Resource Loading by Branch-and-Price Techniques

Research School for Operations
Management and Logistics

This thesis is number D-45 of the thesis series of the Beta Research School for Operations Management and Logistics. The Beta Research School is a joint effort of the departments of Technology Management, and Mathematics and Computing Science at the Eindhoven University of Technology and the Centre for Production, Logistics, and Operations Management at the University of Twente. Beta is the largest research centre in the Netherlands in the field of operations management in technology-intensive environments. The mission of Beta is to carry out fundamental and applied research on the analysis, design, and control of operational processes.

Dissertation committee

Chairman / Secretary	Prof.dr.ir. J.H.A. de Smit
Promotor	Prof.dr. W.H.M. Zijm
	Prof.dr. S.L. van de Velde (Erasmus University Rotterdam)
Assistant Promotor	Dr.ir. A.J.R.M. Gademann
Members	Prof.dr. J.K. Lenstra (Eindhoven University of Technology)
	Prof.dr. A. van Harten
	Prof.dr. G.J. Woeginger
	Dr.ir. W.M. Nawijn

Twente University **Press**

Publisher: Twente University Press
P.O. Box 217, 7500 AE Enschede
the Netherlands
www.tup.utwente.nl

Cover: branching of the Ohio river (satellite image)
Cover design: Hana Vinduška, Enschede
Print: Grafisch Centrum Twente, Enschede

ISBN 9036516609

RESOURCE LOADING
BY BRANCH-AND-PRICE TECHNIQUES

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof.dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 5 oktober 2001 te 16:45 uur

door

Elias Willem Hans
geboren op 28 juli 1974
te Avereest

Dit proefschrift is goedgekeurd door de promotoren:

prof.dr. W.H.M. Zijm
prof.dr. S.L. van de Velde

en de assistent-promotor:
dr.ir. A.J.R.M. Gademann

Voor mijn ouders
en Hana

Acknowledgements

If you would not be forgotten,
as soon as you are dead and rotten,
either write things worthy reading,
or do things worth the writing.

- Benjamin Franklin (1706-1790)

When, during the final stage of my graduation at the Faculty of Applied Mathematics at the University of Twente in late 1996, Henk Zijm approached me to inquire whether I would be interested in becoming a Ph.D. student, I immediately told him that that was the last thing on my mind. At that time I was developing a mathematical model of the liberalized electricity market in Western Europe at the Energy Research Foundation (in Dutch: ECN). Although my interest in developing mathematical models and solution techniques was sparked, my image of Ph.D. students consisted of people spending all their time in dusty libraries doing research for four years. Henk quickly persuaded me. Looking back, nearly five years later, I am very glad I accepted the opportunity he offered.

Many people contributed to this thesis. I thank them all. I like to thank the people below in particular.

In the first place I express my gratitude to Henk for giving me the opportunity to work under his inspiring supervision. I have learned a lot from him, and his enthusiasm was always a great motivation.

I consider myself very lucky to have had Noud Gademann as my supervisor. He has an amazing eye for detail, and an unremitting patience for repeatedly correcting my mistakes. Every meeting was a learnful experience, almost always yielding fruitful ideas. It has been most pleasant to work with him, and I hope we will be able to continue this in the coming years.

I thank Steef van de Velde, whose idea of using branch-and-price techniques for resource loading problems formed the starting point for this research. Although we met less often after he transferred to the Erasmus University, his ideas and comments contributed a lot to this thesis.

I thank Marco Schutten, who provided some great ideas, and Mark Giebels, with whom I had many instructive discussions. I also thank Tamara Borra,

who developed and implemented many resource loading heuristics during her graduation.

I express my gratitude to prof.dr. A. van Harten, prof.dr. J.K. Lenstra, dr.ir. W.M. Nawijn, and prof.dr. G.J. Woeginger for participating in the dissertation committee.

During the last year of my research I became a member of the OMST department (faculty of Technology and Management) under supervision of Aart van Harten. I thank all my OMST colleagues for offering a pleasant working environment, and I hope to be able to make my contribution to the work of this department in the coming years.

During my time in Twente I met the closest of my friends, with whom I still spend most of my free time. I thank Wim and Jeroen for the many occasions of playing pool and eating shoarma. I thank Richard for teaching me how to play the guitar, and for all the wonderful time singing and playing guitar together. The 'Grote Prijs van Drienerlo': been there, won that! I thank Henk Ernst for his useful comments on the thesis, and for all his help with the lay-out. I also thank Marco for all the time we spent after work, in the mensa, at the cinema, or playing tennis together.

Finally, very special thanks go to my family. In particular I thank my girlfriend Hana and my parents. I would not have been where I am now without their unconditional support.

Enschede, October 2001
Erwin Hans

Contents

Chapter 1

Introduction

It is easier to go down a hill than up,
but the view is from the top.

- Arnold Bennett (1867-1931)

1.1 Problem description

Many of today's manufacturing companies that produce non-standard items, such as suppliers of specialized machine tools, aircraft, or medical equipment, are faced with the problem that in the order processing stage, order characteristics are still uncertain. Orders may vary significantly with respect to routings, material and tool requirements, etc. Moreover, these attributes may not be fully known at the stage of order acceptance. Companies tend to accept as many customer orders as they can get, although it is extremely difficult to measure the impact of these orders on the operational performance of the production system, due to the uncertainty in the order characteristics. This conflicts with the prevailing strategy of companies to maintain flexibility and speed as major weapons to face stronger market competition (Stalk Jr. and Hout, 1988). Surprisingly enough, the majority of the companies in this situation do not use planning and control approaches that support such a strategy.

A typical example of a way to maintain flexibility in situations with a high data uncertainty is the cellular manufacturing concept (Burbidge, 1979). In cellular manufacturing environments, each cell or group corresponds to a group of, often dissimilar, resources. Usually these are a group of machines and tools, controlled by a group of operators. Instead of planning every single resource, the management regularly assigns work to the manufacturing *cells*. The detailed planning problem is left to each individual cell, thus allowing a certain degree of autonomy. The loading of the cells, the so-called *resource loading*,

imposes two difficulties. In the first place, the manufacturing cells should not be overloaded. Although often the cells have flexible capacity levels, i.e., they may use nonregular capacity, such as overtime, hired staff, or even outsourced capacity, a too high load implies that the underlying detailed scheduling problem within the cell can not be resolved satisfactorily. As a result, cells may not be able to meet the due dates imposed by management, or only at high cost as a result of the use of nonregular capacity. Performing a well-balanced finite capacity loading over the cells, while accounting for capacity flexibility (i.e., the use of nonregular capacity) where possible and/or needed, benefits the flexibility of the entire production system. In the second place, since products are usually not entirely manufactured within one cell, parts or subassemblies may have complex routings between the manufacturing cells. The management should thus account for precedence relations between product parts and other technological constraints, such as release and due dates, and minimal durations.

While much attention in research has been paid to detailed production planning (scheduling) at the operational planning level (short-term planning), and to models for aggregate planning at the strategic planning level (long-term planning), the availability of proper (mathematical) tools for the resource loading problem is very limited. The models and methods found in the literature for the operational and strategic level are inadequate for resource loading. Scheduling algorithms are rigid with respect to resource capacity but are capable of dealing with routings and precedence constraints. Aggregate planning models can handle capacity flexibility, but use limited product data, e.g., by only considering inventory levels. Hence they ignore precedence constraints or other technological constraints.

In practice, many large companies choose material requirements planning (MRP) based systems for the intermediate planning level (Orlicky, 1975). Although MRP does handle precedence constraints, it assumes fixed lead times for the production of parts or subassemblies which results in rigid production schedules for parts or entire subassemblies. A fundamental flaw of MRP is that the lead times do not depend on the amount of work loaded to the production system. There is an implicit assumption that there is always sufficient capacity regardless of the load (Hopp and Spearman, 1996). In other words, MRP assumes that there is infinite production capacity. The main reason for the lack of adequate resource loading methods may be that models that impose both finite capacity constraints and precedence constraints are not straightforward, and computationally hard to solve.

Research at a few Dutch companies (Snoep, 1995; Van Assen, 1996; De Boer, 1998) has yielded new insights with respect to the possibility of using advanced combinatorial techniques to provide robust mathematical models and algorithms for resource loading. These ideas are further explored in this thesis: we propose models that impose the aforementioned constraints, and, more importantly, we propose exact and heuristic solution methods to solve the resource

loading problem.

The manufacturing typology that we consider in this research is the make-to-order (MTO) environment. An MTO environment is typically characterized by a non-repetitive production of small batches of specialty products, which are usually a combination of standard components and custom designed components. The aforementioned difficulties with respect to the uncertainty of data in the order processing stage are typical of MTO environments. There is uncertainty as to what orders can eventually be acquired, while furthermore order characteristics are uncertain or at best partly known. Moreover, the availability of some resources may be uncertain. We present the resource loading models and methods in this thesis as tactical instruments in MTO environments, to support order processing by determining reliable due dates and other important milestones for a set of known customer orders, as well as the resource capacity levels that are required to load this set on the system. Since detailed order characteristics are not available or only partly known, we do not perform a detailed planning, but we do impose precedence constraints at a less detailed level, e.g., between cells. Once the order processing has been completed, the resource loading function can be used to determine the available resource capacity levels for the underlying scheduling problem. Note that in resource loading capacity levels are flexible, while in scheduling they are not. Hence resource loading can detect where capacity levels are insufficient, and solve these problems by allocating orders (parts) more efficiently, or by temporarily increasing these capacity levels by allocating nonregular capacity, e.g., working overtime. As a result, the underlying scheduling problem will yield more satisfactory results.

The remainder of this introductory chapter is organized as follows. In Section 1.2 we present a positioning framework for capacity planning to define resource loading among the different capacity planning functions, and to place this research in perspective. In Sections 1.2.1 and 1.2.2 we elaborate on the capacity planning functions order acceptance and resource loading. Subsequently, in Section 1.3 we discuss the production characteristics of the manufacturing environments under consideration. In Section 1.4 we illustrate the resource loading problem with an example. In Section 1.5, we discuss the existing tools as they occur in practice for the tactical planning level. We conclude this chapter with an outline of the thesis in Section 1.6.

1.2 A planning framework for make-to-order environments

The term capacity planning is collectively used for all kinds of planning functions that are performed on various production planning levels. The words capacity and resources are often used as substitutes in the literature. In this

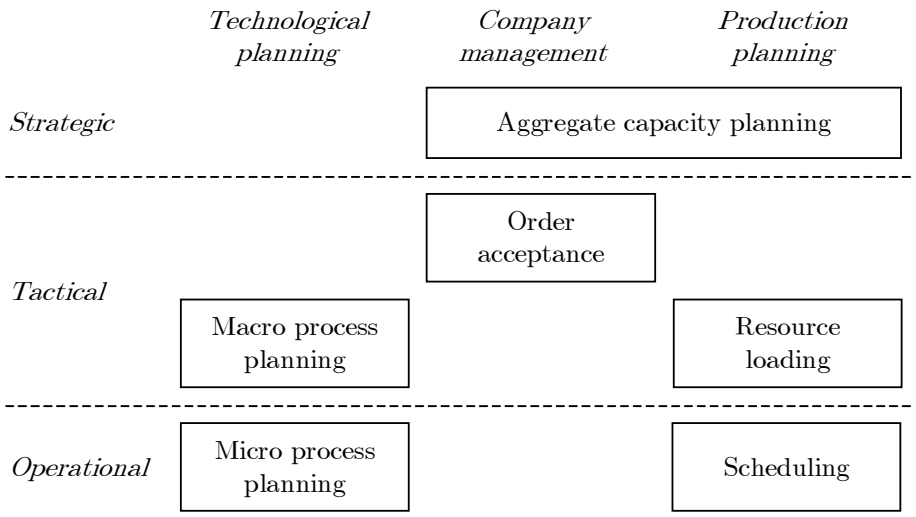

Figure 1.1: Positioning framework (from: Giebels et al., 2000).

thesis we make a clear distinction. While *resources* comprise machines, operators and tools, *capacity* comprises more, e.g., facilities, material handling systems, and factory floorspace. Although much research has been devoted to the topic of capacity planning, there exists no unambiguous definition in the literature. In this section we define the capacity planning functions, and position the resource loading function.

To be able to distinguish the capacity planning functions, we propose the *positioning framework* in Figure 1.1 (see also Giebels et al., 2000). In this framework, we vertically distinguish the three levels/categories of managerial or planning activities as proposed by Anthony (1965): strategic planning, tactical planning, and operational control (see also, e.g., Shapiro, 1993; Silver et al., 1998). Horizontally we distinguish three categories of planning tasks: technological planning, company management, and production planning. In this section we discuss the managerial and planning decisions in the framework relevant to the production area. Of course these decisions also have important interactions with other managerial activities, such as product development and business planning, but these are beyond the scope of this thesis. Furthermore, the technological planning tasks are not discussed in this thesis. We refer to Giebels (2000) for an extensive discussion of this subject.

Strategic planning involves long-range decisions by senior management, such as make-or-buy decisions, where to locate facilities, to determine the market competitiveness strategy, and decisions concerning the available machining capacity, or the hiring or release of staff (see, e.g., Silver et al., 1998). The basic

function of strategic planning is hence to establish a production environment capable to meet the overall goals of a plant. Generally, a *forecasting* module is used to forecast demand and other market information. This demand forecast, as well as additional process requirements, is used by a *capacity/facility planning module* to determine the need for additional machines or tools. The same analysis is performed by a *workforce planning* module to support personnel hiring, firing or training decisions. Finally, an *aggregate planning* module determines rough predictions for future production mix and volume. In addition, it supports other structural decisions, regarding for example which external suppliers to use, and which products/parts to make in-house (i.e., make-or-buy decisions, see Hopp and Spearman, 1996). Hence, aggregate planning is depicted in Figure 1.1 as both a company managerial activity, as well as a production planning activity. Linear programming (LP) often constitutes a useful tool that aims to balance resource capacity flexibility against inventory flexibility. Typically, LP models contain inventory balance constraints, product demand constraints, staff and machine capacity constraints, and an objective function that minimizes inventory costs and the costs of staff working overtime (see, e.g., Shapiro, 1993; Hopp and Spearman, 1996). Workforce planning is usually performed with LP models with similar constraints, but with an objective function that for example minimizes the costs of hiring and firing of staff.

Tactical planning on the medium-range planning horizon, is concerned with allocating sufficient resources to deal with the incoming demand, or the demand that was projected in the (strategic) aggregate planning module, as effectively and profitably as possible. The basic problem to be solved is the allocation of resources such as machines, workforce availability, storage and distribution resources (Bitran and Tirupati, 1993). Although the basic physical production capacity is fixed in the long-term strategic capacity plans, on the tactical planning level capacity can temporarily be increased or decreased between certain limits. Typical decisions in resource loading hence include utilization of regular and overtime labor, hiring temporary staff, or outsourcing parts. We have separated the tactical planning tasks in a managerial module, the (customer) order acceptance, and a production planning module: resource loading. *Order acceptance* is concerned with processing the immediate customer demand. On the arrival of each new customer order, a *macro process planning* step is executed to determine the production activities and the way they are performed (Sheng and Srinivasan, 1996; Van Houten, 1991). Hence, a new production *order* is subdivided into *jobs*, with precedence relations, estimated process times, and, when applicable, additional production related constraints. Using this analysis of the order characteristics, and the current state of the production system, orders are accepted or rejected based on strategic or tactical considerations. We discuss order acceptance in more detail in Section 1.2.1. *Resource loading* on the other hand is concerned with loading a given set of orders and determining reliable internal due dates and other important milestones for these customer

orders, as well as determining the resource capacity levels that are needed to process these orders and their constituting jobs. Resource loading is an important planning tool for the order processing, since it can establish the feasibility and suitability of a given set of accepted orders. In addition it may serve as a tool for determining reliable customer due dates for order acceptance. Silver et al. (1998) distinguish two strategies in medium-range planning: level and chase. For firms that are limited or even completely constrained with regard to their ability to temporarily adjust resource capacity levels (e.g., when the production process is highly specialized), inventories are built up in periods of slack demand, and decreased in periods of peak demand. This strategy is called *level* because of the constant workforce size. It is typical for make-to-stock manufacturers. Other firms that are able to adjust resource capacity levels (e.g., by hiring staff) and that face high inventory costs or a high risk of inventory obsolescence, pursue a strategy that keeps inventory levels at a minimum. This strategy is called *chase*, because production chases, or attempts to exactly meet demand requirements. Of course these situations are extremes, and in practice the strategy is usually a combination of the two. The resource loading methods presented in this thesis have been designed to account for temporary changes in resource capacity levels (by temporarily using nonregular capacity) for at least a part of production. The aim is to minimize the use of nonregular resources, and to meet order due dates as much as possible. We discuss resource loading in more detail in Section 1.2.2.

Finally, *operational planning* is concerned with the short-term scheduling of production orders that are passed on by the resource loading module. Before scheduling takes place, a *micro process planning* is performed to complete the process planning for the products in detail (Cay and Chassapis, 1997), to provide among other things the detailed data for the scheduling module. The resource loading module at the tactical level determines the (regular plus nonregular) operator and machine capacity levels available to scheduling. Baker (1974) defines *scheduling* as the allocation of resources over time to perform a collection of tasks. The resulting schedules describe the sequencing and assignment of products to machines. Scheduling is usually based on deterministic data. Although many intelligent algorithms have been proposed for the (often NP-hard) scheduling problems (see, e.g., Morton and Pentico, 1993; Pinedo, 1995; Brucker, 1995; Schutten, 1998), in practice scheduling systems are often based on simple dispatching rules.

We propose the term *capacity planning* to comprehend the planning activities aggregate planning, order acceptance, resource loading, and scheduling collectively. Hence, capacity planning comprises the utilization and the expansion or reduction of all capacity, as well as the planning of capacity on all managerial/planning levels. Although much research attention has been paid to the strategic level (e.g., facility planning, aggregate planning and workforce planning) and operational level (scheduling) of planning, the tactical planning level has remained rather underexposed. We focus entirely on this planning

level, and on resource loading in particular. In the next two sections we further discuss the two tactical capacity planning functions order acceptance and resource loading.

1.2.1 Order acceptance

In compliance with the strategic decisions taken in strategic planning, order acceptance is concerned with accepting, modifying, or rejecting potential customer orders in such a way, that the goals of management are met as much as possible. Although order acceptance is not a subject of research in this thesis, in this section we point out that the models and methods for resource loading as presented in this thesis may serve as effective tools in order acceptance.

In order processing the continuously incoming orders can be accepted or rejected for various reasons. Especially in make- or engineer-to-order manufacturing environments, orders may vary significantly with respect to their production characteristics. Orders can be initiated in many ways, they may be customer orders, internal orders, suborders, maintenance activities, or engineering activities. Because of the low repeat rates, small batches, and accompanying high product variety, orders in an order mix may vary significantly with respect to the planning goals, priorities, and constraints. This makes the execution of the resource loading and the order acceptance function more complex and asks for dynamic planning strategies that take the individual characteristics of the orders into account. Giebels et al. (2000) present a generic order classification model that is used to recognize these order characteristics. The classification model comprises seven generic dimensions that are of importance for determining the aims, priorities and constraints in the planning of the individual orders. An important dimension is, e.g., the *state of acceptance* (or *progress*) of a customer order. An order may, for example, still be in the state of order acceptance (i.e., the order is being drawn up), and the order prospects may still be uncertain, or it is already accepted and included in the capacity plans, or parts of the order have already been dispatched to the shop floor. Consequently, the state of acceptance shows the ease of cancelling the order or changing its due date. Another example of an order dimension is the *strategic importance* of the order, which, e.g., indicates that this customer order may result in a (profitable) follow-up order from the same customer.

The majority of the order dimensions concern the feasibility of a customer order and the impact of an order on the production system. In this context it is important that order processing is carried out in cooperation with production control. Ten Kate (1995) analyzes the coordinating mechanism between order acceptance and production control in process industries, where uncertainty plays a smaller role at the tactical planning level than in make-to-order environments. He notes that decisions taken by a sales department with respect to the acceptance of customer orders largely determine the constraints for the

scheduling function. Once too many orders have been accepted, it becomes impossible to realize a plan in which all orders can be completed in time. This stresses the need to consider both the actual load and the related remaining available capacity in order to be able to take the right order acceptance decisions. Hence, in process industries scheduling must provide information on which the order acceptance can base its decisions. Analogously, in make-to-order manufacturing environments resource loading must provide information on which the order acceptance can base its decisions. Resource loading can serve as a tool to measure the impact of any set of orders on the production system, by determining the required resource capacity levels and the resulting nonregular production costs by loading such a set of orders. Not only can such an analysis be used to determine what orders to accept from an order portfolio, it can also be used to determine realistic lead times, hence to quote reliable due dates/delivery dates for customer orders. Ten Kate (1995) distinguishes two approaches for the coordination mechanism for a combined use of customer order acceptance and resource loading (or scheduling in process industries). An order acceptance function and resource loading function that are fully integrated is referred to as the *integrated approach* for order acceptance. On the other hand, in the *hierarchical approach* these functions are separated, and the only information that is passed through is the aggregate information on the workload. In this thesis we focus on resource loading algorithms that do not contain order acceptance decisions, i.e., these algorithms involve loading a fixed set of orders. Hence, our algorithms can be used in a hierarchical approach for order acceptance.

The order classification makes clear that orders can not be treated uniformly in order acceptance. The seven dimensions proposed by Giebels et al. (2000) serve as a template to define the objectives and the constraints of the orders concerned. Ideally, in the resource loading problem all these dimensions are taken into account. The crux is that some characteristics are hard to quantify, which is required to impose these characteristics in any quantitative model of the resource loading problem. The resource loading methods presented in this thesis are restricted to measuring the impact of a set of orders on the production system. The methods determine whether a given set of orders *can* be loaded, given current resource capacity levels, and determine what additional nonregular resources are required to do so. Moreover, the methods determine whether order due dates can be met. In Chapter 8 we outline possible extensions of the methods in this thesis, to further support other order characteristics in order acceptance.

1.2.2 Resource loading

As mentioned before, resource loading is a tactical instrument that addresses the determination of realistic due dates and other important milestones for new customer orders, as well as the resource capacity levels that are a result

of the actual set of orders in the system. These resource capacity levels not only include regular capacity, but also overtime work levels, outsourcing, or the hiring of extra workforce. Hence, resource loading can in the first place be used in the customer order processing phase as an instrument to analyze the trade-off between lead time and due date performance on the one hand, and nonregular capacity levels on the other hand. Moreover, once the order acceptance is completed and the set of accepted orders is determined, resource loading serves as a tool to define realistic constraints for the underlying scheduling problem. In the first place it determines the (fixed) resource capacity levels available to the scheduling module. In addition, resource loading determines important milestones for the orders and jobs in the scheduling problem, such as (internal) release and due dates.

So far we have mentioned two important goals of resource loading: the determination of important milestones in time, and the determination of resource capacity levels. In fact, in resource loading, two clearly different approaches can be distinguished: the resource driven and the time driven planning (Möhring, 1984). In *resource driven* planning, the resource availability levels are fixed, and the goal is to meet order due dates. This can, e.g., be achieved by minimizing the total/maximum lateness, or by minimizing the number of jobs that are late. With *time driven* planning, the order due dates are considered as strict, i.e., as deadlines, and the aim is to minimize the costs of the use of nonregular capacity. Since time and costs are equally important at this tactical planning level, in practice both approaches should be used simultaneously (monolithic approach), or iteratively with some interaction (iterative approach). A resource loading tool that uses this approach is an effective tactical instrument that can be used in customer order processing to determine a collection of orders from the entire customer order portfolio, with a feasible loading plan, that yields the most for a firm. After the completion of the customer order processing phase, the resource capacity levels have been set, as well as the (external) release and due dates of the accepted collection of orders. These due dates are then regarded as deadlines in scheduling. In this thesis we present a model that handles time driven planning and resource driven planning simultaneously. An advantage of such an approach is that it can be adapted to time driven planning by regarding order due dates as deadlines, and it can be adapted to resource driven planning by fixing the resource availability levels.

With respect to the positioning framework discussed in Section 1.2, we remark that in the literature much attention has been paid to scheduling problems at a detailed (operational) level, and quite some research has been done in the area of the (strategic) long-term capacity planning. However, the (tactical) resource loading problem remains rather underexposed. The main reason for this is that the LP models that were developed for aggregate planning or workforce planning are not suitable for resource loading. These LP models were developed for a higher planning level, where precedence constraints between production steps, and release and due dates for customer orders are not considered.

The second reason for the lack of adequate resource loading methods is the unsuitability of most scheduling methods for resource loading. These methods require much detailed data, and it is not realistic to assume that all detailed data is already known at the time resource loading is performed. Not only is the progress of current production activities on the shop floor somewhat unpredictable in the long term, the engineering and process planning data that is required to provide the detailed routings and processing times has usually not been generated completely at the time that resource loading should be performed. Another disadvantage of scheduling methods is that they are inflexible with respect to resource capacity levels, which, all things considered, make them unsuitable for resource loading.

Although resource loading uses less detailed data than scheduling, it uses more detailed data than (strategic) aggregate planning. While aggregate planning considers products only, in resource loading the products are disaggregated into (product) orders, which are further disaggregated in jobs. The jobs are described in sufficient detail in macro process planning, such that resource loading can assign them to machine groups or work cells. The jobs may have mutual *precedence relations*, *minimal duration* constraints, and *release and due dates*, which makes resource loading problems much more difficult to solve than aggregate planning models. Ideally, resource loading accounts for as many of these additional constraints as possible, to ensure that when the jobs are further disaggregated (in tasks or operations) for the scheduling task, a feasible detailed schedule for the given loading solution can still be determined. Hence, by first performing a resource loading, the scheduling problem is made easier.

In the next section we describe the production characteristics of the manufacturing environments under consideration in more detail.

1.3 Production characteristics

1.3.1 The resource loading problem

We primarily address the resource loading problem in an MTO manufacturing environment. This problem is concerned with loading a known set of orders onto a job shop in an MTO manufacturing environment. The job shop consists of several machine groups, and several machine operators. The operators are capable of operating one or more machine groups, and may even be capable of operating more than one machine group at the same time. In resource loading both regular and nonregular capacity can be allocated. Regular operator capacity can be expanded with overtime capacity, or by hiring temporary staff. Machine capacity expansions are not allowed. Decisions regarding machine capacity expansions are assumed to be taken at a higher (strategic) level of decision making. Since machines are available for 24 hours a day, all available

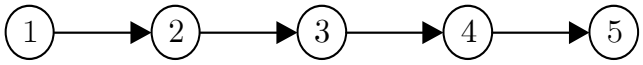

Figure 1.2: Example of linear precedence relations in resource loading.

machine capacity is considered as regular capacity. Machine capacity can thus not be expanded temporarily - if the available machine capacity limits production, the only option to complete all work in time is to outsource. Note that the use of machine time may be constrained by the limited availability of simultaneously needed operators. The costs of the various sources of nonregular capacity may vary.

The orders are subdivided into jobs. These jobs can later be subdivided into activities or operations in the underlying detailed scheduling problem. Jobs have linear precedence relations such as in Figure 1.2. A job may start processing as soon as its predecessor has completed. A job may have a minimal duration. A minimal duration of a job is usually a result of technical limitations of resources, e.g., in a situation when no more than two persons can jointly perform a certain activity. Optionally a minimum time lag between jobs can be taken into account. This is, e.g., the case when a manufacturing company uses a 'one-job-per-period' policy, which prescribes that no more than one job of an order can be processed per period. In this case there is a minimum time of one period between the jobs of an order. Although this policy is quite common in practice, it may have a strong negative impact on the order lead time, particularly when the periods are long.

In the production process jobs require machine and operator capacity at the same time. The processing times of the jobs are estimates, and assumed to be determined in a macro process planning that is performed upon arrival of the order in the customer order processing. The available resource capacity levels of the operators (regular and nonregular) and the machines are pre-specified.

While at the operational planning level the planning horizon ranges from an hour to a week, at the tactical planning level the resource loading planning horizon may range from a week to a year. In this thesis, the planning horizon of resource loading is divided into time buckets (periods) of one week. Besides that a higher level of detail is not required at this planning level, it would probably be impossible to collect all data. Even if so, the resulting resource loading problem may become too hard to solve. Release and due dates of orders are specified in weeks, processing times of jobs are specified in smaller time units (in this thesis: hours). A minimal duration of a job is specified in weeks. Resource capacity levels are specified in hours per week.

As mentioned in Section 1.2.2, in our approach of the resource loading problem we adopt the time driven approach and the resource driven approach simultaneously. We thus aim to minimize the sum of the total cost of the

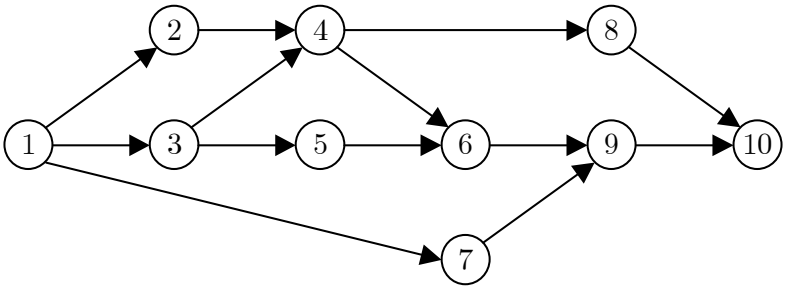

Figure 1.3: Example of generalized precedence relations in RCCP.

use of nonregular capacity and the total cost incurred by tardiness of jobs. The *tardiness* of a job is defined as the *lateness* of the job (i.e., the difference between its completion time and its due date, measured in weeks), if it fails to meet its due date, or zero otherwise. The job due date is an internal due date, i.e., it can be calculated from the order due date and the total minimal duration of the successors of the job. Job due dates may also be imposed externally, e.g., to meet certain deadlines on components or parts of an order.

1.3.2 The rough-cut capacity planning problem

A generalization of the resource loading problem in the previous section is obtained by allowing generalized precedence relations such as in Figure 1.3. These structures typically occur in project environments. The 'analogon' of resource loading in project management is known as the Multi-Project Rough-Cut Capacity Planning (RCCP) problem (De Boer, 1998). We study this problem as well, and therefore extend the scope of this thesis from discrete manufacturing environments to Resource-Constrained Multi-Project Management (RCMPM) systems. Project management was traditionally applied in situations where large projects are carried out over a long period of time by several companies. Examples include the engineering, development and construction of ships, refineries, or military weapons. However, in the last two decades, the popularity of project management extended its application to smaller tasks as well, such as the development of software, or the repair and overhaul of ships. In the context of project management, we speak of *projects* that consist of *activities*, rather than orders that consist of jobs. Project management covers all aspects of the entire life cycle of a project (Lock, 1988):

- **Order acceptance.** In this stage, a reliable due date is determined for each new project. For this purpose, insight must be obtained in the (estimated) capacity requirements for both this new project order, as well as for the projects that have been accepted already. This information is used

as an input to make a rough-cut capacity plan. The rough-cut capacity plan estimates costs, and yields insight with respect to the expected due dates. It must be used to settle a contract with the customer, that is executable for the company.

- **Engineering and process planning.** In this stage, engineering activities provide more detailed input for the material and resource scheduling of the next stage.
- **Material and resource scheduling.** The objective in this stage is to use scheduling to allocate material and activities to resources, and to determine start and completion times of activities, in such a way that due dates are met as much as possible.
- **Project execution.** In this stage, the activities are performed. Rescheduling is needed whenever major disruptions occur.
- **Evaluation and service.** In this stage, the customer and the organization evaluate the progress and the end result, to come closer towards the customer needs, and to improve the entire process in the future.

We address the *rough-cut capacity planning* (RCCP) problem in the order acceptance stage of a project life cycle. In order to position RCCP as a mechanism of planning in project management, we use the framework for hierarchical planning for (semi)project-driven organizations (see Figure 1.4) as proposed by De Boer (1998). Hierarchical production planning models are usually proposed to break down planning into manageable parts. At each level, the boundaries of the planning problem in the subsequent level are set. Ideally, multiple levels are managed simultaneously, using information as it becomes available in the course of time. However, this means that uncertainty in information will be more important as more decisions are taken over a longer period of time. Possibly due to this complexity, project planning theory hardly concentrates on problems at multiple levels. In the hierarchical planning framework in Figure 1.4, vertically we observe the same distinction between strategic, tactical, and operational planning activities as in the positioning framework in Figure 1.1. Again, at the highest planning level, the company's global resource capacity levels are determined. Strategic decisions are made with regard to machine capacity, staffing levels, and other critical resources, using the company's long-term strategy and strategic analyses like market research reports as the input of the planning. The planning horizon usually varies from one to several years, and the review interval depends on the dynamics of the company's environment.

At the RCCP level, decisions are made concerning regular and nonregular capacity usage levels, such as overtime work and outsourcing. Also, due dates and other project milestones are determined. Just as resource loading is a tool in order acceptance, the RCCP is used as a tool in the bidding and order acceptance phase of new projects, to:

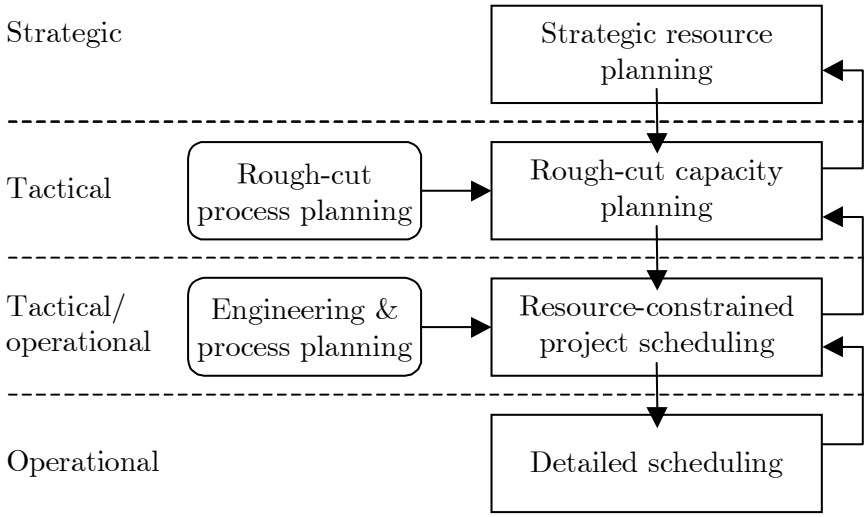

Figure 1.4: Hierarchical planning framework (De Boer, 1998).

- determine the capacity utilization of new and accepted orders,
- perform a due date analysis on new or accepted orders, and/or to quote realistic due dates,
- minimize the usage of nonregular capacity.

In order to support these analyses, just like in resource loading we distinguish two problems in RCCP: resource driven and time driven RCCP. As in resource loading, with *resource driven* RCCP, all (non-)regular resource capacity levels are fixed, and we try to minimize the maximum lateness of the projects, preferably using regular capacity. The resource driven RCCP is applicable in the situation where a customer requests a due date quotation for a project, while the company wants/has to fulfill strict resource constraints. With *time driven* RCCP, deadlines are specified for each activity in a project, while the company is flexible in the resource utilization, in that it may use nonregular capacity in order to meet the deadlines. In this case, the company wants to meet all deadlines, in such a way, that the usage of nonregular capacity is minimized. The input for the RCCP is generated by performing a *rough-cut process planning*. Based on customer demands, the rough-cut process planning breaks up a project into a network (with generalized precedence constraints) of work packages (such as in Figure 1.3), with estimates on the resource utilization (number of work hours) and minimal durations. Minimal duration constraints are a result of technical limitations of resources, e.g., in a situation when no more than two people can be deployed for a certain activity. Depending on the

project duration, the RCCP should cover a time horizon of half a year or more.

RCCP determines the detailed resource availability profiles for the next planning level, the *resource-constrained project scheduling*. Work packages are broken into smaller activities, with given duration and resource rates, and more complex precedence relations. The planning horizon varies from several weeks to several months, depending on the project duration. Resource-constrained project scheduling determines when activities are performed, and it uses *detailed scheduling* to determine which operators or machines of a resource group are assigned to each activity. The planning horizon of detailed planning may vary from one week to several weeks, and is updated frequently, when disturbances occur in the resource-constrained project scheduling plan, or on the shop floor.

From the literature we observe that, just as in the case of discrete manufacturing, the focus has been put primarily on a more detailed level, such as the resource-constrained project scheduling problem RCPSP (e.g., Kolisch, 1995). These techniques work fine at the operational level, but are not suitable for the tactical planning level, for two reasons. First, the RCPSP is inflexible with respect to resource capacity levels, which is desirable at the tactical level. The second reason is that the information that is required to perform resource-constrained project scheduling is not available at the stage of tactical planning. Detailed engineering has not been performed yet at the stage of RCCP, so, e.g., the exact number of required resources and the precedence relations are not known in detail at this stage. Recently, some heuristics have been proposed for RCCP (De Boer and Schutten, 1999; Gademann and Schutten, 2001). In this thesis we consider exact and heuristic methods to solve time driven and resource driven RCCP problems.

1.4 Example

Consider a furniture manufacturer, that produces large quantities of furniture to order. The company has a product portfolio with numerous standard products that can be assembled to order, but it is also capable of designing and subsequently producing products to order.

The sales department of the company negotiates with the customers (e.g., furniture shops and wholesalers) the prices and due dates of the products that are to be ordered. Using up-to-date information and communication systems, the sales department is able to quote tight but realistic customer due dates based upon the latest production information. If a customer order is (partly) non-standard, a macro process planning is performed to determine the production activities and the way they are performed. Each product is then processed in one or more production departments, in which the following production activities are performed:

- sawmilling (to prepare the lumber for assembly),
- assembly,
- cleaning (to sand and clean products),
- painting,
- decoration (e.g., to decorate products with glass, metal or soft furnishings),
- quality control.

The production planner breaks down the customer order into jobs, one for each production activity. He then determines for all jobs in which week a department should process a job. The head of the production department is responsible for subsequently scheduling the jobs over the resources in his department within the week. The processing of a job usually requires only a fraction of the weekly capacity of the concerned production department. However, since more workload may be present in the department in the same week, the production planner gives the head of the production department one week to complete production of all assigned jobs. Consequently, at most one job of the same order may be assigned to any particular week. The planner calculates the lead time of a customer order by counting the number of jobs (in weeks), and he dispatches the customer order for production accordingly, to meet the negotiated customer due date.

After determining the order start times, the production planner performs a capacity check for each production department, to determine if there is sufficient weekly operator and machine capacity to execute his plan. Since the machines in a department are usually continuously available, the capacity of the department is mostly limited by operator capacity. The capacity check is done by simply comparing for each week the capacity of the department with the sum of the processing times of the assigned jobs (both are measured in hours).

We illustrate the planning approach of the planner with some example data. Consider the situation described in Table 1.1, where 7 customer orders are to be dispatched for production. We have indicated the production activities sawmilling, assembly, cleaning, painting, decorating, and quality control by their first letter, i.e., S, A, C, P, D and Q. The production planner calculates the order start times from the customer due date and the number of jobs of that order. The resulting start times are displayed in Table 1.2. As mentioned before, after determining the order start times, the production planner performs a capacity check for each production department, to determine if there is sufficient weekly operator and machine capacity to execute his plan. In Figures 1.5 and 1.6 we see the capacity check for the decoration department (D) and the quality control department (Q) respectively. The bars in the fig-

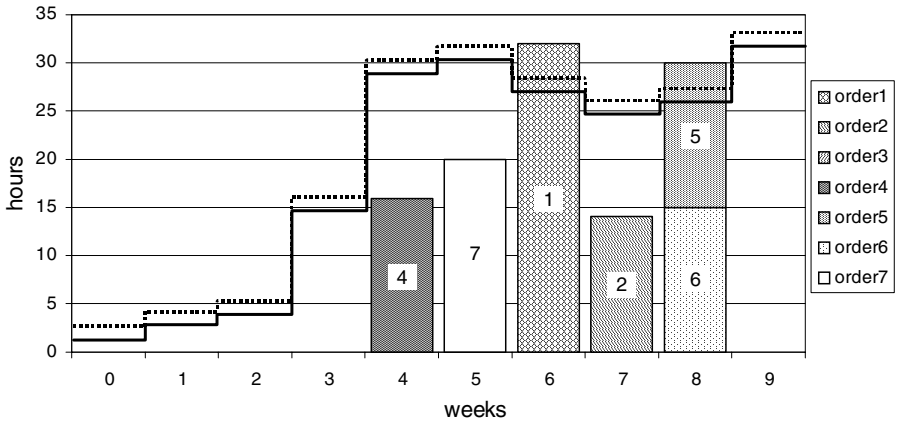

Figure 1.5: Loading of department D by production planner.

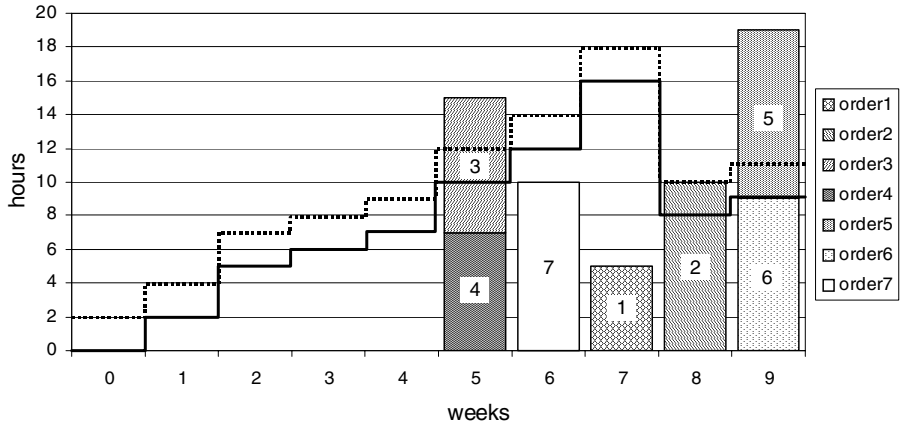

Figure 1.6: Loading of department Q by production planner.

customer	job (processing time in hours)						due date
order	1	2	3	4	5	6	(week)
1	S (37)	A (19)	C (14)	D (32)	Q (5)		7
2	S (18)	A (12)	C (15)	P (10)	D (14)	Q (10)	8
3	S (20)	A (21)	C (17)	Q (8)			5
4	A (10)	C (25)	D (16)	Q (7)			5
5	S (16)	A (33)	C (14)	P (16)	D (15)	Q (10)	9
6	S (36)	A (16)	C (20)	P (15)	D (15)	Q (9)	9
7	S (15)	A (10)	C (15)	P (15)	D (20)	Q (10)	6

Table 1.1: Example customer order data.

customer order	number of jobs	due date	start time (week)
1	5	7	3
2	6	8	3
3	4	5	2
4	4	5	2
5	6	9	4
6	6	9	4
7	6	6	1

Table 1.2: Customer order start times found by production planner.

ure represent jobs of customer orders assigned to certain weeks, and the bold line represents the available regular capacity of the department (in hours per week). By letting operators work in overtime, the department can temporarily increase the available capacity. The dotted line above the bold line represents the total available capacity of the department, i.e., the regular capacity plus the nonregular capacity. Observe that in this example, the available capacity in the first few weeks is lower than in the other weeks. This typically occurs in practice when capacity is reserved for (scheduled) orders that are already on the shop floor, and that may not be shifted in time. In department Q we see that capacity is insufficient in weeks 5, 8 and 9, and in department D in weeks 6 and 8. This plan is thus infeasible. In general, if the production planner establishes that available capacity is insufficient, he has four options to come to a feasible plan:

1. to shift jobs in time, or to split jobs over two or more weeks;
2. to increase the lead time of some customer orders (by decreasing their start time, or by increasing their due date), and then to reschedule;
3. to expand operator capacity in some weeks by hiring staff;
4. to subcontract jobs or entire orders.

Although there may be excess capacity in other weeks, shifting or splitting jobs may have a negative effect on other departments. Consider for example week 9 in department Q, where capacity is insufficient to complete the jobs of orders 5 and 6. Shifting the job of order 5 to week 7 (in which there is sufficient capacity) implies that the preceding job of order 5 in department D must shift to week 6, where capacity is already insufficient. Moreover, the other preceding jobs of order 5 must shift to week 5 and before, which may result in more capacity problems in other departments. Splitting the job of order 2 in department Q over week 7 and 8, implies that the preceding job of order 2 in department D must shift to week 6, where, as mentioned before, capacity is already insufficient.

The option to increase customer order lead times is not desirable, since this may lead to late deliveries, which may result in penalties, imposed by the customers involved. Moreover, this does not guarantee that no overtime work or subcontracting is required to complete all orders in time. Since increasing the lead times often does not lead to feasible plans, this option tends to increase order lead times more and more, which has several undesirable side effects, such as an increasing amount of work-in-process.

The remaining options to make a feasible plan involve using nonregular operator or machine capacity. Assigning nonregular operator capacity enables the use of machines in nonregular operator time. However, hired staff is often not readily available. Moreover, some work can not be subcontracted, and subcontracting a job tends to increase the order lead time.

The example shows that even a small loading problem may be very hard to solve. The approach of the planner is commonly used in practice. It appears to be a reasonable approach, since it accounts for important aspects such as:

- the precedence relations between the jobs,
- meeting customer due dates,
- (regular and nonregular) operator and machine capacity availability.

In this thesis we provide tools for solving similar loading problems, and loading problems that are much larger (with respect to the number of orders and jobs) and more complex due to additional restrictions on jobs and orders. We have used one of our methods, which uses all four of the aforementioned options to come to a feasible plan, to optimally solve the loading problem in this example. Figures 1.7 and 1.8 display a part of the solution, i.e., the optimal loading for the decoration department and the quality control department respectively. Observe that in the optimal plan little overtime work is required in the quality control department to produce all orders in time. Table 1.3 displays the optimal start times of the customer orders, as well as the completion times in the optimal plan, in comparison with the (external) order due dates.

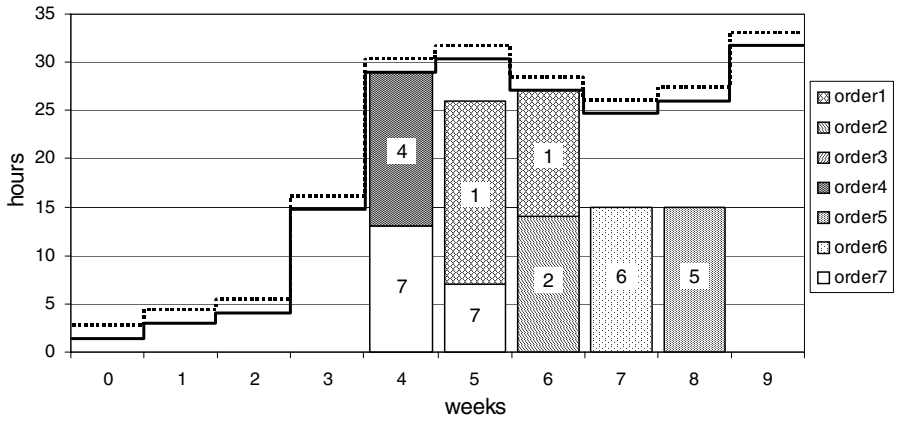

Figure 1.7: Optimal loading of department D.

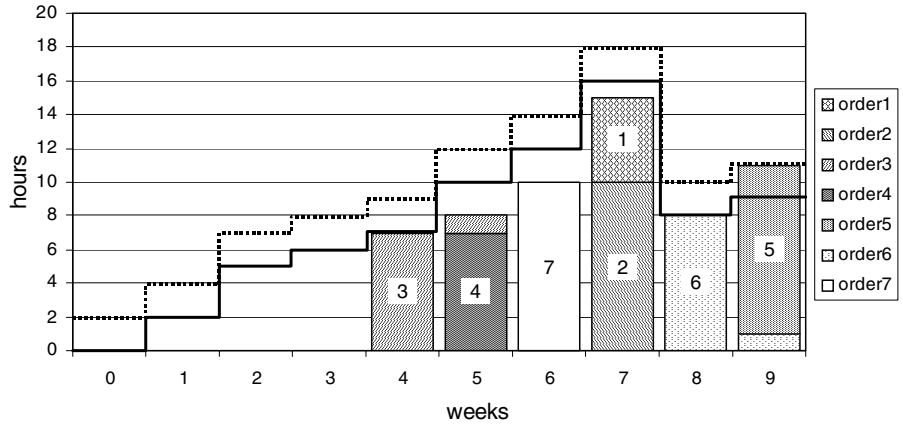

Figure 1.8: Optimal loading of department Q.

customer order	start time (week)	completion time (week)	due date (week)
1	1	7	7
2	1	7	8
3	1	5	5
4	0	5	5
5	4	9	9
6	0	9	9
7	0	6	6

Table 1.3: Optimal customer order start times

1.5 Current practices

The influence of customers on the manufacturing plans has increased enormously during the last two decades. A higher product variety, shorter product life cycles, improved delivery performance and stronger competition force manufacturing organizations to continuously improve their quality and to reduce their lead times. In this context, agile manufacturing has become a major strategy for firms to reduce their lead times and the time-to-market (Van Assen et al., 1999). Not only does time reduction enable a swifter response to changing market conditions, the pursuit of time compression also reveals sources of quality problems and wasted efforts. In addition to time reduction, manufacturing companies need to control the operational costs by efficiently allocating their resources over time. For this purpose, various approaches for production planning and control have been proposed, partly leading to the development of automated production control systems. Notable examples are Material Requirements Planning (Orlicky, 1975) and its successor Manufacturing Resources Planning (Wight, 1981; Vollmann et al., 1988), Hierarchical Production Planning (Hax and Meal, 1975) and, at the shop floor level, Optimized Production Technology (Goldratt, 1988). Unfortunately, most systems have not fulfilled the initial expectations, for various reasons (Zijm, 2000). *Manufacturing Resources Planning* (MRP II) based systems have specifically been developed for make-to-stock assembly production, and are primarily administrative systems. MRP II systems lack intelligent loading algorithms, and can not handle dynamic situations, or situations that involve a high data uncertainty. The material oriented planning techniques of MRP II systems in fact do not even account for capacity constraints. The systems use rough-cut capacity planning (RCCP) at a higher level, while at a more detailed level they use capacity requirements planning (CRP) to check whether capacity constraints are violated. CRP performs an infinite forward loading (Hopp and Spearman, 1996) to predict job completion times using fixed lead times (Karmarkar, 1993; Karni, 1982). CRP compares the loading against available capacity, but does not provide a tool to resolve capacity problems. Often such an infinite planning procedure yields

infeasible plans (Negenman, 2000). Moreover, the feasibility of the plans on the lower planning levels can not be guaranteed satisfactorily, due to specific job shop constraints like complex precedence relations that are not considered on the aggregate planning level. As a result, the fixed lead times in infinite planning tend to inflate (the so-called 'planning loop', see, e.g., Zäpfel and Missbauer, 1993; Suri, 1994). While MRP II is primarily material oriented, ideally, the material in the production process and the use of resources are considered simultaneously in the planning process. To be able to solve the models satisfactorily, the existing methods for this purpose either have a high level of aggregation, or are based on repairing a plan that is infeasible with respect to capacity or precedence constraints (Baker, 1993; Negenman, 2000). A more detailed description of the major drawbacks of MRP II has been given by Darlington and Moar (1996).

Hierarchical Production Planning (HPP) on the other hand, is strongly capacity oriented. In HPP, problems are decomposed by first roughly planning complete product families and then scheduling individual items. Compared to a monolithic system, this not only reduces the computational complexity of the mathematical models, it also reduces the need of detailed data (Bitran and Tirupati, 1993). As recognized by Schneeweiss (1995), high-level decisions need to anticipate the effect on future low-level problems. However, since HPP can not handle complex product structures and product routings, this is hardly possible. Finally, *Optimized Production Technology* (OPT) does consider routing constraints but is primarily shop floor oriented, i.e., it assumes a medium term loading as given.

Workload control concepts buffer a shop floor against external dynamics (such as rush orders) by creating a pool of unreleased jobs (Bertrand and Wortmann, 1981; Wiendahl, 1995). The use of workload norms should turn the queueing of orders on the shop floor into a stationary process that is characterized by an equilibrium distribution. The development of workload control systems shows that adequate finite resource loading tools are needed to control the aforementioned dynamic production situations. However, although workload control does yield more reliable internal lead times by not allowing work to be loaded in the job shop in case of a high load already present, the problem is shifted to the buffers before the job shops, hence due dates still tend to increase (Hendry et al., 1998).

The models in the literature as well as the techniques used in the aforementioned production planning and control systems that explicitly address loading problems are inadequate in situations that require more flexibility due to uncertainty in data. *Linear programming* (LP) is an accepted method in, e.g., aggregate planning and workforce planning to decide upon capacity levels (once a global long-term plan is determined), by considering overwork, hiring and firing, and outsourcing options (see, e.g., Shapiro, 1993; Hopp and Spearman, 1996). However, modeling complex precedence relations in LP models requires

the introduction of many integer variables, which increases the size of the models enormously, and makes them computationally hard to solve (Shapiro, 1993). For this reason, precedence constraints are often omitted in linear programming approaches (see, e.g., Graves, 1982; Bahl and Zionts, 1987). Although, as argued in Section 1.1, at the resource loading level product routings should not be considered in the same level of detail as in shop floor scheduling, it is essential to account for major precedence relations, e.g., with respect to machine groups, in order to define work packages at the loading level that can indeed be handled at the shop floor level. In other words: approaches that ignore these major constraints to reduce complexity may lead to production plans that can not be maintained at a more detailed level. This holds in particular when multiple resources (e.g., machines and operators) are needed to simultaneously process jobs. Dealing with multiple resources without considering job routings often leads to inconsistencies, when eventually these different resources are not available at the same time.

Although we use a deterministic approach in this thesis, it is likely that in future research also *stochastic techniques* (Kall and Wallace, 1994; Infanger, 1994; Buzacott and Shanthikumar, 1993) will be used for resource loading (see, e.g., Giebels, 2000). This may particularly be the case for engineer-to-order (ETO) manufacturing planning, where even more uncertainty in data is involved than in MTO manufacturing planning. While generally at the tactical planning level in MTO environments processing times are deterministic, and routings are known, in ETO environments the macro process planning is generally incomplete at the tactical planning stage. Hence processing times are uncertain, and even the required machines, tools and materials are uncertain. A promising approach that deals with such situations is the EtoPlan-concept, proposed by Giebels (2000).

We conclude this section with a note on the role of material coordination in resource loading. Compared to assemble-to-order (ATO) environments, in MTO environments material coordination plays a less significant role. We will not consider material related constraints. For advanced models for material coordination under capacity constraints in ATO environments we refer to Negenman (2000).

1.6 Outline of the thesis

The remainder of this thesis is organized as follows. In Chapter 2 we present some mathematical preliminaries for the analysis in the next chapters. The discussed techniques include column generation, branch-and-price, Lagrangian relaxation and deterministic dynamic programming. Some example problems in Chapter 2 are tailored to the problems in the remainder of this thesis. In Chapter 3 we discuss the modeling issues of resource loading, and present a Mixed Integer Linear Programming (MILP) formulation. We present column

generation based solution methods for the resource loading problem in Chapter 4. In this chapter we also discuss heuristics that can be used 'stand-alone', or as an integrated part of the column generation based methods. We pose several branch-and-price techniques to find feasible solutions to the entire MILP model. In Chapter 5 we discuss the results of computational experiments with the methods proposed in Chapter 4.

In Chapter 6 we present a generalization of the MILP model of Chapter 3, and use it to solve the RCCP problem. The solution method presented in this chapter is also based on column generation, but we present a generalized pricing algorithm, that is able to generate schedules for projects, that comply with the additional constraints of the RCCP problem. In Chapter 7 we discuss the experiments with, and computational results of the methods proposed in Chapter 6.

Finally, in Chapter 8, we summarize our experiences with the methodology presented in this thesis, discuss its strengths and its weaknesses, and propose topics of further research.

Chapter 2

Preliminaries

Only two things are infinite,
the universe and human stupidity
- and I'm not sure about the former.

- Albert Einstein (1879-1955)

2.1 Introduction to integer linear programming

In this thesis we use several combinatorial optimization techniques to solve the resource loading problem. In this chapter we discuss these techniques, such as branch-and-bound, column generation, Lagrangian relaxation, and dynamic programming. For a more comprehensive overview of combinatorial optimization techniques, we refer to, e.g., Wolsey (1998); Johnson et al. (1998); Nemhauser and Wolsey (1988); Winston (1993).

During the last 50 years, linear programming (LP) has become a well-established tool for solving a wide range of optimization problems. This trend continues with the developments in modeling, algorithms, the growing computational power of personal computers, and the increasing performance of commercial linear programming solvers (Johnson et al., 1998). However, many practical optimization problems require integer variables, which often makes them not straightforward to formulate, and extremely hard to solve. This is also the case for the resource loading problem. The resource loading problem is formulated in Chapter 3 as a *mixed integer linear program* (MILP). A MILP is a generalization of a linear program, i.e., it is a linear program of which some

of the variables are integer, e.g.:

$$\begin{array}{lll} MILP: & \min c^T x & \\ & Ax \geq b & \\ s.t. & l \leq x \leq u & \\ & x_j \ integer \ \ (j = 1, \ldots, p; p \leq n). & \end{array} \tag{2.1}$$

The input data for this MILP is formed by the n-vectors c, l, and u, by $m \times n$-matrix A, and m-vector b. The decision vector x is an n-vector. When $p = n$ the MILP becomes a *pure integer linear program* (PILP). When the integer variables are restricted to values 0 or 1, the problem becomes a (mixed) *binary integer linear program* (MBILP or BILP). Each ILP can be expressed as a BILP by writing each integer variable value as a sum of powers of 2 (see, e.g., Winston, 1993). A drawback of this method is that the number of variables greatly increases. For convenience, in the remainder of this chapter we discuss integer linear programs that are *minimization* problems with *binary* variables.

Formulating and solving ILPs is called *integer linear programming*. Integer linear programming problems constitute a subclass of combinatorial optimization problems. There exist many combinatorial optimization algorithms for finding feasible, or even optimal solutions of integer linear programming models (see, e.g., Wolsey, 1998; Johnson et al., 1998). When optimizing complex problems, there is always a trade-off between the computational effort (and hence running time) and the quality of the obtained solution. We may either try to solve the problem to optimality with an *exact algorithm*, or settle for an *approximation* (or *heuristic*) *algorithm*, which uses less running time but does not guarantee optimality of the solution. In the literature numerous heuristic algorithms can be found, such as *local search* techniques (e.g., tabu search, simulated annealing, see Aarts and Lenstra, 1997), and *optimization-based* algorithms (Johnson et al., 1998).

The running time of a combinatorial optimization algorithm is measured by an upper bound on the number of elementary arithmetic operations it needs for any valid input, expressed as a function of the input size. The input is the data used to represent a problem instance. The input size is defined as the number of symbols used to represent it. If the input size is measured by n, then the running time of an algorithm is expressed as $O(f(n))$, if there are constants c and n_0 such that the number of steps for any instance with $n \geq n_0$ is bounded from above by $cf(n)$. We say that the running time of such an algorithm is of order $f(n)$. An algorithm is said to be a *polynomial time algorithm* when its running time is bounded by a polynomial function, $f(n)$. An algorithm is said to be an *exponential time algorithm* when its running time is not bounded by a polynomial function (e.g., $O(2^n)$ or $O(n!)$). A problem can be classified by its *complexity* (see Garey and Johnson, 1979). A particular class of 'hard' problems is the class of so-called NP-hard problems. Once established that a problem is NP-hard, it is unlikely that it can be solved by a polynomial algorithm.

Before considering what algorithm to use to solve an ILP problem, it is important to first find a 'good' formulation of the problem. Sometimes a different ILP formulation of the same problem requires fewer integer variables and/or constraints, which may reduce or even increase computation time for some algorithms. Optimization algorithms for NP-hard problems always resort to some kind of enumeration. The complexity of NP-hard problems is nicely illustrated by Nemhauser and Wolsey (1988), who show that a BILP (which is proven to be NP-hard) can be solved by brute-force enumeration in $O(2^n nm)$ running time. For example, a BILP with $n = 50$ and $m = 50$ that is solved by complete enumeration on a computer that can perform 1 million operations per second requires nearly 90,000 years of computing time. For $n = 60$ and $m = 50$ this is nearly 110 million years.

A generic categorization for exact algorithms for ILP problems is given by Zionts (1974). He distinguishes three generic optimization methods: constructive algorithms, implicit enumeration, and cutting plane methods.

Constructive algorithms construct an optimal solution by systematically adjusting integer variable values until a feasible integer solution is obtained. When such a solution is found, it is an optimal solution. The group theoretic algorithm is an example of a constructive algorithm (Zionts, 1974).

Implicit enumeration methods enumerate solutions in a highly efficient manner. They generally eliminate subsets of solutions without explicitly enumerating them. In this thesis we use two implicit enumeration methods: dynamic programming and branch-and-bound. *Dynamic programming* is a decomposition technique that first decomposes the problem into a nested family of subproblems. The solution to the original problem is then found by recursively solving the (generally easier) subproblems. We discuss dynamic programming in more detail in Section 2.5. *Branch-and-bound* is structured by an enumeration scheme that involves partitioning the solution space. In the scheme, each partition is further analyzed and partitioned until a (better) feasible solution is found or it is determined that the partition does not contain a (better) solution. The enumeration scheme can nicely be displayed by a *branching tree*. The root node of the tree corresponds to the original problem, and has a solution space that holds all feasible solutions to this problem. As the algorithm progresses, the solution space is partitioned by adding constraints to the problem in the root node, forming two or more new problems in the child nodes. This process of partitioning the solution space, which is called *branching*, is continued in the child nodes, which become parent nodes to new child nodes in a subsequent partitioning of the solution space. A typical example is a branching strategy based on choosing either a value 0 or 1 for a single binary variable (see, e.g., Figure 2.1). Hence in each node the feasible set of solutions is decomposed into smaller (disjoint) subsets. At each node lower and upper bounds can be determined on the objective value of that subset. *Lower bounds* can be obtained, e.g., by relaxations of the original integer linear program. A *relaxation*

of an integer linear program is obtained by simplifying the problem. E.g., the *LP relaxation* of a BILP is obtained by omitting the integrality restrictions on the variables ($x \in \mathbb{B}^n$), and replacing them by: $0 \leq x \leq 1, x \in \mathbb{R}^n_+$. Another technique to obtain a relaxation is *Lagrangian relaxation* (see, e.g., Geoffrion, 1974). We discuss Lagrangian relaxation in more detail in Section 2.3. *Upper bounds* can be obtained by finding feasible solutions in a node (e.g., heuristically). The upper and lower bounds can be used to fathom certain nodes from further consideration. A node can be *fathomed*, if the lower bound in that node is larger than or equal to the best solution found so far in branching or before. The best solution found so far is called the *incumbent solution*. The ILP problems in child nodes of that node have additional constraints, so the solutions in these nodes can only become worse, or remain the same. Nodes can also be fathomed when the problem in a node is infeasible. This process is called *pruning*. The branching terminates when all nodes have been evaluated. The incumbent is then the optimal solution. The number of nodes that are examined in the branching scheme depends on the branching strategy, the order in which the nodes are examined (e.g., *depth first* or *best first search*), dominance rules (applicable when nodes may be fathomed after other dominant nodes have been examined), and the quality of the lower and upper bounds.

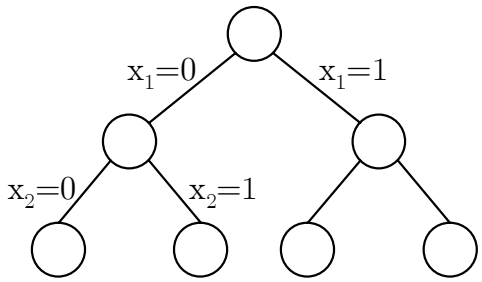

Figure 2.1: Branch-and-bound tree example.

The third category of exact optimization methods for binary problems are the *cutting plane methods*. These methods start from the (generally non-integer) solution of the LP relaxation of the ILP. The basic idea is then to add constraints (cutting planes) to the LP relaxation, to cut off a part of the feasible region of the LP relaxation that includes the optimal LP relaxation solution, and that is not part of the *convex hull* of the feasible region of the ILP. These constraints are found by solving the *separation problem*. The difficulty in this approach is to describe the convex hull of the feasible region of the ILP. The research that focuses on finding the equalities that are needed to describe the convex hull (the so-called *facets*) is known as *polyhedral combinatorics* (see Nemhauser and Wolsey, 1988).

The aforementioned optimization techniques are often used in conjunction

with each other. For example cutting plane methods are almost always used in conjunction with branch-and-bound. In these so-called *branch-and-cut* algorithms, partial descriptions of the convex hull of feasible solutions of the ILP are used to determine strong lower bounds in the branching scheme (see, e.g., Hoffman and Padberg, 1985). Branch-and-cut is the technique that most commercial ILP solvers use to solve (mixed) integer linear programs. In this thesis we use the *column generation* method to solve the LP relaxation of the ILP, a technique very similar to cutting plane methods. Whereas cutting plane methods add *rows* to the LP relaxation of the problem, column generation adds *columns* to a restricted version of the LP relaxation. The separation problem in column generation is called the *pricing problem*. Column generation can be interpreted as a cutting plane technique on the dual of the LP relaxation of the ILP (see, e.g., Barnhart et al., 1998; Beasley, 1996; Bixby et al., 1992). Dantzig and Wolfe (1960) proposed the column generation technique as a scheme for solving large linear programming problems that have (exponentially) many variables. Since then it has been applied to a wide variety of problems. The most well-known is the application to the cutting stock problem (Gilmore and Gomory, 1963). In more recent research, column generation has been applied to, e.g., vehicle routing problems with time windows (Desrosiers et al., 1984; Desrochers et al., 1992) and crew scheduling (Freling, 1997; Desrochers and Soumis, 1989). When column generation is used in conjunction with branch-and-bound, it is referred to as *branch-and-price* (see, e.g., Savelsbergh, 1997; Vance et al., 1994). In this thesis we use *branch-and-price* methods to determine a feasible solution to the ILP from the (fractional) LP relaxation solution.

In the remainder of this chapter we elaborate on the techniques used in this thesis. Since branch-and-price plays a central role in this thesis, we discuss this technique in more detail in Section 2.2. In the branching schemes used in this thesis, we use Lagrangian relaxation to determine lower bounds. In Section 2.3 we elaborate more on Lagrangian relaxation, and in Section 2.4 we discuss how to use Lagrangian relaxation in conjunction with column generation. Since dynamic programming is used in this thesis for optimizing the pricing problem of the column generation algorithm we discuss this subject in Section 2.5.

2.2 Branch-and-price for solving large integer linear programming models

2.2.1 Method outline

Successfully solving large integer linear programs often requires LP relaxations that give a good approximation of the convex hull of feasible solutions (Barnhart et al., 1998). Approaches based on this idea that are often used to solve

large integer linear programs are the aforementioned branch-and-cut approach, and the branch-and-price approach. Typical ILP problems that are solved by branch-and-cut have an LP relaxation with too many valid inequalities to handle efficiently. Since many of these valid inequalities are not binding in an optimal solution anyway, the approach starts by finding a formulation of the LP relaxation where these valid inequalities are left out. This formulation is the so-called *restricted LP relaxation* (RLP). This RLP is then solved to optimality. A subproblem (the *separation problem*) is solved to determine if the resulting optimal solution is feasible to the LP relaxation by identifying violated valid inequalities. If the solution is not feasible to the LP relaxation, at least one violated inequality is identified and added to the RLP. This procedure continues until no more violated inequalities exist. In that case, the optimal solution to the RLP is also optimal to the LP relaxation. Generally this solution is fractional, and hence not feasible for the ILP. Branching can be performed in order to find a feasible solution to the ILP.

The basic idea of *branch-and-price* is very similar to that of branch-and-cut, except that it involves *column generation*, instead of row generation. Typical ILP problems that are solved by branch-and-price have too many columns to handle efficiently, and since the basis has a limited dimension, many of these columns do not play a role in an optimal solution anyway. Analogous to branch-and-cut, the LP relaxation is solved to optimality by first considering a restricted LP relaxation (RLP), where many columns are left out. The RLP is also called the *restricted master problem*. The solution of the RLP is optimal for the LP relaxation if all variables of the LP relaxation have non-negative *reduced cost*. Since only a subset of the columns of the LP relaxation is explicitly available, this can not be checked explicitly. A so-called *pricing algorithm* is used to verify optimality. If the solution is not optimal, the pricing algorithm identifies at least one column with negative reduced cost. This column is added to the RLP, and the procedure continues. The column generation scheme terminates when no columns with negative reduced cost exist anymore. The optimal solution to the RLP is then also optimal to the LP relaxation. This process of adding columns to the RLP when they are needed is called *implicit column generation*. In *explicit column generation*, all columns are generated in advance and kept in computer memory. Subsets of columns are then passed to the RLP until a subset is found that contains the optimal solution.

Analogous to branch-and-cut, the optimal solution to the LP relaxation is generally fractional (i.e., non-integer), and hence not feasible to the ILP, so branching needs to be performed in order to find a feasible solution to the ILP. The column pool is partitioned by adding constraints. The type of constraints that work best for a problem is usually determined by performing experiments. Usually constraints are used that cut off the fractional (optimal) LP relaxation solution, or that fix variables. By partitioning the column pool in the parent node, each child node corresponds to a new LP relaxation. The column pool must be updated accordingly, and the new LP relaxation is also solved by

column generation. When the optimal solution to the new LP relaxation is again fractional, branching continues. A node is explored no further when in that node a feasible ILP solution is found, or when the LP relaxation in the node is infeasible, or when the node can be fathomed. A node is fathomed when a lower bound to the feasible ILP solution in that node has been determined that exceeds the best ILP solution found this far. Lower bounds can be determined in each node, e.g., by Lagrangian relaxation (we discuss the application of Lagrangian relaxation in Section 2.3).

In Figure 2.2 we illustrate the column generation procedure that is performed in each node. The entire branch-and-price procedure is illustrated in Figure 2.3. In the next section we will elaborate on possible branching strategies and speed ups of branch-and-price algorithms.

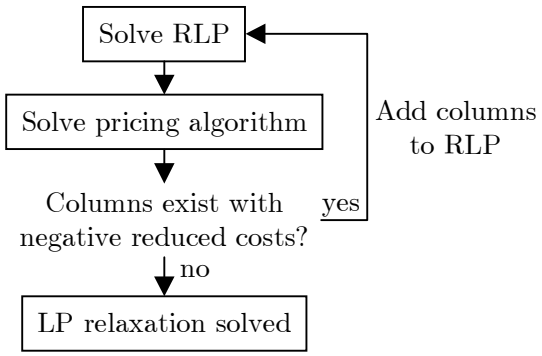

Figure 2.2: Column generation scheme in a node.

2.2.2 Column generation strategies

There are many ways to speed up a branch-and-price algorithm. Most approaches try to reduce the number of nodes that must be examined to prove optimality of an integer solution. There is a trade-off between finding a strong formulation on one hand, by trying to make the difference in the solutions to the LP relaxation and the integer linear program (the *integrality gap*) as small as possible, and the number of nodes to explore in the branch-and-bound tree on the other hand. The latter aspect is usually done by reformulation. Well-known reformulation techniques are the *preprocessing and probing techniques* that focus on identifying infeasibility and redundancy, improving variable bounds and coefficients, and fixing variables (Savelsbergh, 1994; Nemhauser et al., 1994; Wolsey, 1998). These techniques, as well as other reformulation techniques such as constraint generation, have proven to be very effective in reducing both the integrality gap, and the size of the branch-and-bound tree (Savelsbergh,

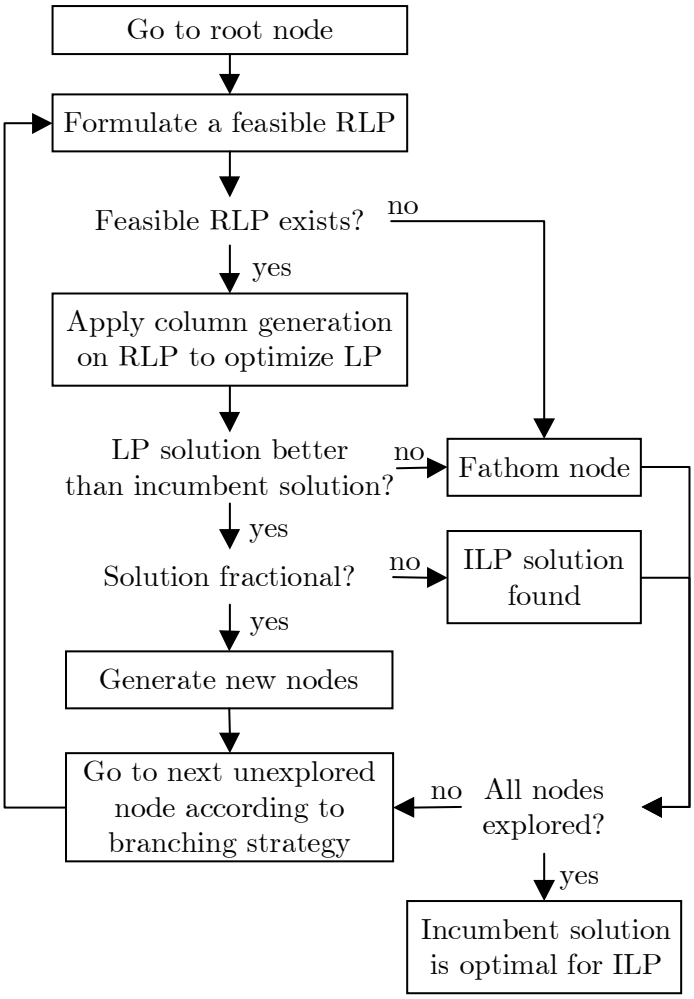

Figure 2.3: Branch-and-price scheme.

1994). Another technique that is used in advance of the branching process to speed up the algorithm is *column management*. By generating and storing many columns in advance without adding them to the RLP, fewer pricing problems have to be solved. In column management, a *pricing strategy* prescribes when columns must be added to the pool, and how many. There is not one 'best' pricing strategy for each problem, and hence computational experiments must be performed to provide insight. In the pricing strategy there is also a trade-off between the number of columns to add in each iteration (hence the computation time of the pricing algorithm) on one hand and the size of the column pool (hence the computation time of the RLP) on the other hand. Moreover, column management may prescribe to delete columns from the pool, e.g., when columns are redundant in the RLP (i.e., when columns are no longer in the basic RLP solution).

Another way to improve the speed of the branch-and-price algorithm is to focus on both improving the lower bound in a node, and on finding good feasible ILP solutions to the original problem as early as possible in the branching process. Recall that a node is fathomed if a lower bound to the LP relaxation in that node exceeds the incumbent. Lower bounds thus apply to a node, and incumbent solutions apply to the entire branching tree. The stronger the lower bounds, the fewer nodes have to be examined, in particular when a good feasible incumbent solution has been found already. However, determining strong lower bounds may be computationally intensive. In Section 2.3 we discuss Lagrangian relaxation for determining a lower bound in a node with little additional computational effort every time the RLP is solved in the column generation scheme. Feasible ILP solutions can be found by solving primal heuristics, or by choosing a branch selection strategy that yields feasible ILP solutions fast (see, e.g., Vanderbeck, 2000). Since such feasible solutions are more likely to be found deep in the tree, a typical selection strategy to find feasible solutions fast is the *depth first* strategy. Moreover, depth first requires relatively little computer memory. Once an incumbent solution is found, the branch selection strategy may be changed to, e.g., *best bound search*. In best-bound search all unfathomed nodes are examined, and the node that has the smallest lower bound is selected as the branch node, since it most likely contains a solution that improves the incumbent. Primal heuristics are usually used to find an incumbent solution in the root of the branching tree. When the heuristic does not require too much computation time, it can also be used to improve the incumbent solution in other nodes of the tree. Hence, as to how often a heuristic is used in the branching scheme, there is always a trade-off between the computation time of the heuristic and the resulting speed up of the branching algorithm.

One side effect of column generation that affects the speed of the branch-and-price algorithm is the so-called *tailing-off* effect, which is due to *degeneracy*. What occurs in such an event is that in each column generation iteration at least one column is found with negative reduced costs, which, after being added to the RLP, enters the basis with a value zero. As a result, it takes

many iterations before optimality can be proven, i.e., before column generation converges. A performance gain can be achieved by detecting tailing-off, such as for example in an approximation method that terminates column generation when the difference between the RLP solution (which is an upper bound on the LP relaxation solution) and a lower bound to the LP relaxation solution is sufficiently small.

2.2.3 Implicit column generation

Consider the following general integer minimization problem (ILP):

$$\begin{array}{rll} ILP: & \min & z_{ILP} = c^T x \\ & s.t. & A_1 x \geq b_1 \\ & & A_2 x = b_2 \\ & & x_j \in \mathbb{Z}_+ \ \ (j \in J)\,, \end{array} \tag{2.2}$$

where the input data is formed by m_1-vector b_1, m_2-vector b_2, n-vector c, $m_1 \times n$ coefficient matrix A_1 and $m_2 \times n$ coefficient matrix A_2. J is an n-set of variable indices. We obtain its LP relaxation (LP) by omitting the integrality constraints $x_j \in \mathbb{Z}_+$ $(j \in J)$, and replacing them by $x_j \geq 0$ $(j \in J)$. Hence we have:

$$\begin{array}{rll} LP: & \min & z_{LP} = c^T x \\ & s.t. & A_1 x \geq b_1 \\ & & A_2 x = b_2 \\ & & x_j \geq 0 \ \ (j \in J) \end{array} \tag{2.3}$$

We show how to use (implicit) column generation to solve LP in a case where the set of columns in the coefficient matrix is large. More specifically, suppose the number of columns in ILP is too large to handle efficiently, and hence its LP relaxation (LP) can not be formulated and solved explicitly. To start the column generation scheme, we formulate a restricted LP relaxation (RLP) that contains a subset of the columns in coefficient matrices A_1 and A_2. For this purpose, a set of columns is required that forms a feasible basis for LP. For the moment we assume that such a set of columns is available. These determine the initial RLP, which can be formulated as follows:

$$\begin{array}{rll} RLP: & \min & \widetilde{z}_{RLP} = \widetilde{c}^T \widetilde{x} \\ & s.t. & \widetilde{A}_1 \widetilde{x} \geq b_1 \\ & & \widetilde{A}_2 \widetilde{x} = b_2 \\ & & \widetilde{x}_j \geq 0 \ \ \left(j \in \widetilde{J} \subseteq J\right). \end{array} \tag{2.4}$$

The restricted LP model (RLP) has the same form as LP, but has a subset $\widetilde{J}$ of the variables, with corresponding submatrices $\widetilde{A}_1$ and $\widetilde{A}_2$, and cost vector $\widetilde{c}$.

The corresponding *dual problem* ($DRLP$) is given by:

$$\begin{array}{rll} DRLP: & \max & \widetilde{z}_{DRLP} = b_1^T u + b_2^T v \\ & s.t. & \widetilde{A}_1^T u + \widetilde{A}_2^T v \leq \widetilde{c} \\ & & u \in \mathbb{R}_+^{m_1}, v \in \mathbb{R}^{m_2} \end{array}$$

where u is an m_1-vector of dual variables, and v is an m_2-vector of dual variables. Observe that this problem can also be written in a compact form with an m-vector w of dual variables, where $m = m_1 + m_2$, and $w = (u, v)$.

We start the column generation scheme by solving RLP to optimality. Note that, since RLP has the same constraints as LP, any basic feasible solution ($\widetilde{x}^*$) to RLP forms a basic feasible solution (x^*) to LP, by letting $x_j^* = 0$ ($\forall j \in J \backslash \widetilde{J}$). However, an optimal solution to RLP is not necessarily optimal to LP. To determine if an optimal solution to RLP is also optimal to LP, we use the following result (see, e.g., Wolsey, 1998) from duality theory:

Theorem 2.1 *Let $x^* = (x_1^*, x_2^*, \ldots, x_n^*)$ be a primal n-vector, and $w^* = (w_1^*, w_2^*, \ldots, w_m^*)$ a dual m-vector. Then x^* is an optimal primal solution and w^* is an optimal dual solution if and only if x^* is primal feasible, w^* is dual feasible and $c^T x^* = w^{*T} b$.*

Since any feasible solution to RLP is also feasible to LP, we have the primal feasibility condition. Moreover, with $\widetilde{x}^*$ optimal to RLP and the corresponding u^* and v^* optimal to $DRLP$, we also have that $c^T \widetilde{x}^* = u^{*T} b_1 + v^{*T} b_2$ and thus that $c^T x^* = u^{*T} b_1 + v^{*T} b_2$. Hence, to determine if an optimal solution $\widetilde{x}^*$ to RLP is also optimal to LP, it remains to verify that the corresponding (optimal) dual solution u^* and v^* to $DRLP$ is feasible to the dual of LP (DLP). The dual solution vectors (u^*, v^*) are feasible to DLP when:

$$A_{1j}^T u^* + A_{2j}^T v^* \leq c_j \ (\forall j \in J),$$

or:

$$c_j - A_{1j}^T u^* - A_{2j}^T v^* \geq 0 \ (\forall j \in J). \tag{2.5}$$

The term $c_j - A_{1j}^T u^* - A_{2j}^T v^*$ in (2.5) is in fact the *reduced cost* $\overline{c}_j$ of variable x_j. The reduced cost value $\overline{c}_j$ of a variable x_j, indicates the change $\Delta\widetilde{z}$ in the objective value $\widetilde{z}$, resulting from a change Δx_j in non-basic variable x_j:

$$\Delta\widetilde{z} = \overline{c}_j \Delta x_j$$

Clearly, $\overline{c}_j \geq 0$ for all $j \in \widetilde{J}$. Therefore, determining if an optimal solution to RLP is also optimal to LP boils down to determining if the reduced cost $\overline{c}_j$ of all variables x_j in LP that are not included in RLP, is non-negative. Hence we can formulate the *optimality condition for column generation* as follows (Gilmore and Gomory, 1961):

Theorem 2.2 *Consider an LP model of a minimization problem with column set J, denoted by $LP(J)$. Let w^* be an optimal dual solution of problem $LP(\widetilde{J})$ for $\widetilde{J} \subset J$, and $\overline{c}^*$ the corresponding reduced cost vector. The solution of $LP(\widetilde{J})$ is optimal for problem $LP(J)$ if $\overline{c}_j^* \geq 0$ $(\forall j \in J \backslash \widetilde{J})$.*

When columns have non-negative reduced costs, we say the columns *price out*. The problem of identifying columns with negative reduced costs is called the *pricing problem*. A pricing algorithm either determines that all $\overline{c}_j^* \geq 0$, or it provides at least one column with associated $\overline{c}_j^* < 0$. Such columns can be added to RLP. Although more than one such column may exist, it is not necessary to generate all columns with negative reduced costs. However, adding more than one negative reduced cost column (called *multiple pricing*) reduces the chance of cycling.

The column generation scheme thus solves the RLP and the pricing algorithm successively in each iteration, and terminates when no negative reduced cost columns can be found in the pricing algorithm. However, since the optimal solution to LP is usually not integral, and thus not feasible to ILP, we need to perform branching or to use heuristics in order to find a feasible solution to ILP.

2.3 Lagrangian relaxation

The basic idea of Lagrangian relaxation is to *dualize* (i.e., relax) those constraints that make a problem hard to solve, in such a way, that good lower bounds can be obtained for the original problem. The idea of dropping constraints is brought about by introducing a *Lagrangian multiplier/parameter* for each constraint that is dropped from the problem. These constraints are then removed from the problem, and put in the objective function, weighted by a vector of Langrangian multipliers. Again, consider the (general) integer linear program ILP:

$$\begin{array}{rl} ILP: & \min\ z_{ILP} = c^T x \\ s.t. & A_1 x \geq b_1 \ (\text{duals } u \in \mathbb{R}_+^{m_1}) \\ & A_2 x = b_2 \ (\text{duals } v \in \mathbb{R}_+^{m_2}) \\ & x \in \mathbb{Z}_+^n \end{array}$$

Suppose constraints $A_1 x \geq b_1$ are the constraints that make the problem hard to solve, and $X = \left\{x | A_2 x = b_2, x \in \mathbb{Z}_+^n\right\}$ has some computationally convenient structure not shared by the entire problem. For the Lagrangian relaxation $L_{ILP}(\lambda)$ of ILP, we dualize constraints $A_1 x \geq b_1$, and introduce a vector of Lagrangian multipliers $\lambda \in \mathbb{R}_+^{m_1}$ to form the following *Lagrangian subproblem*:

$$\begin{array}{rl} L_{ILP}(\lambda): & \min\limits_{x}\ z_{L,ILP}(\lambda) = c^T x + \lambda^T (b_1 - A_1 x) \\ s.t. & x \in X. \end{array}$$

Let $z^*_{L,ILP}(\lambda)$ be the optimal objective function value of $L_{ILP}(\lambda)$. Let x^* denote the optimal solution for ILP. Note that every x that is feasible for ILP is also feasible for $L_{ILP}(\lambda)$. Therefore:

$$z^*_{L,ILP}(\lambda) \leq c^T x^* + \lambda^T (b_1 - A_1 x^*) \leq z^*_{ILP},$$

and thus:

Theorem 2.3 *$z^*_{L,ILP}(\lambda)$ is a lower bound on z^*_{ILP}, the optimal solution to ILP, for any $\lambda \geq 0$, i.e., $z^*_{L,ILP}(\lambda) \leq z^*_{ILP}$ $(\forall \lambda \in \mathbb{R}^{m_1}_+)$.*

The Langrangian multipliers λ^* that provide the best lower bound are found by solving the so-called *Lagrangian dual problem*:

$$z^*_{L,ILP}(\lambda^*) = \max_{\lambda \geq 0} \left\{ z^*_{L,ILP}(\lambda) \right\}.$$

In general, to find strong lower bounds to z^*_{ILP}, it is required to have appropriate Lagrangian multipliers. Common (approximation) techniques for solving the Lagrangian dual of an integer linear programming problem are the *subgradient method* (Held and Karp, 1971) and the *bundle method* (see, e.g., Hiriart-Urruty and Lemarechal, 1993), that both involve iteratively updating the Lagrangian multipliers. In general it can not be guaranteed that $z^*_{L,ILP}(\lambda^*) = z^*_{ILP}$. The difference $z^*_{ILP} - z^*_{L,ILP}(\lambda^*)$ is referred to as the *Lagrangian duality gap*. We refer to Geoffrion (1974) for a more comprehensive discussion on Lagrangian relaxation techniques.

Another important relation in Lagrangian relaxation theory is that of $z^*_{L,ILP}(\lambda^*)$ with z^*_{LP}, the optimal solution to the LP relaxation (LP) of ILP (see, e.g., Geoffrion, 1974):

Theorem 2.4 *If ILP is an integer linear programming problem, then $z^*_{LP} \leq z^*_{L,ILP}(\lambda^*)$.*

Proof

$$\begin{aligned} z^*_{LP} &= \min_x \left\{ c^T x | A_1 x \geq b_1, A_2 x = b_2, x \geq 0 \right\} \\ &= \min_x \left\{ c^T x | - A_1 x \leq -b_1, -A_2 x = -b_2, x \geq 0 \right\} \\ &= \max_{v,u} \left\{ -u^T b_1 - v^T b_2 | - u^T A_1 - v^T A_2 \leq c^T, u \geq 0 \right\}, \end{aligned}$$

where u and v are decision variable vectors (of appropriate length) of the dual problem of LP. Substituting for u the optimal dual solution vector (u^*) that

corresponds to constraints $A_1x \geq b_1$ of LP, we have:

$$\begin{aligned} z^*_{LP} &= \max_v \left\{-u^{*T}b_1 - v^Tb_2| - v^TA_2 \leq c^T + u^{*T}A_1\right\} \\ &= \min_x \left\{c^Tx + u^{*T}(A_1x - b_1)|A_2x = b_2, x \geq 0\right\} \\ &\leq \min_x \left\{c^Tx + u^{*T}(A_1x - b_1)|A_2x = b_2, x \in \mathbb{Z}^n_+\right\} \\ &= z^*_{L,ILP}(u^*) \\ &\leq \max_{\lambda \geq 0} \left\{z^*_{L,ILP}(\lambda)\right\} \\ &= z^*_{L,ILP}(\lambda^*). \end{aligned}$$

□

In particular, when the integrality constraints in $L_{ILP}(\lambda)$ are redundant (we then say that the Lagrangian subproblem $L_{ILP}(\lambda)$ possesses the *integrality property*), all 'smaller than'-relations in the proof of Theorem 2.4 become equality-relations, since we have that:

$$z^*_{L,ILP}(\lambda) = z^*_{L,LP}(\lambda) \leq z^*_{LP} \ (\forall\lambda), \tag{2.6}$$

where $L_{LP}(\lambda)$ is the same as $L_{ILP}(\lambda)$ without the integrality constraints, and $z^*_{L,LP}(\lambda)$ is its optimal objective value. Hence, combining (2.6) and Theorem 2.4:

Corollary 2.1 *If the Lagrangian subproblem $L_{ILP}(\lambda)$ of integer linear programming problem ILP possesses the integrality property, then $z^*_{L,ILP}(\lambda^*) = z^*_{L,ILP}(u^*) = z^*_{LP}$.*

For integer linear programming problems for which the Lagrangian dual problem does not possess the integrality property, the optimal dual solution vector of the dual problem of LP provides a lower bound, since $z^*_{L,ILP}(u^*) \geq z^*_{LP}$ (see the proof of Theorem 2.4).

In the next section we apply the results of Theorem 2.4 and Corollary 2.1 to show how to embed Lagrangian relaxation in a column generation algorithm to obtain lower bounds in a branch-and-price tree, and to speed up convergence of the column generation algorithm (i.e., to alleviate the tailing-off effect).

2.4 Combining Lagrangian relaxation and column generation

It is well known that Lagrangian relaxation can complement column generation in that it can be used in every iteration of the column generation scheme to

compute a lower bound to the original problem with little additional computational effort (see, e.g., Van den Akker et al., 2000; Vanderbeck and Wolsey, 1996). The comparison of the Langrangian lower bound to an upper bound for ILP can be used to *fathom nodes* in a branch-and-price algorithm, to *fix variables*, or to *prove convergence of column generation* to alleviate the tailing-off effect of column generation. In this section we demonstrate the use of Lagrangian relaxation on the integer linear programming problem ILP of the previous section, for a specific constraint set $A_2x = b_2$, given by $\sum_{j \in J(i)} x_j = 1$ (for $i = 1, \ldots, m_2$, with $J = \bigcup_{i=1}^{m_2} J(i)$, and $J(i) \cap J(j) = \phi$ for $i \neq j$). This type of problem will play an important role in the remainder of this thesis, since it has the same form as the formulation of the resource loading problem. Hence we have:

$$\begin{aligned} ILP: \quad & \min \ z_{ILP} = c^T x \\ s.t. \quad & A_1 x \geq b_1 \\ & \sum_{j \in J(i)} x_j = 1 \ (i = 1, \ldots, m_2) \\ & x_j \in \mathbb{Z}_+ \ \left(j \in J = \bigcup_{i=1}^{m_2} J(i) \right). \end{aligned}$$

Note that constraints $\sum_{j \in J(i)} x_j = 1 \ (\forall i)$ and $x_j \in \mathbb{Z}_+$ imply that $x \in \{0,1\}^n$. The set of variable indices J is thus partitioned into m_2 disjoint sets $J(i)$, of which precisely one corresponding variable x_j $(j \in J(i))$ must be 1. Clearly, if we dualize $A_1 x \geq b_1$, the Lagrangian subproblem has the integrality property. Moreover, it can be solved greedily by identifying for each subset $J(i)$ one c_j with the smallest value, and setting the corresponding x_j to 1. We show how to use this in addition to column generation.

Suppose J and the subsets $J(i)$ are exponentially large, and we use branch-and-price to solve the LP relaxation of ILP (LP) to optimality. In each node of the branching tree we apply a column generation scheme to a restricted LP relaxation (RLP) of ILP, that has the following form:

$$\begin{aligned} RLP: \quad & \min_{\widetilde{x}} \ z_{RLP} = \widetilde{c}^T \widetilde{x} \\ s.t. \quad & \widetilde{A}_1 \widetilde{x} \geq b_1 \\ & \sum_{j \in \widetilde{J}(i)} \widetilde{x}_j = 1 \ (i = 1, \ldots, m_2, \widetilde{J}(i) \subseteq J(i), \widetilde{J}(i) \neq \phi) \\ & \widetilde{x}_j \geq 0 \ \left(j \in \widetilde{J} = \bigcup_{i=1}^{m_2} \widetilde{J}(i) \right), \end{aligned}$$

where $\widetilde{J}$ is a non-empty subset of the variables, $\widetilde{A}_1$ is the coefficient matrix and $\widetilde{c}$ the cost vector corresponding to variable vector $\widetilde{x}$.

The special structure of this problem implies that we can solve the pricing problem by solving m_2 separate pricing problems, one for each subset $J(i)$. If

we solve all the separate pricing problems, we identify for $i = 1, \ldots, m_2$ the variable x_j $(j \in J(i) \setminus \widetilde{J}(i))$ that has the lowest reduced costs.

Consider problem ILP once again. By relaxing constraints $A_1 x \geq b_1$ and introducing a Lagrangian multiplier vector λ $(\lambda \in \mathbb{R}_+^{m_1})$, we obtain the following Lagrangian subproblem of ILP:

$$\begin{array}{rll} L_{ILP}(\lambda): & \min\limits_{x} & z_{L,ILP}(\lambda) = c^T x + \lambda^T (b_1 - A_1 x) \\ & s.t. & \sum\limits_{j \in J(i)} x_j = 1 \ (i = 1, \ldots, m_2) \\ & & x_j \in \mathbb{Z}_+ \ (j \in J)\,, \end{array}$$

where $z^*_{L,ILP}(\lambda)$ is the optimal objective value of $L_{ILP}(\lambda)$. We can rewrite $z^*_{L,ILP}(\lambda)$ as follows:

$$\begin{aligned} z^*_{L,ILP}(\lambda) &= \min_x \ c^T x + \lambda^T b_1 - \lambda^T A_1 x \\ &= \min_x \ \left(c^T - \lambda^T A_1\right) x + \lambda^T b_1. \end{aligned}$$

Hence we have:

$$\begin{array}{rll} L_{ILP}(\lambda): & \min\limits_{x} & z_{L,ILP}(\lambda) = \left(c^T - \lambda^T A_1\right) x + \lambda^T b_1 \\ & s.t. & \sum\limits_{j \in J(i)} x_j = 1 \ (i = 1, \ldots, m_2) \\ & & x_j \in \mathbb{Z}_+ \ (j \in J)\,. \end{array} \tag{2.7}$$

The term $c^T - \lambda^T A_1$ is referred to as the *Lagrangian cost term*. For a given vector λ the optimal solution to $L_{ILP}(\lambda)$ can be determined as follows. The term $\lambda^T b_1$ is a constant. We optimize the term $\left(c^T - \lambda^T A_1\right) x$ by finding for each variable set $J(i)$ the x_j $(j \in J(i))$ that has the lowest Lagrangian cost factor. That x_j is set to 1; all other variables x_j $(j \in J(i))$ are set to 0. Hence we may conclude that $L_{ILP}(\lambda)$ possesses the integrality property. From equation (2.6) and Corollary 2.1 we know that:

$$z^*_{L,ILP}(\lambda) \leq z^*_{LP} = z^*_{L,ILP}(u^*) = z^*_{L,ILP}(\lambda^*) \leq z^*_{ILP}, \tag{2.8}$$

where u^* is the optimal dual solution vector of the dual of LP corresponding to constraints $A_1 x \geq b_1$, and $z^*_{L,ILP}(\lambda^*) = \max\limits_{\lambda} \left\{z^*_{L,ILP}(\lambda)\right\}$. The lower bound z^*_{LP} on z^*_{ILP} is obtained upon convergence of the column generation scheme (recall from Section 2.2.3 that upon convergence of column generation we have that $z^*_{RLP} = z^*_{LP} \leq z^*_{ILP}$). In the remainder we show that the advantage of using Lagrangian relaxation in a column generation scheme is that a solution to $L_{ILP}(\lambda)$ can easily be found in each iteration of the column generation scheme, thus providing a lower bound on both z^*_{LP} and z^*_{ILP} in each iteration. This allows nodes to be fathomed in an earlier stage.

The difficulty of solving $L_{ILP}(\lambda)$ in each iteration of the column generation scheme is that it requires that all variable sets $J(i)$ are *entirely* available, as

well as the entire coefficient matrix A_1. However, column generation involves a *subset* of the variables in an RLP. By rewriting $z_{L,ILP}(\lambda)$ into a convenient form, we show that we can determine a lower bound to ILP with little additional effort by using the information from a solution to an RLP and the information provided by solving a pricing algorithm for each variable set $J(i)$ $(i = 1, \ldots, m_2)$ after solving the RLP.

Since $\sum_{j \in J(i)} x_j = 1$ is a constraint in $L_{ILP}(\lambda)$, without loss of generality we can add a term $\sum_i (1 - \sum_{j \in J(i)} x_j)\mu_i$ (with $\mu_i \in \mathbb{R}$) to $z_{L,ILP}(\lambda)$, so that we have:

$$\begin{aligned} z^*_{L,ILP}(\lambda) &= \min_x \left(c^T - \lambda^T A_1\right) x + \sum_{i=1}^{m_2} (1 - \sum_{j \in J(i)} x_j)\mu_i + \lambda^T b_1 \\ &= \min_x \left(c^T - \lambda^T A_1\right) x - \sum_{i=1}^{m_2} \mu_i \sum_{j \in J(i)} x_j + \sum_{i=1}^{m_2} \mu_i + \lambda^T b_1 \\ &= \min_x \left(c^T - \widetilde{\mu}^T - \lambda^T A_1\right) x + \sum_{i=1}^{m_2} \mu_i + \lambda^T b_1, \end{aligned}$$

where $\widetilde{\mu}$ is an n-vector with elements $\widetilde{\mu_j} = \sum_{i \in R_j} \mu_i$, where R_j is the set of row indices that is covered by variable x_j in constraint set $\sum_{j \in J(i)} x_j = 1$ $(i = 1, \ldots, m_2)$. Note that in this example each R_j has one element, thus $\widetilde{\mu_j} = \mu_i$ if $j \in J(i)$ $(i = 1, \ldots, m_2)$. Hence (2.7) becomes:

$$\begin{aligned} L_{ILP}(\lambda): \quad & \min_x \; z_{L,ILP}(\lambda) = \left(c^T - \widetilde{\mu}^T - \lambda^T A_1\right) x + \sum_{i=1}^{m_2} \mu_i + \lambda^T b_1 \\ & s.t. \quad \sum_{j \in J(i)} x_j = 1 \;\; (i = 1, \ldots, m_2) \\ & \quad x_j \in \mathbb{Z}_+ \;\; (j \in J). \end{aligned} \tag{2.9}$$

Substituting the optimal dual solution vectors u^* and v^* of the dual of LP (DLP) for Lagrangian multiplier λ and multipliers μ in (2.9) yields that $c^T - \widetilde{\mu}^T - \lambda^T A_1$ becomes the (non-negative) reduced cost vector of variable vector x, and $\sum_{i=1}^{m_2} \mu_i + \lambda^T b_1$ becomes the objective value of the dual of LP.

After optimizing an RLP and m_2 subsequent pricing problems we optimize (2.9) as follows. Constraints $\sum_{j \in J(i)} x_j = 1$ and $x_j \in \mathbb{Z}_+$ $(j \in J)$ require that we set exactly one variable x_j from each set $J(i)$ to 1 $(i = 1, \ldots, m_2)$. The vector $\left(c^T - \widetilde{\mu}^T - \lambda^T A_1\right)$ with the substituted dual solution becomes the reduced cost vector for variable vector x. Since RLP is solved to optimality, all variables x_j contained in RLP have non-negative reduced costs. For all other variables not contained in RLP, and divided over the sets $J(i)$, we use the outcomes of

the pricing algorithms that were solved to find for each set $J(i)$ the variable with the lowest reduced cost. Hence, when the pricing algorithm regarding a set $J(i)$ finds the variable x_j with the most negative reduced costs, we set that x_j to 1 (and all other variables in that set to zero). If it finds no such variable, than an arbitrary variable of that set $J(i)$ contained in RLP with zero reduced costs is set to 1 (and all other variables in that set to zero). Finally, the term $\sum_{i=1}^{m_2} \mu_i + \lambda^T b_1$ with the substituted dual solution vectors u_{RLP} and v_{RLP} is equal to the objective value of $DRLP$. Since RLP is solved to optimality, we know from linear programming theory that the dual solution is equal to the primal solution. Recall that as long as in an arbitrary column generation iteration columns exist with negative reduced costs that are not yet added to RLP, the optimal solution to $DRLP$ is not feasible to DLP. Feasibility is obtained upon convergence of column generation. Hence from (2.8) we know that the lower bound obtained by using the solution to $DRLP$ as Lagrangian multipliers in $L_{ILP}(\lambda)$ is a lower bound on LP and ILP, i.e., $L_{ILP}(u_{RLP}) \leq z^*_{LP} \leq z^*_{ILP}$. Only upon convergence of column generation we have that $z^*_{L,ILP}(u_{RLP}) = z^*_{L,ILP}(u^*) = z^*_{LP} \leq z^*_{ILP}$. Hence, using the outcomes of the pricing algorithms, and the objective value of RLP, with little effort a Lagrangian lower bound on z^*_{LP} and z^*_{ILP} can be obtained each time an RLP is optimized in branch-and-price. The lower bound obtained by Lagrangian relaxation is known to zig-zag and converge slowly. However, each lower bound found in a node of the branch-and-price scheme applies to each column generation in that node, and to all nodes below that node. In Figure 2.4 we illustrate how to use this lower bound to interrupt the column generation scheme, to alleviate the tailing-off effect of column generation. In the figure we observe that when the Lagrangian lower bound meets the RLP upper bound, column generation has converged. The Lagrangian lower bound can also be used to fathom a node, if it exceeds the incumbent solution ($LB \geq UB$). Finally, the Lagrangian lower bound can also be used to fix variables. When the reduced cost of a variable x_j is larger than $UB - LB$ (where UB is the upper bound/incumbent solution, and LB the Lagrangian lower bound), we know from linear programming theory that $x_j = 0$ in any solution with a value less than UB. Hence, we can fix that variable in the current node and in all nodes below that node. The same result can also be applied to remove columns from the column pool (see, e.g., Freling, 1997). Analogously, when the reduced cost is smaller than $LB - UB$ then $x_j = 1$ in any solution with a value less than UB. There are other ways to exploit Lagrangian relaxation in combination with a column generation algorithm (Van den Akker et al., 2000), but these are beyond the scope of this thesis.

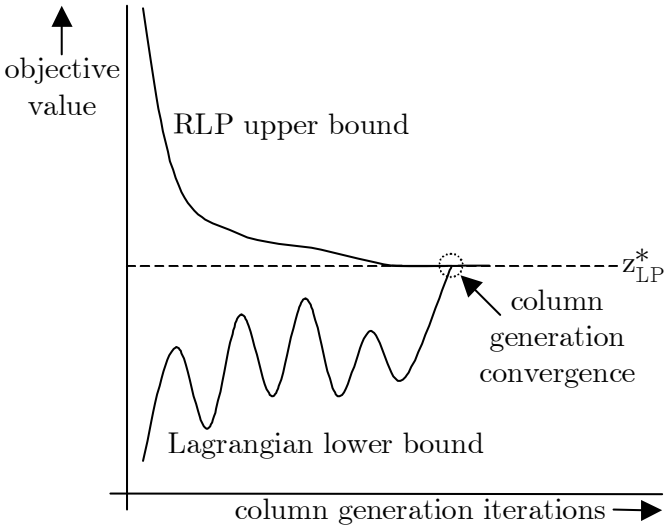

Figure 2.4: Alleviation of tailing-off effect in column generation.

2.5 Deterministic dynamic programming

Dynamic programming (DP) is a technique for solving many optimization problems (Bellman, 1957; Dreyfus and Law, 1977). DP is a decomposition technique that first decomposes the problem into a nested family of subproblems. The solutions to the original problem are obtained by either working backward from the end of the problem to the beginning (*backward dynamic programming*), or forward from the beginning to the end (*forward dynamic programming*). A distinction can be made between deterministic and probabilistic dynamic programming problems. In *stochastic* or *probabilistic DP*, the decisions have a stochastic outcome, and the goal is to determine the decisions that minimize the *expected* cost (or maximize the expected reward), while in *deterministic DP*, all decision variables yield a deterministic contribution to the objective. In this thesis we only discuss deterministic DP problems, and hence in this section we discuss this category of dynamic programming problems only. For literature concerning stochastic DP we refer to Ross (1983) and Sennott (1999).

Five characteristics can be distinguished that are common to dynamic programming applications:

1. The problem can be divided into *stages* t, with a *decision* x_t required at each stage.
2. Each stage t has a set of *states* $\{i_t\}$ associated with it. At any stage, a state holds all the information that is needed to make a decision.

3. The decision chosen at any stage determines how the state at the current stage is transformed into the state at the next stage, as well as the immediately earned reward or cost.

4. Given the current state, the optimal decision for each of the remaining stages must not depend on previously reached states or previously chosen decisions. This is the so-called *principle of optimality* for dynamic programming (Bellman, 1957).

5. If the states for the problem have been classified into one of T stages, there must be a *recursion* that relates the cost or reward earned during stages $t, t+1, \ldots, T$ to the cost or reward earned from stages $t+1, t+2, \ldots, T$.

In a forward DP algorithm, the recursion mentioned in the fifth characteristic can often be written as:

$$F_t(i_t) = \min_{x_t \in S_t} \{c_t(i_t, x_t) + F_{t-1}(i_{t-1}(i_t, x_t))\}, \tag{2.10}$$

where $c_t(i_t, x_t)$ is the cost (or reward in a maximization problem) function that returns the cost (or reward) for moving from state $i_{t-1}(i_t, x_t)$ to state i_t according to decision x_t, where $i_{t-1}(i_t, x_t)$ is the state from which i_t is reached, given decision x_t, and where $F_t(i_t)$ is the minimum cost (or maximum reward in a maximization problem) incurred in stages $1, 2, \ldots, t$, given state i_t in stage t. Optimal decisions can be made as follows. In the forward DP we assume there is a desired state we want the system to be in, in stage T (call it i_T). We first compute the $F_1(i_1)$ for all possible states in stage 1. We then apply (2.10) to calculate the $F_2(i_2)$'s in terms of the $F_1(i_1)$'s, and continue in this fashion until $F_T(i_T)$ (the desired state i_T in T) has been reached. We then determine the optimal decision in stage T that attains $F_T(i_T)$. This decision determines a state i_{T-1} in stage $T-1$ from which we arrive to state i_T during the last stage. We then determine the optimal stage $T-1$ decision, which in turn determines a state i_{T-2} in stage $T-2$, and continue until the optimal stage 1 decision is found.

Analogously, in a backward DP algorithm, the recursion mentioned in the fifth characteristic can often be written as:

$$F_t(i_t) = \min_{x_t \in S_t} \{c_t(i_t, x_t) + F_{t+1}(i_{t+1}(i_t, x_t))\}. \tag{2.11}$$

In a backward DP, we assume that the initial state in stage 1, i_1, is given. We begin by finding the optimal decision for each possible state in the final stage, and subsequently use the recursion in (2.11) to determine the functions $F_{T-1}(i_{T-1}), \ldots, F_1(i_1)$ (along with the optimal decisions for each stage). We then select the optimal decision from the set of decisions attaining $F_1(i_1)$. This decision determines a state i_2 in stage 2. In stage 2 we then choose any decision attaining $F_2(i_2)$, determining a state i_3 in stage 3, and continue until a decision has been chosen for each stage.

Dynamic programming algorithms are computationally efficient, as long as the state space does not become too large. E.g., for shortest path problems, dynamic programming clearly outperforms enumeration (Dreyfus and Law, 1977). However, when the state space becomes too large, implementation problems may occur (e.g., insufficient computer memory), or excessive computational time may be required to use dynamic programming to solve the problem. In that case other algorithms (e.g., branch-and-bound algorithms) may be more suitable.

In this thesis we use dynamic programming to solve pricing problems of branch-and-price algorithms. We show in Chapter 4 that finding a column with negative reduced costs can be translated into finding a longest path in an acyclic network with weights on the nodes. We solve these problems with forward dynamic programming.

Chapter 3

Model description

A theory has only the alternative
of being right or wrong.
A model has a third possibility:
it may be right, but irrelevant.

- Manfred Eigen (1927-)

3.1 Introduction

In this chapter we formulate the resource loading problem presented in Chapter 1 (Sections 1.2.2. and 1.3.1) as a mixed integer linear programming problem (MILP), and discuss the difficulties involved when modeling this problem.

The outline of this chapter is as follows. In Section 3.2 we formalize the problem and discuss the modeling assumptions. In the subsequent two sections we discuss two aspects of modeling the resource loading problem. First, in Section 3.3 we discuss modeling the capacity restrictions, which can easily be modeled with linear constraints and continuous variables. Subsequently, in Section 3.4 we discuss modeling the precedence relations, which is much less straightforward and requires the introduction of integer variables. In Section 3.5 we discuss modeling order tardiness. In Section 3.6 we use the observations of Sections 3.3 to 3.5 to formulate the mixed integer linear programming model of the resource loading problem that will be used throughout the remainder of this thesis. We refer to Appendix A for a complete list of the notation that is introduced in this chapter, and that is used throughout the remainder of this thesis.

3.2 Model assumptions and notations

We consider a job shop that consists of m independent machine groups $M_1, \ldots, M_m$. Independent in this context means that the machine groups do not share tools. On these machine groups, a set of n orders has to be loaded. As usual at the resource loading level, we discretize the planning horizon T into time buckets of (not necessarily) equal length (index $t = 0, \ldots, T$). In the remainder, we refer to these buckets as *periods*. Without loss of generality we assume that the unit of time is one *hour*, and that periods have a length of one *week*. Each order J_j $(j = 1, \ldots, n)$ consists of a chain of n_j jobs $(B_{1j}, \ldots, B_{n_j,j})$. No job can be started unless all predecessors have been completed. Job B_{bj} $(b = 1, \ldots, n_j)$ needs to be performed on machine group $\mu_{bj} \in \{M_1, \ldots, M_m\}$ for a given positive time p_{bj} (hours). In the resource loading problem we regard a machine group as one type of resource, while it may consist of several machines. We assume that the further disaggregation of a job into operations, which must be performed on machines, is done at the operational planning level.

A job B_{bj} may have a prespecified minimal duration of w_{bj} weeks (w_{bj} integer). A *minimal duration* is usually a result of technical limitations of resources, e.g., in a situation when no more than two persons can jointly perform a certain activity. Each order has a *release date* r_j before which its first job can not be started and a *due date* d_j (both are expressed in weeks) before which all its jobs should be completed. From the order release and due dates, and the minimal duration of the jobs, we calculate for each job B_{bj} its internal release date r_{bj} and due date d_{bj}. *Preemption* of jobs is allowed, i.e., the execution of any job may be spread over different, not necessarily subsequent, weeks.

For the resource loading problem we consider two types of resource capacities: *machine group capacity* and *operator capacity*. Since machine capacity investments are made at a higher, strategic level of decision making, they are assumed to be fixed for the resource loading problem. Machine group capacity is thus considered as regular capacity, that can not be expanded temporarily at the level of resource loading. We indicate the total capacity of machine group M_i in week t by $\overline{mc}_{it}$. Without loss of generality we assume that machine operators are fully cross-trained and can thus operate any machine in any machine group. An operator must be present at the machine group during the entire processing time. Moreover, we assume that an operator can operate only one machine at a time. For operators we distinguish regular and nonregular capacity (i.e., working overtime and hiring staff). Machine groups can thus be operated in *regular* and *nonregular operator time*. The total machine group capacity $\overline{mc}_{it}$ of machine group M_i in week t is the machine group capacity in the sum of regular and nonregular operator time. The capacity of machine group M_i in week t in regular operator time is indicated by mc_{it}. As a result, $\overline{mc}_{it} - mc_{it}$ is the machine group capacity in nonregular operator time. To elucidate why it is necessary to make a distinction between machine group

capacity in regular and in nonregular time, consider the situation where a company lets operators work on a machine group in 1 shift of 8 hours per day, 5 days a week, i.e., $mc_{it} = 40$. Suppose there are 3 operators, i.e., the operator capacity (indicated by c_t) is 120 hours. The total machine group capacity is $\overline{mc}_{it} = 100$ hours, and the workload to be performed is 50 hours. Although total operator capacity and total machine group capacity are sufficient to process the workload ($c_t = 120 > 50$ and $\overline{mc}_{it} = 100 > 50$), the machine group can be operated in regular time for only 40 hours. The remaining 10 hours must be performed with nonregular operator capacity. In general, if capacity is insufficient to complete all jobs before their due date in regular time, short-term production capacity may be expanded at extra costs. There are three options to expand short-term production capacity:

- *Working in overtime.* Of course, extra labor costs are involved, and it is assumed that the unit overtime cost is a known constant. Furthermore, collective labor agreements put an upper bound on the number of overtime production hours per week. The assigned number of overtime hours in week t is indicated by O_t.
- *Hiring extra operators.* It is assumed that the cost per hour of hiring one extra operator is a known constant. We distinguish hiring operators in regular operator time and in nonregular operator time. The number of hired production hours in week t in *regular* operator time is indicated by H_t^R, and in *nonregular* time this is indicated by H_t^N. Without loss of generality we assume that the cost of hiring in regular operator time is the same as the cost of hiring in nonregular operator time. We assume there is an upper bound on the total hirable operator capacity.
- *Outsourcing (subcontracting).* While certain jobs can not be outsourced, other jobs can. It is assumed that the total cost of outsourcing a job is proportional to the amount of work. The number of outsourced production hours for machine group M_i in week t is indicated by S_{it}.

To specify a feasible resource loading we use the concept of an order plan, and the concept of an order schedule. The concept of an *order plan* is central in our formulation. An order plan $a_{j\pi}$ ($\pi \in \Pi_j$) for an order J_j ($j = 1, \ldots, n$) specifies for each job B_{bj} ($b = 1, \ldots, n_j$) the weeks in which B_{bj} is allowed to be performed. Accordingly, each order plan is a 0-1 vector $a_{j\pi} = \big(a_{1,j,0,\pi}, \ldots, a_{1,j,T,\pi}, \ldots, a_{n_j,j,0,\pi}, \ldots, a_{n_j,j,T,\pi}\big)$, where $a_{bjt\pi} = 1$ ($b = 1, \ldots, n_j, t = 0, \ldots, T$) if job B_{bj} is allowed to be performed in week t. A feasible order plan for order J_j is an order plan in which no job is allowed to be performed before its release date, and in which all precedence relations are met. We allow jobs to be tardy in the resource loading problem (recall that we aim to minimize both the use of nonregular capacity and the total tardiness of jobs), so we do allow jobs B_{bj} to be performed after their due dates d_{bj}. We further discuss order tardiness in Section 3.5.

There are two possibilities to deal with the precedence relations. First, they can be enforced at the resource loading level by forbidding jobs with a precedence relation to be processed in the same period. For this purpose we introduce a parameter δ that indicates the number of weeks between the completion time of a job, and the start time of its successor. We will only consider the cases $\delta = 0$ and $\delta = 1$. When $\delta = 1$, we speak of a *one-job-per-week policy*. This implies that predecessors (successors) of a job must complete (start) in the week before (after) this job starts (completes). Although the one-job-per-week policy is quite common in practice (for instance, the metal working plant that motivated this research uses this policy), it may have quite a negative impact on the order lead time, particularly when the time buckets are quite long compared to the work contents of the jobs. Therefore, we also consider the case that the first week in which a specific job is allowed to be performed overlaps with the last week in which its preceding job is allowed to be performed. The precedence relation then has to be enforced in the detailed planning. In this case we set δ to 0 weeks. As a result, order plans may be generated in which a number of jobs of the same order are allowed to be performed in the same week. This may be possible if these jobs were allowed to be produced on different machine resources simultaneously. However, it does not guarantee that these jobs can be performed in succession in the same week. Hence it does not guarantee that precedence relations between these jobs are satisfied *within* the week. For example when three jobs of the same order may be performed in the same week, and the processing time of each of these jobs is 40 hours, their total duration is 120 hours. It is unlikely that these jobs can be produced in succession in the same week. To limit these possibilities, when $\delta = 0$, we impose an additional restriction that at most κ jobs of the same order are allowed to be performed in the same week. Either case ($\delta = 0$ and $\delta = 1$) is easily modeled using order plans, and this is a pro of using the concept. The con is that there are exponentially many order plans, and hence any such type of formulation poses a formidable computation challenge.

We consider only order plans $a_{j\pi}$ that can not be dominated by other order plans. This implies that an order plan $a_{j\pi}$ has no element $a_{bjt\pi} = 0$ that can be changed to 1 without making the order plan infeasible.

In addition to the order plans, which specify when jobs are *allowed* to be performed, we use the concept of an *order schedule*. An order schedule specifies a realization of an order plan, i.e., it specifies how many hours of each job are performed in each week. An order schedule for an order J_j is characterized by a vector $Y = \big(Y_{1j0}, \ldots, Y_{1jT}, \ldots, Y_{n_jj0}, \ldots, Y_{n_jjT}\big)$, where Y_{bjt} denotes the fraction of job B_{bj} performed in week t. As a consequence, $p_{bj}Y_{bjt}$ denotes the number of hours processing time of job B_{bj} performed in week t. Obviously we want to make sure that for each order J_j the order plan and order schedule are consistent (we come back to this point in our mathematical model formulation in Section 3.4). A *feasible resource loading* implies that all order plans are feasible, and there is sufficient (regular and nonregular) operator and machine

group capacity to process all (consistent) order schedules. The aim of the resource loading model is to find a feasible loading that minimizes the total costs that result from short-term capacity expansions and the penalties incurred by tardy orders.

The decision problem that corresponds to the resource loading problem is NP-complete. This can be proven by reduction to the 3-machine job shop problem with unit processing times. This problem is proven to be NP-complete by Lenstra and Rinnooy Kan (1979).

3.3 Modeling resource capacity restrictions

Resource capacity restrictions can easily be modeled in linear constraints. This is the most straightforward part of our model. To model the capacity constraints we use the aforementioned order schedule concept with decision variables Y_{bjt} ($b = 1, \ldots, n_j, j = J_1, \ldots, J_n, t = 0, \ldots, T$), that indicate the fraction of job B_{bj} that is assigned to week t. These variables play a key role in resource loading, since all information concerning production schedules and capacity utilization can be calculated from these decision variables. The order schedules can be read directly from the Y_{bjt} variables. With respect to the capacity utilization, note that each (partial) assignment of a job B_{bj} to a week t requires a matching amount of operator and machine group capacity. With p_{bj} denoting the required production processing time (in hours) of job B_{bj}, $p_{bj}Y_{bjt}$ yields the number of hours that job B_{bj} is processed in week t. By adding the $p_{bj}Y_{bjt}$'s for a week t we can determine the assigned workload to machine groups and operators in that week t, and with that the regular and nonregular capacity that is required to meet the capacity demand in that week. The machine group and operator capacity restrictions can now be formulated as follows (decision variables are denoted by capitals):

$$\sum_{b,j} p_{bj} Y_{bjt} \leq c_t + O_t + H_t^R + H_t^N + \sum_{i=1}^{m} S_{it} \ (\forall t)\,, \tag{3.1}$$

$$\sum_{\{(b,j)|\mu_{bj}=M_i\}} p_{bj} Y_{bjt} \leq mc_{it} + U_{it} + S_{it} \ (\forall i, t)\,, \tag{3.2}$$

$$U_{it} \leq \overline{mc}_{it} - mc_{it} \ (\forall i, t)\,, \tag{3.3}$$

$$\sum_{i=1}^{m} U_{it} = O_t + H_t^N \ (\forall t)\,, \tag{3.4}$$

$$\sum_{t=r_j}^{T} Y_{bjt} = 1 \ (\forall b, j)\,, \tag{3.5}$$

$$Y_{bjt} \leq \frac{1}{w_{bj}} \ (\forall b, j, t)\,, \tag{3.6}$$

$$O_t \leq o_t \ (\forall t)\,, \tag{3.7}$$

$$\begin{aligned} H_t^R + H_t^N &\leq h_t \ (\forall t)\,, && (3.8)\\ \sum_{i=1}^{m} S_{it} &\leq s_t \ (\forall i,t)\,, && (3.9)\\ \text{all variables} &\geq 0. && (3.10) \end{aligned}$$

where $t = 0,\ldots,T$, $b = 1,\ldots,n_j$, $j = J_1,\ldots,J_n$, $i = 1,\ldots,m$. Constraints (3.1) stipulate operator capacity restrictions. The left-hand side is the sum of all assigned workload in a week t, and the right-hand side is the sum of all available operator capacity: regular (c_t), overtime (O_t), hiring staff in regular time (H_t^R), hiring staff in nonregular time (H_t^N), and outsourcing capacity (S_{it}). Constraints (3.2)-(3.4) form the machine group capacity restrictions. As mentioned in Section 3.2 we distinguish machine group capacity in regular operator time, and machine group capacity in nonregular operator time. Auxiliary variables U_{it} indicate the used machine group capacity in nonregular operator time. Constraints (3.3) restrict variables U_{it}. The left-hand side of (3.2) is the total amount of workload assigned to machine group M_i, and the right-hand side is the sum of the available machine group capacity in regular operator time (mc_{it}), the used machine group capacity in nonregular operator time (U_{it}), and the outsourcing capacity on machine group M_i (S_{it}). Constraints (3.4) stipulate that the used machine group capacity in nonregular operator time is equal to the sum of the overtime capacity (O_t) and the hiring capacity in nonregular operator time (H_t^N). Constraints (3.5) guarantee that all work is done. Constraints (3.6) guarantee a minimal duration w_{bj} for job B_{bj} by limiting the fraction Y_{bjt} per week t to $\frac{1}{w_{bj}}$ ($w_{bj} \geq 1$). The remaining constraints (3.7)-(3.10) are the variable upper and lower bounds. Note that constraints (3.9) set an upper bound on the *total* subcontracted capacity in each week t. A case where a machine group M_i is unique in that no work assigned to that machine group can be outsourced is easily modeled by setting an additional upper bound on outsourcing for this machine group M_i: $S_{it} \leq 0$ $(\forall t)$.

Let parameters $\overline{o}$, $\overline{h}$, and $\overline{s}$ specify the costs of using one hour of nonregular capacity (overtime, hiring staff, outsourcing respectively). When we omit tardiness penalties, the objective function can be formulated as follows:

$$\min \sum_{t=0}^{T} \left(\overline{o} O_t + \overline{h} \left(H_t^R + H_t^N \right) + \overline{s} \sum_{i=1}^{m} S_{it} \right). \tag{3.11}$$

Objective function (3.11) thus only minimizes the costs of the use of nonregular capacity. In Section 3.5 we add a term to this objective function to penalize order tardiness.

Note that so far we have neglected the precedence constraints among jobs. They will be modeled in the next section, where we discuss the difficulties

in modeling precedence constraints by analyzing different formulations. Unless stated otherwise, constraints (3.1)-(3.10) and objective function (3.11) are used as a basis in all of these formulations.

3.4 Modeling precedence relations

By introducing precedence constraints in the (partial) model of the resource loading in the previous section, the complexity of the problem is increased enormously. A straightforward formulation is to model the precedence constraints in terms of an assignment problem. A difficulty is that we allow jobs to be assigned (partly) to more than one week. To illustrate this, consider the following example formulation for the case where jobs are assigned to one week only:

$$\sum_{t=r_j}^{T} Z_{bjt} = 1 \ (\forall b,j)\,, \tag{3.12}$$

$$\sum_{t=r_j}^{T} tZ_{bjt} + \delta \leq \sum_{t=r_j}^{T} tZ_{sjt} \ (\forall b, s \in DS_b \neq \emptyset, j)\,, \tag{3.13}$$

$$\sum_{b,j} p_{bj} Z_{bjt} \leq c_t + O_t + H_t^R + H_t^N + \sum_i S_{it} \ (\forall t)\,, \tag{3.14}$$

$$\sum_{\{(b,j)|\mu_{bj}=M_i\}} p_{bj} Z_{bjt} \leq mc_{it} + U_{it} + S_{it} \ (\forall i,t)\,, \tag{3.15}$$

$$U_{it} \leq \overline{mc}_{it} - mc_{it} \ (\forall i,t)\,, \tag{3.16}$$

$$\sum_{i=1}^{m} U_{it} = O_t + H_t^N \ (\forall t)\,, \tag{3.17}$$

$$Z_{bjt} \in \{0,1\}, O_t, H_t^R, H_t^N, S_{it} \geq 0 \ (\forall i,t)\,, \tag{3.18}$$

where binary variable Z_{bjt} takes value 1 if and only if job B_{bj} is assigned to week t. Constraints (3.12) assign each job B_{bj} to one week. Constraints (3.13) make sure that job B_{bj} is assigned in or before the week to which its direct successors $s \in DS_b$ are assigned. A minimum time lag between adjacent jobs is added as a parameter δ to the left hand side of this constraint. When $\delta = 1$, two adjacent jobs may not be assigned to the same week (this is the so-called one-job-per-week policy). When $\delta = 0$, this policy is not imposed. Since the set DS_b may contain more than one job, this formulation thus allows generalized precedence relations. Analogous to constraints (3.1)-(3.4), constraints (3.14)-(3.17) form the machine group and operator capacity constraints.

The number of binary variables ($(T+1)\sum_j n_j$), and the number of constraints ($\sum_j n_j + \sum_{b,j} |DS_b| + (m+1)(T+1)$) in this formulation is considerable. The complexity increases even more when we allow jobs to be assigned to more

than one week (recall from Section 3.2 that the number of order plans is exponentially large). We illustrate this with the following example, in which we allow jobs to be assigned to more than one week. We introduce binary decision variables A_{bjt}, that indicate whether a job B_{bj} may be started in week t. In this formulation these variables can thus constitute the order plans, i.e., the time intervals in which the jobs are allowed to be performed. The order schedule is formed by variable Y_{bjt} that indicates the fraction of job B_{bj} that is assigned to week t.

$$\sum_{t=r_j}^{T} A_{bjt} = 1 \ (\forall b, j)\,, \tag{3.19}$$

$$\sum_{t=r_j}^{T} tA_{bjt} + w_{bj} + \delta - 1 \leq \sum_{t=r_j}^{T} tA_{sjt} \ (\forall b, s \in DS_b \neq \emptyset, j)\,, \tag{3.20}$$

$$\sum_{t=r_j}^{T} tA_{bjt} + w_{bj} - 1 \leq d_j \ (\forall b, j | DS_b = \emptyset)\,, \tag{3.21}$$

$$Y_{bjt}, A_{bjt} = 0 \ (\forall b, j, t < r_j)\,, \tag{3.22}$$

$$\sum_{t=r_j}^{T} Y_{bjt} = 1 \ (\forall b, j)\,, \tag{3.23}$$

$$Y_{bjt} \leq \frac{\sum_{\tau=r_j}^{t} A_{bj\tau} - \sum_{\tau=r_j}^{t+\delta-1} A_{sj\tau}}{w_{bj}} \quad (\forall b, s \in DS_b \neq \emptyset, j, t)\,, \tag{3.24}$$

$$Y_{bjt} \leq \frac{\sum_{\tau=r_j}^{t} A_{bj\tau}}{w_{bj}} \ (\forall b, j, t | DS_b = \emptyset) \tag{3.25}$$

$$A_{bjt} \in \{0,1\} \ (\forall b, j, t)\,. \tag{3.26}$$

Constraints (3.19) make sure that each job is started. Constraints (3.20) and (3.21) make sure that job B_{bj} is completed before its successor s ($s \in DS_b$), and due date d_j respectively. In these constraints we use that $\sum_{t=r_j}^{T} tA_{bjt}$ is the week in which job B_{bj} may start. Constraints (3.22) stipulate that variables Y_{bjt} and A_{bjt} can have nonzero values only for $t \in [r_j, d_j]$. Constraints (3.24) stipulate that job B_{bj} is not allowed to be performed, once job B_{sj} may be started. The term $\sum_{\tau=r_j}^{t} A_{bj\tau} - \sum_{\tau=r_j}^{t+\delta-1} A_{sj\tau}$ has value 1 for $t \in [\sum_{\tau=r_j}^{T} \tau A_{bj\tau}, \sum_{\tau=r_j}^{T} \tau A_{bj\tau} + w_{bj} + \delta - 1]$ (i.e., the week in which job B_{bj} is allowed to be performed), and value 0 for other t. Constraints (3.25) form an upper bound on the Y_{bjt}-variables for jobs B_{bj} that have no direct successors. Constraints (3.1)-(3.10) of the previous section must be added to this formulation to model the capacity constraints. Although this formulation has the same number of binary variables as the previous example, the number of constraints has increased drastically ($(T+3)\sum_j n_j + (m+1)(T+1)$ in the

case of path-type precedence constraints). This formulation will demand huge computational effort, and optimal results for cases of reasonable size are not likely to be found. Equivalent formulations are possible, e.g., using integer (non-binary) decision variables that indicate the allowed duration of job B_{bj}, but these lead to formulations of equivalent size and complexity.

Snoep (1995) has tried to model precedence constraints in an LP-formulation for a similar problem by penalizing non-admissable overlap. However, this approach has the nasty side-effect that a job must always be performed in the week just before the start time of its successor. Van Assen (1996) has also made an attempt to model precedence constraints in an LP formulation, by using slack variables to model precedence relations as 'soft' constraints. The slack variables, that measure the extent in which precedence constraints are violated, are then added to the objective function with a penalty. This approximation method also did not lead to satisfactory results. The latter research did, however, lead to the idea to develop a mixed integer linear programming model that uses binary variables to select a feasible *order schedule* for each order, thereby minimizing the use of nonregular capacity that is implied by these order schedules. In our research this idea was further explored, which resulted in a formulation where we use binary variables to select *order plans*. While the number of feasible order schedules is infinite, the number of feasible order plans is not infinite, yet still exponential. Instead of modeling these order plans in our formulation by constraints, such as in the example above, we represent the order plans by binary columns (recall the 0-1 vector $a_{j\pi} = (a_{bjt\pi})$ introduced in Section 3.2), and use these columns as input of the model. Since there are exponentially many order plans, we have opted for implicit column generation. Thus we work with a subset of all feasible columns, and generate new columns when they are needed (see Chapter 2 for a discussion on column generation). This leads to the following formulation:

$$\sum_{\pi \in \Pi_j} X_{j\pi} = 1 \ (\forall j), \tag{3.27}$$

$$Y_{bjt} \leq \frac{\sum_{\pi \in \Pi_j} a_{bjt\pi} X_{j\pi}}{w_{bj}} \ (\forall b, j, t), \tag{3.28}$$

$$\sum_{t=r_j}^{T} Y_{bjt} = 1 \ (\forall b, j), \tag{3.29}$$

$$X_{j\pi} \in \{0,1\} \ (\forall j, \pi \in \Pi_j), \tag{3.30}$$

where binary variable $X_{j\pi}$ takes value 1 if order plan $a_{j\pi}$ is selected for order J_j ($\pi \in \Pi_j$). Constraints (3.27) and (3.30) make sure that exactly one order plan is selected for each order J_j. Constraints (3.28) make sure that the order schedule (formed by variables Y_{bjt}) and the order plan (formed by $\sum_{\pi \in \Pi_j} a_{bjt\pi} X_{j\pi}$) are consistent. Constraints (3.29) make sure that all work is done. Constraints

(3.1)–(3.4) and (3.7)-(3.10) of the previous section must be added to this formulation to model the capacity constraints. Note that in this formulation we no longer need integer variables to define the time intervals in which jobs are allowed to be performed. The order plans that define these are represented by vectors $a_{j\pi}$. In this formulation we have $(T+1)\sum_j n_j$ continuous variables, $\sum_j |\Pi_j|$ binary variables, and $n + (T+2)\sum_j n_j$ constraints.

3.5 Modeling order tardiness

The *lateness* of an order or job is the difference between its completion time and its due date, measured in weeks. The *tardiness* of an order or job is its lateness if it fails to meet its due date, i.e., if the lateness is positive, or zero otherwise. In the literature, there are several commonly used performance measures for order lateness (see, e.g., Baker, 1974):

- maximum (weighted) order lateness,
- maximum (weighted) order tardiness,
- mean (weighted) tardiness,
- mean (weighted) lateness,
- number of tardy orders.

In our model we choose a strategy that complies with the concept of order plans. In the objective function we penalize a selected order plan in which jobs are *allowed* to be tardy. We thus penalize the possibility of tardiness in an order plan, rather than the real tardiness in an order schedule. Of course, if in an optimal solution an order is not tardy in its order schedule, the solution will not contain an order plan for this order in which tardiness is allowed.

We calculate for each order plan $a_{j\pi}$ its tardiness $\rho_{j\pi}$, which is defined as the tardiness of job $B_{n_j,j}$ in order plan $a_{j\pi}$ with respect to the (external) order due date d_j. Thus, when $CT_{bj\pi}$ is the last week in which job B_{bj} is allowed to be produced in order plan $a_{j\pi}$, the allowed tardiness is given by:

$$\rho_{j\pi} = \max\left\{0, CT_{n_j,j\pi} - d_j\right\}.$$

The *tardiness penalty* of order plan $a_{j\pi}$ of order J_j is given as $\rho_{j\pi}\theta$, where θ is a cost multiplier. Using binary variable $X_{j\pi}$ that indicates whether order plan $a_{j\pi}$ of order J_j is selected, we formulate the objective function as follows:

$$\min \sum_{t=0}^{T} \left(\overline{o} O_t + \overline{h}\left(H_t^R + H_t^N\right) + \overline{s} \sum_{i=1}^{m} S_{it} \right) + \sum_{j=1}^{n} \sum_{\pi \in \Pi_j} \rho_{j\pi} X_{j\pi} \theta. \qquad (3.31)$$

3.6 Synthesis

Combining the capacity constraints in Section 3.3, the precedence constraints in Section 3.4 and the objective function in Section 3.5 yields the following (mixed) integer linear programming formulation (ILP) of the resource loading problem:

$$ILP: z^*_{ILP} = \min \sum_{t=0}^{T} \left(\overline{o} O_t + \overline{h} \left(H_t^R + H_t^N \right) + \overline{s} \sum_{i=1}^{m} S_{it} \right) + \sum_{j=1}^{n} \sum_{\pi \in \Pi_j} \rho_{j\pi} X_{j\pi} \theta,$$

subject to:

$$\sum_{\pi \in \Pi_j} X_{j\pi} = 1 \ (\forall j), \tag{3.32}$$

$$Y_{bjt} - \frac{\sum_{\pi \in \Pi_j} a_{bjt\pi} X_{j\pi}}{w_{bj}} \leq 0 \ (\forall b, j, t), \tag{3.33}$$

$$\sum_{t=r_j}^{T} Y_{bjt} = 1 \ (\forall b, j), \tag{3.34}$$

$$\sum_{b,j} p_{bj} Y_{bjt} \leq c_t + O_t + H_t^R + H_t^N + \sum_{i=1}^{m} S_{it} \ (\forall t), \tag{3.35}$$

$$\sum_{\{(b,j)|\mu_{bj}=M_i\}} p_{bj} Y_{bjt} \leq mc_{it} + U_{it} + S_{it} \ (\forall i, t), \tag{3.36}$$

$$U_{it} \leq \overline{mc}_{it} - mc_{it} \ (\forall i, t), \tag{3.37}$$

$$\sum_{i=1}^{m} U_{it} = O_t + H_t^N \ (\forall t), \tag{3.38}$$

$$O_t \leq o_t \ (\forall t), \tag{3.39}$$

$$H_t^R + H_t^N \leq h_t \ (\forall t), \tag{3.40}$$

$$\sum_{i=1}^{m} S_{it} \leq s_t \ (\forall i, t), \tag{3.41}$$

$$\text{all variables} \geq 0, \tag{3.42}$$

$$X_{j\pi} \in \{0, 1\} \ (\forall j, \pi \in \Pi_j \subset \Pi). \tag{3.43}$$

The linear programming relaxation of this problem (LP) is obtained by replacing the integrality constraints (3.43) by $X_{j\pi} \geq 0$ $(\forall j, \pi \in \Pi_j)$. In the next chapter we discuss optimizing the ILP by various branch-and-price techniques. The general idea is that we first optimize the LP by performing column generation on a restricted linear programming relaxation (RLP) of the LP, in which for each order J_j we consider a subset $\widetilde{\Pi}_j$ of all feasible columns Π_j for that order. The pricing algorithm generates columns $a_{j\pi}$ for order J_j $(\pi \in \Pi_j \backslash \widetilde{\Pi}_j)$,

and adds these columns to $\widetilde{\Pi}_j$ when they have negative reduced costs. After optimizing the LP we perform branch-and-bound in conjunction with column generation to find feasible solutions to the ILP.

Chapter 4

Branch-and-price techniques for resource loading

Life is what happens to you
while you're busy making other plans.

- John Lennon (1940-1980)

4.1 Introduction

In this chapter we discuss the branch-and-price techniques that we apply to solve the mixed integer linear programming model (ILP) of the resource loading problem of Section 3.6. Recall that in Section 2.2 we discussed the basic elements of the branch-and-price method applied to a large integer linear programming model. Figure 4.1 displays the basic steps of a branch-and-price algorithm. In this chapter we discuss each of these steps in detail for the branch-and price algorithms for resource loading. The branch-and-price methods solve the LP relaxation of the ILP (Section 3.6) of the resource loading problem by column generation. The column generation scheme starts from a restricted LP relaxation (RLP) that contains at least one order plan for each order. The order plans for the initial RLP must constitute a primal solution to the LP relaxation of the resource loading problem, which we find by applying column generation to a Phase I LP relaxation equivalent to that of the two-phase simplex method.

The LP is optimized by performing column generation on the RLP. A

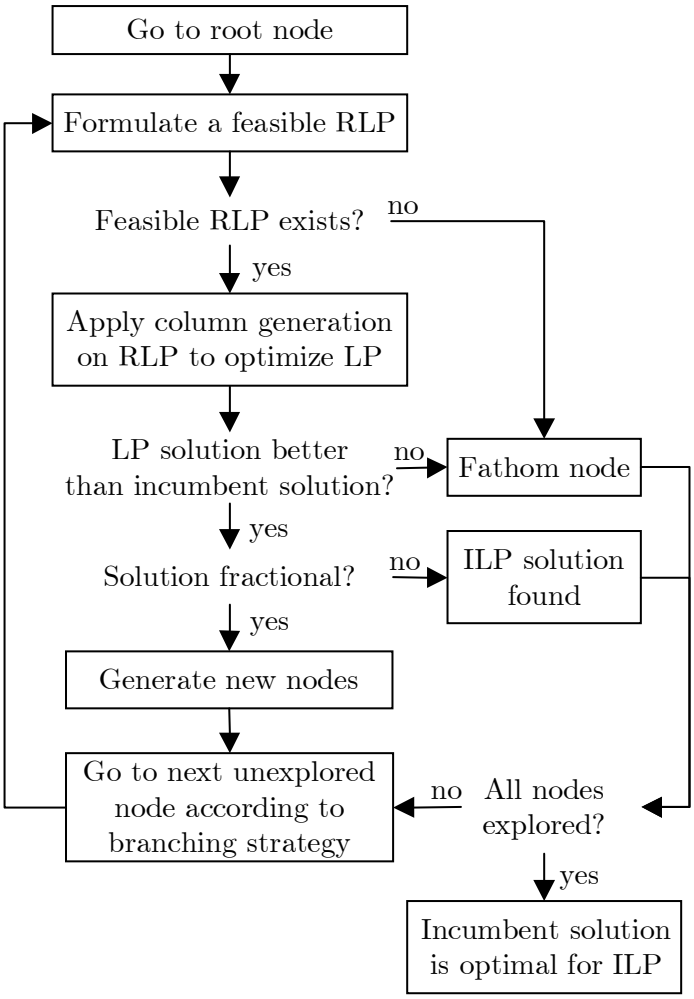

Figure 4.1: Branch-and-price scheme.

pricing algorithm determines whether order plans exist that may improve the solution of the RLP, and, provided that such order plans exist, generates order plans in each iteration. The column generation scheme terminates when no order plans exist that may improve the solution of the RLP. In that case, an optimal solution to the LP has been found.

When an optimal LP solution is found, it is usually not integral, and therefore not feasible to the ILP. The methods presented in this chapter exploit different branch-and-bound strategies to find an optimal or approximate solution to the ILP. A branching scheme partitions the solution space of the LP in a node, and hence in each node we consider a new linear program. As a result, in each node of the branching tree we must perform the column generation scheme to optimize the new LP relaxation in that node. This leads to so-called branch-and-price methods.

In addition to the elements of the branch-and-price methods mentioned above, heuristics can play an important role in these methods. We may use a heuristic to determine a primal solution to the resource loading problem to start up the column generation algorithm. We may also use heuristics to determine or improve the incumbent solution.

Another important element of our branch-and-price methods is the determination of Lagrangian lower bounds, which are used in conjunction with the incumbent solution (upper bound) to fathom nodes or terminate the column generation scheme.

The outline of this chapter is as follows. In Section 4.2 we discuss the basic form of the RLP. In Section 4.3 we present the pricing algorithm, which is the primary issue in the implementation of a column generation algorithm. Our pricing algorithm is a dynamic programming algorithm that may generate more than one candidate column per iteration. We discuss the issue of which and how many columns should be added to the RLP in each column generation iteration. In Section 4.4 we discuss various branching strategies for finding a feasible solution to the ILP. In Section 4.5 we discuss how to obtain lower bounds to the ILP by Lagrangian relaxation. In Section 4.6 we present several heuristics that can be used stand-alone to solve the resource loading problem (e.g., to find the initial RLP), and heuristics that can be used in conjunction with the branch-and-price algorithm to improve upper bounds to the ILP.

4.2 Restricted linear programming relaxation

4.2.1 Basic form

The linear programming relaxation of the ILP is obtained by omitting the integrality constraints on the variables $X_{j\pi}$. As mentioned before, the ILP, and its linear programming relaxation LP have a huge number of columns,

since the number of order plans for an order J_j (i.e., $|\Pi_j|$) is exponentially large. Consider for example an order J_j with n_j jobs, release date r_j and due date d_j. When we assign all jobs as early as possible, and allow them only to be performed for their minimal duration, there are $d_j - r_j + 1 - \sum_{b=1}^{n_j}(w_{bj} + \delta)$ weeks that remain unassigned for order J_j, where $\delta = 0$ when a job may start in the week that its predecessor has completed. We refer to these weeks as *slack weeks*. If we allow only order plans that are not dominated, we must divide the slack weeks over the n_j jobs. The number of ways to divide k slack weeks over n_j jobs is $\binom{n_j+k-1}{k}$. In combinatorics this counting problem can be translated into drawing k items from a collection of n_j different items (the job numbers), where repetitions are allowed, and the arrangement of the items is irrelevant. The result of this problem is known as a redundant combination. Hence, when an order has n_j jobs and k slack weeks, there are $\binom{n_j+k-1}{k}$ different feasible order plans for this order.

In the *RLP* we consider a subset $\overline{\Pi}$ of all feasible columns Π. The general form of the *RLP* is as follows:

$$RLP : z^*_{RLP} = \min \sum_{t=0}^{T} \left(\overline{o} O_t + \overline{h} \left(H_t^R + H_t^N \right) + \overline{s} \sum_{i=1}^{m} S_{it} \right) + \sum_{j=1}^{n} \sum_{\pi \in \overline{\Pi}_j} \rho_{j\pi} X_{j\pi} \theta,$$

subject to:

$$\sum_{\pi \in \overline{\Pi}_j} X_{j\pi} = 1 \ (\forall j), \tag{4.1}$$

$$w_{bj} Y_{bjt} - \sum_{\pi \in \overline{\Pi}_j} a_{bjt\pi} X_{j\pi} \leq 0 \ (\forall b, j, t), \tag{4.2}$$

$$\sum_{t=r_j}^{T} Y_{bjt} = 1 \ (\forall b, j), \tag{4.3}$$

$$\sum_{b,j} p_{bj} Y_{bjt} \leq c_t + O_t + H_t^R + H_t^N + \sum_{i=1}^{m} S_{it} \ (\forall t),$$

$$\sum_{\{(b,j)|\mu_{bj}=M_i\}} p_{bj} Y_{bjt} \leq mc_{it} + U_{it} + S_{it} \ (\forall i, t),$$

$$U_{it} \leq \overline{mc}_{it} - mc_{it} \ (\forall i, t),$$

$$\sum_{i=1}^{m} U_{it} = O_t + H_t^N \ (\forall t),$$

$$O_t \leq o_t \ (\forall t),$$

$$H_t^R + H_t^N \leq h_t \ (\forall t),$$

$$\begin{aligned} \sum_{i=1}^{m} S_{it} &\leq s_t \ (\forall t)\,, \\ \text{all variables} &\geq 0, \end{aligned}$$

where the sets $\overline{\Pi}_j$ are non-empty. Since the constraint $\sum_{\pi \in \Pi_j} X_{j\pi} = 1 \ (\forall j)$ in LP stipulates that for each order one order plan must be selected, any RLP must contain at least one order plan for each order. We discuss in Section 4.2.2 how we obtain these order plans.

A feasible LP solution in general is not integral. Suppose in an LP solution for an order J_j there are at least two order plans $a_{j\pi}$ ($\pi \in \overline{\Pi}_j$), for which $X_{j\pi} > 0$. We call such an order a *fractional order*. In contrast, we call order J_j an *integral order* if there is a $\pi \in \overline{\Pi}_j$ with $X_{j\pi} = 1$. Observe that the order schedule that corresponds with the selected order plan of an integral order is by definition feasible. For a fractional order this is usually not the case. For each order J_j we define an order plan $a_{j\pi'}$ as follows:

Definition 4.1 *Consider any order J_j ($j = 1, \ldots, n$) and consider the order schedule determined by the realization of the Y_{bjt} values. We define the corresponding order plan $a_{j\pi'}$ as follows: $a_{bjt\pi'} = 1$ if $Y_{bjt} > 0$, else $a_{bjt\pi'} = 0$.*

The following Lemma can be used to verify if the order schedule of a fractional order is feasible.

Lemma 4.1 *The order schedule for order J_j is feasible if and only if $a_{j\pi'}$ is a feasible order plan.* □

For each order J_j with a feasible order plan $a_{j\pi'}$ we define an order plan $a_{j\pi^*}$ as follows:

Definition 4.2 *Consider any order J_j ($j = 1, \ldots, n$) with a feasible order plan $a_{j\pi'}$. We define the corresponding order plan $a_{j\pi^*}$ as follows: extend $a_{j\pi'}$ to $a_{j\pi^*}$ in such a way, that no elements $a_{bjt\pi^*}$ can be changed from 0 to 1 without making the order plan $a_{j\pi^*}$ infeasible.*

Using Lemma 4.1, the following Lemma gives a simple but powerful method to try to transform an LP solution into a completely integral or less fractional solution that has the same objective value.

Lemma 4.2 *Consider any fractional order J_j ($j = 1, \ldots, n$) with a feasible order plan $a_{j\pi'}$, and a corresponding order plan $a_{j\pi^*}$. The solution obtained by setting $X_{j\pi^*} = 1$ and setting $X_{j\pi} = 0$ for each $\pi \in \overline{\Pi}_j$ ($\pi \neq \pi^*$) is an alternative optimal solution to LP in which J_j has become an integral order.* □

Thus if $a_{j\pi'}$ is feasible for all orders J_j ($j = 1, \ldots, n$), then we can transform the fractional solution to a feasible integer solution to the ILP. Observe that there generally is no unambiguous way to transform order plan $a_{j\pi'}$ into $a_{j\pi^*}$. However, Lemma 4.2 holds for any order plan $a_{j\pi^*}$ that dominates $a_{j\pi'}$.

Unfortunately in general not all fractional orders can be transformed to integral orders. We illustrate this by a simple example. Suppose an order J_j consists of two jobs (i.e., $n_j = 2$) where job B_{1j} has to precede job B_{2j}. Suppose moreover that $r_j = 0$, $d_j = 2$, $T = 2$, and $\delta = 0$ (i.e., a job may start in the week in which its predecessor has completed). Recall from Section 3.2 that an order plan is characterized by a 0-1 vector $a_{j\pi} = \left(a_{1,j,0,\pi}, \ldots, a_{1,j,T,\pi}, \ldots, a_{n_j,j,0,\pi}, \ldots, a_{n_j,j,T,\pi}\right)$, where $a_{bjt\pi} = 1$ ($b = 1, \ldots, n_j$, $t = 0, \ldots, T$) when job B_{bj} is allowed to be performed in week t. Suppose we have two order plans in $\overline{\Pi}_j$: $a_{j1} = (1, 1, 1, 0, 0, 1)$ and $a_{j2} = (1, 0, 0, 1, 1, 1)$. Suppose moreover that both order plans are fractionally selected with a corresponding value $X_{j\pi} = 0.5$. In that case we have that constraints (4.2) stipulate for job B_1:

$$\begin{array}{ll} Y_{1jt} \leq \frac{1}{w_{1j}}, & \text{for } t = 0, \\ Y_{1jt} \leq \frac{0.5}{w_{1j}}, & \text{for } t \in \{1, 2\}, \end{array}$$

and for job B_2:

$$\begin{array}{ll} Y_{2jt} \leq \frac{1}{w_{2j}}, & \text{for } t = 2, \\ Y_{2jt} \leq \frac{0.5}{w_{2j}}, & \text{for } t \in \{0, 1\}. \end{array}$$

As a result, for both jobs and for all weeks Y_{bjt} may be nonzero, for example: $Y_{bjt} = \frac{1/3}{w_{bj}}$ ($b = 1, 2$, $t = 0, 1, 2$). The corresponding order plan $a_{j\pi'} = (1, 1, 1, 1, 1, 1)$ is infeasible. Since this order plan does not satisfy the precedence constraints we conclude that this order can not be transformed into an integral order. As a result, in general, a feasible solution to the LP is not necessarily feasible to the ILP. In the remainder of this chapter we propose several methods to construct a feasible solution to the ILP from a fractional LP solution. In Section 4.4 we propose several branch-and-bound strategies, and in Section 4.6 we propose several heuristics for finding a feasible solution to the ILP.

4.2.2 RLP initialization

To obtain a feasible RLP of the aforementioned general form in any node of the branching tree we must find at least one feasible order plan $a_{j\pi}$ for each order, in such a way, that a feasible solution to the RLP exists. One feasible order plan $a_{j\pi}$ per order may not result in a feasible RLP because of the finite capacity levels of the resources. We may use a primal heuristic for the ILP to find a feasible set of order plans. Such a heuristic is not guaranteed to succeed. Note however, that we do not need to find a feasible integer solution, just a feasible RLP. Since it is easy to find a basic solution to the RLP we can find a

feasible RLP by column generation as follows. The basic idea is equivalent to Phase I of the two-phase simplex method. The *two-phase simplex method* is a technique that is commonly used in situations where a basic feasible solution to an LP model is not readily available. Phase I of the method involves optimizing a relaxation of the RLP, which is obtained by adding artificial variables. The objective of this relaxation is to minimize the sum of all artificial variables. The solution to the relaxation is feasible to the RLP if all artificial variables are zero.

The Phase I relaxation of the RLP is obtained by relaxing the work assignment constraints (4.3). We do so by adding nonnegative artificial variables Z_{bj}:

$$\sum_{t=r_j}^{T} Y_{bjt} + Z_{bj} = 1 \ \ (\forall b, j)\,.$$

As a result, the Phase I relaxation becomes trivially feasible, namely set $X_{j\pi} = 1$ for some $\pi \in \overline{\Pi}$, $Z_{bj} = 1$ $(\forall b, j)$, and all other variables to 0. The Phase I relaxation is thus feasible for any set of order plans, provided that at least one arbitrary feasible order plan is available per order. These order plans can simply be constructed by, for example, setting the job start times as early as possible, and allowing the jobs to be performed for the minimal duration.

The Phase I objective function becomes:

$$\min \sum_{b,j} Z_{bj}.$$

We optimize the Phase I relaxation by column generation. We can use the same pricing algorithm as we use for the original RLP, using the duals of the Phase I relaxation as input. After termination of the column generation scheme in Phase I, we evaluate the objective value of the Phase I relaxation. If the objective function value is zero, the current solution is also feasible to the RLP. Since the order plans in the Phase I relaxation yield a feasible solution to the RLP, we remove the artificial variables and restore the objective function of the RLP, and continue by performing column generation on the RLP. However, if the objective function is larger than zero, we have proven that no feasible solution exists to the LP. In that case we may conclude that no feasible solution exists to the resource loading problem.

4.3 Pricing algorithm

From linear programming theory it is known that a solution to a minimization problem is optimal if the reduced cost of each variable is nonnegative (see also Section 2.2). Since an RLP in a column generation scheme contains for each order only a subset $\overline{\Pi}_j$ of all feasible order plans Π_j, we need to determine

whether order plans exist with negative reduced cost that are not in $\overline{\Pi}_j$. The reduced cost $c_{j\pi}$ of any order plan $a_{j\pi}$ ($\pi \in \Pi_j$, $j = 1, \ldots, n$) in LP is given by:

$$c_{j\pi} = \rho_{j\pi}\theta + \alpha_j - \sum_{b=1}^{n_j} \sum_{t=r_j}^{T} \beta_{bjt} a_{bjt\pi}, \tag{4.4}$$

where α_j is the (known) value of the dual variable corresponding to condition (4.1) and β_{bjt} are the (known) non-negative values of the dual variables corresponding to conditions (4.2).

To test optimality of the current RLP solution, we determine whether there exists an order plan $a_{j\pi}$ ($\pi \in \Pi_j \backslash \overline{\Pi}_j$) for some order J_j ($j = 1, \ldots, n$) with negative reduce costs, that is, with $c_{j\pi} < 0$. To this end, we solve the pricing problem of finding an order plan in $\Pi \backslash \overline{\Pi}$ with minimum reduced cost, which boils down to solving n independent subproblems, one for each order. The subproblem for order J_j ($j = 1, \ldots, n$) is to minimize:

$$\rho_{j\pi}\theta + \alpha_j - \sum_{b=1}^{n_j} \sum_{t=r_j}^{T} \beta_{bjt} a_{bjt\pi},$$

subject to the release date and due date of order J_j, the minimal duration of its jobs, the precedence constraints between its jobs, and the minimum time lag δ. The (binary) decision variables in the pricing problem are the $a_{bjt\pi}$'s that form the order plan $a_{j\pi}$ for order J_j. These variables also determine the values $\rho_{j\pi}$ ($\rho_{j\pi} \geq 0$), which denote the number of weeks that an order is allowed to be tardy. When the order plan $a_{j\pi}$ allows order J_j to be tardy by $\rho_{j\pi}$ weeks, a tardiness penalty $\rho_{j\pi}\theta$ is added to the pricing objective.

Note that an order plan may also be represented by the first and last weeks in which each job is allowed to be produced. For convenience we speak of the *start* and *completion times* of jobs in the order plan, although the actual start and completion times in the order schedule may respectively be higher and lower.

In the pricing algorithm we account for internal release dates (r_{bj}) and due dates (d_{bj}) of jobs. These may be calculated from the minimal duration of the jobs, the δ parameter, and the (external) release date and due date of the order, but these may also be imposed externally, e.g., to meet certain deadlines on components or parts of an order. Note that since we allow an order to be tardy, we must also allow jobs to be produced after their internal due date d_{bj}. We define the *job deadline* $\overline{d}_{bj}$ as the last week in which a job is allowed to be produced. This deadline is thus not externally given. Accordingly, we define the *order deadline* $\overline{d}_j$ as the deadline of job n_j. By default we set: $\overline{d}_j = \overline{d}_{bj} = T$ ($b = 1, \ldots, n_j$), however, we may also set $\overline{d}_{bj}$ to some number smaller than T to limit the number of feasible order plans, and thus simplify the pricing problem.

We use implicit column generation, i.e., we generate order plans as they are needed. An explicit column generation approach would require that all

order plans are generated in advance and kept in computer memory. Since the number of feasible order plans is exponentially large this is not a serious option. Our pricing algorithm for solving the subproblem for order J_j is based on dynamic programming and uses a forward recursion. In this chapter we consider the case of linear precedence relations. In Section 4.3.1 we discuss the pricing algorithm for $\delta = 1$. As discussed in Section 3.2, in the case $\delta = 0$ we impose an additional restriction that at most κ jobs of the same order are allowed to be performed in the same week. In Section 4.3.2 we discuss the pricing algorithm for this special case. In Chapter 6 we present a generalized pricing algorithm for the case with generalized precedence constraints.

4.3.1 Pricing algorithm for $\delta = 1$

The pricing problem can be translated into finding a longest path (when maximizing $-c_{j\pi}$) or shortest path (when minimizing $c_{j\pi}$) in a directed acyclic network with weights on the nodes. Consider for example an order J_j with 3 jobs, a release date $r_j = 1$, due date $d_j = 3$ and a planning horizon $T = 4$. The minimal duration w_{bj} of each job B_{bj} is 1 week. From these data, we calculate the release dates and due dates of the jobs as follows:

$$\begin{aligned} r_{1j} &= r_j, \text{ and } r_{bj} = r_{b-1j} + w_{b-1j} - 1 + \delta \ (b = 2, \ldots, n_j), \\ d_{n_j,j} &= d_j, \text{ and } d_{bj} = d_{b+1j} - (w_{b-1j} - 1) - \delta \ (b = 1, \ldots, n_j - 1). \end{aligned} \tag{4.5}$$

Hence, for jobs B_{1j}, B_{2j} and B_{3j} the values (r_{bj}, d_{bj}) are $(1,1)$, $(2,2)$ and $(3,3)$ respectively. The corresponding longest path problem can be described using the network depicted in Figure 4.2. We represent each job B_{bj} and each week t in which it is allowed to be performed by a node (b,t). Hence we have $n_j = 3$ sets of $T - r_j + 1 = 4$ nodes. A path through a node (b,t) indicates that job b is allowed to be performed in week t, which results in a contribution $-\beta_{bjt}$ to the reduced cost of the order plan. A path through a node (n_j, t), with $t > d_{n_j,j} = d_j$, contributes a tardiness penalty $\rho_{j\pi}\theta = (t - d_j)\theta$ to the reduced cost of the order plan, in addition to the contribution $-\beta_{n_jjt}$ for that job. A source and a target node are added to complete the directed network. A path from the source to the target node represents a complete order plan (see Figure 4.2).

The reduced cost of the order plan is the sum of α_j, the contributions $-\beta_{bjt}$ on the longest path, and, if applicable, a tardiness penalty $\rho_{j\pi}\theta$ for job n_j.

Since we only consider those order plans that can not be dominated by other order plans (see Section 3.2), in Figure 4.2 we have removed all redundant arcs. We could have, for example, drawn an arc from node $(1,1)$ to $(2,3)$. However, a path from $(1,1)$ to $(2,3)$ via $(2,2)$ or $(1,2)$ will yield as least as much as a path from $(1,1)$ directly to $(2,3)$. Hence, all arcs not involving the source and target node connect node (b,t) with node $(b+1, t+\delta)$ or node $(b, t+1)$.

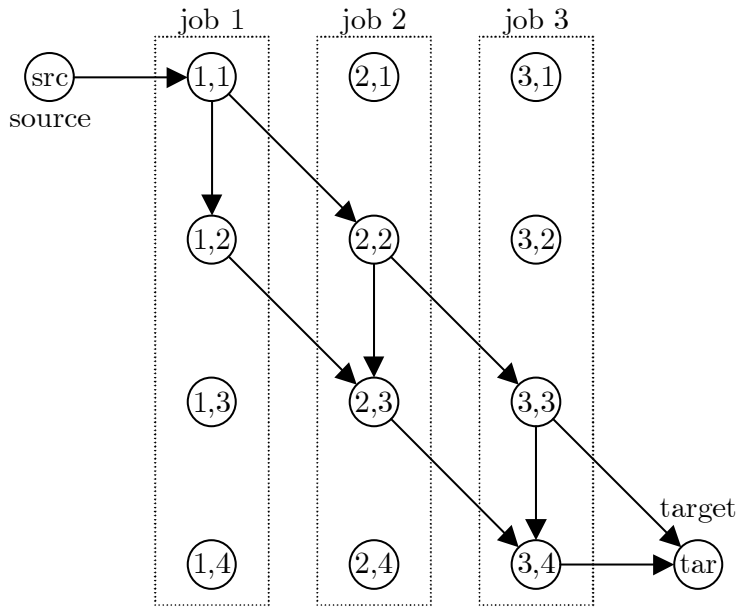

Figure 4.2: Example pricing problem ($\delta = 1$).

We solve the pricing problem by dynamic programming (DP). Each job B_{bj} represents a *stage* b. Since the information needed to represent an order plan consists of the allowed start and completion times of the jobs, the decision in each stage must regard either the start time or the completion time of the job. Since the pricing problem is symmetric with respect to this decision (a start time of a job implies the completion time of its predecessor, and vice versa), we say that a *decision* in a stage concerns the start time of the job in that stage. A *state* (j, b, t) in a stage b describes a completion time t of job B_{bj}.

Let $F_j(b,t)$ be the minimum reduced cost for all feasible *partial* order plans for the jobs $B_{1j}, \ldots, B_{bj}$ with t the completion time of job B_{bj}, i.e., t is the last week in which job B_{bj} is allowed to be performed ($b \leq n_j, r_{bj} + w_{bj} - 1 \leq t \leq \overline{d}_{bj}$). Note that t must be at least $r_{bj} + w_{bj} - 1$ to comply with the minimal duration constraint. Let $a_{j\pi}$ be an order plan in state (j, b, t) with value $F_j(b,t)$. Then, $a_{j\pi}$ must allow B_{bj} to be performed in weeks $s, \ldots, t$ for some $r_{bj} \leq s \leq t - w_{bj} + 1$ (i.e., s and t are the start and completion time of job B_{bj}). The start time s can be at most $t - w_{bj} + 1$ to comply with the minimal duration constraint. Accordingly, the previous state must be $(j, b-1, s-\delta)$ for some s with $r_{bj} \leq s \leq t - w_{bj} + 1$. Allowing B_{bj} to be performed in weeks $s, \ldots, t$ contributes $-\sum_{u=s}^{t} \beta_{bju}$ to the reduced cost of the partial order plan of the previous state ($F_j(b-1, s-\delta)$).

We now give the dynamic programming recursion to solve the j-th subproblem. The initialization is:

$$F_j(b,t) = \begin{cases} \alpha_j, & \text{if } b = 0, \\ \infty, & \text{otherwise.} \end{cases}$$

The recursion for $b = 1, \ldots, n_j$, $t = r_{bj} + w_{bj} - 1, \ldots, \overline{d}_{bj}$ is then:

$$F_j(b,t) = \min_{r_{bj} \le s \le t - w_{bj} + 1} \left\{ F_j\left(b-1, s-\delta\right) + \Delta_{bjt} - \sum_{u=s}^{t} \beta_{bju} \right\}. \tag{4.6}$$

For all other t:

$$F_j(b,t) = \infty.$$

Where Δ_{bjt} in (4.6) is defined as follows:

$$\Delta_{bjt} = \begin{cases} (t - d_{bj})\theta, & \text{if } b = n_j \text{ and } t > d_{bj} \\ 0, & \text{otherwise.} \end{cases} \tag{4.7}$$

Parameter Δ_{bjt} thus adds a tardiness penalty to $F_j(b,t)$. Note that without loss of generality this pricing algorithm may also be applied in cases where $\delta > 1$.

Lemma 4.3 *The optimal solution to the j-th pricing subproblem is found as:*

$$F_j^* = \min_{d_j \le t \le \overline{d}_j} F_j(n_j, t), \ (j = 1, \ldots, n).$$

□

Lemma 4.4 *The optimal solution to the pricing problem is found as:*

$$F^* = \min_{1 \le j \le n} F_j^*.$$

□

Accordingly, if $F^* \ge 0$, then the current RLP solution is optimal. If $F^* < 0$, then the current RLP solution may not be optimal, and we need to introduce new columns (order plans) to the problem. Candidates are associated with those orders J_j for which $F_j^* < 0$, and they can be found by backtracking. Note that the pricing algorithm for the j-th subproblem uses $O(n_j T)$ space and can be solved in $O(n_j T^2)$ time. Hence the pricing problem uses $O(\max_j(n_j)Tn)$ space and can be solved in $O(\max_j(n_j)T^2 n)$ time. Although the pricing algorithm usually demands little computation time, some speed up can be obtained by identifying jobs that can not contribute to the reduced cost of the order plan, that is, jobs B_{bj} for which $\beta_{bjt} = 0$ $(\forall t)$. For such jobs, given their state (the

completion time (b,t)), the corresponding decision (the start time s) is always chosen as $s = t - w_{bj} + 1$, where w_{bj} is the minimal duration of the job.

Solving the n pricing problems may give us more than one column with negative reduced cost. Therefore, we face the issue to decide on the number of columns to add to the linear program after having solved the pricing problem. In general, the more columns we add per iteration, the fewer linear programs we need to solve, but of course the bigger the linear programs become. As long as the size of the RLP remains manageable, we add all columns (at most one per order) with negative reduced costs. However, when the computation time that is required to solve the RLP increases too much, column disposal may be required. We discuss in Section 4.4 that the growth of the 'column pool' depends largely on the branching strategy.

4.3.2 Pricing algorithm for $\delta = 0$

For the case $\delta = 0$ we also use dynamic programming to solve the pricing problem. The main difference in the pricing algorithm is caused by the additional restriction that we allow at most κ jobs of the same order to be performed in the same week. Note that $\delta = 0, \kappa = 1$ is equivalent to $\delta = 1$. If we would relax this constraint ($\kappa = 0$), we could use the pricing algorithm of the previous section. Also, if the number of jobs n_j of an order J_j is equal to or smaller than κ, we may use the pricing algorithm of the previous section. Hence, in the remainder of this section we assume that $\delta = 0$, and $n_j > \kappa$.

As in the previous section, each job B_{bj} represents a stage b. Moreover, the decision in a stage b regards the start time of job B_{bj}. The additional restriction that at most κ jobs of the same order are allowed to be performed in the same week implies that we may only allow a job B_{bj} to be started in a week t, when at most $\kappa - 1$ predecessors are allowed to be produced in that week t. Thus to be able to make a decision regarding a start time of a job B_{bj}, the state (j,b,t) must be extended with an additional parameter $\sigma < \kappa$, that indicates the number of predecessors that are allowed to be produced in week t. Note that since j is fixed in a pricing problem, the state (j,b,t,σ) is in fact 3-dimensional.

Let $a_{j\pi}$ be an order plan in state (j,b,t,σ) with value $F_j(b,t,\sigma)$, and let $b > 1$. To determine from which possible previous states $a_{j\pi}$ can be created, we must distinguish two cases, i.e., $\sigma = 0$ and $\sigma > 0$. When $\sigma = 0$, the predecessor $(B_{b-1,j})$ is not allowed to be performed in week t, so the completion time of the predecessor must be smaller than t. As a result, π must have been created by allowing B_{bj} to be performed in weeks $s, \ldots, t$ for some $r_{bj} \leq s \leq t - w_{bj} + 1$, and the completion time of the predecessor must be $\min\{t-1, s\}$. Accordingly, the previous state must be $(j, b-1, \min\{t-1,s\}, \sigma')$ for some s with $r_{bj} \leq s \leq t - w_{bj} + 1$, and some $\sigma' < \kappa$. When $\sigma > 0$, the completion time of the predecessor of job B_{bj} is t. Accordingly, the previous state must be

$(j, b-1, t, \sigma-1)$.

We now give the dynamic programming recursion to solve the j-th subproblem. The initialization is:

$$F_j(b,t,\sigma) = \begin{cases} \alpha_j, \text{ if } b = 0, \\ \infty, \text{ otherwise.} \end{cases}$$

The recursion for $\sigma = 0$, $b = 1, \ldots, n_j$, $t = r_{bj} + w_{bj} - 1, \ldots, \overline{d}_{bj}$ is then:

$$F_j(b,t,0) = \min_{r_{bj} \le s \le t - w_{bj} + 1} \left\{ \min_{\sigma' < \kappa} \left\{ F_j(b-1, \min\{t-1, s\}, \sigma') \right\} + \Delta_{bjt} - \sum_{u=s}^{t} \beta_{bju} \right\}$$

where Δ_{bjt} is defined as in (4.7). The recursion for $0 < \sigma < \kappa$, $b = 1, \ldots, n_j$, $t = r_{bj} + w_{bj} - 1, \ldots, \overline{d}_{bj}$ is then:

$$F_j(b,t,\sigma) = \min_{r_{bj} \le s \le t - w_{bj} + 1} \left\{ F_j(b-1, t, \sigma-1) + \Delta_{bjt} - \sum_{u=s}^{t} \beta_{bju} \right\}.$$

For all other σ, b, t:

$$F_j(b,t,\sigma) = \infty.$$

Lemma 4.5 *When $\delta = 0$, and $n_j > \kappa$, the optimal solution to the j-th pricing subproblem is found as:*

$$F_j^* = \min_{d_j \le t \le \overline{d}_j, 0 \le \sigma \le \kappa} F_j(n_j, t, \sigma), \ (j = 1, \ldots, n).$$

□

Analogous to the pricing algorithm in the previous section, when $F_j^* < 0$, a candidate order plan can be found by backtracking. The pricing algorithm for the j-th subproblem uses $O(n_j T \kappa)$ space and can be solved in $O(n_j T^2 \kappa^2)$ time. Hence the pricing problem uses $O(\max_j(n_j) T \kappa n)$ space and can be solved in $O(\max_j(n_j) T^2 \kappa^2 n)$ time.

4.4 Branching strategies

After termination of the column generation scheme on an RLP we evaluate the resulting optimal LP solution. If the optimal LP solution is integral with respect to the $X_{j\pi}$ variables, the solution also forms an optimal solution to the resource loading problem, and we are done. If the solution is fractional, we apply the procedure of Lemma 4.2 to try to transform the fractional solution

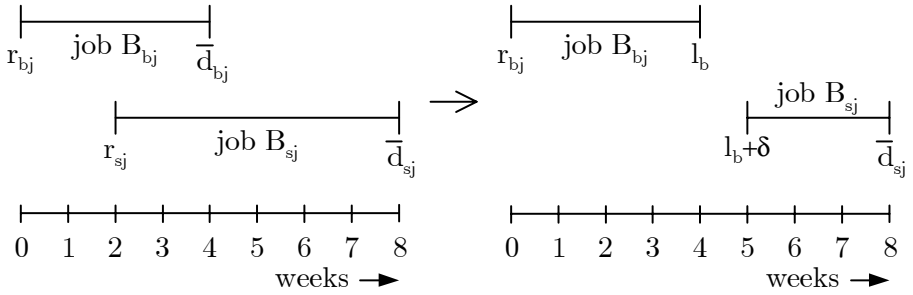

Figure 4.3: Repair of a violated precedence relation ($\delta = 1$)

into an integral solution without changing the objective function. If this is not possible, i.e., a part of the solution remains fractional, we need a branch-and-bound algorithm to find an optimal integral solution. In the next sections we discuss the exact (Section 4.4.1) and heuristic (Section 4.4.2) branching strategies to find a feasible integral solution from an optimal fractional LP solution.

4.4.1 Branching strategy

It is well known that a 'direct' partitioning of the solution space by fixing or bounding binary variables is generally not a good strategy, because it tends to yield an unbalanced branch-and-bound tree (see, e.g., Barnhart et al., 1998; Vanderbeck, 2000). Hence, instead of branching on the fractional variables $X_{j\pi}$ of an order J_j, we use a branching strategy (referred to as BS) that aims at fixing violated precedence relations between jobs of fractional orders. When, in an RLP solution in any node of the branching tree, an order plan $a_{j\pi'}$ (as defined in Definition 4.1) is not feasible, there is at least one pair of jobs, say jobs B_{bj} and B_{sj}, for which a precedence relation is violated. In BS we alter the allowed intervals for jobs B_{bj} and B_{sj} such that they no longer overlap. We analyze each possible alteration in the child nodes. Recall that job B_{bj} is allowed to be performed in interval $[r_{bj}, \overline{d}_{bj}]$, and job B_{sj} is allowed to be performed in the (overlapping) interval $[r_{sj}, \overline{d}_{sj}]$. To repair the violated precedence relation between these jobs, we must alter job deadlines, which were introduced to limit the number of feasible order plans and thus reduce the pricing problem size, and job release and due dates as follows. Each child node corresponds to a new deadline l_b of job B_{bj} ($l_b \in \left[\max\left\{r_{bj} + w_{bj} - 1, r_{sj}\right\}, \min\left\{\overline{d}_{bj}, \overline{d}_{sj} - w_{sj} + 1 - \delta\right\}\right]$). This is illustrated in Figure 4.3. The values $\left(r_{bj}, d_{bj}, \overline{d}_{bj}\right)$ for job B_{bj} thus become $\left(r_{bj}, \min\left\{d_{bj}, l_b\right\}, l_b\right)$, and the values $\left(r_{sj}, d_{sj}, \overline{d}_{sj}\right)$ for job B_{sj} become $\left(\max\left\{r_{bj}, l_b + \delta\right\}, d_{sj}, \overline{d}_{sj}\right)$. Subsequently, we eliminate all order plans $a_{j\pi}$ that do not comply with the updated job release dates, due dates and deadlines. As

a consequence, the (modified) RLP or even the (modified) LP may no longer be feasible. To determine whether the modified LP is feasible we apply column generation to the Phase I LP relaxation of the two-phase simplex method, as described in Section 4.2.2, and account for the modified job release dates, due dates and deadlines in the pricing algorithm. When the modified LP is infeasible, we prune the node. Otherwise, we continue in this node by performing column generation on the RLP. When we prune a node, we return to the parent node in the branching tree.

We illustrate the branching strategy BS with an example. Consider an RLP solution in which $a_{j\pi'}$ is infeasible for order J_j, due to a precedence relation violation of jobs B_{1j} and B_{2j}. Suppose the values $\left(r_{bj}, d_{bj}, \overline{d}_{bj}\right)$ for jobs B_{1j} and B_{2j} are $(0,2,3)$ and $(1,3,4)$ respectively, the minimal duration w_{bj} of each job is 1 week, and $\delta = 1$. Branching strategy BS imposes in each child node a deadline l_1 for job B_{1j}, with:

$$\begin{aligned} l_1 &\in \left[\max\left\{r_{1j} + w_{1j} - 1, r_{2j}\right\}, \min\left\{\overline{d}_{1j}, \overline{d}_{2j} - w_{2j} + 1 - \delta\right\}\right] \\ &= \left[\max\left\{0,1\right\}, \min\left\{3,3\right\}\right] \\ &= \left[1,3\right]. \end{aligned}$$

Hence the branching strategy results in 3 child nodes, in which the modified values $\left(r_{bj}, d_{bj}, \overline{d}_{bj}\right)$ for jobs B_{1j} and B_{2j} are:

$$\begin{aligned} &(0,1,1) \text{ and } (2,3,4) \text{ in child node 1,} \\ &(0,2,2) \text{ and } (3,3,4) \text{ in child node 2,} \\ &(0,2,3) \text{ and } (4,4,4) \text{ in child node 3.} \end{aligned}$$

Note that the modified job release dates, due dates and deadlines exclude an overlap of jobs B_{1j} and B_{2j} in any order schedule.

For the BS method we apply the depth-first node selection strategy. When we select a violated precedence relation to be repaired in a node, we try to select a fractional order for which the repair of a precedence relation violation is most restrictive. We expect that this strategy reduces the depth of the branching tree. There are probably many intuitive strategies that comply with this strategy. We select in each node the fractional order J_j that maximizes $\sum_{\pi \in \overline{\Pi}_j} (X_{j\pi}(1 - X_{j\pi}))$. When there are more than one violated precedence relation for this order, we repair the violated precedence relation for which the concerned pair of jobs has the largest overlap (in weeks).

We may try to find an initial incumbent solution before we start the BS method by applying a heuristic. This usually enables to fathom nodes at an early stage in the branching scheme, at the cost of little additional computation time. We discuss the application of heuristics in Section 4.6. In Section 4.5 we discuss the application of Lagrangian relaxation for lower bound determination.

4.4.2 Branch-and-price based approximation algorithms

For large instances, branch-and-bound methods typically require a long computation time to find an optimal solution, or to prove that the incumbent solution is optimal. We found that this is also the case for the resource loading problem. We therefore tested some branch-and-price based heuristics to find a good solution in little time. We use the integrality gap as an indication of the quality of the solutions of these algorithms.

The first and most straightforward branch-and-price based heuristic we tested is the *truncated BS* (TBS), that truncates the BS algorithm of Section 4.4.1 after a certain amount of time. The resulting solution of the TBS method is the incumbent solution at the moment of truncation. The TBS method enables to test the BS method on large problems.

Another heuristic based on the BS method is the *ϵ-approximation strategy* (EAS). In this heuristic we fathom all nodes in which the lower bound approaches the incumbent (upper bound) within $\epsilon\%$ (i.e., $LB \cdot (1+\epsilon) \geq UB$). On termination of EAS, its incumbent solution differs at most $\epsilon\%$ from the optimal ILP solution. Hence, this method allows us to find a compromise between the computation time required, and the quality of the solution obtained.

We also tested six variants of an incomplete branching strategy that is not based on the aforementioned BS method. This branching strategy is based on selecting one order plan for an order in each layer of the branching tree. Hence each layer corresponds to an order J_j for which we set one $X_{j\pi}$ variable to 1. After selecting an order plan for an order in a node of the branching tree, we apply column generation on the remaining orders (i.e., the orders for which no order plans have been fixed) to prevent that fixing variables leads to infeasible RLPs. We use a depth-first node selection strategy to find a feasible solution as fast as possible.

In the first two variants of this branching strategy we branch on the fractional order that has the highest number of (fractionally) selected order plans in the current RLP solution. In $HBS1$ we branch on all order plans that are available for the selected order in the current RLP. In the second variant ($HBS2$) we branch only on those order plans that are (fractionally) selected in the current RLP solution. Observe that the part of the solution space that is analyzed by these branching strategies strongly depends on the number of order plans that are available for each order at the moment the order is branched on. When an order that is branched on has few order plans available in the current RLP, a large part of the solution space may be discarded. For this purpose we implemented variant $HBS1+$ and $HBS2+$, which are are the same as $HBS1$ and $HBS2$ respectively, except that before branching we add for each order up to five order plans to the column pool, so that sufficient order plans are available to branch on. These order plans are found by five stand-alone heuristics that were proposed by Borra (see Borra, 2000, and Section 4.6.1). Variants

$HBS3$ and $HBS4$ are the same as $HBS1+$ and $HBS2+$ respectively, except that the next order to branch on in each layer is the order that has the most order plans in the current RLP.

4.5 Lower bound determination by Lagrangian relaxation

Lagrangian relaxation can complement column generation, in that it can provide a lower bound to the original problem in every iteration of the column generation procedure with little computational effort (cf. Section 2.4). We note that the example ILP model in Section 2.4 has the same basic form as the resource loading model, where the disjoint sets of variables $J(i) \subset J$ in the example correspond to the sets $\Pi_j \subset \Pi$ of variables $X_{j\pi}$. Hence we apply the same procedure to the resource loading model, and we obtain the Lagrangian relaxation of the ILP by relaxing all constraints but constraints $\sum_{\pi \in \Pi_j} X_{j\pi} = 1$ $(\forall j)$ and variable restrictions $X_{j\pi} \in \{0,1\}$ $(\forall j, \pi \in \Pi_j \subset \Pi)$. The resulting Lagrangian relaxation $L_{ILP}(\lambda)$ possesses the integrality property. Hence, analogous to the example in Section 2.4, we can obtain a Lagrangian lower bound to the linear programming relaxation LP each time an RLP has been solved to optimality, by using the dual solution of the RLP as Lagrangian multipliers, and by performing a pricing algorithm on all orders. The lower bound can be determined as follows:

$$z^*_{RLP} + \sum_j c_j \Theta_j,$$

where z^*_{RLP} is the optimal objective value of the RLP, c_j is the objective value of the j-th pricing algorithm (i.e., the lowest reduced cost factor of all order plans of order J_j), and Θ_j is a binary variable that is set to 1 when c_j is nonnegative, and set to 0, otherwise.

We apply column generation in each node of the branching tree. Hence, in each column generation iteration in each node we may determine a lower bound to the LP relaxation of the resource loading model *for that node* and for all nodes below that node (note that the branching changes the LP relaxation in every node). Hence, when branching downwards in the branching tree, we must store the best lower bound found so far. We compare the best lower bound to an upper bound (e.g., an incumbent solution, or a heuristic solution of the resource loading problem) after every column generation iteration, and terminate column generation and fathom the node when the best lower bound exceeds the best upper bound. This also alleviates the tailing-off effect of column generation.

We also use the Lagrangian lower bound to fix variables. When the reduced cost of a variable is larger than $UB - LB$ (i.e., the difference between the upper and lower bound), we fix that variable to 0, since we know from linear

programming theory that that variable must be 0 in any solution with a value less than UB. Analogously, we fix each variable with a reduced cost smaller than $LB - UB$ to 1.

4.6 Application of heuristics

We propose several heuristics to be used as 'stand-alone' heuristics that solve the resource loading problem without using the linear programming model of the resource loading problem, as well as heuristics that we can use in various stages of the branch-and-price algorithm. The latter category of heuristics aims either at finding an integral solution from a fractional solution (rounding heuristics), or at improving an existing integral solution (improvement heuristics).

4.6.1 Stand-alone heuristics

Stand-alone heuristics can be used to determine an a priori feasible solution to the resource loading problem. Such heuristics can also be used in the branch-and-price algorithm, e.g., to find an initial feasible set of orders for the RLP in the root or in any node. The heuristic solutions serve as an upper bound to the solution of the original problem, which, in conjunction with a Lagrangian lower bound, may reduce the number of nodes to explore in the branch-and-price algorithm. The heuristic solutions can also be used to evaluate the quality of solutions found by heuristic branching strategies, such as the truncated BS method.

The stand-alone heuristics proposed by Borra (2000) are all *priority rule based heuristics*, that load the jobs in a sequence determined by the priority rule, and hereby construct an order schedule. As an example we mention the *earliest due date* (EDD) priority rule, that sorts orders or jobs according to their due dates in an increasing order, and plans the orders or jobs according to this sequence. Since the objective here is to minimize the costs of extra capacity, we first try to process each job in regular operator time. If this is not possible without exceeding its due date, we use whichever type of capacity is cheaper to complete the job in time. If we still can not meet the due date, then we subcontract the entire job.

The solution of a stand-alone heuristic determines exactly one feasible order schedule for each order. When a heuristic solution is used to initialize an RLP in any node of the branching tree, order plans must be constructed from these order schedules. We construct the order plans $a_{j\pi^*}$ for each order J_j from the order schedule for J_j using the method described in Definition 4.2. By optimizing the initial RLP that only has the order plans $a_{j\pi^*}$, we find a feasible resource loading solution that is at least as good as the solution found by the

stand-alone heuristic that was used to generate the order plans. Hence, in this way the RLP model can be used in conjunction with a stand-alone heuristic to improve its solution without much effort.

4.6.2 Rounding heuristics

We propose four *rounding heuristics* that are based on constructing an integral solution from a fractional RLP solution by rounding variables $X_{j\pi}$ ($j = 1,\ldots,n$, $\pi \in \overline{\Pi}_j$). These heuristics may help to speed up the branch-and-price methods, by trying to find better incumbent solutions (with little computational effort) in any node of the branching tree. All rounding heuristics attempt to converge to an integral solution in a number of iterations, in which we round a number of variables (i.e., fix variables $X_{j\pi}$ either to 0 or to 1), after which we re-optimize the (modified) RLP. In each iteration we apply the procedure of Lemma 4.2 to try to eliminate fractional orders. Since we do not add order plans in the iterations, rounding variables may lead to an infeasible RLP. The rounding heuristics thus do not always lead to a feasible integral solution.

In the rounding heuristic that we refer to as $RH1$, we round in each iteration all variables $X_{j\pi}$ ($j = 1,\ldots,n$, $\pi \in \overline{\Pi}_j$) with a value smaller than some positive threshold ϵ to 0, and round each variable $X_{j\pi}$ with a value larger than $1-\epsilon$ to 1. When no variables can be rounded by this strategy, we round the variable $X_{j\pi}$ with the highest value. The larger we set ϵ, the faster the heuristic becomes, however, too many variables may be rounded at once. This usually leads to an infeasible RLP.

In heuristic $RH2$ we take into account that order plans that are not (fractionally) selected may for a large part comply with the order schedule (formed by variables Y_{bjt}) in the current solution. In other words, these non-selected order plans may for a large part be the same as the order plan $a_{j\pi'}$ (defined in Definition 4.1). The $X_{j\pi}$ variables thus do not provide much information as to what order plans are appropriate. We calculate for each order plan of each fractional order a value $Q_{j\pi}$ (that takes a value between 0 and 1) that indicates to what extent the order plan complies with the order schedule in the current RLP solution. The value $Q_{j\pi}$ is calculated as follows:

$$Q_{j\pi} = \frac{\sum_{t=r_j}^{\overline{d}_j} \sum_{b=1}^{n_j} a_{bjt\pi} Y_{bjt}}{n_j}. \tag{4.8}$$

We thus sum up all the Y_{bjt}'s that are matched by a corresponding 1 in the order plan, and divide the sum by the number of jobs. Hence, when the order plan matches the order schedule, the resulting Q-value is 1. Note that although an order plan may not be (fractionally) selected, it may still have a high Q-

value. We use the value $Q_{j\pi}$ instead of $X_{j\pi}$ as the rounding variable, and apply the same rounding strategy as in $RH1$.

Rounding heuristic $RH3$ is basically the same as $RH2$, but in addition it solves a pricing algorithm in each iteration of the rounding scheme, for all fractional orders. Hereby we try to prevent that the rounding procedure yields an infeasible solution after some iterations.

In heuristic $RH4$ we round in each iteration a fixed number of variables, say k, until no variables need to be rounded. For this purpose we calculate for each variable $X_{j\pi}$ that has not yet been fixed the value $Q'_{j\pi} = \max\{Q_{j\pi}, 1 - Q_{j\pi}\}$, with $Q_{j\pi}$ as in (4.8). We then sort the variables on their $Q'_{j\pi}$ value in decreasing order, and we round the first k variables to:

$$\begin{aligned} 1, \text{ when } Q'_{j\pi} &= Q_{j\pi}, \\ 0, \text{ when } Q'_{j\pi} &= 1 - Q_{j\pi}. \end{aligned}$$

When the first k variables $X_{j\pi}$ are integral, we round the subsequent k variables as well, until the RLP basis is modified and it needs to be re-optimized. We choose k to be a small value. The higher the number k the faster the heuristic, however, the higher the risk of an infeasibility.

4.6.3 Improvement heuristic

In our computational experiments we also use the *improvement heuristic* proposed by Gademann and Schutten (2001). We refer to this heuristic as IH. IH starts from a feasible solution and tries to improve the solution by iteratively changing the start and completion times of the jobs in the order plans of the solution. Suppose we want to change the start time s_{bj} or the completion time t_{bj} of job B_{bj} of order J_j in the time window $[s_{bj}, t_{bj}]$ of that job. There are four possible cases that result from decreasing or increasing s_{bj} or t_{bj} by one week:

- a *decrease in the start time* s_{bj} of job B_{bj} is allowed when the completion time of the predecessor B_{uj} may be decreased by one. This condition holds when $(t_{uj} - 1) - s_{uj} + 1 \geq w_{uj}$, i.e., when the modified time window for job B_{uj} is at least as large as its minimal duration w_{uj}.
- an *increase in the start time* s_{bj} of job B_{bj} is allowed when the modified time window satisfies $t_{bj} - (s_{bj} + 1) + 1 \geq w_{bj}$, i.e., when the modified time window for job B_{bj} is at least as large as its minimal duration. This change only makes sense if the completion time of the predecessor B_{uj} can be increased, i.e., $t_{uj} + 1 \leq \overline{d}_{uj}$.
- a *decrease in the completion time* t_{bj} of job B_{bj} is allowed when the modified time window satisfies $(t_{bj} - 1) - s_{bj} + 1 \geq w_{bj}$, i.e., when the

modified time window for job B_{bj} is at least as large as its minimal duration. This change only makes sense if the start time of the successor B_{vj} of job B_{bj} can be decreased, i.e., $s_{vj} - 1 \geq r_{vj}$.

- an *increase in the completion time* t_{bj} of job B_{bj} is allowed when the start time of the successor B_{vj} of job B_{bj} can be increased by one. This condition holds when $t_{vj} - (s_{vj} + 1) + 1 \geq w_{vj}$.

Each time we change a start or completion time of a job in an order plan, we determine the corresponding order schedules by solving an RLP that only contains the order plans of the initial solution. We use the dual solution of the RLP to evaluate the expected change in the objective function value for all possible changes in the order plans. Each of the aforementioned changes in an order plan results in two modified coefficients in the coefficient matrix of the model. More specific, in Constraints (4.2) one coefficient $\sum_{\pi \in \overline{\Pi}_j} a_{bjt\pi}$ of a variable $X_{j\pi}$ is set from 1 to 0 (for a job B_{bj}), and one coefficient of the same variable $X_{j\pi}$ is set from 0 to 1 (for a predecessor or successor B_{uj}). Thus with β_{bjt} the given non-negative values of the dual variables corresponding to conditions (4.2), the expected change in the objective value is $-\beta_{bjt} + \beta_{ujt}$.

In heuristic IH we evaluate all possible changes in the time windows of all jobs, and sort these by increasing value of the expected change in the objective value.

We accept the first change according to this sorting that actually leads to an improvement. This is checked by reoptimizing the corresponding RLP. We repeat this procedure until no more improvement is found. Gademann and Schutten propose several other strategies for accepting changes, each of which leads to different heuristics. We do not use these other strategies.

We presented the heuristic for the case of linear precedence relations, although the heuristic was originally proposed by Gademann and Schutten for the case of generalized precedence relations. The only difference in the case of generalized precedence relations is that when we change the time window of a job in the aforementioned procedure, it may have more than one successor or predecessor.

Gademann and Schutten show that the search space is not connected, i.e., IH may get stuck in a local minimum. For example, when the time window of job B_{bj} in the initial order plan is $[s_{bj}, t_{bj}]$, the optimal time window is $[s_{bj} - 2, t_{bj}]$, and the expected change in the objective for changing s_{bj} to $s_{bj} - 1$ is positive, we may never find an optimal solution. The outcome of the heuristic thus depends on the initial feasible solution it starts from. It thus may pay off to perform this heuristic at various stages of the branch-and-price algorithm.

Chapter 5

Resource loading computational results

All of life is an experiment.
The more experiments you make, the better.

- Ralph Waldo Emerson (1803-1882)

5.1 Test approach

In this chapter we present the computational results of the algorithms for resource loading presented in Chapter 4. All algorithms have been coded and tested in the Borland Delphi 5.0 C/S programming language on a Pentium $III - 600$ MHz personal computer, running Windows NT 4.0. The application interfaces with the ILOG CPLEX 7.0 callable library, which we use for optimizing linear programming models and retrieving the optimization results.

By using different branching strategies and combining branch-and-price algorithms with heuristics, we obtain various algorithms for resource loading. To test these, we implemented an instance generator that allows us to produce classes of instances with various parameter settings. Table 5.1 lists the parameters that characterize the test instances. Each parameter influences the complexity of the test instances. We aim at generating test instances that are not too easy to solve with any parameter setting. Some parameters mainly determine the problem size (e.g., the length of the planning horizon), other parameters only influence the solution space (e.g., the release and due dates of the orders determine how much slack the jobs have). Our test approach is to first perform preliminary experiments in which we test all algorithms on a small number of instances. For these instances we vary the length of the plan-

n	the number of orders.
m	the number of machine groups.
n_j	the number of jobs of order J_j.
$T+1$	the length of the planning horizon (in weeks).
r_j	release date of order J_j.
d_j	due date of order J_j.
μ_{bj}	the machine group on which job B_{bj} must be processed.
p_{bj}	the processing time of job B_{bj}.
w_{bj}	minimal duration of job B_{bj} (in weeks).
δ	minimum time lag (0 or 1 week) between adjacent jobs, to impose a one-job-per-week policy.
$\overline{mc}_{it}$	total regular capacity of machine group M_i in week t.
mc_{it}	capacity of M_i in week t in regular operator time.
c_t	available regular operator capacity in week t (in hours).
o_t	available overtime capacity in week t (in hours).
h_t	available hiring capacity in week t (in hours).
s_t	available subcontracting capacity in week t (in hours).
$\overline{o}_t$	cost of overtime per hour.
$\overline{h}_t$	cost of hiring one extra operator per hour.
$\overline{s}_t$	cost of outsourcing one hour of work.
κ	maximum number of jobs of the same order that are allowed to be produced in the same week.

Table 5.1: Test case parameters.

ning horizon, the number of orders to be loaded and the number of available machine groups. Based on the preliminary test results we aim to select the best version of the branch-and-price based algorithm for further evaluation. We also choose an interesting class of instances, which we use for more extensive testing of various other parameters.

We restrict ourselves to time driven resource loading problems. As a result, in the test instances, the orders are not allowed to be tardy. The testing of the algorithms on resource driven resource loading problems is subject of further research.

The outline of this chapter is as follows. In Section 5.2 we discuss the test instance generation procedure. In Section 5.3 we give an overview of all algorithms that we tested. In Section 5.4 we determine the best version of the algorithm by performing preliminary computational experiments with various heuristics and branch-and-price methods. Finally, in Section 5.5 we present the extensive computational results of this algorithm, by performing sensitivity analyses on various parameters.

5.2 Test instance generation

In this section we discuss the test instance generation procedure. We categorize the test instances in classes that correspond to different parameter settings. We particularly vary the length of the planning horizon $T+1$, the number of orders n and the number of machine groups m. We consider the following values for T and m:

$$\begin{aligned} T &\in \{5, 10, 15, 20, 25, 30\}, \\ m &\in \{3, 5, 7, 10\}. \end{aligned}$$

We generate the order data by simulating an order arrival process. The first order arrives in week 1, and the subsequent orders arrive with a Poisson arrival distribution at a mean rate of λ. This implies that the interarrival times have an exponential distribution with an average interarrival time of $1/\lambda$ weeks. Furthermore, for the number of orders n we have that $E(n) = \lambda(T+1)$. The release date r_j of an order J_j is found by rounding the arrival time of the order to the nearest whole number. We consider the following values for parameter λ:

$$\lambda \in \{0.5, 1, 2\}.$$

The remaining order data for an order J_j is then generated as follows:

- $n_j \in \{1, 2, \ldots, 10\}$.

The number of jobs of order J_j (n_j) is uniformly drawn from $\{1, 2, \ldots, 10\}$. Observe that when we solve an instance for $\delta = 1$, orders can have at most $T + 1$ jobs.

- $p_{bj} \in \{10, 11, \ldots, 60\}$.

We give the jobs a processing time of a similar magnitude as the size of a time period (i.e., a week). Accordingly, we draw p_{bj} uniformly from $\{10, 11, \ldots, 60\}$ hours.

- $\mu_{bj} \in \{1, 2, \ldots, m\}$.

A machine group number μ_{bj} is uniformly drawn for each job B_{bj} from $\{1, 2, \ldots, m\}$. Subsequent jobs must have different machine group numbers.

- $w_{bj} = 1$.

All jobs have a minimal duration w_{bj} of 1 week, so that, when solving the test instance for $\delta = 0$, varying parameter κ (i.e., the maximum number of jobs of the same order that are allowed to be produced in the same week) will generally have significant impact on the solution space.

- $d_j = r_j + n_j + k - 1$, $k \in \{\lceil 0.5 \cdot n_j \rceil, \lceil 0.5 \cdot n_j + 1 \rceil\}$.

The order due date d_j must be large enough to allow the order to complete between the release and due date. For this purpose we first calculate the order minimal duration, and then add a number of slack weeks to determine the order due date. Although we solve the test instances for both $\delta = 0$ and $\delta = 1$, for the determination of the order minimal duration we assume that $\delta = 1$. As a result, when we solve the test instance for $\delta = 0$ (and $\kappa > 1$) instead of $\delta = 1$, the order will have more slack weeks. Hence, for $\delta = 1$ and $w_{bj} = 1$ $(\forall b, j)$ the order minimal duration is $\sum_{b=1}^{n_j} w_{bj} - (n_j - 1)(1 - \delta) = n_j$ weeks. The order due date thus becomes $d_j = r_j + n_j + k - 1$, where k is the number of slack weeks that we add. Without adding slack weeks (i.e., $k = 0$), the planning of the order would be trivial. To prevent this, we give each order at least 1 week slack. Also, we want orders with a small number of jobs to generally have fewer slack weeks than orders with more jobs. For this purpose, we relate the number of slack weeks of an order J_j to its number of jobs n_j. Accordingly, we uniformly draw a value from $\{\lceil 0.5 \cdot n_j \rceil, \lceil 0.5 \cdot n_j + 1 \rceil\}$, and round the value to obtain the number of slack weeks k.

- $\delta = 0 \Rightarrow \kappa \in \{2, 3, 10\}$.

The values we consider for the maximum number of jobs of the same order that are allowed to be produced in the same week (κ) are $\{2, 3, 10\}$. This only applies when a test instance is solved for $\delta = 0$. Note that, since $n_j \leq 10$ and $w_{bj} = 1$ $(\forall b, j)$, we have that when $\kappa = 10$, all jobs are allowed to be processed in the same week.

From the order release date r_j and the order due date d_j we determine the internal job release dates r_{bj} and due dates d_{bj}. We perform computational experiments for $\delta = 0$ and $\delta = 1$ on the same test instances. When $\delta = 0$ and $w_{bj} = 1$ $(\forall b, j)$ the minimal duration of the order is $\lceil \frac{n_j}{\kappa} \rceil$. As mentioned before, when $\delta = 0$ and $\kappa > 1$, the order has more internal slack. Note that $\delta = 0$ and $\kappa = 1$ corresponds to $\delta = 1$.

The number of orders that are allowed to be processed in a week will be smaller in the first weeks of the aforementioned order arrival process. We therefore simulate an order arrival process for $t = 0, \ldots, 60$, and discard the first 30 weeks and consider, for the test instances with parameter T, all orders that are allowed to be produced in weeks $\{30, 30 + T\}$. Each simulation of the order arrival process generates one test instance for each value of the parameter T ($T \in \{5, 10, 15, 20, 25, 30\}$). When the interval in which a job B_{bj} is allowed to be produced (i.e., $\{r_{bj}, d_{bj}\}$) falls *partly* inside the planning horizon $\{30, 30 + T\}$, we consider that job completely if at least half of the interval $\{r_{bj}, d_{bj}\}$ falls inside the planning horizon $\{30, 30 + T\}$.

For the determination of the various capacity profiles of the resources, we first measure how much capacity Q_{it} is approximately required per week t in the given realization of the order arrival process, per machine group M_i $(i = 1, \ldots, m)$. Note that each job B_{bj} requires p_{bj} hours of processing time on machine group μ_{bj} in the time window $\{r_{bj}, d_{bj}\}$. We calculate Q_{it} as follows:

$$Q_{it} = \sum_{\{(b,j)|\mu_{bj}=M_i\}} \frac{p_{bj}}{d_{bj} - r_{bj} + 1},$$

where $\frac{p_{bj}}{d_{bj}-r_{bj}+1}$ is the average required capacity per week for job B_{bj} in its time window $\{r_{bj}, d_{bj}\}$. Note that $\sum_i Q_{it}$ is the approximate required operator capacity in week t. From Q_{it} we calculate $\overline{Q}_i$, the average required machine group capacity from week 30 to week 60 (i.e., corresponding to $T = 30$) in the given realization of the order arrival process, i.e., $\overline{Q}_i = \sum_{t=30}^{60} \frac{Q_{it}}{31}$. For the experiments, unless noted otherwise, we generate instances where the utilization rate of the total machine group capacity is approximately 80%, and the utilization rate of the total machine group capacity in regular time is approximately 100%. Accordingly, we set the machine group capacity in regular time mc_{it} to $\overline{Q}_i$ and the total machine group capacity $\overline{mc}_{it}$ to $1.25\overline{Q}_i$. Furthermore, for the experiments we generate instances where the utilization rate of the operators in regular time is approximately 125%. Accordingly, we set the regular operator capacity c_t to $0.8 \sum_i \overline{Q}_i$. The utilization rate of the machine groups in regular time is thus limited to 80% by the operator capacity. Observe that we do not vary the values of mc_{it}, $\overline{mc}_{it}$, and c_t over the time periods t, so these values are the same for all periods t, regardless of the length of the planning horizon.

We draw the available operator overtime capacity per week o_t and hiring capacity per week h_t uniformly from $[0, 0.5c_t]$. We draw the available subcontracting capacity per week s_t uniformly from $[2c_t, 5c_t]$, which is sufficiently high in order to be able to easily obtain an initial feasible solution.

We complete the test instance generation procedure by choosing the non-regular capacity cost parameters $(\overline{o}_t, \overline{h}_t, \overline{s}_t, t = 0, \ldots, T)$ as follows:

$$\begin{aligned} \overline{o}_t &= 1, \ (\forall t)\,, \\ \overline{h}_t &= 2, \ (\forall t)\,, \\ \overline{s}_t &= 3, \ (\forall t)\,. \end{aligned}$$

In Section 5.4 we discuss the preliminary experiments in which we select a branch-and-price based algorithm for further evaluation. Subsequently, in Section 5.5 we discuss various experiments, amongst which experiments with various combinations of the remaining parameters $\overline{mc}_{it}$, mc_{it}, c_t, δ and κ (if $\delta = 0$).

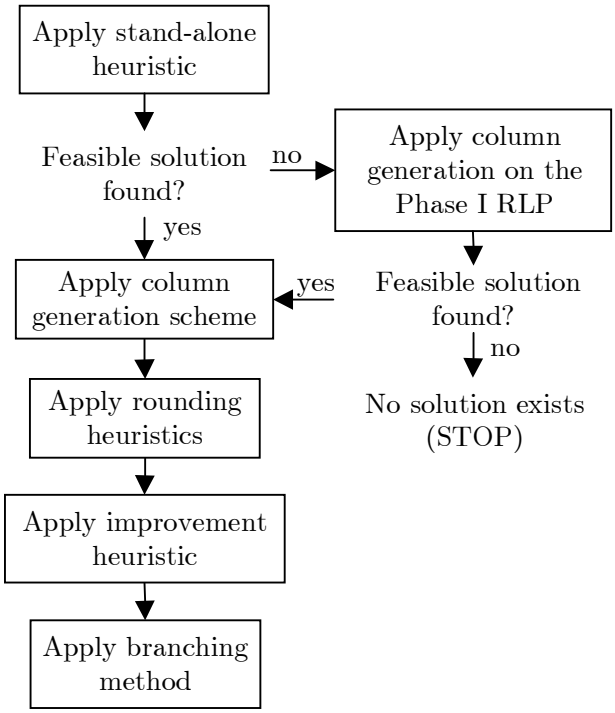

Figure 5.1: Algorithm scheme.

5.3 Algorithm overview

In this section we give an overview of the algorithms that we use for preliminary testing. We refer to Sections 4.4 to 4.6 for an extensive description of these algorithms.

Each branch-and-price algorithm that we tested is executed as a chain of subalgorithms. Figure 5.1 displays a scheme of the branch-and-price algorithm. We initialize the *RLP* in the root of the branching tree with order plans that constitute a feasible primal solution to the LP relaxation. We try to find such a feasible solution with a stand-alone heuristic. In all branch-and-price algorithms we apply the earliest due date heuristic (*EDD*) for this purpose. If this heuristic fails to find a feasible solution, we apply column generation on the Phase I *RLP* to either find a feasible primal solution to the LP relaxation, or prove that no such solution exists. In any other node of the branching tree, we do not use the *EDD* heuristic, because it tends to fail to find a feasible solution more often when more constraints are added to the problem in the branching scheme. Instead, we immediately apply column generation on the

Phase I RLP.

Table 5.2 shows the approximate size of the initial RLP for instances with $T \in \{10, 20, 30\}$. It shows that the size of the RLP increases considerably with T.

T	#nonzeros	#variables	#rows
10	4000	1000	1000
20	11000	2800	2800
30	25000	6500	6500

Table 5.2: Size of the initial RLP.

After initializing the RLP, we apply column generation on the RLP in order to determine the optimal solution to the LP. Since this solution is usually fractional, we may use one of the branch-and-price methods to find a feasible integral solution to the ILP. However, in order to find an incumbent solution that may fathom many nodes in the branching scheme, we first apply various rounding heuristics and an improvement heuristic.

The branch-and-price methods use Lagrangian lower bounds (see Section 4.5) to fathom nodes, and use a depth-first node selection strategy.

Table 5.3 gives an overview of the heuristics that we use to determine an incumbent solution before branching. Table 5.4 gives an overview of all branch-and-price algorithms that we use for the preliminary testing.

EDD	stand-alone heuristic based on the earliest due date priority rule
IH	improvement heuristic that uses the EDD solution as an initial solution
$RH1\,(\epsilon)$	rounding heuristic 1 with rounding threshold value $0 < \epsilon < 1$
$RH2\,(\epsilon)$	rounding heuristic 2 with rounding threshold value $0 < \epsilon < 1$
$RH3\,(\epsilon)$	rounding heuristic 3 with rounding threshold value $0 < \epsilon < 1$
$RH4\,(k)$	rounding heuristic 4 with parameter k $(k \in N)$

Table 5.3: Overview of heuristics.

5.4 Preliminary test results

In this section we present the computational results for the preliminary experiments with various combinations of the algorithms on the instance classes discussed in Section 5.2.

For the preliminary experiments we repeat the order arrival process twice for each value of parameters m ($m \in \{3, 5, 7, 10\}$) and λ ($\lambda \in \{0.5, 1, 2\}$) to

TBS	truncated (after 30 minutes) branching strategy BS with Lagrangian lower bound determination,
$TBS(noLLB)$	TBS without Lagrangian lower bound,
$EAS(\epsilon)$	TBS with an ϵ-approximation strategy,
TBS+IH	TBS algorithm that solves IH every 3 minutes to improve the upperbound,
$TBS90$	same as TBS, but truncated after 90 minutes,
$HBS1, \ldots, HBS4$	incomplete branching strategies based on selecting one order plan in each node of the branching tree,
$HBS1+, HBS2+$	same as $HBS1$, $HBS2$, but with additional order plans available for each order in the initial RLP.

Table 5.4: Overview of branch-and-price methods.

obtain $2 * 12$ classes, each containing 6 instances (one for each value of T). We solve all 144 instances for both $\delta = 0$ and $\delta = 1$. When we solve an instance for $\delta = 0$, we allow only two jobs of the same order to be produced in the same week, i.e., in the preliminary experiments we set $\kappa = 2$. This parameter setting comes closest to the parameter setting $\delta = 1$.

In conjunction with the test instances, we generate two feasible order plans (i.e., for $\delta = 0$ and $\delta = 1$) for each order in each test instance, to form the initial RLP for a branch-and-price algorithm. For this purpose we use the EDD heuristic, or, if this algorithm can not find a feasible solution, we apply column generation on the Phase I RLP.

The outline of this section is as follows. We first test the performance and speed of the heuristics that are listed in Table 5.3. We present the computational results of these experiments in Section 5.4.1. Subsequently, we test the branch-and-price methods that are listed in Table 5.4. We present the computational results of these experiments in Section 5.4.2. Finally, in Section 5.4.3 we compare all computational results, and choose a branch-and-price algorithm that we use for more extensive testing in Section 5.5.

5.4.1 Preliminary test results for the heuristics

In this section we compare the performance of the heuristics that are listed in Table 5.3. IH must start from a feasible solution. Gademann and Schutten tested IH, as well as various variants of this improvement heuristic, extensively, and showed that the heuristic performance strongly depends on the initial feasible solution that the heuristic starts from (Gademann and Schutten, 2001). They showed that IH performs well when it starts from an initial solution in which the slack of the jobs is divided evenly. Here we use the solution found by the rounding heuristic $RH4(1)$ as the initial solution for the rounding heuristic IH. The rounding heuristics each use the columns and the LP solution that

are found after solving the LP in the root node via column generation. We test the performance of the rounding heuristics $RH1(\epsilon)$, $RH2(\epsilon)$ and $RH3(\epsilon)$ for rounding parameter $\epsilon = 0.01$, and $\epsilon = 0.05$, and rounding heuristic $RH4(k)$ for parameter $k \in \{1, 2, 3\}$. Table 5.5 compares the average and maximum execution times of the heuristics, tested on all 288 instances (i.e., 144 instances solved for both $\delta = 0$ and $\delta = 1$). Table 5.5 shows that the rounding heuristics

Heuristic	Average	Maximum
EDD	0	0.01
IH	132.87	3608.89
$RH1(0.01)$	27.14	1116.69
$RH1(0.05)$	30.10	1615.72
$RH2(0.01)$	31.85	2314.64
$RH2(0.05)$	37.98	2314.13
$RH3(0.01)$	100.44	7526.77
$RH3(0.05)$	103.53	6319.65
$RH4(1)$	74.81	5631.74
$RH4(2)$	32.78	2108.23
$RH4(3)$	27.87	1662.10

Table 5.5: Execution time of heuristics (in seconds).

all require small execution times. For $RH4(k)$ the computation time drops significantly when k increases. This may be expected, since a larger value of k implies that in each iteration more variables are rounded at once, which will lead to a solution faster. Table 5.5 also shows that the improvement heuristic IH requires significantly more computation time than the other heuristics. Whether this leads to improved solution values is reflected in Table 5.6. Table 5.6 compares the solution values found by the algorithms. We compare the number of times (again out of 288) that each heuristic finds the best solution value among all the heuristic solutions, as well as the number of times it finds the unique best solution value among all heuristic solutions. The last column displays the number of times that each heuristic finds a solution value that is proven to be optimal by one of the complete branching strategies.

Table 5.6 shows that IH finds a better solution value than the other heuristics for the majority of the instances. Its average solution value is significantly smaller, and it finds the optimal solution for far more instances than the other heuristics. The additional computational effort for the IH heuristic thus clearly pays off. We also note that IH almost always improves the solution value of $RH4(1)$.

Observe that for $k \in \{2, 3\}$, $RH4(k)$ finds a solution for significantly fewer instances than $RH4(1)$. The performance also drops for $k \in \{2, 3\}$, so, based on these aspects, $k = 1$ is clearly preferred.

There is not much difference in performance between the two parameter set-

Heuristic	# Solutions found	Avg. solution value	# Best	# Unique best	# Worst[a]	# Unique worst	# Proven optimal
EDD	288	1094.7	3	1	194	82	2
IH	288	596.5	265	212	0	0	70
$RH1\,(0.01)$	281	826.6	21	0	38	3	16
$RH1\,(0.05)$	279	815.0	21	0	36	4	16
$RH2\,(0.01)$	288	768.6	39	3	5	2	30
$RH2\,(0.05)$	288	773.0	36	0	6	1	30
$RH3\,(0.01)$	288	764.9	41	0	13	1	38
$RH3\,(0.05)$	287	762.3	50	4	20	3	41
$RH4\,(1)$	288	784.0	47	0	20	5	41
$RH4\,(2)$	196	702.4	30	1	96	2	24
$RH4\,(3)$	159	704.5	31	6	135	3	22

[a] We subtracted the number of times that the solution found was also the best.

Table 5.6: Comparison of heuristic solutions.

tings for the rounding heuristics $RH1\,(\epsilon)$, $RH2\,(\epsilon)$ and $RH3\,(\epsilon)$. For $RH1\,(\epsilon)$ and $RH2\,(\epsilon)$ the CPU-time/solution performance trade-off is best for $\epsilon = 0.01$. For $RH3\,(\epsilon)$ it is best for $\epsilon = 0.05$. Finally, the tables show that although the EDD heuristic requires hardly any computation time, it clearly has the worst solution performance of all heuristics. The EDD heuristic does perform better after we translate its order schedules into order plans using the method described in Lemma 4.1, and let the initial RLP find the optimal order schedules along with these order plans. Nevertheless, even with this improvement the EDD heuristic is outperformed by the other heuristics.

We thus conclude that the rounding heuristics $RH1\,(0.05)$, $RH2\,(0.05)$, $RH3\,(0.01)$ and $RH4\,(1)$ are the best rounding heuristics. In all branch-and-price based algorithms before we start branching we execute these heuristics in this order, followed by IH, which uses the solution found by $RH4\,(1)$ as the initial solution.

5.4.2 Preliminary test results for the branch-and-price methods

In this section we compare the performance of the branch-and-price algorithms that are listed in Table 5.4. Before the branching starts, we optimize the LP in the root by performing column generation on the initial RLP (see also Figure 5.1). In other words, we start by solving the LP in the root node. Table 5.7 displays some characteristics of this first column generation iteration. In the table we compare the CPU-time that is required to solve the LP in the root node for both $\delta = 0$ and $\delta = 1$, as well as the average number of columns (order

plans) that are generated per order in the column generation scheme in order to solve the LP. The final two columns compare the average LP solution value for both $\delta = 0$ and $\delta = 1$.

The size of the LP model is largely determined by the number of orders, as well as the size of the order plans. Since both are closely related to T, we divide the 144 test instances in 6 classes of 24 instances, each class representing instances with a different T ($T \in \{5, 10, 15, 20, 25, 30\}$).

T	# Cases	Avg. CPU-time (sec.)		Avg. #columns per order		Avg. LP solution value	
		δ=0	1	0	1	0	1
5	24	0.07	0.09	1.62	1.79	269.95	318.19
10	24	0.64	1.28	1.82	2.18	326.42	387.24
15	24	2.55	5.01	1.93	2.53	252.95	338.39
20	24	11.96	22.05	2.08	2.74	176.18	357.57
25	24	21.42	39.09	2.12	2.84	234.94	457.28
30	24	57.80	89.95	2.13	3.09	442.26	740.10

Table 5.7: Computational results of solving the LP in the root node.

Table 5.7 clearly shows that the CPU-time increases with the length of the time horizon. We note that the average CPU-time in the table is often increased by a small number of instances in each class that require much more CPU-time. This is a typical symptom of solving LP problems with the simplex method. For example, in the class of instances with $T = 30$ and $\delta = 0$, there are only 5 instances that require more CPU-time than the average. The maximum CPU-time of these 5 instances is 335.04 seconds. If we omit the 5 instances of cycling for $T = 30$ and $\delta = 0$, the average CPU-time is only 20.26 seconds.

For $\delta = 0$, the jobs of the orders have more slack, and therefore, per definition, the LP solution value for $\delta = 0$ is never larger than the LP solution value for $\delta = 1$. Table 5.7 shows that the average LP solution value is smaller for $\delta = 0$ (and $\kappa = 2$) than for $\delta = 1$. The difference between the average LP solution value for $\delta = 0$ and $\delta = 1$ increases non-proportionally with T. Apparently, the instances with $\delta = 0$ benefit more from considering a longer planning horizon.

Table 5.7 also shows that only few order plans are generated per order. This demonstrates the strength of the approach: apparently only few order plans are needed to be able to find the optimal order schedules of the LP.

The difference between $\delta = 0$ and $\delta = 1$ is also apparent in the average number of columns that are added per order in the column generation scheme. This can be attributed to the increase in slack: a dominant order plan for $\delta = 0$ allows more order schedules than an order plan for $\delta = 1$. Thus, in general, fewer order plans are necessary for $\delta = 0$.

We truncate each branch-and-price algorithm after 30 minutes, except for $TBS90$, which is truncated after 90 minutes. We first analyze the general computational results for these algorithms. Tables 5.8 and 5.9 show the general computational results for all 288 instances, i.e., 144 instances solved for both $\delta = 0$ and $\delta = 1$, for all branch-and-price methods, except for $TBS90$. Since the computational results may diverge for $\delta = 0$ and $\delta = 1$, we have also compared the algorithms for both $\delta = 0$ and $\delta = 1$. This, however, leads to the same conclusions for $\delta = 0$ and $\delta = 1$. Therefore we show the general results for $\delta = 0$ and $\delta = 1$ combined.

	#Truncated	Avg. CPU-time (sec.)[a]	Avg. solution value	#Improvement found	Avg. last improvement time (sec.)	#Proven optimal
TBS	196	44.33	598.85	111	249.6	96
$TBS(noLLB)$	213	65.59	602.52	83	254.6	81
$EAS\,(0.05)$	186	38.88	600.15	102	295.8	83
$TBS{+}IH$	193	68.07	597.05	115	273.2	97
$TBS90$	191	148.31	596.61	118	1068.9	98
$HBS1$	127	102.91	590.26	250	483.7	87
$HBS2$	72	120.73	593.06	248	323.6	82
$HBS1{+}$	136	118.80	596.95	252	438.8	85
$HBS2{+}$	72	155.79	595.91	262	355.2	83
$HBS3$	153	105.98	597.87	240	472.6	82
$HBS4$	80	154.19	597.63	257	368.5	82

[a]These results concern the instances that were not truncated.

Table 5.8: General computational results of the branch-and price methods (1).

The first column of Table 5.8 shows the number of instances (out of 288) that are truncated after 30 minutes. The second column is the average CPU-time for the instances that are not truncated. This does not include the CPU-time required for the heuristics in the root node. The remaining columns in Table 5.8 indicate the average solution value, and the number of times the algorithm finds an improvement over the best heuristic solution, the average last improvement time, and the number of times that the solution found is the optimal solution. A solution is optimal if this is proven by one of the complete branching strategies. Table 5.9 shows the number of times the method finds the best, unique best, worst and unique worst solution value of all methods.

From the first and second column of Table 5.8, the importance of the Lagrangian lower bound determination is clear: TBS is not only truncated less often than $TBS(noLLB)$, it also solves the non-truncated instances in less time.

$EAS\,(0.05)$ shows a small decrease in the number of truncated instances as compared to TBS, and a decrease in average CPU-time. The approach of

	#Best solution	#Unique best solution	#Worst solution[a]	#Unique worst solution
TBS	167	0	61	0
$TBS(noLLB)$	138	0	96	14
$EAS\,(0.05)$	146	3	83	14
$TBS{+}IH$	171	0	59	0
$TBS90$	173	1	55	0
$HBS1$	178	27	35	0
$HBS2$	155	11	54	2
$HBS1+$	153	9	61	7
$HBS2+$	145	12	75	5
$HBS3$	146	7	69	8
$HBS4$	141	8	85	7

[a] We subtracted the number of times that the solution found was also the best.

Table 5.9: General computational results of the branch-and price methods (2).

cutting off nodes in situations where the lower bound approaches the upper bound within 5% on the one hand speeds up the branching. On the other hand, this same approach allows that nodes are discarded, below which a better solution value may be found. The incumbent solution (upper bound) thus may improve less often, which slows down the branching. Apparently, with respect to computation time, the former effect has the upper hand.

$TBS{+}IH$ calls IH every 3 minutes to improve the current upper bound. As a result, fewer nodes can be evaluated since the method is truncated after 30 minutes. Table 5.8 shows that the instances that are not truncated, on average require more CPU-time because of applying IH. However, since $TBS{+}IH$ does have a better solution performance, it is preferred over the TBS method.

The $TBS90$ method truncates 191 instances, which is only 5 instances fewer than TBS. It finds a better solution value than TBS in 10 of the 288 instances. In these 10 instances the average solution improvement is 7.27%. The increased branching time thus hardly pays off.

The six incomplete branching strategies HBS are truncated significantly less often than the best BS branching strategy $TBS{+}IH$. The best HBS method is $HBS1$, which finds the most unique best solution values of all methods (see Table 5.9), and which finds the least number of (unique) worst solution values among the HBS methods. The solution performance of the other 5 HBS methods diverges much more. On the one hand they find the unique best solution value more often, on the other hand they also find the (unique) worst solution value more often. Recall from Chapter 4 that $HBS1+$ and $HBS2+$ are the same as $HBS1$ and $HBS2$, except that before branching, we add up

to 5 additional order plans per order to the column pool. This apparently does not give any improvement: more instances are truncated, and the solution performance clearly decreases. In $HBS3$ and $HBS4$ we change the selection criterion for selecting the next order to branch on. This experiment also does not give any improvements in the solution performance.

The average last improvement time of $TBS{+}IH$ is approximately 5 minutes. Apparently, $TBS{+}IH$ either finds a solution in short time, or it branches for a long time without finding an improvement. $HBS1$, on the other hand, keeps improving the solution even after a much longer time. Based on this observation, we perform an experiment in which we apply $TBS{+}IH$, truncated after 5 minutes, followed by $HBS1$, which we truncate after 10 minutes. $HBS1$ initially uses the final column pool of $TBS{+}IH$. We refer to this method as $Comb1$. A drawback of this algorithm is that, unless $TBS{+}IH$ finds an optimal solution within 5 minutes, it can never prove that the solution found is optimal. We therefore also perform an experiment in which we first use $HBS1$, which we truncate after 10 minutes, followed by $TBS{+}IH$, which we truncate after 5 minutes. $TBS{+}IH$ thus uses the column pool of $HBS1$. In this method, $TBS{+}IH$ has 5 minutes to improve the solution found by $HBS1\,(10)$, or to prove that the solution found by $HBS1\,(10)$ is optimal. We refer to this method as $Comb2$. Table 5.10 shows the computational results of these two algorithms, tested on the 288 instances.

	Avg. solution value	#Truncated	#Better than		#Same solution as		#Unique best method
			$TBS{+}IH$	$HBS1$	$TBS{+}IH$	$HBS1$	
$Comb1$	594.87	186	49	53	209	166	13
$Comb2$	589.56	190	87	40	173	221	5

Table 5.10: Computational results for $Comb1$ and $Comb2$.

Table 5.10 shows that $Comb2$ performs slightly better than $Comb1$. Since $Comb2$ performs just as well, or even better than $TBS{+}IH$ and $HBS1$ in almost all instances, it may be preferred over these methods.

In Table 5.11 we focus on some specific computational results of $TBS{+}IH$. It shows various performance characteristics of this method on 6 classes of instances, each of which represent 24 instances with a different planning horizon ($T \in \{5, 10, 15, 20, 25, 30\}$).

The first two columns show that the number of truncated instances increases with the length of the planning horizon. When T is 15 weeks and larger, less than half of the instances are solved to optimality. For the majority of the classes, the instances are truncated more often for $\delta = 1$ than for $\delta = 0$. On the one hand, $\delta = 0$ offers more flexibility to plan the jobs, which makes problems easier to solve. As a result, there will be more solutions (i.e., order schedules) that yield the same objective value. On the other hand, the solution

	#Truncated		Avg. #nodes[a]		Avg. #cuts[a]		Avg. tree depth[a]	
T	δ=0	1	0	1	0	1	0	1
5	1	0	190898	-	134878	-	37.0	-
10	13	10	115426	109469	85857	70769	29.4	28.1
15	17	20	56268	59371	42323	40137	40.4	39.4
20	19	23	30826	40006	23181	27637	54.0	46.0
25	21	24	17505	20810	13338	14695	58.1	47.8
30	22	23	10380	14321	8067	10180	60.8	52.1

[a]These results only concern the instances that were truncated.

Table 5.11: Computational results for $TBS{+}IH$.

space becomes larger, so more branching is required. Table 5.11 shows that the former effect dominates the latter.

The average number of nodes that are explored generally decreases with T. This may be expected since the problem size (as well as the LP) increases with T. For $\delta = 1$ more nodes are cut off. As a result, the number of nodes that are explored for $\delta = 0$ is usually fewer than for $\delta = 1$. Approximately 70% of all nodes are cut off (i.e., fathomed or pruned), regardless of the length of the planning horizon.

The average maximum tree depth significantly increases with the length of the planning horizon. The table shows that when the tree depth is too large, although good solutions may have been found, it becomes impossible to prove that the solution is optimal within 30 minutes. The average tree depth is significantly smaller for the instances that are solved to optimality, although in a small number of instances the optimal solution is found even though the tree depth is large (i.e., up to 61).

The computational results of all methods together show that in 229 instances out of 288, the branching methods either improved the best heuristic solution, or proved that it is optimal. This clearly demonstrates the contribution of the branching methods. We performed an experiment to analyze the importance of using heuristics in the root node, for the performance of the branching methods. For this purpose we tested a variant of $TBS{+}IH$ in which we do not use EDD to initialize the RLP. Instead we start from an arbitrary solution, which may not be feasible. Furthermore, in this algorithm we do not use the various rounding heuristics and IH before branching. The results showed that this method finds a worse solution than $TBS{+}IH$ in 211 out of the 288 instances, and a better solution in 3 instances. The average solution found by this method is 865.24, which is much larger than the average of 597.05 for $TBS{+}IH$. Finally, the method without the heuristics is also truncated more often: in 211 versus 193 instances. We conclude that the use of heuristics in

the root node clearly pays off.

The branching methods together solve 116 instances out of 288 to optimality. For the remaining 172 instances we perform an additional experiment, in which we use $TBS{+}IH$ and the best solution found this far as an initial upper bound. In this experiment, only for one instance optimality is proven. For the remaining 171 instances we use $EAS(\epsilon)$ to determine the size of the gap between the optimal solution and the best solution found. $EAS\,(0.05)$ determines that for 12 instances the gap is at most 5%, and $EAS\,(0.1)$ determines that for 8 instances the gap is at most 10%. For the remaining 151 instances we are not able to prove within 30 minutes that the gap is less than 10%.

5.4.3 Conclusions and algorithm selection

In Section 5.4.2 we showed that the best branching strategy with respect to the solution values is the $HBS1$ strategy. The biggest drawback of this method is that it is an incomplete branching strategy. Hence, unless there is no gap between the LP lower bound and the best solution value found, it can not be proven that the solution that is found by $HBS1$ is optimal. The $TBS{+}IH$ method is the best complete branching strategy. The best algorithm is $Comb2$, which combines the computational effectiveness of $HBS1$ with the possibility (of $TBS{+}IH$) to prove optimality, while the computation times are still acceptable. We thus use $Comb2$ for the remaining computational experiments.

The preliminary results show that the larger the number of machine groups in the instances, the harder the instances are to solve. When the number of machine groups is smaller, there is more planning flexibility. Also, the smaller the number of orders in the instances, the harder the instances are to solve. For example, the computational results (not mentioned in the previous sections) show that for the instances with $\lambda = 0.5$, 26 instances out of 96 are solved to optimality, as opposed to 32 instances for $\lambda = 1$, and 58 instances for $\lambda = 2$. When there are more orders (and jobs) in an instance, the proportional size of the jobs is smaller with respect to the machine group capacity. As a result, there is more flexibility to plan the jobs. This is underlined by the fact that for 53 instances out of the 58 instances that are solved to optimality for $\lambda = 2$ there is no gap between the LP lower bound and the optimal solution value. For $\lambda = 0.5$, there is no gap between the LP lower bound and the optimal solution for only 9 instances out of the 26 that are solved to optimality. Of course, the larger n, the larger the LP model will become. As a result, when λ is increased further, the instances will eventually become harder to solve.

Based on the preliminary results we generate a class of instances with $\lambda = 1$, $m = 5$ and $T \in \{5, 10, 15, 20, 25, 30\}$. This class appears to be the most interesting of the instance classes: the instances are not too easy, and not too difficult. In Section 5.5 we perform various analyses to determine how sensitive $Comb2$ is to various parameter settings.

5.5 Sensitivity analyses

In this section we present the results of various sensitivity analyses on various parameters in the test instances. Unless noted otherwise, the experiments are performed on a class of instances with $\lambda = 1$ and $m = 5$, which is determined by repeatedly simulating an order arrival process. The class contains 120 instances, i.e., 20 instances for each value of T ($T \in \{5, 10, 15, 20, 25, 30\}$). In conjunction with the test instances, we generate two feasible order plans (i.e., for $\delta = 0$ and $\delta = 1$) for each order in each test instance, to form the initial RLP for a branch-and-price algorithm. For this purpose we use the EDD heuristic.

When we solve an instance for $\delta = 0$, we allow only two jobs of the same order to be produced in the same week, i.e., we set $\kappa = 2$.

5.5.1 Length of the planning horizon

Table 5.12 displays the computational results of the $Comb2$ method. The results are divided into 6 categories, one for each value of T ($T \in \{5, 10, 15, 20, 25, 30\}$). Each category thus contains 20 instances that are solved for both $\delta = 0$ (and $\kappa = 2$) and $\delta = 1$.

	#Optimal		Avg. #nodes[a]		Avg. #cuts[a]		Avg. tree depth	
T	δ=0	1	0	1	0	1	0	1
5	19	20	49684	-	33533	-	2.3	4.8
10	14	13	183988	143738	127801	94623	11.2	15.7
15	7	2	154986	233463	103510	156342	24.1	28.7
20	2	0	91833	108364	67057	73913	45.6	40.0
25	2	0	39923	39375	28153	27588	55.6	44.6
30	0	0	22792	19233	15355	13084	61.9	43.7

[a] These results only concern the instances that were truncated.

Table 5.12: Computational results for various T.

From Table 5.12 we may draw the same conclusions as from Table 5.11 in Section 5.4.2: the complexity of the instances increases with the length of the planning horizon. The model size increases with T, and as a result, the required CPU-time to optimize the LP in each node increases as well. The number of nodes that are explored thus decreases with T. Also, the larger the problem becomes, the more violated precedence relations must be repaired. Table 5.12 indeed shows that the average tree depth increases with T.

Table 5.12 shows that in general, instances with a planning horizon of up to 10 weeks, and some instances with a planning horizon of 15 weeks can be solved to optimality. Instances with a larger planning horizon are usually truncated. Note that it is questionable whether in practice, resource loading problems

with a planning horizon of more than 10 weeks are very relevant. The longer the planning horizon, the less accurate the problem data may become, since disturbances in the production process may occur, or, more likely, changes in the demand requirements may occur. Hence, from a practical point of view, a planning horizon of 10 weeks is quite large, and solving instances with a larger planning horizon is more a mathematical challenge.

Since all instances with a planning horizon of 30 weeks are truncated, we perform an additional experiment to determine if splitting up the planning horizon in three, and solving the three 'easier' subproblems instead, will improve the overall solution. In other words, in this experiment we split up the problems with $T = 30$ in three subproblems, and compare the solution value of the problem with $T = 30$ to the sum of the solution values of the 3 subproblems. When the interval in which a job B_{bj} is allowed to be produced (i.e., $\{r_{bj}, d_{bj}\}$) falls partly inside the planning horizon of one of the subproblems, we only consider that job in the subproblem if at least half of the interval $\{r_{bj}, d_{bj}\}$ falls inside its planning horizon. Table 5.13 and 5.14 display the computational results of this experiment.

	#Truncated		Avg. solution value		Avg. CPU-time (sec.)		#Best solution	
T	δ=0	1	0	1	0	1	0	1
30	20	20	588.6	932.1	900	900	12	11
3×10	9[a]	17[a]	897.1	1013.7	496	1086	8	9

[a]This is the number of times that at least one of the subproblems was truncated.

Table 5.13: Computational results for splitting up the planning horizon of 30 weeks in three (1).

	Avg. operator overtime (hrs.)		Avg. hiring (hrs.)		Avg. subcontracting (hrs.)	
T	0	1	0	1	0	1
30	421.3	584.9	75.5	140.0	5.5	22.4
3×10	368.0	423.6	175.4	189.9	59.5	70.2

Table 5.14: Computational results for splitting up the planning horizon of 30 weeks in three (2).

Note that splitting up the planning horizon reduces the planning flexibility of the jobs. Nevertheless, Table 5.13 shows that the sum of the solution values of the 3 subproblems is smaller than the solution value found for the entire problem in nearly 50% of the instances. This demonstrates that it is useful to try to solve the problems to optimality. If we would use overlapping periods

of 10 weeks, some planning flexibility can be regained. This is a more realistic approach, which will likely yield even better solutions. The challenge of this approach is how to connect the solutions for the overlapping periods, and make them consistent. This experiment is subject of further research.

5.5.2 Number of machine groups

Table 5.15 shows the computational results of four classes of instances, each containing 60 instances with a different number of machine groups m ($m \in \{3, 5, 7, 10\}$). We solve each instance for $\delta = 0$ and $\delta = 1$. In the table we compare the number of times the instances are solved to optimality, and the use of nonregular machine group and operator capacity.

	#Proven optimal		Avg. operator overtime (hrs.)		Avg. regular time hiring (hrs.)		Avg. non-regular time hiring (hrs.)		Avg. sub-contracting (hrs.)	
m	δ=0	1	0	1	0	1	0	1	0	1
3	45	30	214.0	253.4	1.7	3.8	15.3	38.1	3.5	4.9
5	30	19	223.1	331.9	0.0	1.3	11.9	42.6	1.3	4.7
7	12	12	303.3	362.4	1.0	0.5	44.5	102.7	8.4	40.9
10	10	11	380.6	419.0	0.0	0.2	51.4	140.7	36.0	123.7

Table 5.15: Sensitivity analysis of m.

The table shows that the larger the number of machine groups, the more difficult it becomes to solve the instances to optimality. The larger m, the larger the jobs are relative to the machine group capacity. As a result, there is less flexibility to plan the jobs. For the same reason, also the use of nonregular machine group capacity increases. The increase in the use of machine group capacity in nonregular operator time is apparent in Table 5.15 from the increase in the use of operator overtime, and the use of hiring capacity in nonregular operator time. Also the use of subcontracting clearly increases. For $\delta = 0$ there is more planning flexibility, and the machine group capacity can therefore be utilized more efficiently. As a result, the required nonregular operator and machine group capacity is lower for $\delta = 0$ in all instances.

We conclude that in general, instances with 3 machine groups, and a large part of the instances with 5 machine groups can be solved to optimality. In the instance generation procedure, jobs are assigned to machine groups randomly. As a result, there is no clear bottleneck. Instead, the instance generation procedure makes that in fact all machine groups may have capacity problems, and form bottlenecks. It is likely that in practical situations, only a few machine groups are bottlenecks. It is likely that such problems, with, e.g., more than 5 machine groups, of which only a few (e.g., 3) are bottlenecks, are computationally easier than when all the machine groups are bottlenecks. This experiment

is subject of further research.

5.5.3 Operator capacity

In this section we analyze various regular operator capacity profiles. We solve the 120 instances for both $\delta = 0$ and $\delta = 1$. In Table 5.16 we compare the computational results for these instances with various regular operator capacity profiles. The first row corresponds to the experiment where the regular operator capacity per week is $0.7 \sum_i \overline{Q}_i$. In the second and third row the regular operator capacity per week is $0.8 \sum_i \overline{Q}_i$ and $\sum_i \overline{Q}_i$ respectively. This corresponds to operator utilization rates 140%, 125% and 100% respectively. Observe that increasing the operator capacity per week to more than $\sum_i \overline{Q}_i$ by definition makes no sense, since the machine groups can not handle more workload than $\sum_i \overline{Q}_i$ in regular time. We note that the average sum of the processing times of the jobs in the instances is approximately 3220 hours, i.e., on average: $\sum_i \overline{Q}_i \approx 3220$.

Operator capacity per week	# Proven optimal		Avg. solution value		Avg. operator overtime (hrs.)		Avg. regular time hiring (hrs.)		Avg. nonregular time hiring (hrs.)		Avg. subcontracting (hrs.)	
c_t	δ=0	1	0	1	0	1	0	1	0	1	0	1
$0.7 \sum_i \overline{Q}_i$	57	42	444.8	613.4	345.8	383.7	5.5	3.1	38.1	85.3	3.9	17.6
$0.8 \sum_i \overline{Q}_i$	44	35	263.4	439.6	209.2	299.4	0.2	1.4	23.7	52.6	2.1	10.8
$\sum_i \overline{Q}_i$	35	32	132.8	291.0	112.3	201.0	0.0	0.0	9.4	36.5	0.6	5.7

Table 5.16: Sensitivity analysis of the operator capacity profile.

Table 5.16 shows that the number of instances that are solved to optimality decreases when the available regular operator capacity increases. The additional operator capacity on the one hand increases the planning flexibility, which should make the instances easier to solve. On the other hand, the size of the solution space also increases, which makes it harder to prove optimality. Since the nonregular capacity usage is much lower for the class of instances with the highest operator capacity, from a practical point of view, proving optimality does not yield much.

As may be expected, the use of nonregular capacity decreases when the available regular operator capacity increases. This is most visible for the average required operator overtime capacity: the machine groups can be utilized more in regular time, and as a result, less overtime work is required.

For $\delta = 0$ there is more planning flexibility, and the operator capacity can therefore be utilized more efficiently. As a result, the required nonregular operator and machine group capacity is lower for $\delta = 0$ in almost all instances.

5.5.4 Machine group capacity

Similar to the sensitivity analysis of the regular operator capacity profiles, in this section we analyze various regular machine group capacity profiles. Table 5.17 shows the computational results for three machine group capacity profiles. The first row corresponds to the experiment where the machine group capacity for machine group M_i $(i = 1, \ldots, m)$ in regular operator time per week mc_{it} is $0.8\overline{Q}_i$, and the total machine group capacity per week $\overline{mc}_{it}$ is $\overline{Q}_i$. This corresponds to a utilization rate of 125% for the machine group capacity in regular time, and a utilization rate of 100% for the total machine group capacity. The second row corresponds to the experiment in which the utilization rates are 100% and 80% respectively. Finally, the third row corresponds to the experiment in which the utilization rates are 80% and 60% respectively. The operator capacity per week c_t in each row is $0.8 \sum_i \overline{Q}_i$. Each row corresponds to 120 instances, which are solved for both $\delta = 0$ and $\delta = 1$.

Machine group capacity per week		# Proven optimal		Avg. solution value		Avg. operator overtime (hrs.)		Avg. regular time hiring (hrs.)		Avg. nonregular time hiring (hrs.)		Avg. subcontracting (hrs.)	
mc_{it}	$\overline{mc}_{it}$	δ=0	1	0	1	0	1	0	1	0	1	0	1
$0.8\overline{Q}_i$	$\overline{Q}_i$	25	30	420.1	659.9	307.2	367.6	0.3	0.0	39.1	90.2	11.3	37.3
$\overline{Q}_i$	$1.25\overline{Q}_i$	44	35	263.4	439.6	209.2	299.4	0.2	1.4	23.7	52.6	2.1	10.8
$1.25\overline{Q}_i$	$1.66\overline{Q}_i$	79	60	200.5	316.8	163.3	228.9	4.2	4.7	13.4	31.8	0.7	5.0

Table 5.17: Sensitivity analysis of machine capacity profiles.

Table 5.17 shows that the number of instances that are solved to optimality increases with the machine group capacity. In the previous section we observed the opposite effect with respect to the operator capacity. Since we assume that operators are generic, the solution space increases more when we increase the operator capacity than when we increase the machine group capacity. Apparently with respect to machine group capacity this does not make it harder to prove optimality.

From a practical point of view, a production system where the machine group capacity utilization in regular time is 125%, is overloaded, which is not unusual. As may be expected, the average solution value clearly decreases with the increase of machine group capacity. Also the use of nonregular machine group capacity and operator capacity decreases when the machine group capacity increases. We note that the impact of increasing the machine group

capacity on the average solution value is generally smaller than the impact of increasing the operator capacity.

5.5.5 Internal slack of an order

In this section we analyze the relation between the average slack of the instances, and the solution performance. The average slack of an instance is calculated as: $\frac{\sum_j k_j}{n}$, where k_j is the slack of order J_j. We generated 100 instances with $\lambda = 1$, $T = 10$ and $m = 5$, calculated their average slack, and categorized the computational results of the instances in Table 5.18.

Average slack (range)	# Cases		%Proven optimal		Avg. treedepth	
	δ=0	1	0	1	0	1
$[1,2)$	0	31	-	51.6	-	12.5
$[2,3)$	20	53	65.0	45.3	5.4	6.3
$[3,4)$	50	15	66.0	53.3	4.7	0.0
$[4,5)$	27	1	66.7	100	2.8	1.0
$[5,6)$	3	0	66.7	-	0.0	-

Table 5.18: Sensitivity analysis of the average slack of jobs.

Per definition, the average slack is larger for the instances with $\delta = 0$. Table 5.18 shows that there is no relation between the slack and the number of instances solved to optimality. Hence, the increase in planning flexibility does not make it harder to prove optimality.

5.5.6 Nonregular capacity cost parameters

In this section we analyze the nonregular capacity cost parameters. Table 5.19 shows the computational results of 3 different settings of the cost parameters $(\overline{o}, \overline{h}, \overline{s})$: $(1,1,1)$, $(1,2,3)$ and $(1,2,10)$ respectively. Table 5.19 shows that for

Cost parameters			#Proven optimal		Avg. operator overtime (hrs.)		Avg. regular time hiring (hrs.)		Avg. non-regular time hiring (hrs.)		Avg. sub-contracting (hrs.)	
$\overline{o}$	$\overline{h}$	$\overline{s}$	δ=0	1	0	1	0	1	0	1	0	1
1	1	1	49	36	0.0	0.2	0.0	0.0	0.0	0.3	230.4	378.1
1	2	3	44	35	209.2	299.4	0.2	1.4	23.7	52.6	2.1	10.8
1	2	10	45	35	204.0	282.7	1.2	1.7	23.1	56.6	1.6	8.7

Table 5.19: Sensitivity analysis of the nonregular capacity cost parameters.

the setting $(\overline{o}, \overline{h}, \overline{s}) = (1, 1, 1)$, hardly any operator overtime or hiring capacity is used. Subcontracting capacity is used instead, because it can be used as both operator and machine group capacity. As soon as the cost of subcontracting increases, such as for $(\overline{o}, \overline{h}, \overline{s}) = (1, 2, 3)$, the use of subcontracting capacity decreases, and the use of nonregular operator capacity increases.

The number of instances that are solved to optimality is slightly larger for $(\overline{o}, \overline{h}, \overline{s}) = (1, 1, 1)$, from a practical point of view this parameter setting is not realistic. In practice, subcontracting usually is the last option for capacity expansion, and hiring is usually more expensive than working overtime, because hired staff is often not readily available.

5.5.7 Parameter κ

Finally, we analyze the parameter κ, which indicates the maximum number of jobs of the same order that are allowed to produced in the same week. Table 5.20 shows the computational results of the 120 instances, solved for $\delta = 0$ and $\kappa \in \{1, 2, 3, 10\}$.

κ	#Proven optimal	Avg. operator overtime (hrs.)	Avg. regular time hiring (hrs.)	Avg. non-regular time hiring (hrs.)	Avg. sub-contracting (hrs.)
1	35	299.4	1.4	52.6	10.8
2	44	209.2	0.2	23.7	2.1
3	49	188.4	0.5	18.6	1.1
10	50	188.3	1.6	17.5	4.3

Table 5.20: Sensitivity analysis of κ.

Table 5.20 shows that the number of times that the instances are solved to optimality increases with κ. The use of nonregular operator and machine group capacity usually decreases when κ increases. When κ is larger, there is more planning flexibility, and the operator and machine group capacity can be utilized more efficiently.

The value of κ that will most likely be used in practice is 1 or 2. Order schedules for larger κ may become impracticable when three or more jobs are assigned to the same week. It may be relevant to use large values of κ further in the planning horizon.

5.6 Conclusions

The computational results show that in most cases, the branch-and-price methods improve heuristic solutions. The best branch-and-price method with respect

to the solution performance is $Comb2$. It embeds the best characteristics of $HBS1$ and $TBS{+}IH$. $HBS1$ is a fast branching method, which improves the solution value even after a longer branching time than the other branching methods. $TBS{+}IH$ typically either improves the solution value within short time (and/or proves that the solution found is optimal), or it branches for a long time. The computational results show that $Comb2$ clearly outperforms the other branch-and-price methods. Also, with respect to solution performance, the branch-and-price methods also clearly outperform the heuristics. Only the heuristic IH has a satisfactory performance.

One of the objectives of this thesis is to provide algorithms that can solve resource loading problems of a size that is typically encountered in practice. The most important parameters that determine whether an instance can be solved to optimality are T and m. The computational results show that, in general, instances with a planning horizon of up to 10 weeks, and some instances with a planning horizon of 15 weeks can be solved to optimality. Furthermore, most instances with 3 machine groups and a large part of the instances with 5 machine groups can be solved to optimality. In Section 5.5.2 we argued that the instance generation procedure makes that in the resource loading instances all machine groups form bottlenecks. We conclude that with the aforementioned parameter settings, even large size resource loading problems can be solved to optimality. From a practical point of view, using a longer planning horizon (than 10 weeks) hardly makes sense, because the longer the planning horizon, the less accurate the problem data may become (see Section 5.5.1). It may be a mathematical challenge to solve instances with a larger planning horizon. For this purpose we suggest further research into methods that use a rolling horizon approach (see Section 5.5.1).

Chapter 6

Rough Cut Capacity Planning

Every fact that is learned
becomes a key to other facts.

- Edward L. Youmans (1821-1887)

6.1 Introduction

In Section 1.3.2 we introduced Multi-Project Rough-Cut Capacity Planning (RCCP) in project management as an analogue of resource loading in make-to-order production planning. In this chapter we show that with some modifications we can use the algorithms for resource loading (see Chapter 4) to solve RCCP problems.

We use basically the same integer linear programming model (presented in Section 3.6) to formulate RCCP problems, but the terminology is somewhat different. In the context of project management, we speak of *projects* subdivided into *activities*, rather than orders that consist of jobs. While in resource loading both machine capacity and operator capacity is required simultaneously to produce a job, in RCCP each activity is performed by one or more (renewable) *resources*. Typical resources in RCCP are labor, machines, equipment, space, and so on. We introduce parameter v_{bji} to indicate the fraction of activity B_{bj} of project J_j that is performed on resource M_i $(i = 1, \ldots, m)$. This yields the following ILP model for RCCP:

$$ILP : z^*_{ILP} = \min \overline{s} \sum_{t=0}^{T} \sum_{i=1}^{m} S_{it} + \sum_{j=1}^{n} \sum_{\pi \in \Pi_j} \rho_{j\pi} X_{j\pi} \theta, \tag{6.1}$$

subject to:

$$\sum_{\pi \in \Pi_j} X_{j\pi} = 1 \ (\forall j), \tag{6.2}$$

$$Y_{bjt} - \frac{\sum_{\pi \in \Pi_j} a_{bjt\pi} X_{j\pi}}{w_{bj}} \leq 0 \ (\forall b, j, t), \tag{6.3}$$

$$\sum_{t=r_j}^{T} Y_{bjt} = 1 \ (\forall b, j), \tag{6.4}$$

$$\sum_{j=1}^{n} \sum_{b=1}^{n_j} p_{bj} \upsilon_{bji} Y_{bjt} \leq \overline{mc}_{it} + S_{it} \ (\forall i, t), \tag{6.5}$$

$$\sum_{i=1}^{m} S_{it} \leq s_t \ (\forall i, t), \tag{6.6}$$

$$\text{all variables} \geq 0, \tag{6.7}$$

$$X_{j\pi} \in \{0, 1\} \ (\forall j, \pi \in \Pi_j \subset \Pi). \tag{6.8}$$

The objective function (6.1) minimizes the outsourcing costs and the total tardiness penalty costs. Constraints (6.2), (6.3) and (6.6)-(6.8) are the same as in the resource loading model ILP. Constraints (6.5) are the capacity restrictions for the resources. The term $p_{bj}\upsilon_{bji}Y_{bjt}$ in (6.5) is the number of hours that activity B_{bj} is performed on resource M_i in week t. Hence, the left hand side of (6.5) is the total workload assigned to resource M_i in week t. The right hand side of (6.5) is the sum of the total resource capacity in week t ($\overline{mc}_{it}$) and the subcontracting capacity S_{it}.

Note that the constraints that provide the duals for the pricing problem in resource loading are still embedded in the ILP for RCCP. In fact, the only algorithmic implications are the generalized precedence relations in RCCP (see Section 1.3.2), which have to be accounted for in the pricing algorithm, as opposed to the linear precedence relations in resource loading. We therefore use a generalization (with respect to the precedence relations) of the pricing algorithm of Section 4.3.1 to solve RCCP problems with the branch-and-price techniques presented in Chapter 4, as well as two other pricing algorithms.

The outline of this chapter is as follows. In Section 6.2 we present three pricing algorithms for RCCP problems. In Section 6.3 we discuss the modifications of the branching strategy, when we use the branch-and-price algorithm presented in Chapter 4 to solve a RCCP problem. We conclude this chapter with Section 6.4, where we discuss various existing heuristics for RCCP.

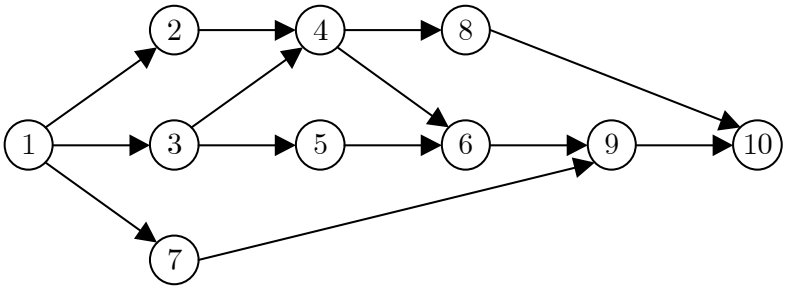

Figure 6.1: Example project with 10 activities.

6.2 Pricing algorithm

In this section we describe three pricing algorithms for finding project plans (in RCCP terminology we refer to an order plan as a *project plan*) with negative reduced costs. As in Section 4.3, the pricing problem can be subdivided into n independent subproblems, i.e., one for each project J_j $(j = 1, \ldots, n)$.

The outline of this section is as follows. In Section 6.2.1 we describe a pricing algorithm that accounts for generalized precedence relations. It is based on a forward dynamic programming (DP) approach, which is in fact a generalization of the algorithm of Section 4.3.1. The generalized DP has a multi-dimensional state space. For larger instances this may lead to computational problems, and therefore we try to improve the DP based algorithm. In Section 6.2.2 we discuss various ways to speed up the DP based algorithm. Furthermore, we present two alternative ways to solve pricing problems. In Section 6.2.3 we discuss a mixed integer linear programming based pricing algorithm, and in Section 6.2.4 we discuss a heuristic pricing algorithm. In Section 6.2.5 we conclude this section with a discussion on how to use a combination of exact pricing algorithms and heuristics to solve a pricing problem.

6.2.1 Pricing by dynamic programming

The generalized precedence constraints mainly influence the stages and the state space. Recall that in the DP algorithm of Section 4.3.1 each job represents a stage, and the state describes the completion time of a job. In Chapter 4, stages were naturally introduced since all jobs must be performed subsequently. This is no longer the case with generalized precedence constraints. In particular, several activities may be performed (partially) in parallel. This is best illustrated by an example. Consider a project with 10 activities. Figure 6.1 displays an acyclic directed graph representation of the precedence relations of the activities. Suppose (just as in Section 4.3.1) we would let each activity

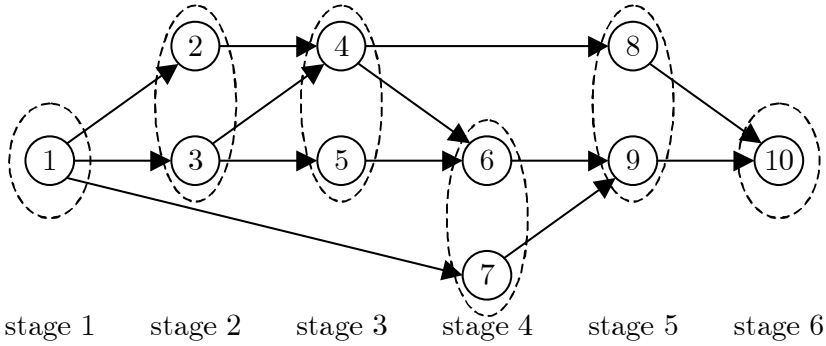

Figure 6.2: Example project with 10 activities and 6 stages.

represent a stage, and let each state describe the completion time of an activity in that state. In stage 2, the optimal decision for activity 2 (concerning its start time) restricts the completion time of activity 1 in stage 1. Since activity 3 in stage 3 may only start after activity 1 is completed (just like activity 2), the optimal decision concerning the start time of activity 3 in stage 3 depends on the previously chosen decision in stage 2. This conflicts with the principle of optimality for DP, which states that in each stage, the optimal decision for each of the remaining stages must not depend on previously reached states or previously chosen decisions (see Section 2.5). Hence the optimal decision for activity 3 can not be made. We conclude that we must redefine stages and states in the case of generalized precedence relations.

To define decisions and states properly, we first determine which activities should be considered in each stage. Hereby we basically have to take into account that an activity and its successors may never be in the same stage together. There are several ways to achieve this. We form the stages as follows. The activities that have no successors form the last stage (stage $\overline{\Gamma}$). For the remaining activities we determine the length of the longest path to any of the activities in the final stage, where the length is defined as the number of activities on the path. Subsequently, we form the other stages by activities that have the same longest distance to the last stage. This guarantees that a (not necessarily immediate) successor of an activity is always in a later stage. Hence, the stage with the longest distance (distance $\overline{\Gamma}-1$) is numbered as stage 1. We refer to the activities of stage Γ as B_Γ ($\Gamma = 1, \ldots, \overline{\Gamma}$). The example precedence graph in Figure 6.2 shows this for the aforementioned example project with 10 activities in 6 stages. The 6 stages are formed by activity sets $B_1 = \{B_{1j}\}$, $B_2 = \{B_{2j}, B_{3j}\}$, $B_3 = \{B_{4j}, B_{5j}\}$, $B_4 = \{B_{6j}, B_{7j}\}$, $B_5 = \{B_{8j}, B_{9j}\}$ and $B_6 = \{B_{10,j}\}$.

With the aforementioned definition of stages, the decision that is to be taken for the activities in each stage concerns either the start times or the

completion times of the activities of project J_j. Since the problem is symmetric with respect to that decision, we choose that the *decision* x_Γ in stage Γ defines the starting times $x_{\Gamma b}$ of the activities $B_{bj} \in B_\Gamma$ in that stage, i.e., $x_\Gamma = \{x_{\Gamma b} | B_{bj} \in B_\Gamma\}$. In forward dynamic programming, the decision x_Γ describes how state $i_{\Gamma-1}$ in stage $\Gamma - 1$ is transformed into a state i_Γ in stage Γ. The maximum completion time of an activity follows directly from the earliest start time of its successors. A decision x_Γ in stage Γ determines the maximum completion times of all immediate predecessors of the activities B_Γ in stage Γ. Note that these predecessors are not necessarily activities in stage $\Gamma - 1$, but these may also be activities from a stage before $\Gamma - 1$ (e.g., in Figure 6.2, activity 1 from stage 1, and activities 4 and 5 from stage 3 are predecessors of stage 4). Accordingly, we form a *state* $i_{\Gamma-1}$ by the maximum completion times $i_{\Gamma-1,b}$ of all activities B_{bj} in *or before* stage $\Gamma - 1$, that have a successor in stage Γ *or* in a stage after Γ. We denote this set of activities as B_Γ^+. We illustrate this with the aforementioned example project in Figure 6.2. The 6 states are formed as follows: $i_1 = \{i_{11}\}$, $i_2 = \{i_{21}, i_{22}, i_{23}\}$, $i_3 = \{i_{31}, i_{34}, i_{35}\}$, $i_4 = \{i_{44}, i_{46}, i_{47}\}$, $i_5 = \{i_{58}, i_{59}\}$ and $i_6 = \{i_{6,10}\}$. Hence, $B_1^+ = \{B_{1j}\}$, $B_2^+ = \{B_{1j}, B_{2j}, B_{3j}\}$, $B_3^+ = \{B_{1j}, B_{4j}, B_{5j}\}$, $B_4^+ = \{B_{4j}, B_{6j}, B_{7j}\}$, $B_5^+ = \{B_{8j}, B_{9j}\}$, and $B_6^+ = \{B_{10,j}\}$. Note that we included the completion time of activity B_{4j} in state i_4 of stage 4. Activity B_{4j} has immediate successors in stage 4 (i.e., activity B_{6j}) as well as in stage 5 (i.e., activity B_{8j}). If we would not include the completion time of activity B_{4j} in state i_4, we could not make an optimal decision for activity B_{6j}. Namely, this decision determines the completion time of activity B_{4j}, which in turn determines the decision for activity B_{8j}. This would imply that in stage 5, the optimal decision would depend on the decision made in stage 4, and would conflict with the principle of optimality for DP. Consequently, we include the completion time of activity B_{4j} in state i_4 of stage 4, to be able to make an optimal decision in stage 5 that does not depend on the decision in stage 4. Analogously, due to the precedence relation between activity B_{1j} and B_{7j} we must include the completion time of activity B_{1j} in the states of stages 2 and 3.

Let $a_{j\pi}$ be a (partial) project plan for project J_j in stage Γ and state $i_\Gamma = \left\{i_{\Gamma b} | B_{bj} \in B_\Gamma^+\right\}$ with value $F_j(i_\Gamma)$. Then $a_{j\pi}$ must allow activities $B_{bj} \in B_\Gamma$ of project J_j to be performed in weeks $x_{\Gamma b}, \ldots, i_{\Gamma b}$ for some $r_{bj} \leq x_{\Gamma b} \leq i_{\Gamma b} - w_{bj} + 1$. Accordingly, the previous state $i_{\Gamma-1} = \left\{i_{\Gamma-1,b} | B_{bj} \in B_\Gamma^+\right\}$ must be defined as follows. When activity $B_{bj} \in B_{\Gamma-1}$ has a successor in a stage after Γ, we have that $i_{\Gamma-1,b} = i_{\Gamma b}$. Otherwise, when activity $B_{bj} \in B_{\Gamma-1}$ only has successors in stage Γ, we have that $i_{\Gamma-1,b} = \min_{s \in DS_{bj} \cap B_\Gamma} x_{\Gamma s} - \delta$, where DS_{bj} are the immediate successors of activity B_{bj} of project J_j. Hence in this case the completion time of the activity is equal to the earliest start time of its successors, minus δ to comply with the one activity per week policy. We thus

have that:

$$\begin{aligned} i_{\Gamma-1} &= \left\{ i_{\Gamma-1,b} | B_{bj} \in B_{\Gamma}^{+} \right\}, \text{ where:} \\ i_{\Gamma-1,b} &= \begin{cases} i_{\Gamma b}, \text{ if } i_{\Gamma b} \in i_{\Gamma}, \\ \min\limits_{s \in DS_{bj} \cap B_{\Gamma}} x_{\Gamma s} - \delta, \text{ otherwise.} \end{cases} \end{aligned} \tag{6.9}$$

Allowing activity B_{bj} to be performed in the weeks $x_{\Gamma b}, \ldots, i_{\Gamma b}$ contributes $-\sum\limits_{u=x_{\Gamma b}}^{i_{\Gamma b}} \beta_{bju}$ to the reduced cost of the project plan. We now give the dynamic programming recursion to solve the j-th subproblem. The initialization is:

$$F_j(i_{\Gamma}) = \begin{cases} \alpha_j, \text{ if } \Gamma = 0, \\ \infty, \text{ otherwise.} \end{cases}$$

The recursion for $\Gamma = 1, \ldots, \overline{\Gamma}$ is then as follows. If $\forall B_{bj} \in B_{\Gamma}$, $r_{bj} + w_{bj} - 1 \leq i_{\Gamma b} \leq \overline{d}_{bj}$:

$$F_j(i_{\Gamma}) = \min_{\substack{r_{bj} \leq x_{\Gamma b} \leq i_{\Gamma b} - w_{bj} + 1 \\ B_{bj} \in B_{\Gamma}}} \left\{ F_j(i_{\Gamma-1}) + \sum_{B_{bj} \in B_{\Gamma}} \Delta_{bj i_{\Gamma b}} - \sum_{B_{bj} \in B_{\Gamma}} \sum_{u=x_{\Gamma b}}^{i_{\Gamma b}} \beta_{bju} \right\}, \tag{6.10}$$

otherwise:

$$F_j(i_{\Gamma}) = \infty.$$

Where $i_{\Gamma-1}$ in (6.10) is defined as in (6.9), and:

$$\Delta_{bjt} = \begin{cases} (t - d_{bj})\theta_{bj}, & \text{if } t > d_{bj} \\ 0, & \text{otherwise.} \end{cases}$$

Parameter $\theta_{bj} = \theta$, when tardiness of activity B_{bj} must be penalized (for example when the completion time of this activity is considered as a project milestone), and $\theta_{bj} = 0$ for activities that are allowed to be tardy without a penalty cost. Parameter Δ_{bjt} thus adds a tardiness penalty to $F_j(i_{\Gamma})$.

Lemma 6.1 *The optimal solution to the j-th pricing subproblem is found as:*

$$F_j^* = \min_{d_j \leq t \leq \overline{d}_j} F_j(i_{\overline{\Gamma}}), \ (j = 1, \ldots, n).$$

□

Accordingly, if $F^* = \min\limits_{1 \leq j \leq n} F_j^* \geq 0$, then the current RLP solution is optimal. If $F^* < 0$, then the current RLP solution is not optimal, and we need to introduce new columns (project plans) to the problem. Candidates are associated with those projects J_j for which $F_j^* < 0$, and they can be found by backtracking. The pricing algorithm for the j-th subproblem uses $O(n_j T^{\omega})$ space,

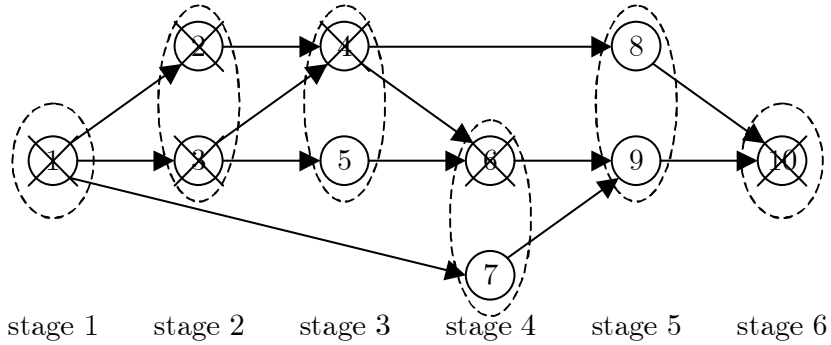

Figure 6.3: Example project with redundant activities.

where ω is the maximum number of activities in all states, i.e., $\omega = \max_{\Gamma} \left|B_{\Gamma}^{+}\right|$. The pricing algorithm can be solved in $O(n_j T^{2\omega})$ time. As can be expected, this pricing algorithm requires significantly more computation time than the pricing algorithms in Section 4.3 for instances of a similar size. This is mainly due to the increase in the number of possible states in each stage. In the next section we discuss some methods that significantly speed up the pricing algorithm.

6.2.2 Pricing algorithm speed up

In a pricing problem, there may be activities B_{bj} for which $\beta_{bjt} = 0$ $(\forall t)$ and $\theta_{bj} = 0$. Such activities will never contribute to the objective function of the pricing problem. We refer to these activities as *redundant activities*. Recall that in each iteration of a column generation scheme a project plan is added to the *RLP*, thereby introducing new possibilities to plan the activities. As a result, typically after a number of column generation iterations the number of redundant activities increases. We show that without affecting the solution we may eliminate some redundant activities from the pricing problem and hereby reduce the problem size. We illustrate this with the example project from Section 6.2.1 with 10 activities and 6 stages. Suppose in a certain column generation iteration the activities $\{1, 2, 3, 4, 6, 10\}$ are redundant. We have marked the redundant activities by a cross in Figure 6.3. We distinguish 3 types of redundant activities:

1. *Redundant activities to which there is no path from a non-redundant activity.* These can be planned as early as possible. In the example project these are activities $\{1, 2, 3, 4\}$.

2. *Redundant activities from which there is no path to a non-redundant activity* (activity 10 in the example project). These can be planned in

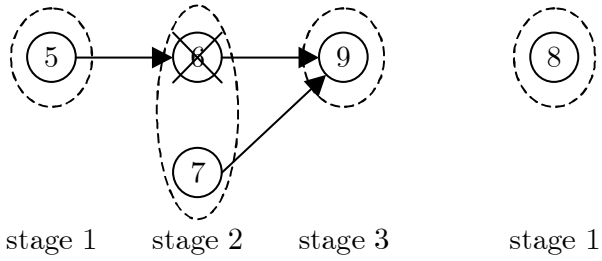

Figure 6.4: Subnetworks after eliminating redundant activities.

the project plan as late as possible. Note that such redundant activities do not exist when the lateness of activities at the end of the project is penalized.

3. The *remaining redundant activities* (activity 6 in the example project). These can not be planned as early or as late as possible, since this would also fix some non-redundant activities. These redundant activities can thus not be removed from the pricing problem. However, given the state $i_{\Gamma b}$ (completion time) of such an activity B_{bj} in a stage Γ, the corresponding optimal decision is always $x_{\Gamma b} = i_{\Gamma b} - w_{bj} + 1$, i.e., the start time is the completion time minus the minimal duration of the activity.

Removing activities $\{1, 2, 3, 4, 10\}$ from the example project yields two independent subnetworks in the pricing problem (see Figure 6.4). For each subnetwork we must redefine the sets of activities B_Γ and B_Γ^+ that respectively form the stages and are involved in the states. The subnetworks can be optimized independently, and the solutions must be brought together to form the resulting project plan and calculate its reduced cost.

Another speed up of the pricing algorithm can be obtained by discarding *redundant states*. A redundant state may occur when certain activities have exactly the same successors. Suppose for instance that an activity B_{bj} has a completion time $i_{\Gamma b}$ in state i_Γ, and there are other activities that have the same successors. In this case, the states in which these activities have the same completion time as activity B_{bj} dominate the states in which these activities have a smaller completion time, unless these activities have a deadline that is smaller than $i_{\Gamma b}$. We illustrate this with an example. In the example project in Figure 6.2 redundant states may occur in stage 4, with respect to activities 6 and 7, and in stage 5, with respect to activities 8 and 9. Suppose for activities 6 and 7 in stage 4 we have release dates $r_{6j} = 2$ and $r_{7j} = 2$, deadlines $\overline{d}_{6j} = 5$ and $\overline{d}_{7j} = 6$, and minimal durations $w_{6j} = 2$ and $w_{7j} = 1$. The possible completion times for activities 6 and 7 are $\{3, 4, 5\}$ and $\{2, 3, 4, 5, 6\}$ respectively. The non-redundant states $i_4 = \{i_{46}, i_{47}\}$ in stage 4 are: $\{3, 3\}$, $\{4, 4\}$, $\{5, 5\}$, and

$\{5,6\}$. For example state $\{3,3\}$ dominates state $\{3,2\}$, hence state $\{3,2\}$ is a redundant state. Although activities 6 and 7 do not have the same completion time in state $\{5,6\}$, this state is not redundant, since activity 6 is not allowed to be completed in week 6.

A similar speed up can be obtained in cases where activities in the same stage have the same predecessors and the same activity release date. In this case we may discard all decisions in which these activities have a different start time. We refer to these decisions as *redundant decisions*. In the example project redundant decisions may originate in state 2, with respect to activities 2 and 3.

With the aforementioned procedures we not only try to speed-up the pricing algorithms, we also hope to be able to optimize pricing problems for larger projects, since the procedures also require less computer memory. We expect that the network reduction procedure has the largest impact. However, typically in the first column generation iterations there are only a few redundant activities, and here the network reduction technique hardly has any effect on the problem size. In these cases the pricing problem may be too hard to solve by dynamic programming. To still be able to optimize pricing problems for large projects, in the next section we propose a pricing algorithm that may be less efficient (on small pricing problems), but that may be able to solve pricing problems for large projects.

6.2.3 Pricing by mixed integer linear programming

In this section we present a mixed integer linear programming (MILP) model of the pricing problem for a project J_j. As in Section 4.3 we use the binary decision variables $a_{bjt\pi}$ to form project plan $a_{j\pi}$ in the pricing algorithm. We introduce non-integer variables S_{bt} and C_{bt} ($S_{bt}, C_{bt} \in [0,1]$) to define variables $a_{bjt\pi}$ as follows:

$$a_{bjt\pi} = S_{bt} + C_{bt} - 1 \ \ (\forall b,t)\,,$$

where:

$$S_{bt} = \begin{cases} 1, & \text{if } t \text{ is larger than or equal to the start time of } B_{bj}, \text{ and} \\ 0, & \text{otherwise,} \end{cases}$$

and where:

$$C_{bt} = \begin{cases} 1, & \text{if } t \text{ is smaller than or equal to the completion time of } B_{bj}, \text{ and} \\ 0, & \text{otherwise.} \end{cases}$$

Accordingly, the following constraints must hold with respect to S_{bt} and C_{bt}:

$$\begin{aligned} S_{bt} &\geq S_{b,t-1} \ \ (\forall b, t>0)\,, \\ C_{bt} &\leq C_{b,t-1} \ \ (\forall b, t>0)\,, \\ S_{bt} &= 0 \ \ (t < r_{bj})\,, \qquad (6.11) \\ C_{bt} &= 0 \ \ \left(t > \overline{d}_{bj}\right). \qquad (6.12) \end{aligned}$$

Thus the variables S_{bt} and C_{bt} determine the start and completion times of the activities B_{bj} respectively. Observe that Constraints 6.11 stipulate that an activity may not be performed before its release date r_{bj}, and Constraints 6.12 stipulate that an activity may not be performed after its deadline $\overline{d}_{bj}$.

We model the precedence relation between activities B_{bj} and B_{sj} ($s \in DS_b$) as follows:

$$S_{st} \leq 1 - C_{bt} \ \ (\forall b, s \in DS_b \neq \emptyset, t)\,.$$

Furthermore, using variables $a_{bjt\pi}$, we model the minimal duration of activity B_{bj} as follows:

$$\sum_{t=r_{bj}}^{\overline{d}_{bj}} a_{bjt\pi} \geq w_{bj} \ \ (\forall b)\,.$$

We use variables $\rho_{bj\pi}$ ($\rho_{bj\pi} \geq 0$) to denote the number of weeks that an activity is allowed to be tardy. This variable is constructed as follows:

$$\rho_{bj\pi} \geq tC_{bt} - d_{bj} \ \ (\forall b, t \geq r_{bj} + w_{bj} - 1)\,.$$

The objective of the pricing problem for project J_j is to minimize the reduced cost of the project plan $a_{j\pi}$, i.e.:

$$\min \sum_{b} \rho_{bj\pi}\theta_{bj} + \alpha_j - \sum_{b,t} \beta_{bjt} a_{bjt\pi}\,.$$

The resulting MILP has $2n_j\,(T+1) + n_j$ non-integer variables, $n_j\,(T+1)$ binary variables and $O\left(n_j^2 T\right)$ constraints. We use the commercial solver ILOG CPLEX 7.0. to solve the MILP to optimality.

6.2.4 Heuristic pricing algorithm

The application of the explicit pricing algorithms of Sections 6.2.1 and 6.2.3 do prove on the one hand existence of a project plan with negative reduced costs and find the best project plan, but may on the other hand be very time-consuming. Observe that it is not necessary to find the project plan with the lowest reduced costs in order to improve the RLP and continue a column generation scheme. Instead a heuristic may be used to find a project plan with negative reduced costs using little computation time. Obviously, when a heuristic pricing algorithm does not find such a project plan, the column generation scheme may not be terminated as we would do with an explicit pricing algorithm, since there may exist a column with negative reduced costs.

We propose a heuristic pricing algorithm that is based on an improvement heuristic proposed by Gademann and Schutten (2001). Since this heuristic is an improvement heuristic (see Section 4.6.3) it requires a start solution, which in this algorithm is a project plan. Similar to heuristic IH in Section 4.6.3

it tries to improve the initial solution by iteratively changing the start and completion times of the activities in the project plan. It evaluates all possible changes in the time windows of all activities, and sorts these by increasing value of the expected change in the reduced cost. We accept the first change according to this sorting that leads to an improvement, i.e., that leads to a decreasing reduced cost. We repeat this procedure until no more improvement is found. We may in fact stop when a project plan is found with marginal negative reduced costs. However, such a project plan can only lead to a small improvement in the RLP. Since this may lead to many column generation iterations it seems better to find a project plan with significantly low reduced costs.

6.2.5 Hybrid pricing methods

With the three pricing algorithms presented in this section we hope to be able to optimize pricing problems of any reasonable size. We have shown that the efficiency of the DP based pricing algorithm of Section 6.2.1 depends rather on the project structure than on the number of activities in the project. Especially when a project network has a large ω-value (i.e., the highest number of activities in all states), the DP based algorithm may become less efficient. The efficiency of the MILP based pricing algorithm Section 6.2.3 rather depends on the number of activities and the number of precedence relations in the project network. This pricing algorithm may on the one hand be less efficient, but on the other hand it may be able to solve larger pricing problems.

In addition to the aforementioned pricing methods we propose a hybrid pricing strategy that uses the three aforementioned pricing methods in collaboration. In this method we take into account that the complexity of the pricing problem typically changes in each column generation iteration. Also, after applying the network reduction technique a pricing problem may fall apart into subproblems of different size and complexity. If the ω-value of the network in the pricing (sub)problem is not too large, we use the DP based algorithm. If the ω-value is too large we use the MILP based pricing algorithm. If the pricing (sub)problem is even too large for this algorithm, i.e., the algorithm does not find a project plan with negative reduced costs or does not prove that no such project plan exists within reasonable time, we use the heuristic pricing method as a final remedy to find a project plan with negative reduced costs. In Chapter 7 we evaluate the different solution methods for the pricing problem.

6.3 Branching strategy

The branching strategy that we use to construct a feasible ILP solution from the LP relaxation solution is virtually the same as the BS strategy presented

in Section 4.4.1. Recall that this branching strategy fixes a precedence relation in each node of the branching tree. Since we consider generalized precedence relations in RCCP problems, usually other activities are involved when fixing a precedence relation between two activities, and thus some small modifications are needed to still be able to apply the BS strategy. Suppose in a node of the branching tree we analyze in each child node a possible repair of the precedence relation violation of activities B_{bj} and B_{sj}. In each child node we modify the deadline (and if necessary the due date) of activity B_{bj} and the release date of activity B_{sj} in such a way, that these activities can no longer overlap (see Section 4.4.1). When there are activities that have the same predecessors as B_{sj}, without affecting the solution we can modify the release date of these activities along with the release date r_{sj} of activity B_{sj}, when these are smaller than the modified release date r_{sj} of activity B_{sj}. When a modification in the release date of one of these other activities is not possible, e.g., due to a minimal duration restriction or a deadline restriction, we may prune the child node. Analogously, when there are activities that have the same successors as B_{bj}, without affecting the solution we can modify the deadline (and if necessary the due date) of these activities along with the deadline $\overline{d}_{bj}$ of activity B_{bj} when $\overline{d}_{bj}$ is smaller than the existing deadline of these activities. Again, when a modification in the deadline of one of these other activities is not possible, we may prune the child node.

6.4 Heuristics

As far as we know, the first heuristics for the type of RCCP problems discussed in this thesis are presented by De Boer and Schutten (1999). They have presented various heuristics for time driven RCCP. Their first heuristic, called $ICPA$ (incremental capacity planning algorithm), starts by sorting all activities in order of non-decreasing deadlines, and subsequently plans the activities in this order in at most two phases. In the first phase each activity is planned in its time window as much as possible, without using nonregular capacity. This is repeated for all unplanned activities. In the second phase the remaining unplanned (parts of) activities are planned in their time window by assigning nonregular capacity. De Boer and Schutten test two variants of this heuristic, which result from different criteria for the order in which the activities are planned. The second heuristic proposed by these authors is an LP-based heuristic. The LP formulation that is used determines the project schedules that minimize the use of nonregular capacity. However, the precedence constraints between activities are ignored. The heuristic iteratively updates the release dates and deadlines of activities in order to repair violated precedence constraints, and re-optimizes the LP model after each change to update the order schedules. De Boer and Schutten present different variants of the heuristic, which result from different procedures to repair the violated precedence

constraints.

Gademann and Schutten (2001) propose several improvement heuristics, and several heuristics that start with infeasible solutions and convert these to feasible solutions. All these heuristics are proposed for time driven RCCP. The improvement heuristics start from a feasible solution, and try to improve the solution by iteratively changing the start and completion times of the activities in the project plans of the solution. The heuristics only differ in the procedure to obtain an initial feasible solution. We tested these heuristics on resource loading problems as well as on RCCP problems. For a detailed description of the heuristic we refer to Section 4.6.3. The other type of heuristics proposed by Gademann and Schutten are basically variants of the LP based heuristics proposed by De Boer and Schutten. They differ slightly with respect to the procedure to repair violated precedence constraints.

Chapter 7

RCCP computational results

Get your facts first,
and then you can distort them
as much as you please.

- Mark Twain (1835-1910)

7.1 Introduction

In this chapter we present the computational results for the branch-and-price methods for RCCP, presented in Chapter 6. As discussed in Chapter 6, we use basically the same integer linear programming model for RCCP problems, as for resource loading. The only algorithmic implications are the generalized precedence relations in RCCP, which have to be accounted for in the pricing algorithm. In Section 6.2 we proposed three pricing algorithms. The first is a dynamic programming based pricing algorithm (see Section 6.2.1), which is basically a generalization of the pricing algorithm for resource loading. The second is based on a mixed integer linear programming model (see Section 6.2.3). Finally, the third is a heuristic pricing algorithm (see Section 6.2.4), which is based on IH. We refer to the three pricing algorithms as PDP, $PMILP$ and PH respectively.

In Chapter 5 we showed that for resource loading problems, the best branching strategy is $Comb2$. It makes no sense to use this branching strategy for the RCCP test instances, because each test instance holds the data for one project, as will be explained in Section 7.2. As a result, if we would use the $Comb2$ method, we could only branch on the project plans of one project in the

$HBS1$ method (which is embedded in $Comb2$). We therefore use the $TBS{+}IH$ branching strategy with one of the three pricing algorithms proposed in Section 6.2.

Recall that in the $TBS{+}IH$ method, we initialize the RLP with the EDD heuristic, or, if EDD does not lead to a feasible solution, we apply column generation on the Phase I RLP to either find a feasible primal solution to the LP relaxation, or to prove that no such solution exists. Subsequently, we apply the rounding heuristics $RH1\,(0.01)$, $RH2\,(0.01)$, $RH3\,(0.05)$, $RH4\,(1)$ and the improvement heuristic IH. IH uses the solution found by $RH4\,(1)$ as initial solution. We use the best solution value found by the heuristics as an upper bound, and use the branching strategy of $TBS{+}IH$. $TBS{+}IH$ branches by repairing violated precedence constraints in each node. We truncate the branching scheme after 30 minutes.

We restrict ourselves to testing only time driven RCCP problems. As a result, in the test instances discussed in this chapter, the projects are not allowed to be tardy. The testing of the algorithms on resource driven RCCP problems is subject of further research.

The outline of this chapter is as follows. First, in Section 7.2 we discuss the test instances that we use for the experiments. In Section 7.3 we analyze the computational results of optimizing the initial LP (i.e., the LP in the root node) with the three pricing algorithms. Subsequently, in Section 7.4 we analyze the computational results of the three pricing methods embedded in the $TBS{+}IH$ algorithm. We compare the three versions of the algorithm mutually, and compare the computational results with results known from the literature.

7.2 Test instances

As compared to the instance generation procedure for resource loading (Chapter 5), the instance generation procedure for RCCP is even more complicated due to the general precedence relations between activities in RCCP. For testing and benchmarking of the algorithms for RCCP we use the set of instances generated by De Boer (1998). De Boer uses a network construction procedure that was developed by Kolisch et al. (1992) to generate random networks. Each test instance holds the data for only one project. The test instances are subdivided into classes, each of which is characterized by the parameters n_j (number of activities in the project), m (the number of resources) and ϕ (the average slack). The average slack is defined as:

$$\phi = \frac{\sum_{b=1}^{n_j} \left(\overline{d}_{bj} - w_{bj} - r_{bj} + 1\right)}{n_j},$$

where $\overline{d}_{bj} - w_{bj} - r_{bj} + 1$ is the slack of activity B_{bj}. The parameter values are displayed in Table 7.1. The test set contains 10 instances for each combination of the parameters, which comes to a total of 450 instances. The length of the planning horizon for these instances varies from 12 to 72 periods.

n_j	10	20	50		
m	3	10	20		
ϕ	2	5	10	15	20

Table 7.1: Parameter values.

Table 7.2 shows the approximate size of the initial RLP for instances with $n_j \in \{10, 20, 50\}$. It shows that the size of the RLP increases considerably with n_j.

n_j	#nonzeros	#variables	#rows
10	3500	400	600
20	6500	900	1200
50	21000	2400	2900

Table 7.2: Size of the initial RLP.

De Boer considers only time driven RCCP problems, with $\delta = 1$. Gademann and Schutten use the same instances for their computational experiments. Although our algorithms work for the case $\delta = 0$ and for resource driven problems, we restrict ourselves to testing the time driven problems for $\delta = 1$. This allows us to compare the computational results of our algorithm with the computational results of the existing algorithms from the literature.

7.3 Optimization of the initial LP

In this section we analyze the computational results of optimizing the LP in the root node. We first analyze the required CPU-time, and the number of times that the LP is solved to optimality using the pricing algorithms PDP (Section 7.3.1), $PMILP$ (Section 7.3.2) and PH (Section 7.3.3). Finally, in Section 7.3.4 we compare the pricing algorithms mutually.

7.3.1 Pricing by dynamic programming

In this section we analyze the computational results of optimizing the initial LP in the root node using the dynamic programming based pricing algorithm (PDP). Before we solve a pricing problem we apply the speed up techniques discussed in Section 6.2.2 to reduce the problem size. The experiments show

that when the problem size increases, it becomes more and more difficult to solve pricing problems with PDP. Particularly when the maximum number of activities in all states (ω) is large, the pricing problem becomes harder or even impossible to solve. Without applying the speed up techniques, we are able to solve pricing problems for projects with up to 20 activities. The speed up techniques not only allow pricing problems of larger projects to be solved, they also significantly reduce the average required CPU-time to solve the LP. Table 7.3 shows the computational results for applying the $TBS+IH$ branching method on 30 RCCP instances with $n_j = 20$, using the PDP pricing algorithm both with and without the speed up techniques. Table 7.3 shows that the average CPU-time required in the root node to optimize the initial LP is much higher when the speed up techniques are not used. Also the CPU-time required for branching is much higher when the speed up techniques are not used. The speed up techniques allow even some pricing problems for projects with 50 activities to be solved. Furthermore, since the speed up techniques reduce the required CPU-time to solve a pricing problem, much more nodes can be explored in the branching scheme in the same time span.

speed up techniques used	avg. CPU-time root node (sec.)	avg. CPU-time branching (sec.)
yes	0.1	12.5
no	18.5	52.0

Table 7.3: Computational results PDP with and without the speed up techniques.

Recall from Section 6.2.1 that the pricing problem uses $O(n_j T^{\omega})$ space, and can be solved in $O(n_j T^{2\omega})$ time. Without applying the speed up techniques, the average ω for the instances with $n_j = 10$ is 3.8. For $n_j = 20$ and $n_j = 50$ the average ω is 6.9 and 18.0 respectively. We also measured the average ω for the problem in the first column generation iteration in the root node, after applying the speed up techniques. In this case, the average ω is 3.7, 6.3 and 14.7 respectively. The speed up techniques thus have more effect when the projects have more activities. When applying the speed up techniques, ω usually decreases in subsequent column generation iterations. Without applying the speed up techniques, ω remains the same in all column generation iterations. For example, when applying the speed up techniques, the average ω during all column generation iterations in the root node is 1.9 for $n_j = 10$, and 2.0 for $n_j = 20$, which is significantly lower than the ω in the first iteration.

In order to prevent computer memory problems that may stop the test-procedure we skip solving a pricing problem when we observe a priori that the state space becomes too large to handle in computer memory. As a result, the column generation scheme may terminate, and the solution found is an

upper bound on the optimal LP solution. An important implication for the branching algorithm is that when a pricing algorithm is skipped, no Lagrangian lower bound is available for that node. In Table 7.4 we present the number of times (out of 450 instances) that a pricing problem is solved successfully in the root node. We categorize the results for n_j, m and ϕ, so each value is the number of instances out of 10 that a pricing problem is solved successfully in a node.

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	10	10	10	10	10	10	10	10	10
5	10	10	10	10	10	10	8	0	0
10	10	10	9	1	2	2	0	0	0
15	9	8	9	5	7	1	0	0	0
20	10	10	9	0	0	0	0	0	0

Table 7.4: The number of times the pricing problem is solved successfully by PDP.

Table 7.4 shows that the larger n_j and ϕ, the more often a pricing problem is skipped in the root node. This may be expected, because the larger n_j, the larger ω generally becomes. As a result, the larger the state space of the pricing problem becomes. The state space of the pricing problem also becomes larger for larger ϕ.

Table 7.5 shows the CPU-time (in seconds) required for solving the LP in the root node with PDP. We categorize the test instances for n_j, m and ϕ. We only consider the instances for which the pricing problem is solved successfully, because the other instances would distort the analysis.

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	0.0	0.0	0.1	0.1	0.2	0.1	0.5	0.6	0.7
5	0.1	0.1	0.1	3.2	0.3	0.2	1.6	-	-
10	0.1	0.6	1.4	3.0	2.0	10.3	-	-	-
15	9.8	3.3	5.6	0.6	3.1	23.5	-	-	-
20	0.4	0.8	5.5	-	-	-	-	-	-

Table 7.5: Average required CPU-time (sec.) for solving the root LP by PDP.

Table 7.5 shows that the average CPU-time required for solving the LP is little for small n_j and ϕ, and generally increases with n_j and ϕ. The average number of column generation iterations required to optimize the LP is 3.2. The maximum number of iterations of all instances is 29.

We conclude that, from a practical point of view, we are able to solve pricing problems for projects of quite large size. However, when there is too much slack,

the pricing problem often becomes too hard to solve. It is questionable whether it makes sense to give a project as much slack as $\phi \geq 10$. It is likely that when the project due date is made tighter, the pricing problem becomes easier to solve. We also note that the pricing problems usually become easier to solve deeper in the branching tree, because more and more constraints are added to the problem. As a result, the activities will have less slack deeper in the branching tree.

7.3.2 Pricing by mixed integer linear programming

In this section we analyze the computational results of optimizing the initial LP in the root node using the mixed integer linear programming based pricing algorithm ($PMILP$). The computational results for $PMILP$ show that when the pricing problem size grows, it becomes harder to solve the pricing problem using this method. When the solver can not solve the pricing problem within one minute, we skip the pricing problem. This only occurs in two instances, both of which concern a project with 50 activities and $\phi = 20$.

Table 7.6 shows the average CPU-time required for solving the LP in the root node with $PMILP$. We again categorize the results for n_j, m and ϕ, and discard the instances for which the pricing problem is skipped.

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	0.2	0.1	0.1	0.5	0.6	0.4	5.4	4.4	4.0
5	0.3	0.3	0.3	2.9	3.3	1.3	27.2	66.0	39.7
10	0.8	1.3	1.0	4.0	7.4	8.7	404.4	452.5	1312.4
15	2.2	2.3	2.1	16.6	30.3	21.6	250.3	5194.2	4381.2
20	2.8	4.5	4.2	31.5	81.2	56.6	571.9	1366.4	18796.2

Table 7.6: Average required CPU-time (sec.) for solving the root LP by $PMILP$.

Table 7.6 shows that the CPU-time generally increases with n_j and ϕ. In particular for the instances that could not be solved by PDP the CPU-times are large. For small n_j and ϕ, $PMILP$ on the one hand requires more CPU-time than PDP. On the other hand, $PMILP$ solves the LP in far more instances than PDP, and for larger n_j and ϕ.

We note that the average number of column generation iterations required to optimize the LP is 10.2. The maximum number of iterations of all instances is 66.

Just as forPDP, for $PMILP$ the pricing problem becomes harder to solve for larger n_j and ϕ. We did not implement the speed up techniques for this pricing algorithm, so some speed up is possible. A faster variant of $PMILP$ can

also be obtained by stopping the integer program solver as soon as a solution has been found that corresponds to a feasible project plan with negative reduced costs. Such a solution is sufficient to continue the column generation scheme, however, it may require more column generation iterations to solve the pricing problem. This method is subject of further research.

7.3.3 Heuristic pricing

In this section we analyze the computational results of optimizing the LP in the root node using the heuristic pricing algorithm (PH). Table 7.7 shows the average required CPU-time (in seconds).

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	0.0	0.1	0.1	0.2	0.2	0.1	0.2	0.2	0.1
5	0.0	0.1	0.1	0.0	0.0	0.0	0.1	0.1	0.1
10	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1	0.1
15	0.0	0.0	0.0	0.1	0.2	0.1	0.1	0.7	0.2
20	0.1	0.1	0.1	0.0	0.1	0.1	0.3	0.2	0.2

Table 7.7: Average required CPU-time (sec.) for solving the root LP by PH.

Table 7.7 shows that the average CPU-time is less than one second for all the classes of instances. This is an advantage of this method. Table 7.8 shows the solution performance of this pricing algorithm. We categorize the instances for n_j, so each row corresponds to 150 instances. Table 7.8 shows that the difference

n_j	Avg. optimal LP solution[a]	Avg. solution found by PH	#Times LP optimal
10	1005.5	1462.4	46
20	980.8	1657.7	22
50	842.6	1623.4	0

[a]This is the average optimal LP solution found by the other pricing algorithms.

Table 7.8: Root LP solution performance of PH.

between the optimal LP solution and the solution found by PH increases with n_j. Furthermore, the number of times that the optimal LP solution is found by PH decreases with n_j. Both effects are similar for parameter ϕ. The average number of column generation iterations before termination of the column generation scheme is 1.6.

We conclude that, although PH is fast, its solution performance is unsatisfactory. We recommend it only to be used in cases when both PDP and

$PMILP$ can not solve a pricing problem.

7.3.4 Comparison of pricing algorithms

In the previous sections we showed that, when solving the LP for small projects, $PMILP$ on the one hand requires more CPU-time than PDP. On the other hand, $PMILP$ solves the LP in far more instances than PDP. $PMILP$ is therefore favorable for large instances, and PDP is favorable for small instances. In two instances the pricing problem is so large that it can not be solved by $PMILP$.

PH never skips a pricing problem, but it may not find a project plan with negative reduced costs when such a project plan exists. In Section 7.3.3 we showed that only in 68 of the 450 instances it finds the optimal LP solution. Even though this number is small, since its execution time is little, an advantage of using PH is that it may allow many more nodes to be explored in a branch-and-price algorithm, and may thus eventually lead to a better ILP solution. However, an important implication of using PH in a branch-and-price algorithm is that the Lagrangian lower bound is no longer valid. Hence, fewer nodes may be fathomed in the branching scheme, which may cancel out the advantage.

A hybrid approach, in which we would use PDP for small problems, $PMILP$ for larger problems, and PH for problems that $PMILP$ can not solve, may be more efficient. We restrict ourselves, however, to using one pricing algorithm per method. The testing of a hybrid pricing approach is subject of further research.

7.4 Computational results of the branch-and-price methods

In this section we analyze the computational results for the branch-and-price methods. As discussed in Section 7.1, we use the three pricing algorithms PDP, $PMILP$ and PH in the $TBS{+}IH$ branching method. We refer to the three branch-and-price methods as BPD, BPM and BPH respectively. We also test a variant of BPM in which we add 3 project plans to the column pool before branching. We refer to this branch-and-price method as $BPM{+}$. The 3 project plans are constructed as follows. One project plan is added in which all activities are allowed to be produced for w_{bj} periods (i.e., as long as their minimal duration), and as early as possible. Only the activities without successors are allowed to be produced until the project due date. A similar project plan is added, in which all activities are allowed to be produced for w_{bj} periods, and as late as possible. A third project plan is added, in which the

slack is divided evenly over the activities on the critical path. By adding these 3 project plans we try to improve the performance of the rounding heuristics.

Before we start branching we execute the rounding heuristics $RH1\,(0.01)$, $RH2\,(0.01)$, $RH3\,(0.05)$, $RH4\,(1)$ and the improvement heuristic IH. Since we execute the heuristics in the root node of each of the 4 branch-and-price methods (each of which may have a different column pool), we get at most 4 solutions for these heuristics. For all heuristics, the best average solution is obtained in the $BPM+$ method. In this method, the number of project plans in the column pool is the largest at the moment the heuristics are applied. Table 7.9 shows the performance of the heuristics in $BPM+$. We compare the solutions with the best known heuristic solutions from the literature. We refer to the solution data of the best method of De Boer (1998) as DB. We refer to the solution data of the best method of Gademann and Schutten (2001) as GS. The bottom row of Table 7.9 shows the best known solutions from all the methods from the literature. There are no proven optimal solutions available for these test instances from the literature.

	Avg. solution			Avg. CPU-time (sec.)			# Best	# Unique best	# Proven optimal
	n_j=10	20	50	10	20	50			
$RH1\,(0.01)$	1387.7	1612.8	1632.6	0.1	0.6	8.2	58	0	46
$RH2\,(0.01)$	1387.8	1518.3	1501.9	0.1	0.7	11.8	60	0	42
$RH3\,(0.05)$	1403.7	1520.6	1505.7	3.1	24.3	720.2	65	0	45
$RH4\,(1)$	1379.0	1478.0	1409.5	0.1	0.7	13.2	63	0	42
IH	1316.2	1398.2	1251.7	0.6	2.5	132.2	133	11	83
DB	1435.9	1552.3	1423.1	3.0	8.8	49.0	5	0	5
GS	1305.2	1366.2	1177.4	13.4	104.8	1851.7	423	317	118
best known	1297.9	1360.2	1173.8	-	-	-	-	-	126

Table 7.9: Solution performance of the heuristics.

Table 7.9 shows that $RH4\,(1)$ is the best rounding heuristic. The execution times of the rounding heuristics are usually small, because the test instances contain only one project. As a result, the rounding heuristics converge quickly. $RH3\,(0.05)$ requires significantly more CPU-time when n_j increases, because it solves a pricing problem in each rounding iteration.

On average, IH improves the solution found by $RH4\,(1)$ in approximately half of the instances. The rounding heuristics and IH find the same or a better solution than the best known solutions from the literature in 137 instances, and find a proven (by one of the exact branch-and-price methods) optimal solution in 87 instances. The best heuristic is GS. On average it requires much CPU-time, but finds the best solution in the majority of the instances. From a practical point of view, IH may be preferred, because on the one hand it requires much less CPU-time, while on the other hand it finds solution values

that on average require at most 74.3 (for $n_j = 50$) more hours of nonregular capacity.

Tables 7.10 and 7.11 show the computational results for the 4 branch-and-price methods. In Table 7.11 we also show the solutions of GS, and best known solutions from the literature. For the instances where a branch-and-price algorithm could not solve the LP in the root node, we obtain a solution by applying IH on the initial RLP solution (i.e., the EDD solution). This allows us to compare the solution performance of the algorithms in all instances.

	# Solutions found	#Truncated			Avg. #nodes[a] (×1000)			Avg. tree depth		
		n_j=10	20	50	10	20	50	10	20	50
BPD	450	76	114	142	46.7	34.7	10.6	7.2	14.2	32.0
BPM	448	76	119	149	21.9	9.3	1.3	7.2	13.4	18.4
$BPM+$	418	78	119	117	16.2	7.1	2.0	7.1	13.0	17.5
BPH	450	97	123	150	34.7	22.8	8.6	7.4	14.5	29.3

[a]These results concern the instances that were not truncated.

Table 7.10: Test results of the branch-and-price methods (1).

	Avg. solution			Avg. CPU-time (sec.)[a]			# Best	# Unique best	# Unique worst	# Proven optimal
	n_j=10	20	50	10	20	50				
BPD	1317.2	1518.9	1532.2	133.2	170.6	632.8	166	6	83	121
BPM	1318.8	1431.2	1252.1	229.1	166.9	444.1	145	3	12	111
$BPM+$	1309.7	1397.6	1249.1	226.1	166.1	363.5	172	11	0	117
BPH	1365.6	1541.4	1503.6	167.2	117.5	-	106	0	181	96
GS	1305.2	1366.2	1177.4	13.4	104.8	1851.7	410	253	0	118
best known	1297.9	1360.2	1173.8	-	-	-	-	-	-	126

[a]These results concern the instances that were not truncated.

Table 7.11: Test results of the branch-and-price methods (2).

Table 7.10 shows that $BPM+$ finds a solution in fewer instances than BPM. It also generally explores fewer nodes than BPM. On the other hand, the solution performance of $BPM+$ is better than that of BPM.

BPH is truncated more often than the other methods. Only for $n_j = 50$, its solution performance is better than that of BPD. For $n_j \in \{10, 20\}$ BPH is clearly outperformed by the other methods. BPH usually explores more nodes, however, it does not use Lagrangian relaxation to determine lower bounds.

For instances with $n_j = 10$ the fastest method is BPD. The solution performance of BPD on these is only outperformed by $BPM+$. For instances

with $n_j = 20$, $BPM+$ also has the best solution performance. For instances with $n_j = 50$, BPM has the best solution performance.

For 126 out of 450 instances optimality could be proven by BPD, BPM and $BPM+$. In 7 instances the best known solution from the literature is improved, and in 174 instances the best solution of the branching methods is the same as the best known solution from the literature. Table 7.12 shows the number of instances that are solved to optimality by $BPM+$. We categorized the results for n_j, m and ϕ, so each number corresponds to the number of times out of 10 that the instances are solved to optimality.

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	10	10	10	10	10	10	6	2	0
5	10	9	10	6	0	0	0	0	0
10	9	3	4	0	0	0	0	0	0
15	1	0	2	0	0	0	0	0	0
20	1	1	1	0	0	0	0	0	0

Table 7.12: Number of times that an optimal solution is found by $BPM+$.

Table 7.12 shows that instances with small n_j and ϕ can often be solved to optimality. Instances with large n_j and ϕ can almost never be solved to optimality.

Considering the solution performance, $BPM+$ is the best branch-and-price method, and GS is the best overall method. Table 7.13 compares the absolute difference between the solution found by $BPM+$ and the solution found by GS. A positive number means that the average solution found by $BPM+$ is larger than that of GS. We categorize the results for n_j, m and ϕ.

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	0.0	0.0	0.0	0.0	0.0	0.0	0.1	−0.8	4.9
5	0.0	0.0	−1.2	1.3	9.5	2.5	9.6	26.5	19.8
10	1.5	1.6	−0.4	8.9	32.9	43.6	22.6	48.0	79.5
15	13.7	13.4	12.6	45.9	32.2	71.7	40.3	109.9	121.8
20	−2.4	7.7	21.6	41.7	103.4	78.5	89.1	223.1	279.8

Table 7.13: Absolute difference between the average solutions of $BPM+$ and GS.

Table 7.13 shows that $BPM+$ and GS perform comparably for small n_j or for small ϕ. The larger n_j and ϕ, the more GS outperforms $BPM+$.

In Table 7.14 we show the number of times that $BPM+$ and GS find the best solution. We categorize the results for n_j, m and ϕ. For example "6,8" in

Table 7.14 means that $BPM+$ finds the best solution in 6 out of 10 instances, and GS in 8 instances out of 10.

n_j	10	10	10	20	20	20	50	50	50
$\phi \mid m$	3	10	20	3	10	20	3	10	20
2	10,10	10,10	10,10	10,10	9,10	10,10	9,8	7,9	6,10
5	10,10	9,10	10,9	5,8	2,9	3,7	0,10	0,10	3,8
10	6,8	8,8	6,8	2,9	1,10	1,9	0,10	0,10	1,9
15	1,8	4,10	5,7	0,10	1,9	2,8	0,10	0,10	0,10
20	1,7	4,6	4,6	1,9	0,10	1,9	0,10	0,10	0,10

Table 7.14: Number of times that $BPM+$,GS find the best solution.

Just as Table 7.13, also Table 7.14 shows that $BPM+$ and GS perform comparably for small n_j and small ϕ. The larger n_j and ϕ, the more GS outperforms $BPM+$.

7.5 Conclusions

The heuristic pricing method PH performs unsatisfactorily. Research into alternative heuristic pricing methods is therefore recommended. A hybrid pricing approach, in which we would use PDP for small problems, $PMILP$ for larger problems, and PH for problems that $PMILP$ can not solve, may be the most efficient pricing method. The testing of a hybrid pricing approach is subject of further research.

The computational results show that the branch-and-price methods are suitable for solving RCCP problems with $n_j \leq 20$, and with $\phi \leq 10$. For larger RCCP problems, the branch-and-price method $BPM+$ performs satisfactorily, but heuristics like GS are preferred. With respect to the solution performance, the best branch-and-price method is $BPM+$. The absolute difference between the average solution values of $BPM+$ and GS is usually less than 50, which is quite small from a practical point of view.

The speed up techniques have not been implemented for $PMILP$. Hence, the performance of $BPM+$ may be improved further by using the speed up techniques to split up the ILP model of the pricing problem whenever this is possible. This extension is subject of further research.

Chapter 8

Epilogue

There never seems to be enough time to do the things
you want to do once you find them.

- Jim Croce (1943-1973)

The main objective of the research described in this thesis is to develop models for resource loading problems with finite capacity constraints and complex precedence and routing constraints, and, more importantly, to propose exact and heuristic solution methods to solve such models. The models and techniques used were highly motivated by recent insights regarding the use of advanced combinatorial methods for these types of problems. Another motivation of this research originates from shop floor scheduling research. Shop floor scheduling algorithms are rigid with respect to resource capacity. Without the use of resource loading tools to determine the required resource levels and the (possible) size of the workload to be scheduled, shop floor scheduling problems may become too hard to solve and solutions may become unacceptable.

We propose resource loading methods that support shop floor scheduling by determining the resource capacity levels available to the scheduling level, as well as important milestones for the orders and jobs in the scheduling problem, such as (internal) release and due dates. Resource loading can also be used to support order acceptance. It can be used as an instrument to make trade-offs between lead time and due date performance on the one hand, and nonregular capacity levels on the other hand.

In this final chapter we give an overview of our results. In Section 8.1 we give a summary of this thesis. We conclude this thesis in Section 8.2 with suggestions for further research.

8.1 Summary

The main contribution of this thesis is twofold. First, we propose a modeling approach that offers a generic framework to formulate various types of resource loading and RCCP problems as ILP models. Second, we propose various algorithms that can solve problems of reasonable size, i.e., typically encountered in practice, to optimality.

We position resource loading between strategical capacity planning (aggregate planning) and operational capacity planning (scheduling) as a tactical capacity planning problem. Resource loading has been rather underexposed both in the literature and in practice. As a tactical instrument it benefits the flexibility of the entire production system for various reasons. It serves as a tool in the customer order processing stage, which, in make-to-order manufacturing environments, is typically characterized by much uncertainty. On the demand side there is uncertainty as to what orders can eventually be acquired, while furthermore order characteristics are uncertain or at best partly known. On the supply side there is uncertainty in the availability of the resources. In these situations, a resource loading tool can be used to match production capacity and customer demand, while minimizing the cost of customer order tardiness and the use of nonregular capacity. Resource loading analyses can thus be used to accept/reject orders, or to quote reliable due dates. Resource loading can also serve as a tool to define realistic constraints for the underlying scheduling problem. The resource capacity levels and important milestones (such as release and due dates) are usually supposed to be fixed in scheduling. Resource loading can detect when and where difficulties will occur in scheduling at an early stage, and allocate orders or order parts more efficiently, and, if necessary, properly adjust resource capacity levels (by assigning nonregular capacity) and/or milestones.

In this thesis we propose a deterministic approach for modeling and solving resource loading problems. In order to smooth out the aforementioned uncertainty in resource loading problems we formulate resource loading at a higher level of aggregation than scheduling problems (i.e., the tactical level vs. the operational level). In resource loading problems we distinguish (customer) orders that consist of jobs. Jobs are in fact work packages at a higher level of aggregation. In the underlying shop floor scheduling problem, jobs may be further disaggregated into operations or tasks.

The difficulty of formulating the resource loading problem as an integer linear programming model is that modeling precedence relations is not straightforward, and the resulting formulations are often extremely hard to solve. We propose a modeling approach that offers a generic framework for modeling various resource loading problems. The proposed model can handle a large variety of practical aspects, such as generalized precedence constraints, various forms of capacity flexibility, tardiness penalties, and minimal duration constraints. The

model can handle resource driven resource loading and time driven resource loading simultaneously, which allows making trade-off analyses between due date performance on the one hand, and nonregular capacity levels on the other hand. In this modeling approach we make a distinction between order plans and order schedules. Order plans indicate in which periods a job of an order is allowed to be processed. Order schedules indicate which (part of the) jobs of the order are actually processed in each period. We propose a mixed-integer linear programming (MILP) model of the resource loading problem with an exponential number of integer variables. A relatively small and fixed part of these variables determine the required nonregular capacity usage per resource per period. The remaining variables are binary variables that correspond to selecting an order plan for an order. The order plans are thus columns of the coefficient matrix of the model, which are feasible with respect to precedence constraints and order release and due date constraints. The MILP model selects precisely one order plan per order, and determines order schedules that are consistent with these order plans. The model determines the nonregular capacity usage from the order schedules.

The precedence relations and release and due date constraints thus do not have to be applied to the order schedules by the model, since they are embedded in the order plans. However, since there are exponentially many feasible order plans, an explicit model of a problem of regular size is impossible to formulate and solve. We therefore propose various exact and heuristic solution methods, which are all based on first solving the linear programming (LP) relaxation of this formulation by column generation. The pricing problem comprises the determination of feasible order plans with negative reduced costs. The idea of a column generation scheme is that only a small set of variables are required to determine the optimal solution of the LP. It starts from a restricted LP formulation (RLP), which has at least one order plan per order. After each RLP optimization, order plans with negative reduced costs are added to the RLP. The column generation scheme terminates when no order plans with negative reduced costs exist. The optimal solution of the LP is then found.

Clearly, if the optimal solution of the linear programming relaxation happens to be integral, we have found an optimal solution for the resource loading problem. Otherwise, we apply a branch-and-price algorithm to determine an optimal solution. We propose various exact and heuristic branching strategies. Furthermore, we propose various approximation techniques that are based on the column generation approach, such as approximation algorithms that proceed by judiciously rounding the linear programming solution to obtain a feasible solution for the original problem.

Computational experiments with the resource loading methods show that large resource loading problems with a planning horizon of up to 15 weeks and 5 machine groups can usually be solved to optimality. For larger problems, the branch-and-bound methods usually have to be truncated. Various sensitivity

analyses show that adding planning flexibility in some cases makes cases easier to solve, and in other cases makes it harder to prove optimality. The best resource loading method is a combination of two branching strategies. This (exact) method generally outperforms all approximation methods.

In resource loading problems we assume linear precedence relations. We also propose extensions of the algorithms that are able to deal with generalized precedence relations. This allows us to use the same model and solution methods to solve Multi-Project Rough-Cut Capacity Planning (RCCP) problems, for which some heuristics have already been proposed in the literature. The main algorithmic implication of the generalized precedence constraints is the generalization of the pricing algorithm. The pricing problem becomes much harder to solve, especially when the project size increases. We propose three different pricing algorithms, so that pricing problems of many sizes can be solved. We propose branch-and-bound algorithms that use one of these pricing algorithms to solve the linear program. We also propose approximation techniques, such as the rounding heuristics that we proposed for resource loading problems, and an improvement heuristic, which tries to improve an existing feasible solution.

Computational experiments with the branch-and-bound methods show that RCCP problems for projects of reasonable size can be solved to optimality. For larger problems, the branch-and-bound methods compete with the heuristics from the literature. For RCCP problems with very large projects solving the pricing problem often becomes too computational intensive. As a result, for large RCCP problems the branch-and-bound methods are outperformed by the heuristics from the literature. We note that, from a practical point of view, it is questionable whether it makes sense to solve such large problems to optimality, since information regarding resource availability and project characteristics are usually uncertain in the long term. Solving RCCP problems with a long planning horizon is thus more a mathematical challenge.

8.2 Further research

An important question remains as to what extent a resource loading plan is consistent with the next planning stage, i.e., the scheduling stage. Resource loading tools give an indication of the resource usage and important milestones of customer orders at a more aggregated level. As with other planning problems, the data for a resource loading problem (especially problems with a long planning horizon) will become less accurate towards the end of the planning horizon. Disturbances in the production process may occur, or, more likely, changes in the demand requirements may occur (e.g., rush orders may arrive). Since long-term orders usually have less detailed (i.e., more aggregated) specifications than short-term orders, ideally they are not treated uniformly by the resource loading. As a result, resource loading preferably has a planning

horizon with various non-decreasing levels of aggregation, corresponding to the actual capacity flexibility. In such an approach we could, for example, consider periods of a day in the first two weeks of the planning horizon, periods of a week in the subsequent weeks, and perhaps even periods of a month after that. This would allow a more detailed loading for orders in the short term, and a rough planning in the long term. Not only could we then distinguish orders of various levels of aggregation; this would also bring the resource loading problem closer to the underlying scheduling problem. In this thesis we make no such distinction of aggregation levels in the planning horizon, since we considered planning horizons that consist of periods of equal size (i.e., a week). As a result, we would ignore any available data for the short term that is at a lower level of aggregation. It is likely that this will affect the consistency of the resource loading plan and the scheduling stage. This would be avoided if a resource loading method would be used that uses a more detailed loading in the short term, and non-decreasing levels of aggregation for the remainder of the planning horizon. Since there exist no proper resource loading methods even within our problem definition, this is perhaps the logical next step in further research.

An approach in which the size of the planning periods increases during the planning horizon can easily be adapted by the branch-and-price techniques proposed in this thesis. In fact the only implication is that the planning horizon must be extended, and redefined. When, for example, we would consider periods of a day in the first week of the planning horizon, and periods of a week in the subsequent 10 weeks, we could redefine the planning horizon ($t = 0, \ldots, 10$ weeks) as follows: $t = 0, \ldots, 6$ days, $t = 7, \ldots 16$ weeks. An important implication of this extension is that the size of the entire model increases, and may become harder to solve. Whether this extension leads to a higher consistency between resource loading and scheduling is subject of further research. An alternative approach that may be more appropriate for this extension is to split up the planning horizon in parts that partly overlap, and that have an appropriate level of aggregation. Instead of solving the entire problem we could solve the problem in each part of the planning horizon, and connect the overlapping plans. The question as to whether this leads to a better solution than when we would solve the problem in its entirety, is subject of further research.

In this research we primarily deal with resource loading problems from the supply side (i.e., the demand is given): we consider the timing and coordination of the resources in order to load a predetermined set of orders. In practice, especially when the planning horizon is long, it is often possible to reject or postpone orders, possibly with additional costs. Further research is therefore recommended into resource loading tools that support order acceptance or postponement decisions.

An ideal environment to test resource loading tools would be a simulation model of an order processing environment. A simulation model allows a pro-

duction system to be studied with a long time frame in compressed time, or alternatively to study the detailed workings of a system in expanded time (Law and Kelton, 1991). It can be adapted easily, which allows experiments with different factory layouts and operating policies. We could model the production system that we used throughout this thesis for resource loading problems (see Section 1.3.1). The simulation model could generate customer order arrivals at irregular intervals. These orders may have various characteristics, such as size, several possible delivery dates (and corresponding prices), etc. A resource loading tool can be embedded to aid a customer order processor to determine possible delivery dates, and corresponding capacity usage, and (nonregular) capacity costs. Based on some decision criteria, such as costs or profit, the customer order processor may then accept or reject a customer order. If the order is accepted, and the delivery date is agreed upon, the same resource loading tool may be used to load the new order, and adjust the capacity levels if necessary. Various online planning aspects must then be dealt with, such as the question how to deal with existing orders after a new order is accepted, i.e., whether they should be fixed or reloaded. A scheduling tool can also be embedded in the simulation model to determine the short-term capacity plans. This allows the consistency of the resource loading plans with the scheduling plans to be studied.

The computational results for the three pricing algorithms for RCCP in Chapter 7 show that each pricing algorithm is suitable for pricing problems of a specific size. Since, because of the speed up techniques, the size of the pricing problems vary significantly during the branch-and-price scheme, we believe that a hybrid pricing approach may be more efficient. The testing of a hybrid pricing approach for RCCP problems, which selects one of the three pricing algorithms dependent on the pricing problem size, is subject of further research.

In this thesis we studied resource loading problems in a make-to-order manufacturing environment. We argued in Chapter 1 that many manufacturing companies that produce non-standard items are faced with the problem that in the order processing stage, order characteristics are still uncertain. This uncertainty can be even more eminent for engineer-to-order manufacturers. These manufacturers even perform product design activities to order. Giebels (2000) proposes a stochastic resource loading model, which uses stochastic variables to model the start and processing times. The optimization of such a stochastic model for resource loading, perhaps using some of the techniques proposed in this thesis, is subject of further research.

Bibliography

Aarts, E.H.L. and J.K. Lenstra (Eds.) (1997). *Local Search in Combinatorial Optimization*. John Wiley & Sons, Chichester, UK.

Anthony, R.N. (1965). *Planning and Control Systems: A Framework for Analysis*. Harvard University Graduate School Of Business, Boston, Mass.

Bahl, H.C. and S. Zionts (1987). Multi-item scheduling by Benders' decomposition. *Journal of the Operational Research Society 38*(12), 1141–1148.

Baker, K.R. (1974). *Introduction to Sequencing and Scheduling*. Wiley, New York.

Baker, K.R. (1993). *Requirements Planning*. Logistics of Production and Inventory. North-Holland, Amsterdam.

Barnhart, C., E.L. Johnson, G.L. Nemhauser, M.W.P. Savelsbergh, and P. Vance (1998). Branch-and-price: Column generation for solving huge integer programs. *Operations Research 46*, 316–329.

Beasley, J.E. (1996). *Advances in Linear and Integer Programming*. Oxford Lecture Series in Mathematics and its Applications. Clarendon Press, Oxford.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press.

Bertrand, J.W.M. and J.C. Wortmann (1981). *Production Control and Information Systems for Component Manufacturing Shops*. Elsevier.

Bitran, G.R. and D. Tirupati (1993). *Hierarchical Production Planning*, Volume 4 of *Logistics of Production and Inventory, Handbooks in Operations Research and Management Science*. North-Holland, Amsterdam.

Bixby, R.E., J.W. Gregory, I.J. Lustig, R.E. Marsten, and D.F. Shanno (1992). Very large-scale linear programming: A case study in combining interior point and simplex methods. *Operations Research 40*, 885–897.

Borra, T. (2000). Heuristic methods for the resource loading problem. Master's thesis, University of Twente, Faculty of Applied Mathematics.

Brucker, P. (1995). *Scheduling Algorithms*. Springer-Verlag, Berlin.

Burbidge, J.L. (1979). *Group Technology in the Engineering Industry*. Mechanical Engineering Publications LTD.

Buzacott, J.A. and J.G. Shanthikumar (1993). *Stochastic Models of Manufacturing Systems*. Prentice Hall, Englewood Cliffs.

Cay, F. and C. Chassapis (1997). An IT view on perspectives of computer aided process planning research. *Computers in Industry 34*, 307–337.

Dantzig, G.B. and P. Wolfe (1960). Decomposition principles for linear programming. *Operations Research 8*, 101–111.

Darlington, J. and C. Moar (1996). *MRP Rest in Peace*. Management Accounting (UK).

De Boer, R. (1998). *Resource-Constrained Multi-Project Management - A Hierarchical Decision Support System*. Ph. D. thesis, University Of Twente.

De Boer, R. and J.M.J. Schutten (1999). Multi-project rough-cut capacity planning. In *Flexible Automation and Intelligent Manufacturing: Proceedings of the Ninth International FAIM Conference*, pp. 631–644. Center for Economic Research (CentER): Begell House Inc., New York.

Desrochers, M., J. Desrosiers, and M. Solomon (1992). A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research 40*, 342–354.

Desrochers, M. and F. Soumis (1989). A column generation approach to the urban transit crew scheduling problem. *Transportation Science 23*, 1–13.

Desrosiers, J., F. Soumis, and M. Desrochers (1984). Routing with time windows by column generation. *Networks 14*, 545–565.

Dreyfus, S.E. and A.M. Law (1977). *The Art and Theory of Dynamic Programming*. Academic Press, Inc. Ltd.

Freling, R. (1997). *Models and Techniques for Integrating Vehicle and Crew Scheduling*. Ph. D. thesis, Erasmus University Rotterdam.

Gademann, A.J.R.M. and J.M.J. Schutten (2001). Linear programming based heuristics for multi-project capacity planning. *Working paper TBK01W-004 OMST-002, University of Twente*.

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

Geoffrion, A.M. (1974). Lagrangian relaxation for integer programming. *Mathematical Programming Study 2*, 82–114.

Giebels, M.M.T. (2000). *EtoPlan: A Concept for Concurrent Manufacturing Planning and Control*. Ph. D. thesis, University of Twente.

Giebels, M.M.T., E.W. Hans, M.P.H. Gerritsen, and H.J.J. Kals (2000, June). Capacity planning for make- or engineer-to-order manufacturing; the importance of order classification. Stockholm. 33rd CIRP Manufacturing Systems Conference.

Gilmore, P.C. and R.E. Gomory (1961). A linear programming approach to the cutting stock problem. *Operations Research 9*, 849–859.

Gilmore, P.C. and R.E. Gomory (1963). A linear programming approach to the cutting stock problem: Part II. *Operations Research 11*, 863–888.

Goldratt, E.M. (1988). Computerized shop floor scheduling. *International Journal of Production Research 26*(3), 433–455.

Graves, S.C. (1982). Using Lagrangian techniques to solve hierarchical production planning problems. *Management Science 28*(3).

Hax, A.C. and H.C. Meal (1975). *Hierarchical Integration of Production Planning and Scheduling*. Studies in Management Sciences, Vol. 1, Logistics. North Holland TIMS.

Held, M. and R.M. Karp (1971). The traveling salesman problem and minimum spanning trees: Part II. *Mathematical Programming 1*, 6–25.

Hendry, L.C., B.G. Kingsman, and P. Cheung (1998). The effect of workload control (WLC) on performance in make-to-order companies. *Journal of Operations Management 16*, 63–75.

Hiriart-Urruty, J.B. and C. Lemarechal (1993). *Convex Analysis and Minimization Algorithms*, Volume I and II. Springer Verlag, Berlin.

Hoffman, K. and M. Padberg (1985). LP-based combinatorial problem solving. *Annals of Operations Research 4*, 145–194.

Hopp, W.J. and M.L. Spearman (1996). *Factory Physics - Foundations of Manufacturing Management*. IRWIN, Boston.

Infanger, G. (1994). *Planning under Uncertainty*. The Scientific Press.

Johnson, E.L., G.L. Nemhauser, and M.W.P. Savelsbergh (1998). Progress in linear programming based branch-and-bound algorithms: An exposition. *submitted to INFORMS J. on Computing*. Available for download at Savelsbergh's webpage: http://tli.isye.gatech.edu/faculty/savelsbergh.cfm.

Kall, P. and S.W. Wallace (1994). *Stochastic Programming*. Wiley, New York.

Karmarkar, U.S. (1993). *Manufacturing Lead Times, Order Release and Capacity Loading*. Logistics of Production and Inventory (Chapter 6). North-Holland, Amsterdam.

Karni, R. (1982). Capacity requirements planning - a systematization. *International Journal of Production Research 20*(6), 715–739.

Kolisch, R. (1995). *Project Scheduling under Resource Constraints: Efficient Heuristics for Several Problem Classes*. Physica-Verlag, Heidelberg.

Kolisch, R., A. Sprecher, and A. Drexl (1992). Characterization and generation of a general class of resource-constrained project scheduling problems: Easy and hard instances. Technical Report 301, Manuskripte aus den Instituten fuer Betriebswirtschaftslehre.

Law, A.M. and W.D. Kelton (1991). *Simulation Modeling and Analysis*. McGraw-Hill Inc, New York.

Lenstra, J.K. and A.H.G. Rinnooy Kan (1979). Computational complexity of discrete optimization problems. *Annals of Discrete Mathematics 4*, 121–140.

Lock, D. (1988). *Project Management* (4th ed.). Gower House.

Möhring, R.H. (1984). Minimizing costs of resource requirements in project networks subject to a fixed completion time. *Operations Research 32*, 89–120.

Morton, T.E. and D.W. Pentico (1993). *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management*. Wiley, New York.

Negenman, E. (2000). *Material Coordination Under Capacity Constraints*. Ph. D. thesis, Eindhoven University of Technology.

Nemhauser, G.L., M.W.P. Savelsbergh, and G.S. Sigismondi (1994). MINTO, a Mixed INTeger Optimizer. *Operations Research Letters 15*, 47–58.

Nemhauser, G.L. and L.A. Wolsey (1988). *Integer and Combinatorial Optimization*. Wiley, New York.

Orlicky, J. (1975). *Material Requirements Planning*. McGraw-Hill, London.

Pinedo, M. (1995). *Scheduling: Theory, Algorithms, and Systems*. Prentice Hall, Englewood Cliffs.

Ross, S. (1983). *Introduction to Stochastic Dynamic Programming*. Academic Press.

Savelsbergh, M.W.P. (1994). Preprocessing and probing for mixed integer programming problems. *ORSA Journal on Computing 6*, 445–454.

Savelsbergh, M.W.P. (1997). A branch-and-price algorithm for the generalized assignment problem. *Operations Research 45*, 831–841.

Schneeweiss, C. (1995). Hierarchical structures in organizations: A conceptual framework. *European Journal of Operational Research 86*, 4–31.

Schutten, J.M.J. (1998). Practical job shop scheduling. *Annals of O.R. 83*, 161–177.

Sennott, L.I. (1999). *Stochastic Dynamic Programming and the Control of Queueing Systems*. Wiley, New York.

Shapiro, J.F. (1993). *Mathematical Programming Models and Methods for Production Planning and Scheduling*. Logistics of Production and Inventory. North-Holland, Amsterdam.

Sheng, P. and M. Srinivasan (1996). Hierarchical part planning strategy for environmentally conscious machining. *Annals of the CIRP 45*(1), 455–460.

Silver, E.A., D.F. Pyke, and R. Peterson (1998). *Inventory Management and Production Planning and Scheduling* (3rd ed.). Wiley, New York.

Snoep, M. (1995). Produktiebesturing en -beheersing in Machinefabriek Noord-Oost Nederland (in Dutch). Master's thesis, University of Twente, Faculty of Mechanical Engineering.

Stalk Jr., G. and T.M. Hout (1988). Competing against time: How time-based competition is reshaping global markets. *Harvard Business Review*, 41–51.

Suri, R. (1994, November). Common misconceptions and blunders in implementing quick response manufacturing. *SME AUTOFACT '94 Conference*.

Ten Kate, H.A. (1995). *Order Acceptance and Production Control*. Ph. D. thesis, University of Groningen.

Van Assen, M.F. (1996). Produktiebesturing van semi-autonome produktieteams bij Urenco Nederland B.V. (in dutch). Master's thesis, University of Twente, Faculty of Mechanical Engineering.

Van Assen, M.F., E.W. Hans, and S.L. Van de Velde (1999). An agile planning and control framework for customer-order driven discrete parts manufacturing environments. *International Journal of Agile Management Systems 2*(1).

Van den Akker, M., H. Hoogeveen, and S.L. Van de Velde (2000). Combining column generation and Lagrangian relaxation. *submitted to INFORMS Journal on Computing*.

Van Houten, F.J.A.M. (1991). *PART: A Computer Aided Process Planning System*. Ph. D. thesis, University of Twente.

Vance, P.H., C. Barnhard, E.L. Johnson, and G.L. Nemhauser (1994). Solving binary cutting stock problems by column generation and branch-and-bound. *Computational optimization and applications 3*, 111–130.

Vanderbeck, F. (2000). On Dantzig-Wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Operations Research 48*(1), 111–128.

Vanderbeck, F. and L.A. Wolsey (1996). An exact algorithm for IP column generation. *Operations Research Letters 19*, 151–159.

Vollmann, T.E., W.L. Berry, and D.C. Whybark (1988). *Manufacturing Planning and Control Systems* (2nd ed.). IRWIN, Homewood, Illinois.

Wiendahl, H.-P. (1995). *Load-Oriented Manufacturing Control*. Springer Verlag, Berlin.

Wight, O. (1981). *MRP II: Unlocking America's Productivity Potential*. CBI Publishing, Boston.

Winston, W.L. (1993). *Operations Research: Applications and Algorithms* (3rd ed.). International Thomson Publishing.

Wolsey, L.A. (1998). *Integer Programming*. Wiley, New York.

Zäpfel, G. and H. Missbauer (1993). New concepts for production planning and control. *European Journal of Operations Research 67*, 297–320.

Zijm, W.H.M. (2000). Towards intelligent manufacturing planning and control systems. *OR Spektrum 22*, 313–345.

Zionts, S. (1974). *Linear and Integer Programming*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

Appendix A

Glossary of symbols

A.1 Model input parameters

n	the number of orders (index $j = 1, \ldots, n$).
m	the number of machine groups (index $i = 1, \ldots, m$).
n_j	the number of jobs of order J_j (index $b = 1, \ldots, n_j$).
T	the end of the planning horizon (index $t = 0, \ldots, T$ weeks).
B_{bj}	the b-th ($b = 1, \ldots, n_j$) job of order J_j.
μ_{bj}	the machine group on which job B_{bj} must be processed.
v_{bji}	the fraction of B_{bj} that must be performed on resource M_i (RCCP).
p_{bj}	the processing time of job B_{bj}.
Π	the set of all feasible order plans.
Π_j	the set of feasible order plans for J_j (index $\pi = 1, \ldots, \Pi_j$).
$a_{j\pi}$	π-th order plan for order J_j, with elements $a_{bjt\pi}$.
$\overline{mc}_{it}$	total regular capacity of machine group M_i in week t.
mc_{it}	capacity of machine group M_i in week t in regular operator time.
c_t	regular operator capacity in week t.
o_t	upper bound on the number of overtime hours in week t.
h_t	upper bound on the number of hirable hours in week t.
s_{it}	upper bound on the number of production hours that can be outsourced in week t.
$\overline{o}_t$	cost of overtime per hour.
$\overline{h}_t$	cost of hiring one extra operator per hour.
$\overline{s}_t$	cost of outsourcing one hour of work.

continued on next page

continued from previous page

w_{bj}	minimal duration (weeks) of job B_{bj} ($w_{bj} \geq 1$).
δ	minimum time lag (0 or 1 week) between adjacent jobs, to impose a one-job-per-week policy.
r_j	release date of order J_j.
d_j	due date of order J_j.
$\overline{d}_j$	deadline of order J_j.
r_{bj}	internal release date of job B_{bj}.
d_{bj}	internal due date of job B_{bj}.
$\rho_{j\pi}$	tardiness of order plan $a_{j\pi}$ for order J_j.
θ	penalty cost for one week of order tardiness.
κ	maximum number of jobs of the same order that are allowed to be produced in the same week.

A.2 Model ouput variables

O_t	number of overtime hours in week t.
H_t^R	number of hired hours in week t in regular operator time.
H_t^N	number of hired hours in week t in nonregular operator time.
S_{it}	number of outsourced production hours in week t for machine group M_i ($i = 1, \ldots, m$).
U_{it}	number of hours on machine group M_i in week t in nonregular operator time.
$X_{j\pi}$	0-1 variable that assumes the value 1 when order plan $a_{j\pi}$ is selected for order J_j.
Y_{bjt}	the fraction of job B_{bj} processed in week t.

Index

S

T

W

Samenvatting

Het resource loading probleem is een middellange termijn capaciteitsplanningsprobleem, waarbij een job shop met verschillende resources, zoals machines en operators, een aantal klantorders moet verwerken. We richten ons in dit onderzoek op het resource loading probleem in de make-to-order (MTO) productieomgeving. Deze productieomgeving wordt gekenmerkt door een niet-repetitieve productie van kleine batches van klantspecifieke producten, die worden samengesteld uit bestaande componenten en speciaal voor de klant ontworpen componenten. Bij de klantorderverwerking worden MTO bedrijven doorgaans geconfronteerd met een grote mate van onzekerheid. Er is onzekerheid over welke orders uiteindelijk verworven zullen worden, en bovendien zijn klantorderkarakteristieken vaak niet of slechts gedeeltelijk bekend. Daarnaast is de beschikbaarheid van machinecapaciteit en personeel op deze middellange planningstermijn vaak niet gegarandeerd.

Het doel van dit onderzoek is het ontwikkelen van wiskundige modellen en algoritmen die als tactische instrumenten de klantorderverwerking ondersteunen bij het bepalen van betrouwbare levertijden, en benodigde machine- en personeelscapaciteit om de klantorders te produceren binnen de gestelde levertijden. Omdat gedetailleerde orderkarakteristieken vaak onzeker zijn en gedetailleerde inputgegevens voor de resource loading daarom meestal niet beschikbaar zijn, voeren we geen gedetailleerde planning of scheduling uit, maar een planning op een hoger aggregatieniveau. Hierbij houden we wel rekening met complexe technologische restricties, zoals routeringen en lineaire volgorderelaties tussen machinegroepen. Bij de resource loading is de capaciteit van machines en personeel flexibel door het toestaan van overwerken, inhuren van tijdelijk personeel, en uitbesteden. Na de klantorderverwerking kan een resource loading instrument worden ingezet om de capaciteitsniveaus vast te stellen voor het onderliggende korte termijn schedulingsprobleem, waarin de capaciteitsniveaus niet meer flexibel zijn, en als gegeven worden verondersteld. Resource loading biedt daarom inzicht in waar capaciteitsniveaus onvoldoende zijn, en kan oplossingen bieden door efficiënte allocatie van klantordercomponenten en niet-reguliere capaciteit. Dit vereenvoudigt het onderliggende schedulingsprobleem.

We positioneren het onderzoek in een raamwerk voor capaciteitsplanning functies, en analyseren bestaande instrumenten voor de resource loading

(Hoofdstuk 1). Terwijl er in de literatuur veel aandacht is besteed aan de korte termijn productieplanning (scheduling) op het operationele niveau, en de geaggregeerde lange termijn planning op het strategische niveau, is de beschikbaarheid van modellen en algoritmen voor de tactische planning zeer beperkt. Onderzoek bij een aantal Nederlandse bedrijven heeft nieuwe inzichten opgeleverd met betrekking tot het gebruik van geavanceerde combinatorische technieken (Hoofdstuk 2) voor de resource loading. Deze ideeën worden verder uitgewerkt in dit proefschrift. We ontwikkelen een geheeltallig lineair programmeringsmodel van het resource loading probleem (Hoofdstuk 3), en presenteren verschillende exacte methoden en approximatiemethoden (heuristieken) voor het oplossen van dit probleem. Het gepresenteerde model maakt een op kosten gebaseerde afweging tussen levertijd-performance enerzijds, en het gebruik maken van niet-reguliere capaciteit anderzijds. Hierbij houdt het model rekening met de eerdergenoemde complexe technologische restricties.

De moeilijkheid van het formuleren van het resource loading probleem als een (geheeltallig) lineair programmeringsmodel is dat de volgorderelaties niet rechtstreeks zijn te modelleren. De resulterende modellen zijn door hun omvang niet of nauwelijks oplosbaar. Onze aanpak is gebaseerd op een geheeltallig lineair programmeringsmodel met een exponentieel aantal geheeltallige variabelen. We lossen de lineaire programmeringsrelaxatie van dit model op met een kolomgeneratiealgoritme, dat een dynamisch programmeringsalgoritme gebruikt voor het oplossen van het corresponderende pricing probleem (Hoofdstuk 4). Wanneer de oplossing van de lineaire programmeringsrelaxatie geheeltallig is, hebben we een optimale oplossing gevonden van het resource loading probleem. In het andere geval passen we een branch-en-bound algoritme toe, om de optimale oplossing te vinden. Naast enkele exacte methoden presenteren we enkele approximatiemethoden die gebaseerd zijn op het zoeken van geheeltallige oplossingen uit de fractionele oplossing van de lineaire programmeringsrelaxatie.

Terwijl de eerder genoemde algoritmen uitgaan van lineaire volgorderelaties, presenteren we tevens een generalisatie van de algoritmen, zodat deze om kunnen gaan met generieke volgorderelaties (Hoofdstuk 6). Deze algoritmen zijn geschikt voor het oplossen van middellange termijn capaciteitsplanningsproblemen in projectomgevingen, i.e., de zogenaamde rough-cut capacity planning (RCCP) problemen. Gebleken is dat instanties van een redelijke omvang optimaal zijn op te lossen, echter grote instanties vergen te veel tijd om optimaliteit van de oplossing te bewijzen.

We presenteren rekenresultaten voor zowel de resource loading algoritmen (Hoofdstuk 5) als de RCCP algoritmen (Hoofdstuk 7), en vergelijken de rekenresultaten met die van bestaande heuristieken.

Curriculum Vitae

Erwin Hans was born on July 28, 1974 in Avereest, the Netherlands. In 1992 he obtained his Atheneum diploma at De Nieuwe Veste in Coevorden. From August 1992 to October 1996 he studied Applied Mathematics at the University of Twente, and specialized in mathematical programming and combinatorial optimization. From January 1996 until October 1996 he did his graduation assignment at the Energy Research Foundation in Petten, under supervision of prof.dr. J.J. Bisschop. In November 1996 he graduated after completing his Master's thesis, entitled 'ELMA - model of a liberalized electricity market'.

In January 1997 he started as a Ph.D. student under the supervision of prof.dr. W.H.M. Zijm, prof.dr. S.L. van de Velde, and dr.ir. A.J.R.M. Gademann. He started on a research project called 'Capacity Planning and Material Coordination in Complex Manufacturing Systems'. The project resulted in this thesis, entitled 'Resource Loading by Branch-and-Price Techniques'. From January 2001 he is employed as an assistant professor at the University of Twente.