be called flow shop instances anymore. Only the first machines in the machine sets exhibit a flow shop character.

The eight OS instances in the test set are T31, T38, T39, T40 (10×10), T45, T49, T50 (15×15), and T52 (20×20) by Taillard [1993].

For each OS instance, an OSh instance is created in the following way. Let M be the number of machines. The operations of each job are divided into two groups. The first group contains the operations that must be processed by one of the machines in $\{1, \ldots, \lfloor M/2 \rfloor\}$, the second group contains the operations that must be processed by one of the other machines. All operations in the first group have to be completed before any of the operations in the second group can be started. There are no restrictions on the order in which operations within one group are to be performed. The precedence graph for an arbitary job consisting of 5 operations in an OSh instance is depicted in Figure .2.a. For each job, we introduce a dummy operation with zero processing time.

OSp instances are created from OS instances in the following way. Machine 1 is designated to be the preprocessor and machine M is the postprocessor. For each job, the operation that must be performed on the preprocessor is processed first. Then the operations on the machines 2 up to $M - 1$ are to be performed in any desired order. Finally, the postprocessor finishes the job. The precedence graph for an arbitary job consisting of 5 operations in an OSp instance is depicted in Figure .2.b.

Figure .2: **a.** A job in an OSh instance. **b.** A job in an OSp instance.

The JS, FS, and OS instances are well known benchmark instances. Optimal solution values are known for most of these instances, and for the hardest instances good lower and upper bounds exist. The instances of the other problem types are newly created, and upper bounds for these instances are obtained by performing many long runs of our taboo search algorithm with both the swap and the jump neighborhood. The 'best known solution values' obtained in this way are not guaranteed to be the optimal values. In fact, they may be substantially higher than these optimal values. However, since we are mainly interested in determining the relative quality of different solution methods, the quality of the obtained upper bounds is sufficient for our analysis.

References

J. Adams, E. Balas, D. Zawack (1988). The shifting bottleneck procedure for job shop scheduling. *Management Science* **34**, 391-401.

R.K. Ahuja, T.L. Magnanti, J.B. Orlin (1993). *Network Flows - Theory, Algorithms, and Applications*. Prentice Hall, Englewood Cliffs.

S.B. Akers (1956). A graphical approach to production scheduling problems. *Operations Research* **4**, 244-245.

M. Dell'Amico, M. Trubian (1993). Applying tabu search to the job-shop scheduling problem. *Annals of Operations Research* **41**, 231-252.

J.M. Anthonisse, K.M. van Hee, J.K. Lenstra (1988). Resource-constrained project scheduling: an international exercise in DSS development. *Decision Support Systems* **4**, 249-257.

J.M. Anthonisse, J.K. Lenstra, M.W.P. Savelsbergh (1988). Behind the screen: DSS from an OR point of view. *Decision Support Systems* **4**, 413-419.

D. Applegate, W. Cook (1991). A computational study of the job-shop scheduling problem. *ORSA Journal on Computing* **3**, 149-156.

K.R. Baker, G.D. Scudder (1990). Sequencing with earliness and tardiness penalties. *Operations Research* **38**, 22-36.

M. Bartusch, R.H. Möhring, F.J. Radermacher (1988). Scheduling project networks with resource constraints and time windows. *Annals of Operations Research* **16**, 201-240.

M.S. Bazaraa, J.J. Jarvis, H.D. Sherali (1990). *Linear Programming and Network Flows,* 2nd ed. Wiley, New York.

J. Błażewicz, W. Cellary, R. Słowiński, J. Weglarz (1986). Scheduling under resource constraints-deterministic models. *Annals of Operations Research* **7**.

H. Bräsel, T. Tautenhahn, F. Werner (1993). Constructive heuristic algorithms for the open shop problem. *Computing* **51**, 95-110.

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, B. Sievers (1994). A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics* **49**, 107-127.

J. Carlier (1978). Ordonnancements à contraintes disjonctives. R.A.I.R.O. Recherche opérationelle / Operations Research 12, 333-351.

H. Fisher, G.L. Thompson (1963). Probabilistic learning combinations of local job-shop scheduling rules, in: J.F. Muth, G.L. Thompson (eds.), *Industrial Scheduling*, Prentice Hall, Englewood Cliffs.

M.L. Fisher (1985). Interactive optimization. *Annals of Operations Research* **5**, 541-556.

M.L. Fisher, M.B. Rosenwein (1989). An interactive optimization system for bulk-cargo ship scheduling. *Naval Research Logistics Quarterly* **36**, 27-42.

H.L. Gantt (1919). *Organizing for Work.* Harcourt, Brace and Howe, New York.

T. Gonzales, S. Sahni (1976). Open shop scheduling to minimize finish time. *Journal of the Association for Computing Machinery* **23**, 665-979.

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

R.D. Hurrion (1986). Visual interactive modelling. *European Journal of Operational Research* **23**, 281-287.

J.R. Jackson (1955). Scheduling a production line to minimize maximum tardiness, Research Report 43, Management Science Research Project, University of California, Los Angeles.

P. Jackson, J.A. Muckstadt, C.V. Jones (1989). COSMOS: a framework for a computer-aided logistics system. *Journal of Manufacturing and Operations Management.* **2**, 122-148.

S.M. Johnson (1954). Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly* **1**, 61-98.

C.V. Jones, W.L. Maxwell (1986). A system for manufacturing scheduling with interactive computer graphics. *IIE Transactions* **18**, 298-303.

C.V. Jones (1988). The three-dimensional Gantt chart. *Operations Research* **36**, 891-903.

C.V. Jones (1990). An introduction to graph-based modeling systems, Part I: Overview. *ORSA Journal on Computing* **2**, 136-151.

C.V. Jones (1992). User interfaces, in: E.G. Coffman Jr., J.K. Lenstra, A.H.G. Rinnooy Kan (eds.), *Handbooks in Operations Research and Management Science; Vol. 3: Computing*, North-Holland, Amsterdam.

C.V. Jones (1994). Visualization and optimization. *ORSA Journal on Computing* **6**, 221-257.

P.J.M. van Laarhoven, E.H.L. Aarts, J.K. Lenstra (1992). Job shop scheduling by simulated annealing. *Operations Research* **40**, 113-125.

E.L. Lawler (1973). Optimal sequencing of a single machine subject to precedence constraints. *Management Science* **19**, 544-546.

E.L. Lawler, J.K. Lenstra, A.H.G.Rinnooy Kan, D.B. Shmoys (1993).Sequencing and scheduling: algorithms and complexity, in: S.C. Graves, A.H.G. Rinnooy Kan, P.H. Zipkin (eds.), *Handbooks in Operations Research and Management Science; Vol. 4: Logistics of Production and Inventory*, North-Holland, Amsterdam.

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

R. Lioce, C. Martini (1995). Heuristic methods for machine scheduling problems with processor sets: a computational investigation. Memorandum COSOR 95-10, Eindhoven University of Technology.

H. Matsuo, C.J. Suh, R.S. Sullivan (1988). A controlled search simulated annealing method for the general jobshop scheduling problem. Working Paper 03-04-98, Graduate School of Business, Univ. of Texas, Austin.

N.A. Moreira, R.C. Oliveira (1991). A decision support system for production planning in an industrial unit. *European Journal of Operational Research* **55**, 319-328.

M. Nawaz, E.E. Enscore Jr, I. Ham (1983). A heuristic algorithm for the m-machine, n-job flow-shop sequencing problem. *Omega* **11**, 91-95.

W.P.M. Nuijten (1994). *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach.* Ph.D. Thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology.

S.S. Panwalkar, W. Iskander (1977). A survey of scheduling rules. *Operations Research* **25**, 45-91.

B. Roy, B. Sussmann (1964). Les problèmes d'ordonnancement avec contraintes disjonctives. Note DS no. 9 bis, SEMA, Montrouge.

W.E. Smith (1956). Various optimizers for single-stage production. *Naval Research Logistics Quarterly* **3**, 59-96.

E. Taillard (1993). Benchmarks for basic scheduling problems. *European Journal of Operational Research* **64**, 278-285.

E.R. Tufte (1983). The Visual Display of Quantitive Information. Graphics Press, Cheshire.

E.R. Tufte (1990). Envisioning Information. Graphics Press, Cheshire.

R.J.M. Vaessens, E.H.L. Aarts, J.K. Lenstra (1994). Job shop scheduling by local search. Memorandum COSOR 94-05, Eindhoven University of Technology.

F. Viviers (1983). A decision support system for job shop scheduling. *European Journal of Operational Research* **14**, 95-103.

M. Wennink, M.W.P. Savelsbergh (1994). A planning board generator - part I: problem instances and types. Memorandum COSOR 94-42, Eindhoven University of Technology.

M. Wennink, R. Vaessens (1995). An efficient insertion algorithm for scheduling problems with multiprocessor operations. Working paper, Eindhoven University of Technology.

F. Werner, A. Winkler (1995). Insertion techniques for the heuristic solution of the jobshop problem. *Discrete Applied Mathematics* **58**, 191-211.

A.P. Woerlee (1991). *Decision Support Systems for Production Scheduling.* Ph.D. Thesis, Econometric Institute, Erasmus University Rotterdam.

Samenvatting

Processen die zich afspelen in de tijd kunnen vaak op elegante wijze gerepresenteerd worden in een zogenaamde Gantt chart. Op bladzijde 1 van dit proefschrift wordt een voorbeeld van een Gantt chart gegeven. Het gaat daarbij om een tijdschema voor een muziekfestival. Op de verticale as staan de vier podia waarop optredens plaatsvinden en op horizontale as staat de tijd van vier uur 's middags tot twee uur 's nachts uitgezet. De rechthoeken in de Gantt chart geven aan welke artiesten waar en wanneer optreden. In het algemeen worden op de verticale as van een Gantt chart *hulpmiddelen* gerepresenteerd en de horizontale as is de tijdas. De rechthoeken geven een toewijzing van *processen* aan hulpmiddelen en tijdintervallen weer.

In dit proefschrift besteed ik aandacht aan planborden, interactieve planningsystemen die de Gantt chart als belangrijkste representatie hanteren. Planborden zijn instrumenten die gebruikt kunnen worden bij het oplossen van een grote verscheidenheid van problemen. Voorbeelden zijn het maken van een lesrooster, waarbij lessen moeten worden toegewezen aan leraren, klaslokalen en lesuren, en het bepalen van een produktieschema, waarbij operaties door machines moeten worden uitgevoerd in bepaalde tijdintervallen.

Ik beschouw een planbord niet alleen als een 'assistent' die een planner de mogelijkheid biedt om plannen te creëren en op te slaan om ze later weer aan te kunnen passen. Een planbord moet ook als 'adviseur' kunnen optreden door zelf plannen te genereren en mogelijke verbeteringen in een bestaand plan aan te geven.

Hoewel planborden gebruikt kunnen worden voor problemen uit heel verschillende toepassingsgebieden, zijn er toch veel overeenkomsten in de wijze waarop ze ingericht zijn en in de structuur van de problemen waarvoor ze bedoeld zijn. Het gaat er altijd om een goede toewijzing van processen aan hulpmiddelen en tijdintervallen te vinden. Dit heeft als gevolg dat op verschillende planborden vaak dezelfde representaties worden aangeboden en dezelfde manipulaties kun-

nen worden uitgevoerd. Het uitgangspunt van dit onderzoek was dat het mogelijk moet zijn om die gemeenschappelijke kenmerken te benutten en het ontwikkelen van planborden te vereenvoudigen. Daarbij was de ontwikkeling van een planbordgenerator (PBG) het uiteindelijke doel. Een PBG moet, op basis van informatie over het probleemtype waarvoor een planbord ontwikkeld moet worden en de wensen van de uiteindelijke gebruiker van het planbord, automatisch een eerste versie van een planbord kunnen genereren.

Twee conflicterende doelstellingen komen naar voren als we de functionaliteit van een PBG beschouwen. Aan de ene kant moet een PBG een zo groot mogelijke verscheidenheid van planningsproblemen ondersteunen; aan de andere kant moeten de planborden die gegenereerd worden voor bepaalde probleemtypen goed uitgerust zijn voor de specifieke eigenschappen van die problemen.

Een aanpak die het mogelijk maakt om een juiste balans te vinden tussen beide doelstellingen, is de volgende. Voor elke procedure die in een planbord gebruikt wordt bevat de PBG een basis-implementatie. Op grond van de probleemspecifieke informatie die aan een PBG wordt aangeboden wordt die implementatie verder toegesneden op het probleemtype waarvoor een planbord ontwikkeld moet worden. Dit toesnijden kan gepaard gaan met het bepalen van de juiste waarden van bepaalde parameters of het opnemen van een van te voren gedefinieerde subroutine.

Een belangrijke rol is daarbij weggelegd voor de probleemspecificatie. Een PBG zal slechts in staat kunnen zijn om een goed planbord te genereren voor een bepaald probleemtype als dat probleemtype op een goede wijze is gespecificeerd. Het is dus essentieel dat er een specificatiemethode is die het mogelijk maakt om die eigenschappen van probleemtypen te beschrijven die van belang zijn voor de uiteindelijke invulling van de procedures die representaties en manipulaties ondersteunen.

In hoofdstuk 2 behandel ik zo'n specificatiemethode. Eerst wordt er een methode voor het specificeren van probleeminstanties geïntroduceerd. Deze methode maakt gebruik van de algemene structuur van planningsproblemen: processen moeten worden toegewezen aan hulpmiddelen en tijdintervallen waarbij aan een aantal restricties voldaan moet worden. We kunnen een instantie dan ook beschrijven door aan te geven welke processen en hulpmiddelen er zijn en aan welke restricties toewijzingen van processen aan hulpmiddelen en tijdintervallen moeten voldoen. Deze informatie kan verwerkt worden in een zogenaamde instantiegraaf. In een instantiegraaf worden processen en hulpmiddelen gerepresenteerd door knopen, en hun eigenschappen worden weergegeven in termen van attributen van die knopen. De beperkingen waaraan plannen moeten voldoen worden, voor zover ze niet in termen van eigenschappen van individuele proces-

sen of hulpmiddelen kunnen worden uitgedrukt, gerepresenteerd door kanten, pijlen of andere graafstructuren die relaties tussen verschillende objecten kunnen weergeven. Omdat een probleemtype gezien kan worden als een verzameling instanties die aan bepaalde beperkingen voldoen, kan een probleemtype vervolgens gespecificeerd worden door beperkingen op de instantiegraaf te formuleren.

Het is dus mogelijk gebleken om de vele overeenkomsten in de structuur van planningsproblemen te benutten bij het ontwikkelen van een algemeen toepasbare specificatiemethode. Voor een aantal procedures die in een planbord gebruikt worden is zo'n algemene aanpak ook mogelijk. Voor die procedures kan dan een basis-implementatie in een PBG opgenomen worden die door het eenvoudig invullen van de juiste parameterwaarden toegesneden kan worden op bepaalde probleemtypen. In andere gevallen is zo'n 'top-down' benadering niet mogelijk. Vooral als het gaat om de algoritmische ondersteuning van manipulaties zullen zulke algemeen toepasbare procedures leiden tot een te geringe efficiëntie. Omdat het aantal probleemtypen vrijwel oneindig is, is de alternatieve aanpak waarbij voor elk probleemtype de meest geschikte algoritmen in een bibliotheek worden opgenomen evenmin te gebruiken. We kunnen echter proberen om een beperkt aantal redelijk brede probleemklassen aan te wijzen en voor elke klasse algoritmen te ontwikkelen die voldoende efficiënt zijn voor alle problemen in die klasse.

Een 'bottom-up' benadering lijkt geschikt om tot een goede definitie van zulke probleemklassen te komen. We beginnen met een probleemtype waarvoor efficiënte algoritmen ontwikkeld zijn of ontwikkeld kunnen worden, en proberen dan een bredere probleemklasse te vinden waarvoor dezelfde algoritmen, in licht aangepaste vorm, ook gebruikt kunnen worden. Zodra de gewenste efficiëntie niet meer haalbaar blijkt, wordt afgezien van verdere generalisatie.

Een voorbeeld van zo'n bottom-up benadering is te vinden in hoofdstuk 3. Als uitgangspunt is het bekende job shop scheduling probleem genomen. Veel oplossingsmethoden voor dit probleem maken gebruik van de eigenschap dat slechts de volgordes waarin de operaties op de verschillende machines uitgevoerd worden bepaald moeten worden. Zodra dit is gedaan kunnen de bijbehorende starttijden gevonden worden door een langste-padprobleem op te lossen. Het langste-padprobleem is een speciaal geval van het probleem van het vinden van een stroom van maximale kosten. Deze observatie heeft geleid tot de formulering van het 'general scheduling probleem'. De belangrijkste eigenschap van het general scheduling probleem is dat wederom slechts de volgordes waarin de operaties op de verschillende machines worden uitgevoerd bepaald moeten worden, waarbij de bijbehorende starttijden nu echter gevonden kunnen worden door een stroom van maximale kosten in een netwerk te vinden.

Het general scheduling probleem generaliseert het job shop scheduling probleem op vele manieren. Zo is het niet meer noodzakelijk dat elke operatie verwerkt wordt op één machine; we kunnen ook omgaan met situaties waarin een operatie op een aantal machines tegelijk wordt uitgevoerd. Bovendien kunnen in het general scheduling probleem naast de bekende niet-negativiteits- en precedentiebeperkingen ook vroegst mogelijke starttijden, uiterste voltooiingstijden en grenzen op de duur van de periode tussen twee operaties meegenomen worden. De belangrijkste generalisatie betreft echter de doelstellingsfunctie. Binnen het general scheduling probleem is het mogelijk om verscheidene niet-reguliere kostenfuncties te formuleren.

Oplossingsmethoden die toegepast kunnen worden op het job shop scheduling probleem kunnen op eenvoudige wijze geschikt gemaakt worden voor het general scheduling probleem. In hoofdstuk 4 worden constructieve en lokale zoekmethoden behandeld. Ik geef aan onder welke voorwaarden toegelaten oplossingen gegarandeerd kunnen worden gevonden met behulp van constructieve methoden die gebruik maken van dispatching- of insertietechnieken. Voor een bepaalde deelklasse van het general scheduling probleem wordt aangetoond dat de beste insertiepositie voor een operatie op efficiënte wijze gevonden kan worden. Op het gebied van lokale zoekmethoden heb ik aandacht besteed aan twee soorten buurruimtes. In de 'swap'-buurruimte worden buuroplossingen gevonden door de volgorde waarin twee operaties die na elkaar op een machine worden uitgevoerd om te keren. Ik geef aan onder welke voorwaarden zo'n omkering toegelaten is en tot een verbetering kan leiden. In de 'jump'-buurruimte worden buren verkregen door een operatie uit een oplossing te verwijderen en op een andere positie weer in te voegen. Voor de klasse van problemen waarvoor de efficiënte insertiemethode is ontwikkeld zijn experimenten uitgevoerd die duidelijk maken dat de jump-buurruimte vaak betere resultaten geeft dan de swap-buurruimte, vooral als operaties op meer machines tegelijkertijd moeten worden uitgevoerd.

Lokale zoekmethoden kunnen niet zonder meer worden toegepast op problemen waarin bepaalde moeilijke beperkingen, zoals bijvoorbeeld uiterste voltooiingstijden, voorkomen. Het vinden van een toegelaten oplossing is voor dergelijke problemen al $\mathcal{NP}$-lastig, en als er al een toegelaten oplossing gevonden kan worden, dan kan de toegelatenheid van buuroplossingen niet worden gegarandeerd. Zoals ik aan het eind van hoofdstuk 4 aangeef, is het misschien mogelijk om tot redelijke oplossingen te komen door gebruik te maken van Lagrangiaanse relaxatie of strafkostenmethoden.

De twee onderdelen die ik in dit proefschrift behandel, probleemspecificatie en algoritmische ondersteuning van manipulaties voor een deelklasse van de

klasse van planningsproblemen, zijn niet voldoende om tot daadwerkelijke implementatie van een PBG te komen. In hoofdstuk 5 bespreek ik op informele wijze hoe andere onderdelen aangepakt kunnen worden. Bovendien beschrijf ik daar hoe de resultaten die zijn verkregen voor het general scheduling probleem bijdragen aan de theorie van de machinevolgordeproblemen en geef ik enkele suggesties voor verder onderzoek.

Stellingen

behorende bij het proefschrift

Algorithmic Support for Automated Planning Boards

van

Marc Wennink

I

Het 'general scheduling problem' zoals dat is gedefinieerd in hoofdstuk 3 van dit proefschrift is een machinevolgordeprobleem in de letterlijke betekenis van het woord.

II

Zij Q een verzameling van n paren (a_i, b_i), met $a_i, b_i \in \mathbb{N}$ voor $i = 1, \ldots, n$. Een deelverzameling $Q' \subseteq Q$ waarvoor $\max_{i \in Q'} a_i + \max_{i \in Q \setminus Q'} b_i$ minimaal is kan in $\mathcal{O}(n \log n)$ tijd gevonden worden.

M. WENNINK, R. VAESSENS (1995). An efficient insertion algorithm for scheduling problems with multiprocessor operations. Ongepubliceerd manuscript.

III

De insertietechniek kan met succes gebruikt worden in constructie- en verbeteringsmethoden voor een brede klasse van machinevolgordeproblemen (zie hoofdstuk 4 van dit proefschrift).

IV

Het vinden van de beste insertie van een gehele opdracht in een partiële oplossing van een open shop probleem is een sterk $\mathcal{NP}$-lastig probleem.

V

In veel verhandelingen over de evolutietheorie wordt het beeld geschetst van een verbeteringsproces dat heeft geleid tot een (voorlopig) hoogtepunt, de mens. Daarbij wordt voorbijgegaan aan het feit dat de meeste ontwikkelingen zich hebben afgespeeld in takken van de 'boom des levens' die niets van doen hebben met het kleine twijgje waaraan de mens hangt. Algoritmen voor optimaliseringsproblemen die gebaseerd zijn op een analogie met de evolutietheorie gaan uit van hetzelfde misverstand als zou er in de evolutie sprake zijn van een doelgerichte ontwikkeling en missen dan ook elke grond.

VI

De grootte van instanties van het handelsreizigersprobleem die opgelost kunnen worden is geen maatstaf voor de geboekte vooruitgang binnen de combinatorische optimalisering.

VII

Het is niet alleen goed voor de ontwikkeling van een promovendus om onderzoek te verrichten aan een andere universiteit dan die waar hij zijn opleiding heeft genoten; het komt ook de kwaliteit van een onderzoeksgroep ten goede als promovendi met verschillende achtergronden worden aangesteld.

VIII

a. Hoe compacter de formulering, hoe scherper de stelling.

b. .

IX

Fabrikanten die hun produkten aanprijzen met de term 'duurzaam', waar zij 'lang meegaand' bedoelen, appelleren op oneigenlijke wijze aan het milieubewustzijn van potentiële klanten.

O.S. TROMP. 1995. Towards sustainable quality. Proefschrift Rijksuniversiteit Groningen.

X

Dat de Witrus Igor Zjelezovski tijdens zijn laatste poging om een gouden medaille te halen op de Olympische Spelen veel minder aandacht kreeg dan de Amerikaan Dan Jansen, hoewel dat niet gerechtvaardigd werd door hun prestaties in voorgaande jaren, is te wijten aan de overheersing van het westerse perspectief in de internationale berichtgeving.