Column Generation Algorithms for Machine Scheduling and Integrated Airport Planning

Kolomgeneratiealgoritmen voor Machinevolgordeproblemen en Geïntegreerde Planning op Luchthavens

(met een samenvatting in het Nederlands)

Proefschrift

ter verkrijging van de graad van doctor aan de Universiteit Utrecht op gezag van de rector magnificus, prof.dr. J.C. Stoof, ingevolge het besluit van het college voor promoties in het openbaar te verdedigen op woensdag 10 september 2008 des middags te 12.45 uur

door

Guido Diepen

geboren op 26 mei 1980
te Bergen op Zoom

Promotor:	Prof.dr. J. van Leeuwen
Co-promotoren:	Dr.ir. J.M. van den Akker
	Dr. J.A. Hoogeveen

This work was supported by BSIK grant 03018 (BRICKS: Basic Research in Informatics for Creating the Knowledge Society)

Preface

When I look back at how I became a Ph.D. student in Utrecht, Han Hoogeveen played an important role. During my studies I took his course on 'Optimalisering' and for this course we had to write a program to solve an optimization problem for trains. Together with Remco Vossen I wrote a program that used all available computers of the computer science department for a complete weekend. With the solution found after the weekend of computation we went to Han, but he was not impressed with our solution: without too much effort he could find a much better solution. After that I wrote another program, which produced a near-optimal solution within just a couple of seconds. During the diploma ceremony at the end of my MSc studies Han told that the combination of these two implementations had resulted in asking me for a masters' research project on the gate assignment problem. Little did he know that aircraft were another point of interest to me. After I got my diploma, Han presented me with the opportunity of doing a Ph.D. project on the topic of Supply Chain Optimization. Somewhere during the research, Han Hoogeveen and Marjan van den Akker had the great suggestion of considering the planning problems at Amsterdam Airport Schiphol as a supply chain and this resulted in the possibility of looking even further into the airport planning problems.

There are a couple of people who helped me in many ways to get to the result of this thesis. First of all, I want to thank my promotor Jan van Leeuwen for all his help, advice, and efforts in making the thesis better. Furthermore, I want to thank Han en Marjan for all of their support and great ideas and Marjan in particular for helping me to keep track of my deadlines. Also, a very big thanks goes to the people at Amsterdam Airport Schiphol: Koos Ruiter, Ko Stromer, and Nick Struik. Not only did they provide me with all data that I needed for the research, they also gave me the awesome experience of a tour over the platform and of visiting their offices at the air-side of Schiphol.

I also want to thank a couple of people who created the great atmosphere that made me enjoy the four years as a Ph.D. researcher very much. For some reason it was pretty clear to everybody from the beginning on that I would hardly ever say no when asked to get a cup of coffee. This resulted in lots of discussions and talks with Peter Lennartz while getting the coffee. Also, the games of darts played with Roland and Han provided me with yet another opportunity to drink coffee, in between throws. While initially I had to walk quite a bit to get my coffee, this changed when Hans Bodlaender took the Senseo machine from his home to his office. In little time this resulted in a name change of his office to 'Chez Hans'. A very big thanks goes to Hans and the regulars of Chez Hans for the very nice time and the interesting debates we had during the coffee breaks. Also, I would like to thank Wilke Schram for the daily piece of apple at the lunch table.

When the end of the four years of Ph.D. research got in sight, my brother Sjoerd offered me his help with designing the cover. I owe him a lot of thanks for the wonderful design he came up with and for the efforts he put in getting all the details right. Furthermore, I would like to thank my parents for all of their support throughout the years and for providing me with the opportunities that got me to where I am now. And finally, but certainly not least, I want to thank my wife Justyna for all her support.

CONTENTS

1 Introduction **1**
- 1.1 Optimizing the supply chain . . . 2
- 1.2 Topics in this thesis . . . 5

I Machine Scheduling **9**

2 Minimizing weighted tardiness with equal processing times **11**
- 2.1 Introduction . . . 11
- 2.2 Problem formulation . . . 14
- 2.3 Rewriting fractional non-double-nested solutions . . . 18
- 2.4 Common due date . . . 20
  - 2.4.1 Common due date equal to zero . . . 21
  - 2.4.2 Common due date $d_j = D$ . . . 22
- 2.5 Arbitrary due dates . . . 24
  - 2.5.1 Branching rule . . . 28
- 2.6 Related objective functions . . . 29
- 2.7 Parallel identical machines . . . 32
- 2.8 Column generation . . . 33
- 2.9 Computational experiments . . . 39
- 2.10 Conclusion and further research . . . 43

3 Resource-constrained project scheduling **45**
- 3.1 Introduction . . . 45
- 3.2 Parallel machine scheduling . . . 47
- 3.3 Enhancements . . . 51
  - 3.3.1 Speeding up the column generation procedure . . . 52

3.3.2 Propagating knowledge from column generation 53
3.3.3 Valid inequalities for the exact precedence delays 54
3.4 RCPS problems 54
3.4.1 RCPS problems with one type of resource 55
3.4.2 RCPS problems with multiple resources 57
3.4.3 Other extensions 60
3.5 Computational experiments 64
3.6 Conclusion 69

II Planning at Amsterdam Airport Schiphol 71

4 Solving the gate assignment problem 73
4.1 Problem description 76
4.2 Problem formulation 79
4.2.1 Assigning flights to gate plans 81
4.2.2 Pricing problem 84
4.2.3 Solving the restricted ILP 86
4.2.4 Assigning gate plans to gates 87
4.2.5 Directly assigning flights to gates 89
4.3 Computational experiments 90
4.4 Conclusion and further research 95

5 Solving the bus planning problem 97
5.1 Introduction 97
5.2 Problem description 99
5.3 Problem formulation 102
5.3.1 Pricing problem 104
5.3.2 Solving the ILP 108
5.4 Computational experiments 110
5.5 Conclusion 118

6 Solving the integrated airport planning problem 121
6.1 Introduction 121
6.2 Problem formulation 124
6.3 Solving the problem 129
6.3.1 Assigning flights to gate plans and trips to bus plans . . . 129
6.3.2 Assigning gate and bus plans to the actual gates and buses 133
6.4 Computational experiments 134
6.5 Conclusion and further research 142

Bibliography **145**

Samenvatting **149**

Curriculum vitae **155**

Colofon **157**

CHAPTER

ONE

Introduction

Planning problems appear in our lifes on a daily basis and in numerous forms. These problems can arise explicitly in a planning context, but can also occur implicitly, without us even realizing it. An example of an explicit planning problem is the food preparation of a chef in a restaurant who needs to determine in what order to prepare the dishes such that everybody at one table gets his or her dish at the same time, while all the dishes are still warm.

An example in which planning problems appear implicitly occurs when one buys a product. Before one actually buys the product, this product has a complete history already. It has been transported from a factory, where it has been assembled from subcomponents, which in turn have been assembled also out of even smaller components. The chain of actions that a product undergoes from the raw materials to the final product is referred to as the *supply chain*. When we look at two consecutive stages within a supply chain, these two stages can be executed by the same factory, but they can also be executed by two different factories, where the first factory is the supplier for the second factory.

In the different stages of a supply chain, different types of problems occur. There

are stages in which we have to deal with logistic (or distribution) problems, stages in which we have to deal with storage problems, and stages in which we have to deal with production planning problems in which a scarce resource needs to be shared in some way. Examples of such scarce resources are workers, machines, but also gates or buses. In this thesis we are interested in the planning problems that arise across the different stages of supply chains.

1.1 Optimizing the supply chain

Traditionally the planning in each stage of the supply chain was optimized as a separate planning problem. The reason for this was twofold. One reason was the lack of computing power: generally the separate problems are already difficult to solve, let alone when we integrate the problems. The other problem was more economic and occurs when two consecutive stages of the supply chain are executed by two different companies: normally there is no direct incentive for the first company to consider the preferences of the second company when determining a solution for its planning/scheduling problem.

Because of the increase in available computation power and also of better algorithms, the first reason for only considering stages separately when optimizing the supply chain is becoming less of an issue. And if companies are willing to share the profit gained by solving the integrated problems of consecutive stages, the second reason should be no issue either. When considering preferences of the other company, the planning might become sub-optimal for the first company, but if the solution for the complete problem improves, then the profit can be distributed among the companies involved, meaning all of them will benefit.

Due to the rapidly increasing computer power and, above all, greatly improved algorithms, classical integer linear programming (ILP) has become an important tool for solving practical problems. Many planning problems can be formulated as an ILP. When we look at the ILP formulations of these planning problems, we find that they often are very large and complicated, but also contain a lot of structure in the constraints. This structure in the constraints allows for a Dantzig-Wolfe decomposition of the ILP (cf. Wolsey (1998)). This decomposition can be applied in case a solution consists of a number of building blocks or subplans. Each of these subplans only needs to satisfy a subset of the constraints. The resulting model of this decomposition is called the *master problem*.

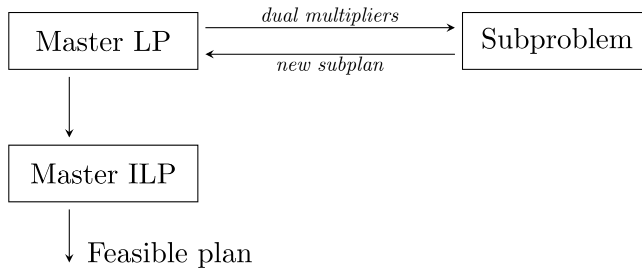

Figure 1.1: Outline of column generation procedure.

It consists of selecting a set of subplans subject to the constraints that are not included in the feasibility constraints of the subplans. This means that we distinguish between a set of constraints satisfied by each subplan ("local constraint") and a set of constraints satisfied by the master problem ("global constraints"). We will use this distinction in solving the problem.

Observe that usually this decomposition results in a master problem with a very large number of variables, as each feasible subplan must be included. To solve the master problem, we first relax the integrality constraints and solve the resulting LP-relaxation with *column generation*, an approach which does not consider all variables explicitly.

Column generation

Column generation proceeds as follows. Consider an LP in its most general form:

$$\min \quad cx$$

subject to:

$$Ax = b$$
$$x \geq 0.$$

From the theory of LP we know that in a basic solution all non-basic variables have value 0. So when we look at the optimal basic solution of the problem, we can see that only a small fraction of all variables will have a non-zero value.

In column generation we do not consider all variables at once, but only consider a subset of all possible variables explicitly. The model containing only this limited subset of variables is called the *restricted master problem* (RMP). When the RMP has been solved to optimality, the found solution might be sub-optimal for the complete problem. To see whether this is the case, we check for the existence of a variable not yet in the RMP that, if added to the RMP, could possibly improve the solution value of the RMP. Each of the constraints in the RMP provides us with a dual multiplier λ_i which can be used to determine the reduced cost of a variable j as follows

$$c_j - \sum_{i=1}^{m} \lambda_i a_{ij},$$

where c_j is the cost of j. From LP theory we know that in a minimization problem only a variable with negative reduced cost can improve the solution value. Thus, finding a variable with negative reduced cost is equal to finding a variable that could improve the solution value of the RMP.

The main idea behind column generation is that we do not check for the existence of a variable with negative reduced cost by just enumerating all variables, but that we determine the existence of such a variable by solving an optimization problem. This problem used for determining whether there exists a variable with negative reduced cost is called the *pricing problem*. Depending on the structure of the pricing problem, the strategy is to find the variable with minimum reduced cost. When the variable with minimum reduced cost is found, two cases can be distinguished. The first case is that the variable that is found has negative reduced cost. In this case we add the variable to the RMP and continue iterating by solving the RMP again. The second case is that the variable that is found has reduced cost ≥ 0. Since the variable has minimum reduced cost, we know that no variable exists that could improve the solution value of the RMP, meaning that the RMP has been solved to optimality. In case the solution is integral, we have an optimal solution for the original problem. Otherwise, all subplans generated by solving the pricing problems can be used to get an approximation for the optimum of the original problem by solving the ILP master problem with only these columns. This whole process is illustrated in Figure 1.1. Other possibilities for solving the ILP master problem are to use the column generation in a branch-and-bound setting. This technique is called *branch-and-price* (cf. Wolsey (1998)).

The name ‘column generation’ comes from solving the pricing problems, which

generate columns that are added to the RMP. The question whether column generation works for a given problem depends for a large part on the fact whether the pricing problem can be solved efficiently. Note that in general we do not have to find the variable with minimum reduced cost, as long as we identify one with negative reduced cost; we only need to solve the pricing problem to optimality if we cannot find one with negative reduced cost in another way.

1.2 Topics in this thesis

In this thesis the first problem we investigate is a *machine scheduling problem* where all of the jobs require the same time of the resource. We consider this problem to be a very basic problem in a supply chain because there exist quite some different real-life problems where this situation occurs. For example, in case we have to dry products after they have been painted, the time needed is exactly the same for all products. We will provide a detailed characterization of the instances that allow for a polynomial time algorithm and for all other instances we provide a branching strategy that can be used in a branch-and-bound search.

Furthermore, we investigate the so-called *resource-constrained project scheduling* (RCPS) problem, where we need to determine which tasks will be assigned to which resources at what time. Also, the tasks need to be executed within a given time interval and there may exist precedence relations between the tasks (i.e. before we can start one task, another task must be finished). For a variety of relevant cases we present the further analysis required that enables us to use a general framework for solving them.

A next problem we consider in this thesis is the *gate assignment problem* at Amsterdam Airport Schiphol (AAS). This problem deals with determining where to place the aircraft after they land at AAS. This gate assignment problem can be seen as a RCPS problem: each of the aircraft is a project that requires a resource (i.e. gate) for a given time interval. We will give a solution method based on column generation to solve real-life instances of the gate assignment problem within a matter of minutes.

When we look at the type of gates at AAS, we can distinguish two different types: there are the gates that are connected to the terminal building and pas-

sengers can embark and disembark the aircraft via a passenger air bridge. The second type of gates is the remote stand, which is not connected to the terminal building, and passengers have to be transported to or from the terminal building by means of buses. The *bus planning problem* is the problem of determining which bus must drive to or from which aircraft. In a way similar way to the gate assignment problem, the bus planning problem can also be seen as a RCPS problem. We will present a model and a solution method for the bus planning problem that are similar to the ones presented for the gate assignment problem. We will show that the solution method is capable of solving real-life instances of the bus planning problem in a matter of minutes.

Finally in this thesis, we investigate the *integration* of the gate assignment problem and the bus planning problem at an airport. The gate assignment problem and the bus planning problem both can be seen as separate stages in a supply chain. These two stages can be integrated by combining the two problems as one large problem. Again, this integrated problem can be seen as a RCPS problem. This time the constraints are a bit more complex, because in the case an aircraft is assigned to a gate that requires buses for transporting the passengers, not only does the aircraft require the gate as a resource, but also one or more buses as resources to transport the passengers. We will show the details of how the separate models for the gate assignment problem and the bus planning problem can be integrated into one large model. Furthermore, we will show that we can solve this integrated problem within acceptable running times.

Outline of the thesis

The remainder of the thesis consists of two separate parts: one part regarding machine scheduling and one part regarding the planning problems occurring at Amsterdam Airport Schiphol.

In the machine scheduling part we consider two separate problems. First, in Chapter 2 we look at the machine scheduling problem in which we have to minimize the *total weighted tardiness* when all jobs require the same processing time on the machine. The second problem we consider in the machine scheduling part is the *RCPS problem.* In Chapter 3 we present a variety of different RCPS problems and provide the analysis needed for using a general framework for solving these problems.

In the second part of this thesis, we consider planning problems at Amsterdam Airport Schiphol. We first look at the gate assignment problem (i.e. where to place each aircraft after it arrives at Schiphol) in Chapter 4. For the aircraft that are assigned to gates that are not equipped with a bridge to the terminal building, passengers have to be transferred to or from the aircraft with buses. In Chapter 5 we investigate the bus planning problem, which concerns the problem of determining which bus will drive to or from which aircraft to transport passengers. Finally, in Chapter 6 we consider the integration of these two problems and solving them as one big problem instead of solving them sequentially.

Part I

Machine Scheduling

CHAPTER

TWO

Minimizing weighted tardiness with equal processing times

2.1 Introduction

The problem we consider is the following: We have a single machine on which we have to process a set $N = \{J_1, J_2, \ldots, J_n\}$ of jobs, where n is the number of jobs. In fact, instead of only a single machine, we can have m parallel, identical machines on which we have to process the jobs. Our results are valid for both cases, but for ease of explanation we will first consider the single machine case and later show how the results hold for the m parallel, identical machines case.

Furthermore, for each job $J_j \in N$ we have a release date r_j, a due date d_j, and a weight w_j. The processing times of the jobs are all equal to p. We assume the release dates, the due dates, and the weights of all jobs to be both integral and non-negative, while we assume the common processing time for N to be integral and positive. We are looking for a feasible schedule, that is, we want to find a set of completion times C_j $(j = 1, \ldots, n)$ such that no job starts before

its release date and no two jobs have an overlap in their execution. Given the completion time C_j of a job J_j, we define the tardiness T_j of J_j as:

$$T_j = \max\{0, C_j - d_j\}.$$

Now the objective is to find the feasible schedule that minimizes the total weighted tardiness $\sum w_j T_j$. To the best of our knowledge it is still unknown whether this problem is $\mathcal{NP}$-hard or not.

To denote the different types of scheduling problems we make use of the three-field notation scheme introduced by Graham et al. (1979). In this three-field notation scheme the problem of minimizing total weighted tardiness on a single machine with release dates and equal-length jobs is denoted as $1|r_j, p_j = p|\sum w_j T_j$.

Recall from the introduction in Chapter 1 that minimizing the total weighted tardiness is interesting in the context of supply chain optimization since it can be considered as a model of a single stage in a chain of processes which a product must undergo before it has reached its final destination.

One example of the minimization problem we consider would be a factory where one needs to dry products after painting or varnishing. For all products this drying process takes exactly the same time, while the color of the paint or other properties of the product are different. So all products have the same processing time, but they become available at different times, are due at different times, and some products might have a higher priority (i.e. a higher weight) than other products.

Over the years quite some research has been done on scheduling problems regarding (weighted) tardiness. Lawler (1977) give a pseudopolynomial algorithm for solving the $1||\sum T_j$ problem and Du and Leung (1990) give a proof for the $1||\sum T_j$ problem to be $\mathcal{NP}$-hard in the ordinary sense. The weighted version of this problem, $1||\sum w_j T_j$, is known to be $\mathcal{NP}$-hard in the strong sense (Lenstra et al., 1977). Akturk and Ozdemir (2001) give a new dominance rule for solving the $1|r_j|\sum w_j T_j$ problem to optimality using branch-and-bound. Kolliopoulos and Steiner (2004) examine the $1||\sum w_j T_j$ problem from the perspective of approximation algorithms and provide a fully polynomial time approximation scheme (FPTAS) for the case where the weights are polynomially bounded.

Also quite some research has been done on scheduling problems with equal processing times: Baptiste (2000) investigates the $1|r_j, p_j = p|\sum T_j$ problem, as

well as the problem with m identical, parallel machines instead of one and shows that both problems can be solved in polynomial time by means of dynamic programming. Baptiste (1999) gives a polynomial time algorithm for the weighted number of late jobs problem based on dynamic programming. In the three-field notation scheme this minimization problem is denoted as $1|r_j, p_j = p|\sum w_j U_j$, where U_j has the value 1 if $C_j > d_j$ (i.e. J_j is late) and 0 otherwise. In Baptiste et al. (2004) ten equal-processing-time scheduling problems are shown to be solvable in polynomial time, among which is the $Pm|r_j, p_j = p|\sum w_j U_j$ problem. An overview of more problems where jobs have equal-processing times can be found in Leung (2004).

Verma and Dessouky (1998) look at common processing time scheduling with earliness and tardiness penalties where earliness E_j for J_j is defined as $E_j = \max\{0, d_j - C_j\}$ and without release dates. They formulate this problem as a time-indexed ILP and show that when certain criteria are met, there exists an integral optimal solution to the LP-relaxation, which means that there exists a polynomial time solution procedure. In this chapter we build upon their approach, and show that if certain criteria are met, then the $1|r_j, p_j = p|\sum w_j T_j$ problem can be solved in polynomial time.

The outline for the rest of this chapter is as follows: In Section 2.2 we give an ILP-formulation of the minimization problem. In Section 2.3 we present an algorithm that can be used to rewrite fractional solutions for the single-machine case that possess a special property of being non-double-nested. In Section 2.4 we discuss the special case of the problem in which the jobs have a common due date and provide a polynomial time algorithm for solving this case. In Section 2.5 we discuss the general problem and show in which cases this general problem can be solved in polynomial time. In Section 2.6 we apply the same techniques to problems with related objective functions for the single machine case, and in Section 2.7 we look at the case with m parallel, identical machines. After this, in Section 2.8 we discuss a way of solving the problem based on column generation, which aims at reducing the amount of memory needed, and in Section 2.9 we present some experimental results based on an implementation of the model and algorithm. Finally in Section 2.10 we draw some conclusions.

2.2 Problem formulation

The problem we consider is to find a feasible schedule for a given set of jobs, each with release date, due date, weight, and a processing time that is the same for all jobs, that minimizes the total weighted tardiness.

Like in Verma and Dessouky (1998), we use a time-indexed formulation to represent this minimization problem as an ILP. We restrict ourselves to the times that can occur as completion times in an optimal solution. Because of the equal processing times, the processing of a job will always occur in an interval with length p. We denote each interval by the end time of the interval. The objective function that we consider is total weighted tardiness, and since this function is non-decreasing in the completion times, there will always exists an optimal schedule that is *left-aligned*, that is, no unnecessary idle time exists.

Since there exists an optimal schedule that is left-aligned, we can restrict ourselves to schedules in which each job either starts at its release date, or immediately after another job. Hence, each release date introduces a set of possible completions times, and in the extreme case some job J_j will start right at its release date after which all other jobs follow contiguously. This means that job J_j introduces $(n-1)$ possible completion times for the other jobs.

Unless another job $J_{j'}$ has a release date that is a multiple of p before r_j, only job J_j can be completed at time $\gamma_j = r_j + p$. We define the set G denoting all first possible completion times of all jobs by

$$G := \bigcup_{j=1}^{n} \{\gamma_j\} .$$

We define the set M_j as the set of possible completion times introduced by job J_j as

$$M_j = \{r_j + 2p, r_j + 3p, \ldots, r_j + np\} ,$$

and the set K containing all these other possible completion times as

$$K := \bigcup_{j=1}^{n} M_j .$$

The set $K \cup G$ gives a *time horizon* in which all jobs must have their completion times. Since the first possible completion time of job J_j is γ_j, we find that the

set of possible completion times of a job J_j is

$$A_j := \{t \in K | t > r_j + p\} \cup \{\gamma_j\}.$$

Since the capacity of the machine is equal to 1, completing a job J_j at time t implies that no other job $J_{j'}$ can have a completion time that is less than p time units before t. For a given time t, we define H_t as the set of preceding completion times conflicting with t as

$$H_t := \left\{t' \in K \cup G | 0 \leq t - t' < p\right\}.$$

Now we formulate the problem of minimizing the weighted tardiness with equal processing times as a time-indexed ILP model. For the ILP formulation we define a variable $x_{j,t}$ for all relevant j and t as

$$x_{j,t} = \begin{cases} 1 & \text{if job } J_j \text{ is completed at time } t \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore we define the cost $c_{j,t}$ of having job J_j completed at time t as

$$c_{j,t} = w_j \max\{0, t - d_j\}.$$

Now the complete ILP model becomes:

$$\min z = \sum_{j=1}^{n} \sum_{t \in A_j} c_{j,t} x_{j,t}$$

subject to:

$$\sum_{t \in A_j} x_{j,t} = 1 \text{ for all } j \in N \tag{2.1}$$

$$\sum_{j=1}^{n} \sum_{t' \in H_t \cap A_j} x_{j,t'} \leq 1 \text{ for all } t \in K \cup G \tag{2.2}$$

$$x_{j,t} \in \{0,1\} \text{ for all } j \in N,\ t \in A_j, \tag{2.3}$$

where constraint (2.1) ensures that all jobs are assigned to exactly one interval and constraint (2.2) ensures that for any time interval no more than one job is processed.

Verma and Dessouky (1998) look at the single machine scheduling of equal-length jobs with both earliness and tardiness penalties, where earliness for job

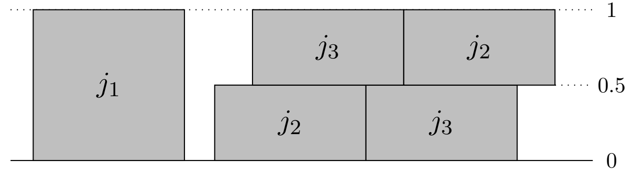

Figure 2.1: Example of a fractional solution in the LP-relaxation.

J_j is defined as E_j =max$\{0, d_j - C_j\}$. In the three-field notation scheme this problem is $1|p_j = p| \sum \alpha_j E_j + \beta_j T_j$, where α_j is the earliness penalty for job J_j and β_j is the tardiness penalty for job J_j. Since they consider earliness penalties, sometimes it can be beneficial to introduce idle time before starting to process a job. They use the same time-indexed formulation. Except for the objective function, which is reflected in the values $c_{j,t}$, the only difference with our problem is the presence of release dates. We build on their results.

One important observation that has to be made is that allowing fractional solutions in the LP-relaxation is not the same as allowing preemption. With preemption one is allowed to process some part of the job and after any time smaller than the processing time, one may preempt it. In our case, if jobs have fractional assignments, no preemption occurs: each job will always have the same processing time, only a certain fraction of the work needed for the job is processed in that time interval. An example of this situation is given in Figure 2.1. The maximum capacity of the machine is equal to 1 (i.e. at most one job can be processed at the same time). Job j_1 has an integer allocation value, and therefore it is completely processed in one given time interval, whereas both jobs J_{j_2} and J_{j_3} each have fractional allocation values, and hence they are both partially allocated to two different time intervals.

First, we repeat some of the definitions introduced by Verma and Dessouky (1998).

Definition 2.2.1. (Verma and Dessouky, 1998) *Let x be a feasible solution to the LP-relaxation. We say that job J_{j_2} is* nested *in job J_{j_1} ($j_1 \neq j_2$), if there exists some $t_k \in K \cup G$, ($k = 1, 2, 3$) such that $t_1 < t_2 < t_3$ and x_{j_1,t_1}, x_{j_2,t_2} and x_{j_1,t_3} are all positive.*

We now define the $\prec$-relation based on the above definition as follows: $j_1 \prec j_2$, if a job J_{j_1} is nested in job J_{j_2}.

Definition 2.2.2. (Verma and Dessouky, 1998) *A feasible solution of the LP-relaxation is non-nested if and only if there exists a pair of jobs J_{j_1} and J_{j_2} which are nested in each other, i.e., $j_1 \prec j_2$ and $j_2 \prec j_1$. If no such pair of two jobs exists, then the solution is a nested solution.*

Verma and Dessouky (1998) use the term non-nested solution for the situation where two jobs are nested in each other. For reasons of clarity we changed this term into *double-nested*.

For ease of writing we furthermore define the notation 1212 to denote that there exist x_{j_1,t_1}, x_{j_2,t_2}, x_{j_1,t_3}, and x_{j_2,t_4} with $t_1 < t_2 < t_3 < t_4$ all having value > 0 in the solution of the LP. In the same way we define the notation 2121 to denote that there exist x_{j_2,t_1}, x_{j_1,t_2}, x_{j_2,t_3}, and x_{j_1,t_4} with $t_1 < t_2 < t_3 < t_4$ all having value > 0 in the solution of the LP.

Verma and Dessouky (1998) show that the $1|p_j = p|\sum \alpha_j E_j + \beta_j T_j$ problem is polynomially solvable if the jobs can be indexed such that $\alpha_1 \leq \alpha_2 \leq \ldots \leq \alpha_n$ and $\beta_1 \leq \beta_2 \leq \ldots \leq \beta_n$. This proof consists of two parts. First they prove that there exists an optimal solution to the LP-relaxation that is non-double-nested, and that in case of a strict ordering, that is, if each inequality is strict, each optimal solution is non-double-nested. Second, they prove that all extremal, non-double-nested solutions to the LP-relaxation are integral. This provides a polynomial time algorithm for solving the problem with the strict ordering; for the case with equal weights they show how these equalities can be removed by adding perturbations such that the resulting solution is still optimal for the problem.

The proof that each non-double-nested, extremal solution of the LP-relaxation is integral is independent from the objective function, as it is solely based on the fact that if a solution is nested, then the constraint matrix for the non-zero columns (i.e. columns for which the value is greater than 0) of the solution can be reordered in such a way that it forms an interval matrix, and interval matrices are known to be totally unimodular (Nemhauser and Wolsey, 1988). Therefore, we can use their results, if we show that the presence of the release dates does not destroy their proofs. Since a release date r_j of job J_j implies that J_j cannot complete its execution at a time $t < r_j + p$, we only have to add

constraints of the form $x_{jt} = 0$. These constraints are represented by unit rows in the constraint matrix and adding a unit row to a totally unimodular matrix results in a matrix which is also totally unimodular. Hence, we get the following lemma.

Lemma 2.2.1. *The results of Verma and Dessouky (1998) are not influenced by adding release dates.*

Using the second part of the proof given by Verma and Dessouky (1998), we will show that certain cases of the minimization of weighted tardiness problem are solvable in polynomial time by either showing that any double-nested solution to the LP-relaxation is sub-optimal. Or, in case of an optimal double-nested solution, we will show that such a solution can be converted in polynomial time into a non-double-nested solution with equal cost, which then can be converted into an integral solution with equal cost.

2.3 Rewriting fractional non-double-nested solutions

In this section we present an algorithm (*rewrite rule*) to convert a fractional solution that is non-double-nested into an integral solution with equal cost. Although for most instances the rewrite rule returns a feasible solution, there are instances for which the rewrite rule does not work. In case the rewrite rule does not return a feasible solution, we must use the method presented by Verma and Dessouky (1998) of adding perturbations to find an integral solution.

We assume that:

- The assignment is left-aligned, that is, given that job J_j is completed at time t, it is not possible to increase any value $x_{j,t'}$ with $r_j + p \leq t' < t$, while keeping the assignment values of the other jobs intact.
- There are no two jobs J_i and J_j that are both (partly) assigned to the same pair of intervals.

It is easily verified that each solution can be modified such that the additional assumptions are satisfied.

Furthermore, we introduce the notion of a job J_i being *consecutive* if there does not exist another job J_j that is nested in job J_i. In case of a non-double-nested solution it is easily seen that at least one consecutive job must exist. Since the assignment is left-aligned, the difference between the completion time and starting time of any consecutive job is no more than $2p$, where the starting time of a job is defined as the starting time of the first interval that it has been assigned to. Note that if this difference is smaller than $2p$, then constraint (2.2) implies that there exists a time-point t and an $\epsilon > 0$ such that J_i is the only job that is processed in the interval $[t - \epsilon, t]$.

We use the following algorithm to find an integral solution.

SELECTION ALGORITHM

- Start with the leftmost (i.e. earliest) interval
- Pick a job assigned to this interval in the following way:
  - If there is only one job, pick this one.
  - If there are two or more jobs assigned to this interval and each one has been picked before, pick any of them.
  - Otherwise, pick a job that has not been picked before; if there are two (or more) such jobs, pick any job of these that has been assigned to some earlier interval.
- Record the job-interval combination and move to the next available and repeat from the second step.

Lemma 2.3.1. *If the rewrite rule selects all jobs exactly once, the found solution is an integral solution with the same cost.*

Proof. Let Z, with corresponding values $z_{j,t}$ for all relevant combinations of j and t, denote the assignment obtained by the selection algorithm; obviously, this corresponds to a feasible schedule. Furthermore, let X with values $x_{j,t}$ denote the assignment found by solving the LP-relaxation. Now construct the assignment Y with values $y_{j,t} = x_{j,t} - \epsilon z_{j,t}$ for all relevant combinations of j and t, where ϵ is equal to the smallest positive assignment value in X. Since $z_{j,t} = 1$ can only occur if $x_{j,t} > 0$, we see that all $y_{j,t} \geq 0$. Moreover, since each job occurs in Z exactly once, we see that the total amount of each job that has

been assigned in Y is equal to $1-\epsilon$. Furthermore, since the algorithm always selects a job whenever it is possible, the total amount of jobs assigned to each set of overlapping intervals is no more than $1-\epsilon$. Hence, if we multiply each $y_{j,t}$ value with a factor $1/(1-\epsilon)$, then we get another feasible solution, Y', to the LP-relaxation. This implies that X can be written as a non-trivial convex combination of Y' and Z; because of the optimality of X, assignment Z must be optimal, too. □

Lemma 2.3.2. *Any fractional non-double-nested solution can be converted into an integral solution with same cost.*

The proof follows directly from combining Lemma 2.3.1 with the fact that we can always use perturbation to find an integral solution.

2.4 Common due date

In this section we look at the special case in which all jobs j have a common due date $d_j = D$. Depending upon the size of D, we distinguish two variants. First, in Section 2.4.1 we look at the case $0 \leq D < r_{\min} + p$, where $r_{\min}$ is the smallest release date of all jobs. For any value of $D < r_{\min} + p$ we know that each job will be tardy, which implies that we obtain an equivalent problem by putting $D = 0$. The $1|r_j, p_j = p, d_j = 0|\sum w_j T_j$ problem is equivalent to the $1|r_j, p_j = p|\sum w_j C_j$ problem, for which Baptiste (2000) already showed that it can be solved in polynomial time ($\mathcal{O}(n^7)$) by means of dynamic programming. In Section 2.4.2 we look at the case where $D \geq r_{\min} + p$; in this case it is possible to complete at least one job at or before the due date. For both cases we assume without loss of generality that the jobs are reindexed such that $w_1 \leq w_2 \leq w_3 \leq \ldots \leq w_n$. If after reindexing the jobs $w_1 < w_2 < w_3 < \ldots < w_n$ holds, we say that a strict order on the weight of the jobs exists.

We will provide a polynomial time algorithm for solving the minimization of weighted tardiness problem for the case where all jobs not only share one common processing time, but also one common due date.

2.4.1 Common due date equal to zero

Lemma 2.4.1. *If there exists a* strict *order on the weight of the jobs, then any optimal solution is nested. When no strict order exists, then there exists an optimal schedule that is nested.*

Proof. Assume first there exists a strict order on the weights of the jobs and that we have an optimal solution that is double-nested. Therefore, according to Definition 2.2.2 there exist jobs J_{j_1} and J_{j_2} that are nested in each other. Without loss of generality we assume $j_1 < j_2$ (i.e. $w_{j_2} \geq w_{j_1}$). Let us now look at the situation where job J_{j_1} is nested in job J_{j_2}, that is, there exist intervals t_1, t_2, and t_3 such that $t_1 < t_2 < t_3$ and x_{j_2,t_1}, x_{j_1,t_2}, and x_{j_2,t_3} are all positive (denoted as situation 212). We define $\varepsilon = \min(x_{j_2,t_1}, x_{j_1,t_2}, x_{j_2,t_3})$.

In Figure 2.2 an example of the situation is depicted where without loss of generality we assumed $\varepsilon = x_{j_2,t_3}$. Job J_{j_1} is nested in job J_{j_2}, and job J_{j_2} is nested in job J_{j_1}. Since both jobs are tardy, the exchange of ε between x_{j_1,t_2} and x_{j_2,t_3} causes a change in the cost of Δ, where we have:

$$\begin{aligned}
\Delta &= \varepsilon(w_{j_2}t_2 + w_{j_1}t_3 - w_{j_1}t_2 - w_{j_2}t_3) \\
&= \varepsilon(w_{j_2}(t_2 - t_3) + w_{j_1}(t_3 - t_2)) \\
&= \varepsilon((t_3 - t_2)(w_{j_1} - w_{j_2})) \\
&\leq 0.
\end{aligned}$$

In the resulting situation job J_{j_2} is still nested in job J_{j_1}, but job J_{j_1} is not nested in job J_{j_2} anymore. The exchange does not violate any constraint as the amount of total allocation at each interval is unaltered, and only its division between the two jobs has been changed. Furthermore, the value of the objective function will not increase by this exchange. Because a strict order exists on the weight of the jobs it can be seen that $\Delta < 0$. Thus, an optimal schedule is always nested.

If there is no strict order on the weight of the jobs, then the optimal schedule might be double-nested. However, this double-nested schedule can always be rewritten into a nested schedule with equal cost by moving parts of jobs with equal weight, by the technique shown above. □

We have now shown that for the problem $1|r_j, p_j = p|\sum w_j C_j$ a double-nested

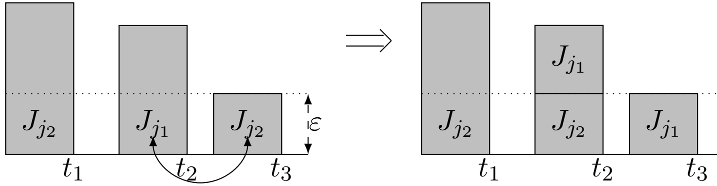

Figure 2.2: Example of rearranging ε between two nested jobs.

solution is either optimal (but can always be rewritten into a nested solution with equal cost in polynomial time) or sub-optimal. Now we can make use of Lemma 2.2.1 to show that this problem can be solved in polynomial time since the constraint matrix for the non-zero columns is totally unimodular. If the LP-solver returns a double-nested solution, then we can convert it into a nested solution as described in the proof Lemma 2.4.1. A nested solution still can be fractional, but by applying the Selection Algorithm of Section 2.3 we can always rewrite a fractional nested solution into an integral solution with equal cost.

Solving an LP with m variables takes $\mathcal{O}(\sqrt{m}\log\frac{m}{\epsilon})$ iterations where each iteration consists of $\mathcal{O}(m^3)$ calculation steps and where ϵ is the maximum allowed difference from the optimal value (Roos, 2005). In our case with n jobs, the number of variables we have is $\mathcal{O}(n^3)$, which means that the total running time for the LP is $\mathcal{O}(n^{10}\sqrt{n}\log\frac{n^3}{\epsilon})$. Although in the worst-case this is more than the previous known result of Baptiste (2000) with dynamic programming, in practice it might be a lot quicker.

2.4.2 Common due date $d_j = D$

Now we consider the situation that there exists a common due date for all jobs that is bigger than the first possible completion time of any job. Hence, one or more jobs can be placed before the common due date while others will become tardy.

Lemma 2.4.2. *A double-nested solution is either sub-optimal, or it can be rewritten into a nested solution with equal cost.*

Proof. Assume that we have an optimal double-nested solution. Therefore, there

are time intervals t_1, t_2, and t_3 such that $t_1 < t_2 < t_3$ and x_{j_2,t_1}, x_{j_1,t_2}, and x_{j_2,t_3} are positive. We will show that we can remove this nesting of J_{j_1} in J_{j_2} and therefore, remove the double-nesting.

Now there are two distinctions to be made:

- $D \geq t_3$. In this case the order of the jobs J_{j_1} and J_{j_2} does not matter because they are both on time. Hence one can switch parts of the jobs around in such a way that we end up with a nested solution of equal cost.
- $D < t_3$. If the weights of the two jobs are equal, we can again exchange parts between intervals t_2 and t_3 at no cost to convert the current double-nested solution into a nested solution. Otherwise, use the same approach as in the proof of Lemma 2.4.1, and define $\varepsilon = \min(x_{j_1,t_2}, x_{j_2,t_3})$.

  The optimal way of allocating ε of job J_{j_2} and ε of job J_{j_1} among the two intervals is to first allocate ε of job J_{j_2} to interval t_2 and then allocate ε of job J_{j_1} to interval t_3. The idea behind this can be seen again in Figure 2.2. The resulting solution after the exchange will have cost that can only have decreased. This implies that we can improve or rewrite the double-nested solution without violating any constraints as the amount of total allocation to each interval is unaltered, only the division between the two jobs has been changed. Furthermore, no job is moved to an interval that is before any interval it was previously allocated to, and hence each release date is satisfied.

This process should be repeated if more occurrences exist of double-nested situations for these two jobs.

We see that either the double-nested solution is not optimal, which contradicts our assumption, or it can be rewritten into a nested solution of equal cost. □

Theorem 2.4.3. *The problem* $1|r_j, p_j = p, d_j = D|\sum w_j T_j$ *can be solved in polynomial time.*

Proof. By combining Lemma 2.4.1 and Lemma 2.4.2 we know that for any value of D there exists an optimal schedule that is nested. By making use of Lemma 2.3.2 we know that any nested, fractional optimal schedule can be rewritten into an integral solution with equal cost. □

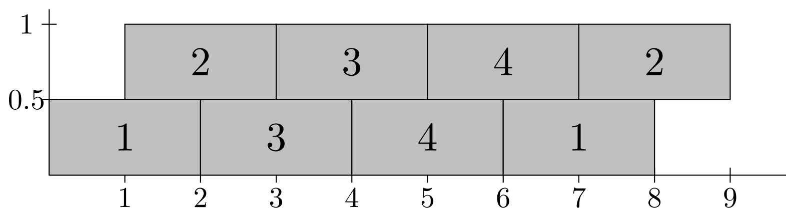

Figure 2.3: Optimal LP solution that is double-nested.

2.5 Arbitrary due dates

Now we consider the situation where all jobs can have arbitrary due dates. In this case we cannot provide a general polynomial time algorithm for solving the minimization problem. We will show that if certain properties hold for the jobs a polynomial time algorithm can be used to solve the problem, while for the other cases we suggest a branching strategy that makes use of the structure of the problem.

In the case of arbitrary due dates, and unlike the case with one common due date, a fractional solution of the LP-relaxation can have a better value than the optimal integer solution. The following example shows that this can indeed occur.

Job	Release date	Due date	Weight	Processing time
1	0	8	100	2
2	1	3	1	2
3	2	5	100	2
4	4	7	100	2

The optimal LP solution with cost 3 is shown in Figure 2.3. Hower, the possible integral solutions are:

- start with job J_1. After that process job J_3 and job J_4 and finally process job J_2, which will be tardy. This solution has cost 5.
- start with job J_2. After that one of the jobs J_1, J_3, and J_4 will become

tardy and due to the large weight of these jobs, the cost of this solution will be greater than 5.

- start with job J_3 or J_4. Since there is enough free capacity to put job J_1 in front without causing a delay, this is dominated by the solution in which job J_1 is put first.

Thus the optimal ILP solution value equals 5, which is different from the optimal LP solution with value 3. Furthermore, we can see in Figure 2.3 that the jobs J_1 and J_2 in the optimal LP solution are double-nested.

In the following, we analyze when it can be optimal for two jobs J_{j_1} and J_{j_2} to be double-nested; we will derive conditions under which such a double-nested situation can be shown to be sub-optimal. We number the jobs such that $d_{j_1} \leq d_{j_2}$. There are two possible double-nested situations for the two jobs:

- Situation 1212
- Situation 2121

In both cases, we assume that the jobs have been assigned in parts to intervals t_1, t_2, t_3, t_4 with $t_1 < t_2 < t_3 < t_4$.

Lemma 2.5.1. *In the 1212 situation a double-nested solution is either sub-optimal, or it can be rewritten into a nested solution with equal cost.*

Proof. Assume $d_{j_2} \geq t_3$. In this case job J_{j_2} will be on time when completed at time t_3. Exchanging ε between x_{j_2,t_2} and x_{j_1,t_3} is profitable or yields the same cost, since:

- Job J_{j_2} will still be on time.
- Job J_{j_1} is placed earlier in time and thus its cost can only decrease or remain the same.

Now assume $d_{j_2} < t_3$. In this case both jobs are tardy at times t_3 and t_4. We have to distinguish between two separate cases, based on the weight of the jobs:

- $w_{j_1} \geq w_{j_2}$. A straightforward calculation shows that, because $d_{j_1} \leq d_{j_2}$ and $w_{j_1} \geq w_{j_2}$, the value of the objective function will not increase when exchanging ϵ between x_{j_2,t_2} and x_{j_1,t_3}.

- $w_{j_1} < w_{j_2}$. Since both jobs are tardy when completed at time t_3, exchanging ε between x_{j_1,t_3} and x_{j_2,t_4} will increase the cost for job J_{j_1} by $\epsilon w_{j_1}(t_4 - t_3)$ and decrease the cost for job J_{j_2} by $\epsilon w_{j_2}(t_4 - t_3)$. Since $w_{j_1} < w_{j_2}$, the total cost decreases.

Hence, when $d_{j_1} \leq d_{j_2}$, the situation 1212 can always be rewritten to a nested solution of equal or less cost. □

In the case of $d_{j_1} = d_{j_2}$ we can choose the situation (1212 or 2121) arbitrarily and thus we can always choose the 1212-situation. This means that the case of $d_{j_1} = d_{j_2}$ can always be rewritten to a nested solution of equal or smaller cost.

Lemma 2.5.2. *In the 2121 situation a double-nested solution is always sub-optimal, or it can be rewritten into a nested solution of the same cost, unless* $w_{j_1} < w_{j_2}$ *and* $r_{j_1} > r_{j_2}$.

Proof. When $w_{j_1} = w_{j_2}$, we can always transform the current solution into a new solution of equal or smaller cost by exchanging parts of the jobs between the intervals t_3 and t_4, since $d_{j_1} < d_{j_2}$. We now have to check the cases where the weights of the jobs are not equal.

Assume $w_{j_1} > w_{j_2}$. Now there are two options:

- $d_{j_2} \geq t_2$. Because $w_{j_1} > w_{j_2}$ and $d_{j_1} < d_{j_2}$ it will be profitable to exchange a bit of the last two parts of the jobs: x_{j_2,t_3} and x_{j_1,t_4}. This exchange will never cause an increase in cost.

- $d_{j_2} < t_2$. In this case both jobs are tardy at times t_3 and t_4. Because $w_{j_1} > w_{j_2}$ it will always be profitable to exchange between x_{j_2,t_3} and x_{j_1,t_4}.

Now assume $w_{j_1} < w_{j_2}$. Again there are two options:

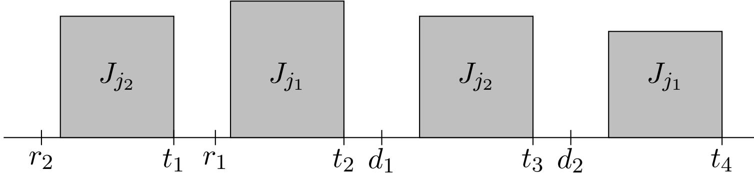

Figure 2.4: Example of situation where rewriting is not profitable.

- $d_{j_2} < t_2$. In this case both jobs are tardy at times t_2, t_3, and t_4. It will always be profitable to exchange x_{1,t_2} and x_{2,t_3}, because one wants the job with the larger weight (J_{j_2} in this case) more to the front.

- $d_{j_2} \geq t_2$. In this case job J_{j_2} will be on time at time t_2. Generally speaking we would like to exchange x_{j_2,t_1} and x_{j_1,t_2} because job J_{j_2} is still on time at time t_2 and job J_{j_1} is put earlier which is always at least as profitable. This is only guaranteed to be possible when $r_{j_1} \leq t_1 - p$ holds. This condition holds for sure when $r_{j_1} \leq r_{j_2}$.

The only case when the situation 2121 cannot always be improved or changed for free into a nested solution is when $d_{j_1} < d_{j_2}$, $w_{j_1} < w_{j_2}$, and $r_{j_1} > r_{j_2}$. An example of such a situation is depicted in Figure 2.4. If w_{j_2} is greater than or equal to $\frac{(t_4-t_3)w_{j_1}}{(t_4-d_{j_2})}$, exchanging between the two intervals will not be profitable. □

Theorem 2.5.3. *When there are no two jobs J_{j_1} and J_{j_2} with $d_{j_1} < d_{j_2}$, $w_{j_1} < w_{j_2}$, and $r_{j_1} > r_{j_2}$, then the problem $1|r_j, p_j = p|\sum w_j T_j$ can be solved in polynomial time.*

Proof. Solve the LP-relaxation. When no pair of jobs exist for which the condition holds, whereas the LP-relaxation yields a double-nested solution, then we can convert it into a non-double-nested solution by combining Lemma 2.5.1 and Lemma 2.5.2. We can then apply Lemma 2.3.2 to this nested, fractional solution to convert it to an integral solution with equal cost. □

Corollary 2.5.4. *The special cases of $1|r_j|\sum w_j T_j$ in which either all release dates, all weights, or all due dates are equal is solvable in polynomial time.*

The two cases with either all release dates being equal or all weights being equal are known to be solvable in polynomial time: Baptiste (2000) presents a polynomial time algorithm for the $1|r_j, p_j = p|\sum T_j$ problem, and it can easily be seen that the $1|p_j = p|\sum w_j T_j$ problem can be solved as an assignment problem.

2.5.1 Branching rule

In case we have a double nested, optimal fractional solution that cannot be converted into a nested solution with equal cost by using the rewrite rule presented in Section 2.3, we will make use of *branch-and-bound* for finding the optimal integral solution.

First we define a pair of two jobs J_{j_1} and J_{j_2} to be a *complicating* pair if their weights, due dates, and release dates satisfy the following condition:

$$d_{j_1} < d_{j_2},\ w_{j_1} < w_{j_2}, \text{ and } r_{j_1} > r_{j_2}.$$

There is only need for branching when we have a fractional solution in which there is a complicating pair of jobs J_{j_1} and J_{j_2} that are double-nested, and in which at least one of the jobs has a tardy assignment.

When we have a complicating pair of jobs J_{j_1} and J_{j_2} that are double-nested, and of which at least one part of the jobs is tardy, then we need to branch. We branch by creating two sub-nodes from the current node by dividing the execution interval of job J_{j_2} into two parts:

- A node where job J_{j_2} has deadline $r_{j_1} + p - 1$.
- A node where job J_{j_2} has release date r_{j_1}.

This means that job J_{j_2} will either start before the release date of job J_{j_1} or it will start at or after it. In the first case we ensure that the two jobs cannot be processed in the same time interval, whereas in the second case we create a new version of the problem where the release dates of the two jobs have become equal and thus in this new version these two jobs are not a complicating pair anymore.

There could be several possibilities for selecting the complicating pair of jobs to branch on. In this case we select to branch on the complicating pair of jobs that span the largest number of jobs, which is defined as the number of (partial) assignments to intervals that lie between the first completion interval and the last completion interval of the two jobs of the pair. The larger this number, the more jobs are influenced by branching on this complicating pair of jobs.

Furthermore, contrary to generic branching rules, this branching rule uses specific information about the structure of the weighted tardiness problem. Typically the usage of such additional information results in fewer branches needed to find the optimal integral solution because unnecessary branches are less likely.

2.6 Related objective functions

Considering other objective functions besides total weighted tardiness, we have found that the approach by Verma and Dessouky (1998) can also be used for the following objective functions:

- Total weighted number of late jobs $\sum w_j U_j$
- Total weighted late work $\sum w_j V_j$, where $V_j = \max\{0, \min\{p_j, C_j - d_j\}\}$.

We will discuss both objective functions in more detail below.

Total weighted number of late jobs

When we look at the weighted number of late jobs problem $1|r_j, p_j = p|\sum w_j U_j$, the cost of having job J_j completed at time t is:

$$c_{j,t} = \begin{cases} w_j & \text{if } t > d_j \\ 0 & \text{otherwise.} \end{cases}$$

Our approach is again to solve the LP-relaxation and, if fractional, to try and convert the solution into a non-double-nested one with equal cost from which we then derive an optimal integral solution. Unfortunately, a solution in which

J_{j_1} and J_{j_2} are double-nested can be better than a non-double-nested solution in case $d_{j_1} < d_{j_2}$ and $r_{j_1} > r_{j_2}$ regardless of the weights w_{j_1} and w_{j_2} and we will have to use branching again to find the optimal integral solution. Baptiste (1999) presents a dynamic programming algorithm that solves the total weighted number of late jobs problem in polynomial time ($\mathcal{O}(n^7)$).

Suppose that the jobs J_{j_1} and J_{j_2} are double-nested. We apply the same analysis as in Section 2.5 to remove double nestings, if possible. The proofs for Lemma 2.5.1 and Lemma 2.5.2 also hold for this objective function with the following two changes:

- Due to the binary character of this objective function (i.e. a job is either late or not and if it is late it does not matter how late the job is) exchanging parts of the jobs between two intervals will not always yield a decrease but never an increase in cost.
- The case where $w_{j_1} \geq w_{j_2}$ cannot always be rewritten in the 2121 situation. The reason for this is again the binary character of the objective function. In case of total weighted tardiness an exchange between two jobs both assigned to intervals that are later than their due date (*tardy intervals*) never increases the objective function, since d_{j_1} is strictly smaller than d_{j_2}. In case of weighted number of late jobs it is not guaranteed that this exchange does not increase the objective function. An example of a situation where the objective function increases is the same as the one depicted in Figure 2.4 in Section 2.5. An exchange between the last two intervals will increase the cost because job J_{j_1} will still be tardy and job J_{j_2} will become tardy after the exchange. An exchange between the middle two intervals will also increase the cost since job J_{j_1} will become tardy while job J_{j_2} will still be on time after the exchange.

We see that regardless of the weights, the 2121 situation cannot always be rewritten into a non-double-nested situation. Therefore, we have to redefine the concept of a complicating pair of jobs for this objective function, as follows. A pair of two jobs J_{j_1} and J_{j_2} is said to be *complicating* if their due dates and release dates satisfy the following condition:

$$d_{j_1} < d_{j_2} \text{ and } r_{j_1} > r_{j_2}.$$

We need to extend the branching rule introduced in Section 2.5.1 a little. From

initial experiments we saw that some instances took very long to solve. Looking at the branching information we saw that this was caused by jobs with a small time interval for being on time (i.e. a job J_j with $d_j - r_j < 2p$). If such a job J_j existed and was partially assigned to an interval before or at its due date (*on-time interval*) and a tardy interval, then job J_j would be always selected as a possible candidate for branching with another job $J_{j'}$, where J_j and $J_{j'}$ are a complicating pair of jobs. To try and prevent such unnecessary branching, we change the branching rule in the following way: In a node we first check whether there exists a job J_j with $d_j - r_j < 2p$ and both tardy and non-tardy completion times. If such a job exists, we create two child nodes from the current node:

- A child node where job J_j is on-time
- A child node where job J_j is tardy.

If such a job does not exist, we go on with the branching rule suggested in Section 2.5.1.

Total weighted late work

When we look at the total weighted late work problem $1|r_j, p_j = p|\sum w_j V_j$, the cost for a job J_j being completed at the t is:

$$c_{j,t} = w_j \max\{0, \min\{p_j, t - d_j\}\}.$$

Since this objective in its most extreme form (when $C_j > d_j + p$ for a job J_j) behaves similarly to the weighted number of late jobs (cost will not increase anymore by placing job J_j even later), we find that also for this objective function we cannot guarantee that the 2121 situation can always be rewritten into a nested solution of equal or smaller cost; the counterexample is the same as the one given for the sum of weighted late jobs, depicted in Figure 2.4. This means that also for this objective function we have to redefine the concept of a complicating pair of jobs, in the same way as for the case of weighted number of late jobs.

For both of these objective functions the case where the release dates and the due dates are equally ordered (i.e. jobs can be reindexed such that $r_{j_1} \leq r_{j_2} \leq$

$\ldots \leq r_{j_n}$ and $d_{j_1} \leq d_{j_2} \leq \ldots \leq d_{j_n}$) means that no complicating pair of jobs can exist, which means that the solution to the LP relaxation will either be already integral, or the fractional solution can be rewritten to an integral solution of same cost. This means that the cases where an equal ordering exists are polynomially solvable.

2.7 Parallel identical machines

A nice advantage of the time-indexed formulation is that it can be used for problems with m parallel, identical machines as well. In this machine environment, there are m machines available, and a job J_j can be processed by any one of these machines, using an uninterrupted period of length $p_j = p$ independent of the machine that executes it. We can formulate the problem as an ILP problem by using assignment variables $x_{j,t}$, which model the decision of executing job J_j in period t. The only difference with the single-machine variant is that we now can process m jobs at the same time, which can be modeled by adjusting the right-hand side in constraint (2.2) from 1 to m. This leads to the following ILP-formulation

$$\min z = \sum_{j=1}^{n} \sum_{t \in A_j} c_{j,t} x_{j,t}$$

subject to:

$$\sum_{t \in A_j} x_{j,t} = 1 \text{ for all } j \in N \tag{2.4}$$

$$\sum_{j=1}^{n} \sum_{t' \in H_t \cap A_j} x_{j,t'} \leq m \text{ for all } t \in K \cup G \tag{2.5}$$

$$x_{j,t} \in \{0,1\} \text{ for all } j \in N,\, t \in A_j. \tag{2.6}$$

It is readily verified that any integral solution can be converted into a feasible schedule by greedily assigning job J_j to any machine that is available in the period t for which $x_{j,t} = 1$.

Since the constraint matrix has not been changed, Lemma 2.2.1 still holds. Hence, we have the following corollary.

Corollary 2.7.1. *If any double-nested solution to the LP-relaxation is either suboptimal or can be rewritten to a nested solution with equal cost, then there exists an optimal solution to the LP-relaxation that is integral.*

Since the proofs that we have given earlier in this section and in Sections 2.4 and 2.5 to show how to modify double-nested solutions do not depend on the number of machines involved, it follows that all our previous results concerning the resolution of double-nested solutions still hold. Therefore, the only thing left to do is to find the integral optimal solution in case the LP-relaxation yields a fractional solution (which must be double-nested). Unfortunately, the algorithm presented in Section 2.3 for the single-machine case to convert a nested, fractional solution into an integral solution with equal cost could not be generalized to the m-machine case. We can, however, use the method presented by Verma and Dessouky (1998), which consists of adding a small perturbation to all assignment cost $c_{j,t}$. This ensures that each extremal solution has a different solution value, which implies that we will always end up in an extremal, and hence optimal, solution when solving the LP-relaxation.

2.8 Column generation

A big disadvantage of the time-indexed formulation of the minimization of the weighted tardiness problem is that it uses huge amounts of memory. To see whether it is possible to reduce the amount of memory needed for solving the problems we consider the use of column generation.

The idea here is to split up the time horizon into B time-frames. These time-frames do not necessarily need to have the same length but must have a length $\geq p-1$. This idea of dividing the complete time horizon has also been suggested by Bigras et al. (2005) for solving the $1||w_jT_j$ problem by means of column generation. All time-frames combined should cover the complete time horizon completely, and consecutive time-frames may overlap at most $p-1$ time units. Each time frame b ($b = 1, \ldots, B$) has a start time μ_b and and end time ω_b. Without loss of generality we create the time-frames in such a way that two consecutive time-frames have an overlap of exactly $p-1$ (i.e. $\mu_{b+1}+p-1 = \omega_b$).

Given a division of the time horizon into a set of time-frames, we compute for

each time-frame a feasible schedule for a subset of the jobs, such that each job is executed in exactly one time-frame. We formulate this problem as an ILP using the concept of *time-frame plans*. A time-frame plan for a given time-frame consists of the completion times for a subset of the jobs with the following properties. The completion times must be feasible within the time-frame, meaning that the jobs can actually be completed at their respective completion times, and all completion times must lie within the boundaries of the time frame.

If we would have all possible time-frame plans, we can formulate the weighted tardiness problem as follows: select a suitable time-frame plan for each of the time-frames such that all jobs are processed in exactly one of the selected time-frame plans and the total weighted tardiness is minimal. Furthermore, we also have to make sure that we do not violate the capacity constraint in the part where two time-frames overlap. This can be achieved by demanding the total idle time in the part where two time-frames overlap to be $>= p - 1$.

Let s denote the number of possible time-frame plans. For each time-frame plan i we introduce the indicators y_{ji} and $q_{b,i}$, defined as follows:

$$
\begin{aligned}
y_{ji} &= \begin{cases} 1 & \text{if job } J_j \text{ is processed in time-frame plan } i \\ 0 & \text{otherwise,} \end{cases} \\
q_{b,i} &= \begin{cases} 1 & \text{if time-frame plan } i \text{ is for time-frame } b \\ 0 & \text{otherwise,} \end{cases}
\end{aligned}
$$

We denote the cost of time-frame plan i by c_i, and we use y_i and z_i to denote the amount of idle time in the *front overlap* and *end overlap* of the time-frame plan i respectively. The front overlap of a time-frame plan is the part of the time-frame that has an overlap with the time-frame that is before this time-frame, while the end overlap is the part of the time-frame that has a n overlap with the time-frame that is right after this time-frame. We introduce the binary variable x_i to indicate whether or not we include time-frame plan i in our solution. Now the complete ILP model is as follows:

$$
\min \sum_{i=1}^{s} c_i x_i
$$

subject to:

$$\sum_{i=1}^{s} y_{ji} x_i = 1 \qquad j = 1 \dots n \tag{2.7}$$

$$\sum_{i=1}^{s} q_{b,i} x_i = 1 \qquad b = 1 \dots B \tag{2.8}$$

$$\sum_{i=1}^{s} z_i q_{b,i} x_i + \sum_{i=1}^{m} y_i q_{b+1,i} x_i \geq p - 1 \qquad b = 1 \dots B - 1 \tag{2.9}$$

$$x_i \in \{0, 1\} \ \forall i. \tag{2.10}$$

Constraint (2.7) ensures that all jobs will be present in exactly one selected time-frame plan and constraint (2.8) ensures that for each time-frame exactly one time-frame plan will be selected. Finally constraint (2.9) ensures that the idle time in the overlap between two consecutive time-frame plans is at least $p - 1$.

Solving the model with column generation

To approximate the optimum of this integral model, we first relax the integrality constraint (2.10) to $x_i \geq 0$ (the other constraints ensure $x_i \leq 1$). The resulting LP-relaxation we will solve with column generation.

In each iteration of the column generation process we only allow a subset of the time-frame plans and solve the LP-relaxation for this restricted set of variables. We then solve the corresponding *pricing* problem to find out whether there are time-frame plans the addition of which will reduce the objective function. The pricing problem is defined as finding the feasible time-frame plan with minimum reduced cost; if this minimum is non-negative, then we have solved the LP-relaxation to optimality. To compute the reduced cost we need the dual multipliers.

For each type of constraint in the master problem we have a dual multiplier:

- From (2.7) we get a dual multiplier π_j for each job J_j.
- From (2.8) we get a dual multiplier τ_b.

- From (2.9) we get a dual multiplier ϕ_b for each overlap between time-frame b and $b+1$.

We solve the pricing problem for each time-frame individually. It can be solved in a similar fashion as the original problem: for a given time-frame we define the set of possible completion times and then assign these to eligible jobs such that the capacity constraints are satisfied. The changes that have to be made for solving the pricing problem for a given time-frame b are:

- We can discard any completion time $< \mu_b + p$ or $> \omega_b$ and create the following sets:
  - $A_j^b = \{a \in A_j | \mu_b + p \leq a \leq \omega_b\}$ $(j = 1, \ldots, n)$ denoting the possible completion times of job J_j within time-frame b.
  - $H_t^b = \{h \in H_t | \mu_b \leq h \leq \omega_b + p\}$ for all times t denoting the time indices within time-frame b that are conflicting with time t.
  - $G^b = \{g \in G | \mu_b \leq g \leq \omega_b + p\}$ denoting all first possible completion times of all jobs within time-frame b.
  - $K^b = \{k \in K | \mu_b \leq k \leq \omega_b + p\}$ denoting all other possible completion times of all jobs that are within time-frame b.
- We are interested in jobs that are processed in the time-frame b. Possibly not all jobs can be processed, so we relax the constraint that all jobs must be processed exactly once to the constraint that all jobs must be processed at most once.
- Since we need to know something about the available idle time in the overlapping regions, we add two extra jobs α_b and β_b, both with processing time p, for the front overlap and the end overlap of a time-frame respectively. The possible completion times for these two extra jobs are, respectively:
  - $A_\alpha^b : \{\mu_b, \mu_b + 1, \ldots, \mu_b + p - 1\}$.
  - $A_\beta^b : \{\omega_b + 1, \omega_b + 2, \ldots, \omega_b + p\}$.

  It can be seen that these jobs have a special status, since job α starts before the start time of the time-frame and job β ends after the end time of the time-frame. For the first time-frame job α does not exist and for the last frame job β does not exist, because there is no time-frame to overlap with.

Solving the pricing problem should result in a time-frame plan with minimum reduced cost. Hence, we need to set the cost of job J_j being completed at time t such that it corresponds exactly to its contribution to the reduced cost. If a job J_j is not assigned in this time-frame, it does not contribute to the cost function; otherwise, the cost $c_{j,t}$ of assigning job J_j to time t is defined as follows:

$$c_{j,t} = w_j \max\{0, t - d_j\} - \pi_j.$$

Furthermore the cost for assigning the α_b and β_b jobs to a time t are as follows:

$$c_{\alpha_b,t} = (t - \mu_b)\phi_{b+1},$$

$$c_{\beta_b,t} = (\omega_b + p - t)\phi_b,$$

which corresponds to the amount of idle time at the front and the back of the time-frame multiplied by the corresponding dual multiplier, respectively.

The pricing problem we need to solve for a certain time-frame b is as follows:

$$\min \sum_{j=1}^{n} \sum_{t \in A_j^b} c_{j,t} x_{j,t} - \tau_b - \sum_{t \in A_\beta^b} c_{\beta_b,t} x_{\beta_b,t} - \sum_{t \in A_\alpha^b} c_{\alpha_b,t} x_{\alpha_b,t}$$

subject to:

$$\sum_{t \in A_j^b} x_{j,t} \leq 1 \text{ for all } j \in N \quad (2.11)$$

$$\sum_{t \in A_\alpha^b} x_{\alpha_b,t} = 1 \quad (2.12)$$

$$\sum_{t \in A_\beta^b} x_{\beta_b,t} = 1 \quad (2.13)$$

$$\sum_{j=1}^{n} \sum_{t' \in H_t^b \cap A_j^b} x_{j,t'} + \sum_{t' \in H_t^b \cap A_\alpha^b} x_{\alpha_b,t'} + \sum_{t' \in H_t^b \cap A_\beta^b} x_{\beta_b,t'} \leq 1 \text{ for all } t \in K^b \cup G^b \quad (2.14)$$

$$x_{j,t} \in \{0,1\}, \text{for } j \in N \cup \{\alpha, \beta\}, t \in A_j^b. \quad (2.15)$$

Constraint (2.11) ensures that all regular jobs are processed at most once and constraint (2.12) and constraint (2.13) ensure that the α-job and β-job are

processed exactly once. Finally constraint (2.14) ensures that there does not exist a time where more than one job is processed.

When the pricing problem is solved we know which jobs are assigned to which time intervals and we can calculate the cost c_i of the new variable x_i in the master problem as follows

$$\sum_{j=1}^{n} \sum_{t \in A_j^b} c_{j,t} x_{j,t}.$$

Furthermore the two indicators y_i and z_i denoting the idle time in the front overlap region and the rear overlap region can be set according to the assignment of the α and β job.

For creating the initial variables for the master problem we order the jobs by their release dates and assigned them to the earliest possible time interval. From this integral feasible schedule we determined which jobs were processed in which time intervals and from this we created the initial variables. When the LP-relaxation is solved to optimality with the column generation, we add the integrality constraint back again to the master problem and then try to solve the master problem again with the generated set of columns.

Initial tests with a prototype implementation showed that the solutions given by solving the LP-relaxation with column generation often had a fractional solution value lower than the integral solution given by the model introduced in Section 2.2. Because solving the pricing problems consists of solving relatively small problems which do not require large amounts of memory and because the column generation model itself does require large amounts of memory, the total amount of memory needed for solving each problem is less than the memory required for the model introduced in Section 2.2. One major disadvantage the tests showed was that the running time for solving the LP relaxation to optimality was considerably larger compared to the running time needed for solving the same problem with the model introduced in Section 2.2. However, since the problem is divided into a set of smaller, independent, subproblems, we can easily parallelize the solving of the subproblems. Since this LP-relaxation gave a much weaker lower bound, one approach to solve the ILP problem to optimality would be to make use of branch-and-price.

2.9 Computational experiments

For testing the time-indexed formulation, the branching rule, and the rewrite rule a program was written in Java. All tests were run on a Pentium 4 running at 3.00 Ghz with 1 GB of RAM. For solving the ILPs we used CPLEX 9.1(ILOG, 2005). We wanted to create some problems varying in size and did this by choosing the number of jobs n from the set $\{70, 80, 90, 100\}$ and the common processing time p from the set $\{5, 10, 15, 20, 25, 30\}$. For each combination of these two parameters we created 50 instances in the following way:

- r_j was randomly selected from the interval $[0, (n-6)p)$
- w_j was randomly selected from the interval $[0, 120)$
- d_j was randomly selected from the interval $[r_j + p, (n-5)p)$.

This resulted in a total of 1200 instances. Each of these instances was then solved for the three objective functions sum of weighted tardiness, sum of weighted late jobs, and sum of weighted late work.

The results of the tests for the weighted tardiness problem can be found in Table 2.1. The first two columns denote the number of jobs and the common processing time. The next two columns give the average time in milliseconds needed for solving the instances by using pure CPLEX without any extra information in the first column and by using CPLEX enhanced with the branching rule and rewrite rule in the second. The following two give the average number of nodes that needed to be explored before the optimum was found. The next two give the average number of iterations needed before the problem was solved to optimality. The final column gives the average integrality gap for the set of instances.

It can be clearly seen from the table that the average integrality gap is very small to completely non-existent, denoting that the relaxed time-indexed formulation gives a really strong bound for the solution. This can also be seen by looking at the average number of nodes that needed to be explored before the optimum was found.

Jobs	P	Avg. time for solving (ms) CPLEX	B.R.	Avg. number of nodes CPLEX	B.R.	Avg. number of iterations CPLEX	B.R.	Avg. int. gap (‰)
70	5	1891.6	1573.1	0.12	0.00	985.1	944.7	0.8
70	10	4780.1	4036.7	0.00	0.00	1721.2	1641.2	=
70	15	10220.3	6946.7	0.14	0.00	2021.2	1853.6	0.0
70	20	14435.7	10444.1	0.16	0.00	2365.0	2213.5	=
70	25	19684.8	15678.4	0.22	0.04	2721.2	2562.2	0.1
70	30	27206.7	17545.4	0.82	0.04	2897.8	2584.8	0.2
80	5	2840.6	2264.3	0.48	0.00	1334.7	1251.3	0.2
80	10	7877.8	5957.8	0.00	0.00	2002.3	1877.0	0.0
80	15	14807.5	10731.1	0.10	0.00	2454.9	2304.5	0.2
80	20	27111.8	16999.6	0.30	0.00	3043.6	2756.8	0.0
80	25	36674.2	20459.5	0.48	0.00	3237.0	2804.3	=
80	30	37105.6	25986.9	0.24	0.02	3363.5	3075.0	0.2
90	5	4144.5	2926.9	0.00	0.00	1527.3	1404.3	0.0
90	10	13632.4	9153.2	0.20	0.02	2458.7	2251.2	0.1
90	15	25802.9	15212.9	0.40	0.00	3146.6	2758.1	=
90	20	37260.0	23841.7	0.32	0.00	3578.2	3257.1	=
90	25	47910.1	31606.5	0.20	0.00	3939.4	3579.9	=
90	30	72489.8	42054.8	0.80	0.00	4322.4	3817.0	=
100	5	5154.0	4183.3	0.00	0.00	1754.1	1674.8	0.0
100	10	16928.8	12230.3	0.30	0.00	2805.6	2608.7	=
100	15	32375.2	23278.0	0.18	0.00	3566.2	3346.5	0.0
100	20	61929.3	34486.5	1.02	0.00	4463.5	3874.0	=
100	25	100738.1	50501.5	1.12	0.00	5326.6	4302.4	0.0
100	30	122807.8	75730.5	0.44	0.08	5655.2	4979.7	0.2

CPLEX = Results CPLEX 9.1 B.R. = Results with branching rule and rewrite rule
= : Integrality gap was 0 for all instances.

Table 2.1: Results for weighted tardiness.

		Avg. time for solving (ms)		Avg. number of nodes		Avg. number of iterations		Avg. int. gap (‰)
Jobs	P	CPLEX	B.R.	CPLEX	B.R.	CPLEX	B.R.	
70	5	2173.8	1614.4	0.28	0.00	925.3	824.5	19.0
70	10	6336.4	4704.9	0.22	0.04	1584.5	1410.8	5.2
70	15	15812.2	9536.7	24.12	0.18	1853.0	1590.3	11.9
70	20	19019.8	14309.1	2.08	4.06	2177.2	2005.3	7.1
70	25	32457.7	20199.0	33.20	0.14	2592.6	2149.7	18.1
70	30	32067.4	21591.1	0.50	0.10	2335.5	2082.8	4.0
80	5	4115.7	2787.9	17.38	0.14	1251.3	1098.2	14.1
80	10	11372.6	6780.6	14.86	0.04	1903.2	1623.1	1.3
80	15	19108.0	11399.2	0.36	0.00	2292.6	1943.3	0.8
80	20	39983.6	21466.4	23.50	0.34	2880.2	2369.9	23.1
80	25	57937.8	38194.2	2.58	0.36	3281.2	2534.7	13.1
80	30	56392.8	45024.9	2.48	15.24	3182.8	3000.3	4.7
90	5	5204.2	3042.5	0.56	0.00	1436.7	1226.8	11.6
90	10	18414.4	9015.2	1.04	0.00	2364.1	1901.8	3.5
90	15	45076.9	21474.6	19.16	0.12	3167.8	2452.9	17.0
90	20	55830.7	29861.6	0.94	0.18	3482.4	2812.4	12.9
90	25	96773.2	44125.8	40.16	0.14	4263.7	3109.6	0.9
* 90	30	210834.7	74404.1	365.88	0.60	6117.1	3750.8	23.2
100	5	6671.2	4945.9	1.38	0.10	1683.7	1495.8	9.0
100	10	36198.6	18203.2	48.52	0.20	3201.3	2416.1	10.9
100	15	53255.3	26856.2	1.64	0.04	3546.7	2911.3	7.3
100	20	125703.8	44362.7	3.72	0.24	4959.4	3313.7	13.9
100	25	189431.8	78217.9	356.50	0.84	7652.9	4092.9	4.2
*100	30	374531.2	173092.3	326.68	1.04	11306.3	5049.8	21.4

CPLEX = Results CPLEX 9.1 B.R. = Results with branching rule and rewrite rule
*CPLEX could not solve on or more instances.

Table 2.2: Results for weighted late jobs.

		Avg. time for solving (ms)		Avg. number of nodes		Avg. number of iterations		Avg. int. gap (‰)
Jobs	P	CPLEX	B.R.	CPLEX	B.R.	CPLEX	B.R.	
70	5	2080.7	1489.9	0.00	0.00	901.7	816.6	0.9
70	10	5456.0	3844.5	0.00	0.00	1534.3	1380.6	1.3
70	15	12834.6	7676.1	10.92	0.04	1868.0	1637.0	1.0
70	20	16126.8	10842.4	0.12	0.40	2077.1	1877.0	0.4
70	25	30571.5	19438.8	21.00	0.22	2511.4	2217.9	3.1
70	30	28914.5	18249.9	2.86	0.24	2495.6	2169.9	3.7
80	5	3395.7	2322.5	0.54	0.08	1237.9	1081.6	0.9
80	10	10167.5	6547.8	0.36	0.02	1971.6	1695.8	1.2
80	15	16948.3	11093.6	0.24	0.06	2254.7	2026.5	0.2
80	20	27690.6	17443.7	0.28	0.08	2679.9	2351.7	0.8
80	25	73342.7	31503.1	60.88	0.24	3513.0	2628.8	3.3
80	30	59537.1	26996.2	45.68	0.22	3221.0	2599.4	0.7
90	5	5406.2	2836.4	1.26	0.00	1463.5	1228.1	0.2
90	10	17117.9	10031.4	1.06	0.34	2462.5	2007.4	1.1
90	15	31236.9	15361.5	0.72	0.00	2821.8	2371.8	0.2
90	20	55840.7	24638.0	0.80	0.00	3455.4	2759.3	0.0
90	25	74709.7	34966.3	12.50	0.28	3983.9	3074.8	1.2
90	30	100073.2	48197.2	1.28	0.06	4391.5	3459.7	1.0
100	5	5781.4	4376.0	0.68	0.06	1617.7	1468.3	4.4
100	10	29789.3	13383.6	5.40	0.06	3111.6	2428.2	4.2
100	15	46590.9	32824.5	1.30	1.50	3574.3	3086.6	1.4
100	20	95535.3	36192.4	1.96	0.08	4732.1	3283.8	2.3
100	25	184365.4	67680.5	3.98	0.42	6221.9	4079.9	0.9
100	30	181203.7	84013.3	3.24	0.48	5599.7	4349.7	2.8

CPLEX = Results CPLEX 9.1 B.R. = Results with branching rule and rewrite rule

Table 2.3: Results for weighted late work.

If we look at the first line (70 jobs with $P = 5$) in Table 2.1 we see that with the branching rule and rewrite rule, on average 0 nodes need to be explored before the optimum is found, while there still is an integrality gap. The reason for this is that after solving the root node, CPLEX added some cuts, after which the rewrite rule could convert the new solution to an integral solution. In the majority of the instances, after CPLEX finished solving the root-node LP and possibly added some cuts, the rewrite rule could convert the found fractional solution to an integral solution of same cost. In the cases the rewrite rule could not be applied, the number of nodes that needed to be explored to find the optimum never exceeded 4. The number of nodes that were explored when only CPLEX was used was at most 30 and for all instances this number was always greater than or equal to the number of nodes needed with the branching rule and rewrite rule.

The results for the objective functions weighted number of late jobs and weighted late work can be found in Table 2.2 and Table 2.3 respectively. If we look at

these two tables then we see that for some of the problems, CPLEX on average needs to explore fewer nodes before the optimum is found than our branching rule. In all cases this increased average is caused by a single instance for which our branching rule branches on an unfortunate set of variables. This causes a lot of nodes to be explored before the optimum can be found.

Another observation is that for the number of weighted late jobs problem, for two of the sets of larger instances there exist instances that CPLEX could not solve. Initially these instances were not solvable with the original branching rule either; only after changing the branching rule as mentioned in Section 2.6 these instances could be solved.

The implementation was not optimized for speed and some improvements are still possible. These changes would not improve the number of nodes or number of iterations for any of the problems, but would only decrease the time needed for solving the problems. The more nodes that need to be explored, the bigger the decrease in time would be.

Another observation that was made during the tests, is that the memory needed by pure CPLEX was on average considerably more compared to the memory consumption of CPLEX combined with our branching rule. This can be explained by the fact that CPLEX branches on single variables, while our branching rule branches on a set of variables and thus the branching depth stays lower.

2.10 Conclusion and further research

In this chapter we consider the main $1|r_j, p_j = p|\sum w_j T_j$ problem. We have presented a time-indexed ILP-formulation for this problem and showed that if the instance does not contain a complicating pair of jobs J_i and J_j such that $d_i < d_j$ and $w_i < w_j$ and $r_i > r_j$ then the LP-relaxation can be converted by means of a polynomial time rewrite rule into an optimal integral solution, yielding a polynomial time algorithm for this case. For the general case we present an efficient branch-and-bound algorithm with a suitable branching rule.

Next, we showed that the same approach could be used for the single-machine problems of minimizing total weighted number of late jobs and total weighted

late work, respectively. Here the LP-relaxation can be converted into an optimal integral solution in case no two jobs J_i and J_j exist such that $d_i < d_j$ and $r_i > r_j$, resulting in a polynomial time algorithm for this case.

We further showed that the results obtained for the single-machine problems can directly be translated to the corresponding problems with m parallel, identical machines, except for that we could not generalize the single-machine rewrite rule to the m-machine situation. This can be circumvented by using small perturbations in the cost function.

Finally, we have shown that the technique of column generation can be used to solve this kind of problems in case of a lack of memory to perform the standard ILP algorithm.

There are two clear open questions. The first one is to establish the computational complexity of $1|r_j, p_j = p|\sum w_j T_j$. Although we have given a good characterization of the problem, it is still an open question whether the general $1|r_j, p_j = p|\sum w_j T_j$ problem is polynomially solvable. The second question is to come up with a rewrite rule for the m-machine case and prove its correctness.

CHAPTER

THREE

Resource-constrained project scheduling

3.1 Introduction

In this chapter we consider a number of basic problems from project scheduling. We refer to the survey paper by Brucker et al. (1999) for an overview of this area. We further refer to Van den Akker et al. (2005), Baptiste et al. (2001), and Bazaraa et al. (1990) for an overview of the application of column generation in scheduling, of the application of constraint programming in scheduling, and of linear programming in general, respectively.

The basic resource-constrained project scheduling (RCPS) problem we consider is defined as follows. We are given a set of n jobs, which we denote by $J_1, \ldots, J_n$. For each job J_j we are given its processing time p_j, its release date r_j, its due date d_j which denotes a target completion time, its deadline $\bar{d}_j$, and its resource consumption pattern, which gives the amount of resource needed during its execution; for the time being, we assume that there is only one kind of

resource. For each job J_j we are asked to find a valid starting time S_j and completion time $C_j = S_j + p_j$ such that job J_j does not start before its release date ($S_j \geq r_j$), is completed by its deadline ($C_j \leq \bar{d}_j$), and such that the total resource consumption of the jobs at any time t does not exceed the amount of resource available at that time. Moreover, between each pair of jobs J_i and J_j, there can be generalized precedence constraints, which define a lower bound and/or upper bound on $S_i - S_j$. In case the upper and lower bound are equal, we say that there is a *no-wait* constraint between J_i and J_j. The goal is to minimize either the makespan $C_{\max}$ or the maximum lateness $L_{\max} = \max_j L_j$, where the lateness L_j of job J_j is defined as the difference between the completion time C_j and the due date d_j. In fact, our approach can easily be generalized further to deal with any regular minimax function.

The RCPS problem has received attention from both the Operations Research and the Constraint Programming perspective. We only discuss a few contributions. Brucker and Knust (2000), Brucker and Knust (2002), and Brucker and Knust (2003) apply column generation to a number of RCPS problems in which the goal is to minimize the makespan. Here they first formulate the problem as a decision problem and then use linear programming to check whether it is possible to execute all jobs in a feasible preemptive schedule; here the decision variables refer to the length of a time slice during which the jobs in a given set of jobs are executed simultaneously. Cesta et al. (2002) apply constraint programming to the makespan problem. The key here is to determine a schedule that is feasible for all constraints except for those that deal with the resource consumption. Then resource conflicts are determined and resolved.

Van den Akker et al. (2006) look at the special case of the above model in which the available amount of resources is constant over time (say m) and each job has a constant resource consumption pattern of one, that is, during its execution, it consumes one unit of resource. This problem is then equivalent to the parallel machine scheduling with m parallel, identical machines. Van den Akker et al. (2006) present a column generation based method to solve it, which yields a lower bound that turns out to be tight in all their computational experiments. They further give a method to find a feasible solution with value equal to the lower bound.

The outline for the remainder of this chapter is as follows: in Section 3.2 we briefly discuss the method presented by Van den Akker et al. (2006). After that, in Section 3.3 we provide some general improvements to their method. In

Section 3.4 we consider a number of different RCPS problems and provide the further analysis needed before the method of Van den Akker et al. (2006) can be used. An implementation of the presented algorithms has been written and we provide the results of the computational experiments in Section 3.5. Finally, in Section 3.6 we draw some conclusions and present directions for future research.

3.2 Parallel machine scheduling

In this section, we briefly review the column generation approach presented in Van den Akker et al. (2006) for the problem of minimizing $L_{\max}$ on a set of m parallel, identical machines. Here each job needs exactly one machine during its processing. Furthermore, there are release dates, due dates, deadlines, and generalized precedence constraints. The minimization problem is reduced to a feasibility problem by putting an upper bound L on $L_{\max}$; this is equivalent to adding deadlines $\bar{d}_j \leftarrow d_j + L$ $(j = 1, \ldots, n)$. In case a job already had a deadline, we set the deadline to the earliest of this new deadline and the original deadline. Since a feasible schedule corresponds to a collection of at most m feasible, single-machine schedules containing all n jobs, the decision problem can be reformulated as: *is it possible to partition the jobs in at most m subsets such that for each subset we can find a feasible single-machine schedule*? Finally, the latter decision problem is solved by answering the question: what is the minimum number of feasible single-machine schedules that are needed to accommodate all jobs?

The problem of minimizing the required number of feasible single-machine schedules can be formulated as an integer linear programming problem as follows. We call a subset of jobs that allows a feasible single-machine schedule with respect to the release dates and deadlines a *machine schedule*. Let S be the set containing all machine schedules. We introduce binary variables x_s $(s = 1, \ldots, |S|)$ that take value 1 if machine schedule s is selected and 0 otherwise. For each machine schedule s we encode whether job J_j is included (then $a_{js} = 1$) or not $(a_{js} = 0)$, and we encode the starting times S_{js} of the jobs with $a_{js} = 1$ $(j = 1, \ldots, n)$. We set the value of S_{js} to 0 for those jobs that have $a_{js} = 0$ $(j = 1, \ldots, n)$. Since two jobs that are connected through a precedence constraint do not have to be executed by the same machine, the generalized precedence constraints are not included in the feasibility of the machine schedules. To account for them we include a constraint in the integer linear programming formulation for each of

the generalized precedence constraints. We define A^1 as the arc set containing all pairs (i, j) such there exists a precedence constraint of the form $S_j - S_i \geq q_{ij}$ where we assume the *precedence delay* q_{ij} to be non-negative; similarly, we define A^2 and A^3 as the arc sets that contain an arc for each pair (i, j), for which $S_j - S_i \leq q_{ij}$ and $S_j - S_i = q_{ij}$, respectively. Note that $q_{ij} \geq 0$ and also that the intersection of A^1 and A^2 does not have to be empty. We denote the union of A^1, A^2, and A^3 by the multiset A. This leads to the following integer linear programming formulation

$$\min \quad \sum_{s \in S} x_s$$

subject to:

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n \tag{3.1}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s \geq q_{ij}, \text{ for each } (i, j) \in A^1 \tag{3.2}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s \leq q_{ij}, \text{ for each } (i, j) \in A^2 \tag{3.3}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{s \in S} S_{is} x_s = q_{ij}, \text{ for each } (i, j) \in A^3 \tag{3.4}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S. \tag{3.5}$$

We relax the integrality constraints to $x_s \geq 0$; the upper bound $x_s \leq 1$ follows from the other constraints.

The LP-relaxation is solved by applying column generation. To start, we give each job its own machine. Given the outcome of the current LP, we find dual multipliers λ_j for constraint 3.1 and δ_{ij} for the constraints 3.2-3.4 in which jobs J_i and J_j are involved. The reduced cost of machine schedule s is then equal to

$$c'_s = 1 - \sum_{j=1}^{n} a_{js} \lambda_j - \sum_{j=1}^{n} \left[\sum_{h \in Prec_j} \delta_{hj} S_{js} - \sum_{k \in Suc_j} \delta_{jk} S_{js} \right],$$

where $Prec_j$ and Suc_j are defined as the sets containing all predecessors and successors of job J_j in A, respectively. The *pricing problem* is then to find a machine schedule with minimum reduced cost, which is achieved by maximizing

the last part, which can be rewritten into

$$\sum_{j=1}^{n} a_{js}\lambda_j + \sum_{j=1}^{n} S_{js}\left[\sum_{h\in Prec_j} \delta_{hj} - \sum_{k\in Suc_j} \delta_{jk}\right].$$

If we define $Q_j = \sum_{h\in Prec_j} \delta_{hj} - \sum_{k\in Suc_j} \delta_{jk}$, then solving the pricing problem boils down to finding a machine schedule s that maximizes

$$\sum_{j=1}^{n} a_{js}\lambda_j + \sum_{j=1}^{n} Q_j S_{js}. \tag{3.6}$$

Since solving the LP-relaxation by column generation only renders us a lower bound when the column generation procedure has finished, we compute an intermediate lower bound as

$$\sum_{s\in S} x_s \geq \left[\sum_{j=1}^{n} \lambda_j + \sum_{(j,k)\in A} \delta_{jk} q_{jk}\right] / (1-c^*),$$

where c^* denotes the outcome value of the pricing problem. For the details of this we refer to Van den Akker et al. (2006).

Since the pricing problem is $\mathcal{NP}$-hard to solve, we do not solve it to optimality in each iteration. We apply a two-phase Simulated Annealing procedure to find a good solution. In the first phase, we decide which jobs are included in the machine schedule and in which order. In the second phase, we find the optimal starting times of the included jobs. After that, we change the choices made in phase 1, etc. We mostly use the local search procedure to find good solutions to the pricing problem, but after 50 iterations, or when we cannot find any improving column, we turn to a time-indexed linear programming formulation of the pricing problem. Here we use binary variables x_{jt} to indicate whether job J_j starts at time t or not; the corresponding cost coefficients c_{jt} are easily determined by using the definition of the term 3.6 that we have to maximize. The ILP-formulation then becomes

$$\max \sum_{j=1}^{n} \sum_{t=r_j}^{\bar{d}_j - p_j} c_{jt} x_{jt}$$

subject to:

$$\sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt} \leq 1 \quad , \text{for all } j = 1, \ldots, n \tag{3.7}$$

$$\sum_{j=1}^{n} \sum_{s=t-p_j+1}^{t} x_{js} \leq 1 \quad , \text{for all } t = 0, \ldots, T-1 \tag{3.8}$$

$$x_{jt} \in \{0,1\} \quad , \text{for all } j = 1, \ldots, n; \text{for all } t = r_j, \ldots, \bar{d}_j - p_j.$$

Here T denotes the latest point in time at which at least two jobs can be executed. Constraint 3.7 decrees that each job can be chosen at most once, and constraint 3.8 states that at most one job should be executed at any time. Furthermore, the decision variable x_{jt} gets the value 1 if job j is started at time t and 0 otherwise.

Since we need to find out whether there exists a solution using at most m machines, we stop as soon as the outcome of the LP-relaxation has hit m. If the outcome of the current LP is bigger than m, and we cannot find an improving column, then we can compute the outcome value of the pricing problem that we need such that the intermediate lower bound equals m. We can then ask the ILP-solver whether there exists a solution to the pricing problem with that value or less. If it does not exist, then we have proven that m is not achievable, and we are done; if we can find it, then this is an improving column that we add, etc. In this way, we do not have to solve the time-indexed formulation to optimality.

Finally, when we have found the smallest upper bound L on $L_{\max}$ that cannot be proven impossible, then we try to construct a feasible schedule with $L_{\max}$ equal to L. We formulate the problem of finding an optimal schedule as a time-indexed ILP as follows. Again, x_{jt} is used to indicate whether job J_j starts at time t or not.

$$\min \quad L_{\max}$$

subject to:

$$\begin{aligned}
\sum_{t=r_j}^{\bar{d}_j-p_j} x_{jt} &= 1, \text{ for each } j=1,\ldots,n \\
\sum_{j=1}^{n} \sum_{t=t'-p_j+1}^{t'} x_{jt'} &\leq m, \text{ for each } t'=0,\ldots,T-1 \\
\sum_{t=r_j}^{\bar{d}_j-p_j} t x_{jt} + p_j - d_j &\leq L_{\max}, \text{ for each } j=1,\ldots,n \\
\sum_{t=r_j}^{\bar{d}_j-p_j} t x_{jt} - \sum_{t=r_i}^{\bar{d}_i-p_i} t x_{it} &\geq q_{ij}, \text{ for each } (i,j)\in A^1 \\
\sum_{t=r_j}^{\bar{d}_j-p_j} t x_{jt} - \sum_{t=r_i}^{\bar{d}_i-p_i} t x_{it} &\leq q_{ij}, \text{ for each } (i,j)\in A^2 \\
\sum_{t=r_j}^{\bar{d}_j-p_j} t x_{jt} - \sum_{t=r_i}^{\bar{d}_i-p_i} t x_{it} &= q_{ij}, \text{ for each } (i,j)\in A^3
\end{aligned}$$

$$x_{jt} \in \{0,1\}, \text{ for each } j=1,\ldots,n \text{ and each } t=0,\ldots,T-1.$$

Solving this ILP from scratch only works for small instances. But when we insert our knowledge of the lower bound by adding the constraint $L_{\max} \geq L$, then CPLEX finds a feasible solution rather quickly, if it exists.

3.3 Enhancements

In this section we propose a number of enhancements to the method presented by Van den Akker et al. (2006) to improve both on the time needed for determining a lower bound for the $L_{\max}$ as well as on the time needed for solving the resulting time-indexed formulation problem. The effect of the enhancements on the solution times will be discussed in Section 3.5.

One general enhancement we made is to determine a better lower bound that is used for starting the binary search. As will be seen from the results of the experiments, there even are instances for which this lower bound is already tight.

3.3.1 Speeding up the column generation procedure

One small enhancement made in the implementation we wrote is to not only return the column with minimal reduced cost when the ILP formulation of the pricing problem is solved, but also all intermediate integral solutions that are found by the solver. Each of these integral solutions is a feasible column also.

Another enhancement for the column generation is the use of *column deletion*. This works by checking the model every given number of iterations for columns with reduced cost above a certain threshold. Instead of using a dynamic threshold, as used in chapters 4, 5, and 6, we decided, after initial experiments, to use a static value of 0.08 as a threshold. This means that any column with a reduced cost > 0.08 will be removed from the model. After the initial tests, the frequency of deleting columns from the model was set to every 10 iterations of the column generation procedure. The use of column deletion results in smaller problems in memory, which means less time is needed in each iteration of the column generation procedure for solving the LP problem after adding new columns.

Furthermore, as mentioned at the end of Section 3.2, we can use the intermediate lower bound also the other way around; we can compute the outcome value of the pricing problem that we need such that the intermediate lower bound equals m. Instead of only querying the ILP solver for the existence of a column with the required reduced cost, we can add a constraint to the time-indexed ILP-formulation of the pricing problem that explicitly sets the required value as a bound on the objective every time we solve the time-index ILP-formulation of the pricing problem.

Besides the usage of the intermediate lower bound in the time-indexed ILP-formulation of the pricing problem, we can also use the same approach in the local search procedure, in the following way. Again, first we compute the required outcome value of the pricing problem that we need such that the intermediate lower bound equals m. Then, we partition the columns found by the local search procedure into two different sets: one set contains the *promising columns*, which have values that are greater than or equal to the required outcome value computed earlier, and the other set contains the remainder of the columns. If during the local search procedure the number of promising columns becomes equal to the maximum number of columns we want to add in each iteration

of the column generation procedure, we prematurely end the local search and return the set of promising columns. Initial experiments showed that especially in the beginning this decreases the time used for the local search procedure in each iteration of the column generation procedure.

3.3.2 Propagating knowledge from column generation

Although time-indexed formulations are known for their strong LP-relaxation, an obvious drawback is their size. For each possible starting time we need a variable, and we need a constraint for each time t during which at least $m+1$ jobs are available for execution. Restricting L by computing an upper bound, for instance by constructing a feasible schedule through local search, makes already a big difference.

In our computational experiments, we noticed that if there were 'greater than or equal' precedence constraints only (sets A^2 and A^3 are empty), then the lower bound was always equal to the optimum. This led to the idea that we can increase the release date of job J_j to the earliest start time of J_j in any column that was actually used in the LP-solution of the problem of minimizing the number of subschedules that are needed to accommodate all jobs; similarly, the deadline is decreased to the latest completion time of J_j in any one of the used subschedules. Note that this reduction is a lot stronger than requiring that $L_{\max} \leq L$, in which case the deadline of J_j is set equal to the minimum of the current deadline and $d_j + L$. Obviously, if the outcome value for this new set of release dates and deadlines is greater than L, then it may be possible that there exists a solution with value L. But since we may have found a feasible schedule that dominates the one that we used as an initial upper bound on L it may not be a waste of computing time.

It is also possible to use the knowledge obtained by solving the LP-relaxation with column generation to guide the search in a search tree or a branch-and-bound tree. For example we can use the increased release date and decreased deadline for determining better branches in a branch-and-bound tree.

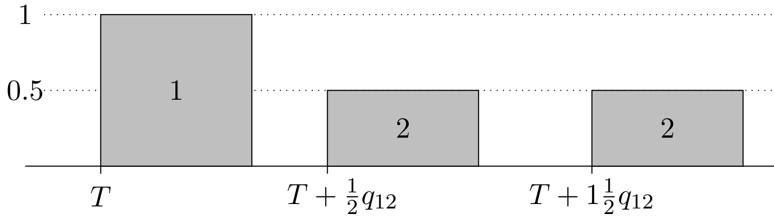

Figure 3.1: Example of averaging of smaller and larger exact precedence delay.

3.3.3 Valid inequalities for the exact precedence delays

In case exact precedence delays are present ($A^3 \neq \emptyset$), one complicating issue that occurred while solving the LP relaxation of the time-indexed formulation was the fact that a fractional solution fulfilled constraint 3.4 by averaging out a small and a large precedence delay. An example of this can be seen in Figure 3.1, where both the first and the second part of job 2 separately do not fulfill the exact precedence delay because the first part starts too early and the second part starts too late. When we look at the combination of the two parts of job 2 we average the starting times of the two parts and we see that this way job 2 does fulfill constraint 3.4 because $0.5 \cdot \frac{1}{2}q_{12} + 0.5 \cdot 1\frac{1}{2}q_{12}$ is exactly equal to q_{12}.

To alleviate this problem we add valid inequalities that prohibit this averaging effect. For each pair of jobs $(i, j) \in A^3$ we add, for all pairs of starting times (t_i, t_j) fulfilling the exact precedence delay, the valid inequality:

$$x_{jt_j} - x_{it_i} = 0. \tag{3.9}$$

For the exact precedence delays this works. Although a similar type of valid inequality is possible for both the $\geq$-precedence and the $\leq$-precedence constraints, initial tests show no performance advantage when valid inequalities were used for these precedence constraints.

3.4 RCPS problems

In this section we consider a number of RCPS problems. We start with the case where there exists only one resource. After that we consider the case with

multiple resources and finally we look at some other possible extensions. For each case we consider what steps are needed to be able to use the solution method presented by (Van den Akker et al., 2006). In some cases a detailed further analysis proves to be needed.

3.4.1 RCPS problems with one type of resource

The jobs can have different types of resource consumption patterns. First we consider the case where each job requires exactly one unit of resource while it is executed. After that, we consider the case where the jobs require an arbitrary integral number of units of resource that is constant while the job is executed. Finally, we consider the case where the resource consumption of a job changes over time.

Unit resource consumption

We first consider the case in which the resource consumption pattern is constantly equal to 1 for each job J_j, but the available amount of nonrenewable resources is not constant over time. We capture this situation in the general framework of Van den Akker et al. (2006) by issuing *dummy jobs*, which 'eat up' the missing resources. This is achieved in the following way. Assume we use machines as resources. We now define the number of machines m to be equal to the maximum amount of resource available at any time. Now we determine the amount of resource that is missing over time with respect to m; this yields a pattern as can be observed in the left part of Figure 3.2. We can model the shown resource unavailability by adding three dummy jobs D_1, D_2, and D_3 with release dates and deadlines that are set such that the jobs are forced to be executed at the exact times corresponding to the time the resource is unavailable. We further assume that each dummy job has a due date that is unrestrictively large to prevent any interference with the $L_{\max}$ value. Note that the choice of dummy jobs is not unique. We can, for example, mingle the dummy jobs D_1 and D_2 to obtain D_1' and D_2' by swapping the deadlines and adjusting the processing times: a solution with these two dummy jobs can be translated into a solution with the two original dummy jobs by applying a 'cross-over' operation of the two involved machine schedules at time $\bar{d}_2$. Similarly, dummy jobs can be split. Anyway, it is easily seen that each feasible solution for this instance that uses

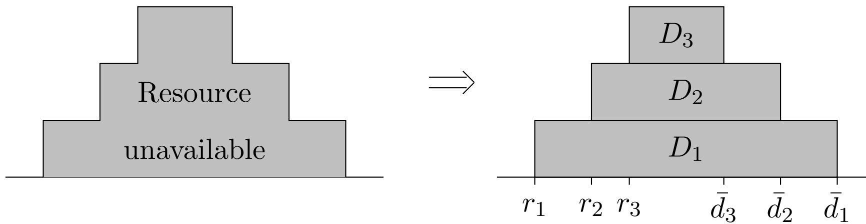

Figure 3.2: Example of modelling resource unavailability with dummy jobs.

no more than m machines corresponds to a feasible solution for the RCPS with equal objective value.

Arbitrary integral resource consumption

We now assume that the resource consumption pattern is constant for each job, but it can be any arbitrary integral value greater than or equal to 1. Suppose that J_j is some job that needs a constant amount of $k \geq 2$ units of resource during its execution. We capture this situation in the general framework by replacing job J_j by job J'_j and $k-1$ additional dummy jobs. Here J'_j is identical to J_j, except for its resource consumption, which we put equal to one. Furthermore, each dummy job has processing time equal to p_j, but it has no release date and deadline, and it is independent of all other jobs, except for J'_j: we force that all these dummy jobs and J'_j are started at the same time by means of a no-wait constraint. It is easily seen that solving the resulting instance with unit resource consumption is equivalent to solving the original instance.

If the resource consumption of job J_j is not constant over time, but can assume arbitrary integral values, then we replace J_j by a set of new jobs with a constant resource consumption pattern equal to 1, and we glue these together by no-wait constraints, such that their joint resource consumption pattern is equivalent to that of the original job J_j.

We can now use the approach of Van den Akker et al. (2006) to find the lower bound on $L_{\max}$. Furthermore, we can use the time-indexed formulation of Van den Akker et al. (2006) to look for a schedule with value equal to the lower bound. Since the dummy jobs that replace an original job J_j are glued together

by no-wait constraints, and since the time-indexed formulation uses variables x_{jt} indicating whether job J_j starts at time t, we can restrict ourselves to the original jobs (with their varying resource consumption patterns) in the time-indexed formulation. Obviously, we must then adjust constraints 3.8 to deal with the consumption patterns.

Note the close connection between the above RCPS problem and the *cumulative constraint* (see the on-line Global Constraint Catalog (Beldiceanu and Demassey, 2007)). The cumulative constraint decrees that for a given set of jobs we should find starting times, which obey the release dates and deadlines, such that the total resource consumption never exceeds the available amount of resource. To filter this constraint, we must check whether a feasible schedule exists for the above RCPS problem without initial precedence constraints.

3.4.2 RCPS problems with multiple resources

Now assume that there are only two resources involved, but this can be easily generalized to deal with any number of resources. Given a problem instance with two resources, we first transform it to an instance in which each job consumes only one resource during its execution: this is easy to achieve by replacing a job that needs both resources with two copies that only need one of the individual resources but are identical otherwise. These copies are then tied together by no-wait constraints such that they start at the same time. Next, we use the transformations described above to achieve that each job uses exactly one amount of resource (either resource 1 or 2) at any time during its execution. We now have transformed the problem into a parallel machine scheduling problem in which there are two different sets of identical machines; we assume that there are m_1 (m_2) machines corresponding to resource 1 (2).

We apply the same solution strategy as Van den Akker et al. (2006). We divide the jobs and the machines into two groups, where jobs are only assigned to machines of the right group (i.e. the corresponding type of resource), which is easily incorporated in the pricing problem. We again minimize the total number of machines that is used, but we add the constraint that we use at least m_1 (m_2) machines of group 1 (2): if we then find a solution using no more than $m_1 + m_2$ machines, then we know that we do not use too much of resources 1 and 2 separately. Note that we could have added constraints decreeing that we

use no more than m_1 (m_2) machines of group 1 (2) instead, but then we run into problems when we look for a feasible solution of the LP-relaxation to start with. Finally, we add some 'empty' columns, such that these two constraints can always be met.

As an illustration, we work things out for the case in which there are two resources, and each job J_j has a constant resource consumption pattern, requiring either 0 or 1 unit of resources 1 and 2. We assume for the ease of exposition that initially there are no precedence constraints. We denote the set of jobs requiring resource 1 only by R_1; similarly, we use R_2 to denote the jobs requiring resource 2 only. We denote the set of jobs that need both resources by $R_{1,2}$; these jobs will be split into two operations. These two operations need only one resource and are connected by a no-wait constraint. We use S and V to denote the set of machine schedules for resources 1 and 2, and we use x_s and y_v as corresponding binary decision variables. Moreover, we use a_{js} and b_{jv} to indicate whether job J_j is included in machine schedules s and v for resource 1 and 2, respectively. This leads to the following ILP-formulation where, with a little abuse of notation, a job $J_j \in R_{1,2}$ in fact consists of its two operations.

$$\min \quad \sum_{s \in S} x_s + \sum_{v \in V} y_v$$

subject to:

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j \in R_1 \cup R_{1,2} \tag{3.10}$$

$$\sum_{v \in V} b_{jv} y_v = 1, \text{ for each } j \in R_2 \cup R_{1,2} \tag{3.11}$$

$$\sum_{s \in S} S_{js} x_s - \sum_{v \in V} S_{jv} y_v = 0, \text{ for each } j \in R_{1,2} \tag{3.12}$$

$$\sum_{s \in S} x_s \geq m_1 \tag{3.13}$$

$$\sum_{v \in V} y_v \geq m_2 \tag{3.14}$$

$$x_s, y_v \in \{0, 1\}, \text{ for each } s \in S \text{ and } v \in V.$$

When we solve the LP-relaxation by column generation, we find that the reduced cost of a schedule $s \in S$ is equal to

$$c'_s = 1 - \lambda_0 - \sum_{j \in R_1 \cup R_{1,2}} a_{js} \lambda_j - \sum_{j \in R_{1,2}} \delta_j S_{js};$$

here λ_0 is the dual multiplier corresponding to constraint 3.13, λ_j $(j \in R_1)$ are the dual multipliers corresponding to the constraints 3.10, and δ_j $(j \in R_{1,2})$ are the dual multipliers corresponding to the constraints 3.12. The reduced cost of machine schedule $v \in V$ is computed in an equivalent way. It is readily verified that the pricing problem is similar to the one of Van den Akker et al. (2006), which implies that the local search procedure and time-indexed formulation to solve it can still be applied. Furthermore, we can compute an intermediate lower bound as follows. Let c_1^* denote the optimal value of the pricing problem for resource 1. If we fill in $c_s' \geq c_1^*$ in the formula of the reduced cost, then we find that

$$1 \geq c_1^* + \lambda_0 + \sum_{j \in R_1 \cup R_{1,2}} a_{js}\lambda_j + \sum_{j \in R_{1,2}} \delta_j S_{js}.$$

Hence,

$$\sum_{s \in S} x_s \geq \sum_{s \in S} \left[c_1^* + \lambda_0 + \sum_{j \in R_1 \cup R_{1,2}} a_{js}\lambda_j + \sum_{j \in R_{1,2}} \delta_j S_{js} \right] x_s =$$

$$(c_1^* + \lambda_0) \sum_{s \in S} x_s + \sum_{j \in R_1 \cup R_{1,2}} \lambda_j \sum_{s \in S} [a_{js} x_s] + \sum_{j \in R_{1,2}} \delta_j \sum_{s \in S} S_{js} x_s =$$

$$(c_1^* + \lambda_0) \sum_{s \in S} x_s + \sum_{j \in R_1 \cup R_{1,2}} \lambda_j + \sum_{j \in R_{1,2}} \delta_j \sum_{s \in S} S_{js} x_s.$$

Similarly, we find that

$$\sum_{v \in V} y_v \geq (c_2^* + \mu_0) \sum_{v \in V} y_v + \sum_{j \in R_2 \cup R_{1,2}} \mu_j - \sum_{j \in R_{1,2}} \delta_j \sum_{v \in V} S_{js} y_v;$$

here μ_0 is the dual multiplier corresponding to constraint 3.14, μ_j $(j \in R_1 \cup R_{1,2})$ is the dual multiplier corresponding to constraint 3.11, and c_2^* is the outcome value of the pricing problem for resource 2. If we add these two inequalities, then the terms containing S_{js} cancel out, because of constraints 3.12. Rearranging the terms, we find that

$$(1 - c_1^* - \lambda_0) \sum_{s \in S} x_s + (1 - c_2^* - \mu_0) \sum_{v \in V} y_v \geq \sum_{j \in R_1 \cup R_{1,2}} \lambda_j + \sum_{j \in R_2 \cup R_{1,2}} \mu_j.$$

If $1 - c_1^* - \lambda_0 = 1 - c_2^* - \mu_0$, then we can divide by this term and find a lower bound, provided that $1 - c_1^* - \lambda_0 > 0$, which is an issue we discuss later. Suppose that $1 - c_1^* - \lambda_0 > 1 - c_2^* - \mu_0$; the other case can be dealt with in the same way.

Then we add to this inequality $(c_2^* - c_1^* + \mu_0 - \lambda_0)$ times inequality 3.14, and we find the intermediate lower bound

$$\sum_{s \in S} x_s + \sum_{v \in V} y_v \geq \frac{((c_2^* - c_1^* + \mu_0 - \lambda_0)m_2 + \sum_{j \in R_1 \cup R_{1,2}} \lambda_j + \sum_{j \in R_2 \cup R_{1,2}} \mu_j}{(1 - c_1^* - \lambda_0)}.$$

What is left to show is that $(1 - c_1^* - \lambda_0) > 0$. We know that $c_1^* \leq 0$, since any column that is used in the current LP solution has zero reduced cost. Moreover, if both λ_0 and μ_0 are positive, then both constraints are binding, which implies that we have found a solution with value $m_1 + m_2$, which means that we can stop. Hence, at least one of λ_0 and μ_0 is zero, which implies that the largest of $1 - c_1^* - \lambda_0$ and $1 - c_2^* - \mu_0$ is positive.

Finally, we look at the problem of finding a feasible solution with this value. It is easily verified that the time-indexed formulation of Van den Akker et al. (2006) to find a feasible solution can be used, but we must split the m machines into two sets representing the m_1 and m_2 units of resources 1 and 2, respectively.

This means that we can use the solution method presented by Van den Akker et al. (2006) to solve the RCPS problem where we have multiple resources.

3.4.3 Other extensions

Machine scheduling with operators

A special case of the RCPS problem with multiple resources is the situation in which each job needs an operator to start it up, which takes 1 time unit per job (which can easily be generalized to the case that the start up takes more than 1 time unit per job). Hence, we can not start more jobs at any moment than there are operators available. We can model the operators as a second resource, but alternatively, since the starting times are included in the machine schedules we can force the restriction on the number of operators by adding constraints. Here, we work out the second option, which has the additional advantage that we can model a varying number of available operators. We again assume without loss of generality that there are no additional precedence constraints. We use o_{st} to indicate whether a job starts at time t in machine schedule s; we use Op_t to denote the number of operators available at time t. We then arrive at the

following ILP-formulation:

$$\min \quad \sum_{s \in S} x_s$$

subject to:

$$\sum_{s \in S} a_{js} x_s = 1, \text{ for each } j = 1, \ldots, n \tag{3.15}$$

$$\sum_{s \in S} o_{st} x_s \leq Op_t, \text{ for all } t = 0, \ldots, T-1 \tag{3.16}$$

$$x_s \in \{0, 1\}, \text{ for each } s \in S,$$

where T denotes a given time horizon. The reduced cost of a machine schedule s is then equal to

$$c'_s = 1 - \sum_{j=1}^{n} a_{js} \lambda_j - \sum_{t=0}^{T} o_{st} \pi_t,$$

where π_t denotes the dual variable corresponding to constraints 3.16. The corresponding pricing problem can be minimized using the local search procedure and the time-indexed formulation of Van den Akker et al. (2006). Furthermore, since $\pi_t \leq 0$ $(t = 0, \ldots, T)$, it is readily determined that

$$\left[\sum_{j=1}^{n} \lambda_j + \sum_{t=0}^{T} \pi_t Op_t \right] / (1 - c^*)$$

is an intermediate lower bound on the outcome of the LP-relaxation.

Finally, we can use the time-indexed formulation of Van den Akker et al. (2006) to find a solution with value equal to the lower bound, but we have to add constraints to ensure that the required number of operators is no more than the available number at any time

$$\sum_{j=1}^{n} x_{jt} \leq Op_t, \text{ for all } t = 0, \ldots, T-1.$$

Set-up times and change-over times

So far, we have assumed that, as soon as a machine has finished a job, it can start the next one. In many applications, however, there can be a mandatory

delay, namely *set-up time* and *change-over time*. A set-up time just depends on the job that is to be started; the change-over time depends on both the job that is to be started and the job that has just been completed.

We first deal with the set-up times, since this is fundamentally easier than the case with change-over times. The basic idea is to add the set-up time to the processing time; we then consider the first part of processing the job as setting it up. We must then update the release date by subtracting the set-up time from it, which might lead to a negative release date. We may further have to update the right-hand-sides of the generalized precedence constraints, but this is simply a matter of administration. An optimal solution for the problem with set-up times is then readily obtained from the optimal solution for the adjusted instance without set-up times.

Change-over times are sequence dependent and therefore much harder to model than the setup-times. Since the change-over time applies to two jobs that are consecutively assigned to the same machine, we only have to deal with the change-over times in the pricing problem by looking for single machine schedules that obey the release dates, deadlines, and the change-over times for consecutively assigned jobs. This implies that the ILP formulation for the feasibility master problem remains the same. The addition of the change-over times to the pricing problem is easily dealt with in the local search procedure that Van den Akker et al. (2006) use to solve the pricing problem approximately, but it cannot be easily incorporated in the time-indexed formulation they use to solve the pricing problem. The reason is that the time-indexed formulation does not allow for determining the order of the jobs without a large number of additional variables. Moreover, we cannot use the time-indexed formulation of Van den Akker et al. (2006) to determine the optimal value of $L_{\max}$ after we have determined a lower bound on $L_{\max}$. This is because in the current time-indexed formulation it is not possible to determine to which machine a job is assigned, meaning that we cannot determine the order of the jobs on one of the machines. For the RCPS problem with change-over times and an additive objective function Pereira Lopes and Valério de Carvalho (2007) use a different ILP formulation and present a branch-and-price algorithm.

Machine unavailability and planned maintenance

Machine unavailabilities are similar to varying resource availabilities, but they are more restrictive, since we put a label on a machine with its unavailability pattern instead of aggregating the capacities of all machines. One way to tackle this problem is to label the machines and determine for each one a separate set of machine schedules, from which we must select one. An alternative and quicker way is to add dummy jobs to the instance which correspond to unavailabilities. In a correct solution, we will have for each unavailability pattern that a feasible machine schedule will be selected that contains the dummy jobs corresponding to this unavailability pattern, which gives us a schedule for the corresponding machine.

In case of a planned maintenance, we know that the machine is being repaired for a given time, but we do not know when this repair starts: we then give the dummy job a release date and deadline corresponding to the earliest start time and the latest completion time of the repair. The only difficulty left is to ensure that a given set of dummy jobs corresponding to the unavailabilities and repairs of a given machine all end up in the same, selected machine schedule. Just like in the previous subsection, we put these constraints in the pricing problem. These additional constraints to a machine schedule are easily being dealt with in the local search procedure. When we want to solve the pricing problem to optimality, we can use the time-indexed formulation, but we must add a constraint for each pair of jobs that must be executed on the same machine or on different machines: if J_i and J_j are to be executed on the same machine, then we add the constraint

$$\sum_{t=r_i}^{\bar{d}_i - p_i} x_{it} = \sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt};$$

if J_i and J_j must go on different machines, then we require

$$\sum_{t=r_i}^{\bar{d}_i - p_i} x_{it} + \sum_{t=r_j}^{\bar{d}_j - p_j} x_{jt} \leq 1.$$

Note that we do not have to solve a pricing problem for each machine separately. Since each job has to be executed, there will be one machine 'executing' the set of dummy jobs that we introduced to mimic the unavailability pattern of this

Problem type	p_j	r_j	d_j	n	m	# prec
00	U[1,20]	U[0,60]	U[50,80]	40	4	20
01	U[1,20]	U[0,40]	U[30,60]	70	5	35
02	U[1,20]	U[0,80]	U[80,150]	80	7	30
03	U[1,20]	U[0,40]	U[60,80]	100	9	40
04	U[1,20]	U[0,60]	U[80,110]	120	9	50
05	U[1,20]	U[0,60]	U[80,110]	140	10	50
06	U[1,20]	U[0,60]	U[80,110]	160	10	50
07	U[1,20]	U[0,60]	U[80,110]	180	10	60
08	U[1,20]	U[0,60]	U[40,80]	60	3	30
09	U[1,20]	U[0,60]	U[40,80]	60	5	30
10	U[1,20]	U[0,60]	U[40,80]	60	7	30
11	U[1,20]	U[0,60]	U[50,80]	30	3	15
12	U[1,40]	U[0,120]	U[100,160]	30	3	15

Table 3.1: Parameters for creating the different problem types.

machine. Unfortunately, after having determined the lower bound, we cannot straightaway use the time-indexed formulation of Van den Akker et al. (2006) to look for a solution with equal value, since we must force the set of dummy jobs representing the machine unavailability pattern on one machine that does not execute any other dummy job. We can use a similar formulation in which we distinguish between the machines by using variables x_{ijt} indicating that job J_j starts at time t on machine i, but this will blow up the model tremendously, since we cannot aggregate the machines and require that at most m are used then anymore.

3.5 Computational experiments

We implemented the algorithms in C++ and ran computational experiments with this implementation. For solving the LP and ILP models we make use of the Concert Technology interface of CPLEX 9.1.2(ILOG, 2005). We tested the case with one type of resource, unit resource consumption, and variable resource availability over time.

To create the instances we use the same types of problem as defined by Van den Akker et al. (2006). The parameters to create the various problem types are given in Table 3.1.

We present the results of the *basic scenario* with full resource availability in Table 3.2. From this table it can be seen that most of the computation time is spent on determining the lower bound. Furthermore, it can be seen that the average number of times we need to solve the ILP formulation of the pricing problem is very low. In the case of problem type 02 and 10 it is even not needed to solve any ILP formulation of the pricing problem. This can be explained by the fact that the initial lower bound used to start binary search for $L_{\max}$ is tight and we never have the need for solving the ILP formulation of the pricing problem to prove m machines are not sufficient for the given lateness.

We consider two scenarios for the case of a *variable resource availability* over time. In the first scenario, there is one pile of dummy jobs (reflecting the resource unavailability) where the pile is located around half of the estimated makespan of the schedule. In the second scenario there are two shorter piles around one third and two third of the estimated makespan, respectively. In both scenarios the maximum amount of unavailable resources is about $\lceil \frac{m}{2} \rceil$. The first scenario is denoted by RU_{A} and the second scenario is denoted by RU_{B}. For a number of the problem types presented, we give the results for both scenarios in Table 3.3. For ease of comparison we included the corresponding results from the full resource availability case in the table and these are denoted by a 'No' in the Resource Unavailability column. It can be seen that the resource unavailability does increase the average time needed for solving the problem, but this still is within acceptable running times.

To investigate the effect of limiting the time indices in the final time-indexed formulation by using the knowledge from the column generation procedure (as suggested in Section 3.3.2), we ran our algorithm twice on the instances of the full resource availability case with only greater than or equal precedence delays. When we look at Table 3.4 it can be clearly seen that limiting the time indices has a significant effect on the solution time of the time indexed formulation, while the optimum is still found in all cases.

In Table 3.5 we present the results for the case of *full resource availability* problem with only exact precedence delays. To get an idea of the influence of this type of precedence constraint on the solution times, we use exactly the same

Problem	Solution time LB (s)		Solution time TIF (s)		Avg
type	Avg	Max	Avg	Max	# ILP
00	7.84	26.49	0.38	0.71	3.0
01	38.42	55.22	4.71	9.34	5.7
02	3.14	4.20	0.51	0.62	0.0
03	32.85	47.32	4.15	8.32	5.0
04	98.11	195.77	7.78	28.88	15.8
05	142.64	223.71	22.44	96.36	15.0
06	183.49	235.44	26.27	43.85	10.1
07	284.01	400.75	68.29	148.91	9.4
08	91.45	205.68	15.39	38.00	8.3
09	10.64	31.46	1.51	4.91	2.7
10	1.32	2.23	0.27	0.45	0.0
11	7.20	22.83	0.25	0.56	3.7
12	20.58	34.62	0.83	1.67	9.3

Table 3.2: Results for solving the RCPS problem in case of only greater than or equal to precedence delays.

instances as for the case where we have only greater than or equal precedence delays and only replace all these delays by exact delays. This direct replacement does have the drawback that certain instances become infeasible due to the very strict nature of this precedence delay (i.e. it results in situations where the precedence delays induce the need of $> m$ machines at particular times).

When we compare Table 3.5 to Table 3.2 we see that the average time needed for determining the lower bound is significant higher for all of the problem types in case of the exact precedence delays. Another major difference that can be seen from the table is that in the case of exact precedence delays, we have instances for which the determined lower bound is not equal to the optimal value.

Finally, in the last two columns in Table 3.5 we present the solution times for the time-indexed formulation for the case where the valid inequalities suggested in Section 3.3.3 are added to the model and for the case where we do not add these valid inequalities. Without adding these valid inequalities there are only a few instances that CPLEX can solve within a time-limit of 30 minutes. When we look at the average time needed for solving the time-indexed formulation we see a large decrease when the valid inequalities are added to the model.

Problem type	Resource Unavailability	Solution time LB (s) Avg	Max	Solution time TIF (s) Avg	Max
00	No	7.84	26.49	0.38	0.71
00	RU_A	14.47	25.13	0.93	2.09
00	RU_B	14.43	29.15	1.03	3.03
01	No	38.42	55.22	4.71	9.34
01	RU_A	57.94	96.17	13.48	44.10
01	RU_B	65.00	98.07	22.62	52.90
03	No	32.85	47.32	4.15	8.32
03	RU_A	74.90	108.31	13.32	21.90
03	RU_B	129.30	170.81	28.86	77.64
07	No	284.01	400.75	68.29	148.91
07	RU_A	408.24	506.12	119.29	293.43
07	RU_B	422.05	614.73	113.86	322.90
09	No	10.64	31.46	1.51	4.91
09	RU_A	35.31	63.66	2.27	5.49
09	RU_B	30.70	67.66	3.57	8.80
11	No	7.20	22.83	0.25	0.56
11	RU_A	20.60	34.70	1.77	4.21
11	RU_B	18.56	37.47	5.28	33.48

Table 3.3: Results with resource unavailability.

Problem type	Average solution time ILP in seconds Limited time indices	All time indices
00	0.38	2.82
01	4.71	41.68
02	0.51	1.09
03	4.15	25.70
04	7.78	201.96
05	22.44	178.50
06	26.27	257.68
07	68.29	528.06
08	15.39	101.71
09	1.51	36.74
10	0.27	0.83
11	0.25	4.76
12	0.83	50.50

Table 3.4: Comparison solution times ILP with and without using knowledge from column generation with only greater than or equal precedence delays.

Problem type	Feasible instances	times LB=OPT	Time (s) LB	Average solution time ILP (s) With inequalities	Without inequalities *(Instances solved)*
00	7	2	27.59	5.55	393.79 *(5)*
01	0	0			
02	9	7	22.34	6.07	121.93 *(4)*
03	7	7	113.04	38.00	732.73 *(1)*
04	9	9	222.47	11.68	*(0)*
05	8	8	388.20	7.92	*(0)*
06	9	9	376.45	14.89	*(0)*
07	9	9	657.25	38.06	*(0)*
08	1	1	91.75	865.43	*(0)*
09	5	1	33.61	21.63	19.96 *(1)*
10	6	3	14.97	7.49	36.40 *(2)*
11	5	0	27.43	9.63	59.09 *(2)*
12	5	0	45.44	28.70	*(0)*

Table 3.5: Comparison solution times with and without the addition of valid inequalities with only exact precedence delays.

3.6 Conclusion

We have described how the framework by Van den Akker et al. (2006) can be used to solve a number of basic RCPS problems. Further, we have shown how to incorporate change-over times and machine maintenance. Except for the case with change-over times, we can use the same tool kit as Van den Akker et al. (2006) to compute the lower bound. In the computational experiments of Van den Akker et al. (2006) this lower bound always coincided with the optimum, and we found the same phenomenon in the case of the strongly related problem with varying amount of resources available with only greater than or equal precedence delays. In case of exact precedence delays we encountered instances for which the lower bound did not coincide with the optimum.

Furthermore, we have given a number of improvements to decrease the time needed for determining the lower bound, as well as the time needed for solving the time-indexed formulation.

When it comes to finding a solution with value equal to the lower bound, we can in many cases use the time-indexed formulation of Van den Akker et al. (2006) in which we specify the wanted optimum beforehand. Van den Akker et al. (2006) conjectured that this is presumably due to the preprocessing step within CPLEX, which suggests that the technique of constraint programming should be able to find such a solution more quickly, or show that it does not exist. Moreover, constraint programming seems to be the most eminent candidate to look for a solution with value equal to the lower bound for the problems with machine unavailabilities and change-over times.

Part II

Planning at Amsterdam Airport Schiphol

CHAPTER

FOUR

Solving the gate assignment problem

Introduction

Between the time an aircraft lands at an airport and the time it departs again many things must happen. One of the most obvious things is that the passengers need to disembark the aircraft. Moreover, the aircraft needs to be refueled, new passengers need to board it, new supplies have to be put on board, and the aircraft has to get cleaned. These latter two actions are taken care of by the so-called *ground handler*. All of these actions take place while the aircraft is standing at a gate. We will refer to the arrival of an aircraft till the following departure of the same aircraft as a *flight*.

Now the *gate assignment problem* is described as follows: assign a given set of flights to a (possibly smaller) set of gates while making sure that certain criteria are met. Examples of such criteria are:

- Gates can handle only flights operated by aircraft of certain sizes.

- Gates can handle only flights for certain origins/destinations (e.g. because of safety regulations).
- Gates can handle only flights assigned to certain ground handlers.
- Two adjacent gates can not be assigned flights operated by big aircraft at the same time.
- Two adjacent gates should not be assigned flights with equal departure times.

Depending on the airport and its characteristics, many variants of the gate assignment problem have been researched and many solution methods have been suggested. A good overview of explored methods and models is given in Van Orden (2002). In this chapter we consider the situation at Amsterdam Airport Schiphol (AAS). Depending on the time horizon of the planning we distinguish between the following three planning problems: *seasonal planning*, *daily planning*, and *tactical planning*. The seasonal planning problem is a capacity planning problem. In this problem the gate planners must decide to accept or decline new requests from airlines to have their aircraft visit AAS. The daily planning concerns the creation of a planning for the upcoming day on the basis of the available information about the flights of that day. Finally, the tactical planning is an online planning problem. This problem concerns the resolving of conflicts that arise in the planned solution (i.e. the planning created the day before) due to disturbances.

The problem we consider in this chapter is daily planning. When looking at this planning problem, different objectives can be taken into account. Examples are minimizing total waiting time for the aircraft (i.e. the time an aircraft has to be held after landing before the gate is free) or minimizing total towing actions. Another objective that is used very often in the literature is to minimize the walking distance for the passengers to and from the gate (Bihr, 1990; Haghani and Chen, 1998; Xu and Bailey, 2001). Moreover, it is also possible to consider multi-criteria objectives. Dorndorf et al. (2007) not only present a state-of-the-art of the gate assignment problem in general but also discuss recent developments with regard to the multi-criteria objectives.

Since an airport is a very dynamic environment, an actual day will hardly ever go completely as planned; flights arrive either earlier or later than planned due to all kinds of reasons. The objective we are concerned with is to create a *robust*

schedule, i.e., a schedule that is able to cope with small perturbations in arrival or departure time without the need to replan big parts of the schedule. During meetings we had with the gate planners at AAS, finding a robust schedule was identified as the main target of their daily planning activities. Moreover, every change needed in the gate assignment during the actual day has an effect on a very broad range of parties: the ground handler, the security personnel, the passengers, etc.

Therefore, creating a schedule that needs fewer changes during the tactical planning is of great importance for a variety of parties. The same objective has been considered by Bolat (2000) and Van Orden (2002).

The question is how to measure the robustness of a schedule. The probability that a conflict arises between two consecutively scheduled flights at the same gate depends on the 'reliability' that the aircraft will stick to their assumed departure and arrival times and on the idle time between these two consecutive flights. The first part is outside the scope of the gate planners, and consequently we focus on optimizing the choice of the pairs of flights that we put consecutively at a gate. One approach for optimizing the individual idle times, and thus finding a robust schedule, is to minimize the variance of the idle time between successive flights at a gate. This approach is used in Bolat (2000) where the problem is formulated as an Integer Linear Program (ILP) using binary decision variables that link flights to positions at a gate. Based on this research a similar approach was followed in Van Orden (2002) for modelling the gate assignment problem for AAS. Although the suggested model showed promising results, a weak point of this model was that it was not possible to solve problems with more than 80 aircraft and 20 gates in a reasonable amount of time.

In this chapter we aim at finding a robust gate assignment schedule for a full day of traffic at AAS, which has about 600 flights. We include all real constraints occurring at the airport that we have identified in discussions with the gate-planners of the airport. To attain robustness, we optimize the idle time between all consecutive flights at the gates. Instead of minimizing the variance, we develop an objective function based on the arctangent, and we make some adjustments to this function to take the 'reliability' of the flights into account. We formulate this problem as an ILP and use a completely different representation from the one used in Van Orden (2002). This different representation is derived from the one used in Freling et al. (2001) for the vehicle and crew scheduling. We present an algorithm based on column generation to find a very

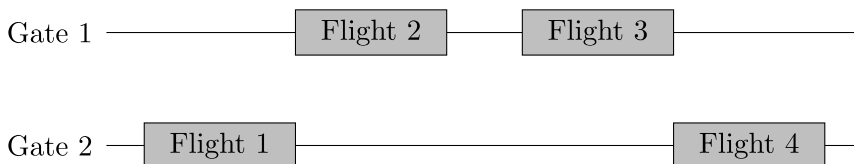

Figure 4.1: Example of a non-robust schedule.

good approximation for the optimum in this model. Our experiments indicate that this algorithm is able to solve real-life instances to near-optimality within a few minutes. Because of the high quality of the approximation even a near-optimal solution will suffice and techniques as branch-and-price (Barnhart et al., 1998) to solve the problem to full optimality will not pay off.

The outline for the remainder of this chapter is as follows: In Section 4.1 we give a detailed description of the gate assignment problem. In Section 4.2 we formulate the gate assignment problem as an ILP and present our algorithm to find an approximate solution. In Section 4.3 we will present the experimental results, and finally in Section 4.4 we draw some conclusions and indicate some future research topics.

4.1 Problem description

As mentioned in the previous section, our objective is to maximize the robustness of a solution to the gate assignment problem. To maximize the robustness we maximize the idle time between all pairs of consecutive flights at a gate, ensuring that each flight can arrive either a bit too early or a bit too late without the need for replanning the schedule. An example of a non-robust schedule can be found in Figure 4.1. This schedule does not have a lot of margin between flight 2 and flight 3. By modifying the schedule such that flight 3 is assigned to gate 2, while flight 4 gets assigned to gate 1 we introduce a lot more robustness in the solution.

One of the things that must be taken into consideration during the gate assignment process at AAS is whether two consecutive flights assigned to the same gate, are operated by the same airline or share the same ground handler. Such a situation is *convenient*, since the airline and the ground handler respectively will have an incentive to make the first flight leave on time. Furthermore, some airlines are known to be unreliable, meaning that if a flight of such an airline is due to depart at a certain time, then there is a great chance that it is delayed.

There are several hard constraints in the gate assignment problem. Obviously, each flight must be assigned to a gate, and two flights cannot be assigned to the same gate at the same time. But there are many more hard constraints, concerning the properties connected with the flights and the gates. The properties that are known for each flight are:

- The region of the origin of the flight,
- The region of the destination of the flight,
- The size category of the flight,
- The ground handler for the flight.

The region can be either Schengen (which refers to the countries that signed the Schengen Agreement), European Union (EU), or Non-EU. Often the region of the origin and the region of the destination of a flight are the same, but there are many exceptions, including e.g. transit flights that do not have AAS as their final destination. With respect to the size category of the flight, there are 8 categories at AAS: category 1 for the smallest aircraft up to category 8 for the biggest aircraft. Finally, the ground handlers are divided into two groups at AAS: KLM Ground Services, and all other companies.

For each of the gates it is known which regions, which size categories, and which ground handlers it can serve. When assigning flights to a certain gate, we need to satisfy the following three essential properties:

- The regions of origin and destination of the flight must match the possible regions of the gate.
- The size category of the flight must match the possible size categories of the gate.

- The ground handler of the flight must match the possible ground handlers of the gate.

One important issue of the gate assignment problem is that some of the flights stay at AAS for a longer period of time; for example, they arrive early in the morning and leave again late in the afternoon. If there are many such *long-stay* flights that stay at the gate, then the number of available gates quickly decreases. To circumvent this problem, the gate planners at AAS have the possibility of *splitting* the stay of long-stay flights into three different parts:

- *Arrival part.* After the aircraft lands, it will stay at the gate for 65 minutes, after which it is towed to some buffer stand.
- *Intermediate part.* During this part the aircraft resides on a buffer stand, where it does not use precious gate capacity.
- *Departure part.* The aircraft is taken from the buffer to the appropriate gate, 95 minutes before the aircraft will depart.

At AAS only flights that stay longer than three hours are considered for such a splitting. The advantages of splitting long-stay flights are three-fold. First, there is the obvious extra capacity that becomes available for the assignment of other aircraft. The second advantage concerns flights with different regions for the origin and destination: such flights normally would have to be assigned to a gate that is multi-purpose with respect to the region property, and these gates are quite scarce. When such a flight is split into parts, the separate parts do not have to be assigned to the same gate and thus can be assigned to two single region gates. Third, the decoupling of the parts yields additional flexibility.

Currently the process of splitting the long-stay flights is done manually. First the gate planners try to solve the gate assignment problem without splitting any flights. If the available capacity is not sufficient to accommodate all flights, the gate planners determine which flights should be split and then solve the problem again. This step is repeated several times until sufficient capacity is available.

4.2 Problem formulation

There are several options to value the idle time between two consecutive flights. Bolat (2000) considers minimizing the variance of the idle time. Here the idle times before the first flight and after the last flight on a gate are taken into account as well, which implies that the total idle time is a constant. As a result, minimizing the variance becomes equivalent to minimizing the sum of the squared idle times.

Another option is to define a cost function to value the idle time, which is the approach we use. This cost function for the idle time must reflect the natural appreciation of a solution. First of all, it should penalize very small idle-times with very high cost and it should only mildly penalize rather large idle times. Second, the function should be steep in the beginning (for small idle times) and then flatten out, to reflect that for small idle times any improvement is very beneficial, whereas for already large idle times an extra increase is of minor importance. Third, it should be possible to combine this with a refinement reflecting the *reliability* of a flight, which is discussed below. We found that a cost-function based on the arctangent fulfills the desired properties best. This function is defined as follows:

$$\mathrm{c}(t) = 1000(\arctan(0.21(5-t)) + \frac{\pi}{2}),$$

where t is the amount of idle time. Initial experiments showed that it was not beneficial to make use of a *cut-off value* (i.e. a threshold after which any increase in idle time will not result in a decrease of the costs). Such a cut-off value resulted in longer running times for the ILP, which can be explained by the fact that the cut-off value introduces a lot of symmetry in the problem.

Recall from the previous section there could be an advantage (or disadvantage) in assigning certain pairs of flights consecutively to the same gate. To model such a relation we introduce a convenience multiplier $conv(v, w)$ for flights v and w. To compute the cost of placing flight w immediately after flight v at the same gate, the cost corresponding to the idle time in between these flights will be multiplied by this multiplier. If putting flight w after flight v is desirable (e.g. flight v and w are operated by the same airline or handled by the same ground handler), then the convenience multiplier is given a value less than 1, thus decreasing the cost. On the other hand, inconvenient situations (e.g. when

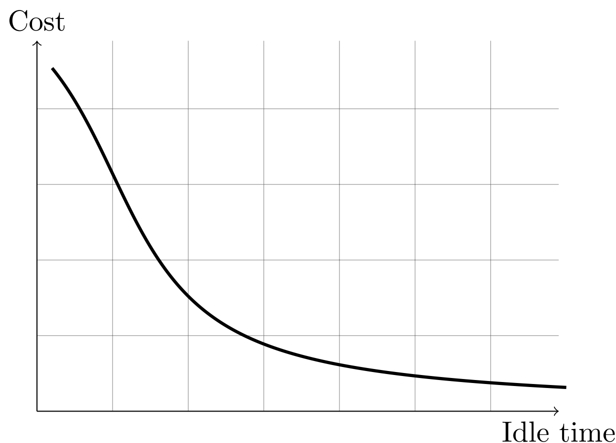

Figure 4.2: Cost function to value idle time.

flight v is operated by an unreliable airline) can be penalized by giving the multiplier for these situations a value greater than 1, thus increasing the cost of such an assignment. An advantage of this approach is that the convenience multiplier is computed beforehand for all possible pairs of successive flights, while during run-time only one additional multiplication per pair of consecutive flights is needed.

Since instances of the complete gate assignment problem consist of around 600 flights and 120 gates, we would like to see if we can decompose the problem into a set of smaller subproblems based on one of the properties associated with each gate (i.e. the possible regions, sizes, and ground handlers). Unfortunately for each of those properties gates exist that are multi-purpose for that particular property. This means that a strict division into multiple independent subproblems based on any of the properties is not possible without discarding available capacity: a given multi-purpose gate can only be assigned to one of the subproblems. Hence, such a static division is not desired.

The standard approach for modeling the problem as an ILP uses variables that denote whether a flight is assigned to a certain gate, see Van Orden (2002). One major disadvantage of this kind of model is that many additional variables are needed to determine the order of the flights, which is necessary to compute the idle time between two consecutive flights and hence the cost of a solution. More-

over, the number of constraints grows rapidly when relations between multiple gates and flights are present in the problem. This rapid growth of both variables and constraints makes it impossible to solve a gate assignment problem of the size that occurs at AAS. Therefore, a new approach is needed.

Based on the approach used by Freling et al. (2001) for the single-depot vehicle scheduling problem, we split the gate assignment problem into two phases. In the first phase we aggregate gates with the same properties (*identical gates*) into groups of gates and each such group we refer to as a *gate type*. We now introduce a *gate plan* as a series of flights that are to be assigned to the same gate of a certain gate type. Gate plans enable us to check feasibility easily: all flights present in the gate plan must satisfy the properties of the gate type that the gate plan corresponds to and no two flights should be assigned at the same time. Furthermore, calculating the cost of a gate plan boils down to summing the cost of all idle times between consecutive flights within the gate plan. With this representation solving the first phase comes down to finding a set of gate plans such that we have a gate plan for each physical gate, and that each flight is present in exactly one of the gate plans.

After we have obtained a solution for the first phase, for each gate type we have as many gate plans as there are gates of that gate type. In the second phase we undo the aggregation of identical gates into groups of gate types and we have to assign each gate plan to a physical gate.

4.2.1 Assigning flights to gate plans

When we look at the number of possible gate plans, we can clearly see that the number of possible gate plans is enormous. Suppose for the moment we do have the complete set of all possible gate plans, the gate assignment problem can be formulated as determining which of these gate plans we must select. Now for each gate plan i we introduce a decision variable x_i as follows:

$$x_i = \begin{cases} 1 & \text{if gate plan } i \text{ is selected} \\ 0 & \text{otherwise.} \end{cases}$$

Let V denote the number of flights, let A denote the number of gate types, let S_a denote the number of gates of type a, and let N denote the number of gate plans. Now the basic model for the gate assignment problem is as follows:

$$\text{Minimize } \sum_{i=1}^{N} c_i x_i$$

subject to:

$$\sum_{i=1}^{N} g_{vi} x_i = 1 \quad \text{for } v = 1, \ldots, V \tag{4.1}$$

$$\sum_{i=1}^{N} e_{ia} x_i = S_a \quad \text{for } a = 1, \ldots, A \tag{4.2}$$

$$x_i \in \{0, 1\} \quad \text{for } i = 1, \ldots, N \tag{4.3}$$

where

$$g_{vi} = \begin{cases} 1 & \text{if flight } v \text{ is in gate plan } i; \\ 0 & \text{otherwise,} \end{cases}$$

$$e_{ia} = \begin{cases} 1 & \text{if gate plan } i \text{ is of type } a; \\ 0 & \text{otherwise.} \end{cases}$$

We will extend this basic model to cover more possibilities. One important issue not addressed by the above model is the fact that it should be possible to give *preference* to placing a flight at a certain gate type. Such preferences can be used to model that certain flights are preferably assigned to certain gate types, e.g. because of the size of the waiting area or because of airlines having their "own" gates where at least a certain percentage of their flights have to be assigned to. Each preference consists of a set of flights, a set of gate types, and the minimum number of flights out of the given set that should be assigned to the given set of gate types.

Preferences can be modelled in the ILP by adding the following constraints to the model:

$$\sum_{i=1}^{N} \sum_{v=1}^{V} \sum_{a=1}^{A} p_{vak} e_{ia} g_{vi} x_i \geq P_k \quad \text{for } k = 1, \ldots, K \tag{4.4}$$

where

$$p_{vak} = \begin{cases} 1 & \text{if flight } v \text{ has preference for gate type } a \text{ in preference } k; \\ 0 & \text{otherwise,} \end{cases}$$

P_k denotes the minimum number of flights that has to be assigned to a given gate type according to preference k, e.g. an airline must have at least 6 flights at their "own" gates. K denotes the total number of preferences.

We extend the above model to deal with the case that there is not enough capacity to accommodate all flights. To solve this problem we add a penalty variable UAF_v (unassigned flight) for every flight v to constraint (4.1) in the following way

$$\begin{aligned} \sum_{i=1}^{N} g_{vi}x_i + \text{UAF}_v &= 1 \quad \text{for } v = 1, \ldots, V \\ \text{UAF}_v &\geq 0 \quad \text{for } v = 1, \ldots, V. \end{aligned} \tag{4.5}$$

The extra variable UAF_v for flight v is added with a very high cost coefficient Q_v in the objective function. Now it is possible to have flights not being assigned to gates, but this option comes at a high price. The unassigned flights present in the final solution are then assigned to gates manually by the gate planners, who have the ability to overrule some of the constraints, if necessary. We can select the cost coefficients of the UAF_v variables to model the effort it takes to assign a flight manually; for example, in general it is easier to assign a flight operated by a small aircraft somewhere manually than a flight operated by a big aircraft. The values for these cost coefficients were determined via preliminary computations.

One thing not addressed in the model yet is the fact that flights with a long stay can be split into three parts of which two parts must be assigned to a gate. To model this possibility, for each long-stay flight v we create two *split flights* v_A and v_B, which refer to the arrival and departure part of flight v, respectively. The intermediate parts of the flights are not modelled because the buffer stands for these intermediate parts are not part of the gate assignment problem.

Since these three flights v, v_A, and v_B concern the same aircraft, we must ensure we model their dependency. This can be achieved by splitting the original constraint (4.1) for flight v into two separate constraints

$$\sum_{i=1}^{N} (g_{vi} + g_{v_A,i})x_i + \text{UAF}_{v_A} = 1 \quad \text{and} \quad \sum_{i=1}^{N} (g_{vi} + g_{v_B,i})x_i + \text{UAF}_{v_B} = 1.$$

Here UAF_{v_A} and UAF_{v_B} indicate the possibility of not assigning (a part of)

flight v; their cost coefficients each get a value half of the value of the original cost coefficient UAF_v.

Furthermore, we have to include the split flights in the preference constraints 4.4. Since each split flight only represents half of the original flight, each gets a coefficient 0.5 by redefining p_{vak} as follows

$$p_{vak} = \begin{cases} 1 & \text{if flight } v \text{ has preference on gate type } a \text{ in preference } k; \\ 0.5 & \text{if the unsplit version of flight } v \text{ has preference on gate type} \\ & a \text{ in preference } k; \\ 0 & \text{otherwise} \end{cases}$$

The ILP formulation presented above models the problem correctly, but unfortunately the size of the problem is enormous, since the number of possible gate plans is enormous. Therefore we try to approximate the optimal solution by taking only *presumably useful* gate plans into account. To identify these, we first relax the integrality constraints (4.3) and then solve the resulting LP-relaxation by column generation. We then reinstate the integrality constraints and solve the resulting ILP formulation with the columns generated. As a side-effect, we can use the value of the LP-relaxation as a measure for the quality of the obtained ILP formulation.

4.2.2 Pricing problem

After we have solved the LP-relaxation for a given set of columns, we find a dual multiplier π_v for the constraint (4.1) corresponding to flight v, a dual multiplier λ_a for the constraint (4.2) corresponding to type a, and a dual multiplier ψ_k for the constraint (4.4) corresponding to preference k. Therefore, the reduced cost for a gate plan i is equal to

$$c_i - \sum_{a=1}^{A} e_{ia}\lambda_a - \sum_{v=1}^{V}(g_{vi}\pi_v + \sum_{k=1}^{K}\sum_{a=1}^{A} g_{vi}e_{ia}p_{vak}\psi_k).$$

Note that for the original parts of a long-stay flight the above must be slightly changed. Since the original part of a long-stay flight is present in two constraints, we must subtract the two dual multipliers π_{v_A} and π_{v_B} instead of only π_v.

It is well-known from the theory of linear programming that we have solved the LP-relaxation to optimality if the reduced cost of each gate plan is greater than

or equal to zero. To check this, we compute the minimum reduced cost over all feasible gate plans; this is called the *pricing problem*. We solve this problem by composing a network such that each feasible gate plan corresponds to a path in this network, and vice-versa. Moreover, we choose the lengths of the arcs such that the length of a path equals the reduced cost of the corresponding gate plan. Hence, we can then solve the pricing problem by solving a shortest-path problem in this network.

We solve the pricing problem for each gate type separately. For each type of gate a we introduce a Directed Acyclic Graph (DAG) $G_a = (V_a, E_a)$. We add a vertex to this graph for every flight v that is allowed to be assigned to a gate of type a. Furthermore, we add vertices s and t, denoting the source and sink respectively. If two flights v and w are allowed on a gate of type a and the arrival time T_w^{arr} of flight w is greater than or equal to the departure time T_v^{dep} of flight v plus the minimum idle time T_v^{min} required after flight v, then a directed edge from vertex v to w is added to the graph. Furthermore, a directed edge from the source vertex s to every vertex v is added, as well as a directed edge from every vertex v to the sink vertex t. Hence,

$$E_a = \{(v, w) | T_w^{\text{arr}} \geq T_v^{\text{dep}} + T_v^{\text{min}}\} \cup \{(s, v), (v, t) | \text{for all } v\}.$$

It can be easily seen that every path from s to t in G_a represents a feasible gate plan of type a and vice-versa. What is left is to set the lengths of the arcs. If we look at the reduced cost, then we see that, if a flight v is selected and succeeded by flight w in a gate plan of type a, then the contribution of flight v to the reduced cost of the gate plan is

$$conv(v, w) c(T_w^{\text{arr}} - T_v^{\text{dep}}) - \pi_v - \sum_{k=1}^{K} p_{vak} \psi_k.$$

Recall that $conv(v, w)$ denotes the convenience multiplier indicating the advantage (or disadvantage) of putting flight v directly before w. We put the cost of the arc (v, w) in G_a equal to the contribution of flight v to the reduced cost. The additional arcs (s, v) and (s, t) get cost $-\lambda_a$, and the additional arcs (v, t) get cost equal to $-\pi_v - \sum_{k=1}^{K} p_{vak} \psi_k$. Note that the cost $c(T_w^{\text{dep}} - T_v^{\text{arr}})$ of putting flight w immediately after flight v in the gate plan is constant; after each iteration, we only have to update the cost terms containing the dual multipliers.

Since exactly one of the outgoing edges of vertex v will be used if v occurs in a path, the total cost of a path equals the reduced cost of the corresponding gate plan.

For solving the pricing problem we need to find the gate plan with minimal reduced cost. Since each path in the graph corresponds to a possible gate plan and the path length corresponds to the reduced cost of the represented gate plan, finding the gate plan with minimal reduced cost comes down to finding the shortest path in the presented graph. Without loss of generality we assume that all flights are sorted by their arrival times. This assumption implies a topological order on the vertices in the graph, namely the order of the flight indices. Because we now have a DAG with a topological order it is possible to find the shortest path in $\mathcal{O}(|V| + |E|)$ time (cf. Cormen et al. (2001)).

When a gate plan with minimum reduced cost has been found, there are two possibilities:

- The new gate plan has *negative reduced cost*. This means that by adding this gate plan to the master problem, the objective value of the master problem might decrease and thus we add this gate plan to the master problem.
- The gate plan has *zero reduced cost* in which case there exists no gate plan with negative reduced cost.

For each of the gate types, we need to check whether a gate plan with negative reduced cost exists. If for none of the gate types a gate plan with negative reduced cost exists, then the master problem has been solved to optimality.

4.2.3 Solving the restricted ILP

After the master problem has been solved to optimality we only have a solution for the LP-relaxation. If this solution happens to be integral, then we have a solution to the original ILP formulation of the gate assignment problem, too. If the solution is fractional, then we have to convert it to an integer solution. One possibility for this is to make use of branch-and-price. But since this is computationally infeasible, and since a good approximation suffices, we go for an approximate solution. To this end, we use the ILP-solver CPLEX to solve the ILP with the limited set of columns. In our preliminary computations, we only used the restricted set of columns generated by the column generation. It turned out that this took too much time and memory. Moreover, if solutions were

found within reasonable running time, then the quality of these solutions was really bad.The large running time and memory consumption can be explained by the fact that the restricted set of columns generated for solving the LP is too restrictive for the ILP: if flight v is part of a selected column, then none of the other columns containing v can be used anymore. Hence, if our *first-choice* column contains a flight that is included in one of the already selected columns, then we need a *second choice* column that does not contain this already covered flight. As the first-choice column was generated once, when solving a pricing problem, we decided to create a set of additional second-choice columns each time when solving the pricing problem.

To create these second-choice columns we used the following procedure. We first solve the pricing problem, that is, we find the shortest path. We then take out the nodes in the shortest path one by one and solve the shortest path problems for each of the resulting graphs. These additional columns are added to a column pool. After the master LP problem has been solved to optimality, we determine the set of unique columns from the ones that are in the column pool. This set of unique columns is then added to the restricted ILP problem. After these unique columns were added to the ILP problem, the ILP solver was able to solve the problem in a matter of minutes and sometimes even seconds instead of running for hours or even days. A second reason for this tremendous speed improvement may be that, since the number of unique gate plans that are in the column pool is quite large, it might be easier for the branch-and-bound subroutine of CPLEX to find a good lower or upper bound quickly, thus speeding up the process.

4.2.4 Assigning gate plans to gates

After solving the first phase, we have a set of gate plans with just as many gate plans of type a as there are physical gates of type a, such that the total schedule is robust against small variations caused by for example delays of flights. In the second phase of the problem we have to determine which gate plan is assigned to which physical gate.

In the first phase we have introduced constraints dealing with just one gate type. We did not consider relations between specific physical gates, like the constraint that two flights having the same departure time can not be assigned to directly opposite gates due to the impossibility of a simultaneous push-back

of both flights. We consider these types of constraints when we assign the gate plans to the physical gates in the second phase.

In Van Orden (2002) some additional constraints have been formulated that need to be addressed in the second phase. These are:

- Avoid putting two flights next to each other that have an overlapping wingspan.
- Avoid putting two flights with equal departure time next to each other because of conflicting push-back.
- Avoid putting two flights that have the same departure time on opposite gates because they cannot have a push-back at the same time.
- Flights from US carriers need extra facilities (e.g. possibility of closing parts of the waiting room at the gates to which they are assigned).
- Minimize the walking distance for the passengers. This concerns both arriving or departing passengers and transfer passengers.

Although the first one of these constraints at first sight seems to be a good example of a second phase constraint, it turned out to be not important at all. After receiving detailed information of the gates at AAS from the gate planners it turned out that this constraint did not exist for any gate at AAS. But there does exist a strongly related constraint at AAS though, which decrees that it is possible to combine two gates of a small category to one gate of a bigger category. This constraint cannot be addressed in the first phase, since we do not know then which two gate plans will be next to each other in the final solution. In theory it is possible to take this constraint into consideration when solving the first phase by creating a new category of gates consisting of the two smaller gates. The pricing problem for this gate type would then consist of two paths that both contain the selected vertices corresponding to the aircraft of a bigger category. This problem is harder to solve. Therefore, and also since there are not so many of these possibilities, we have decided not to take this constraint into consideration.

After completing the second phase it will be known which gate plans and thus which flights will be assigned to the gates that can be combined. When there

are big flights left that are still not assigned, the gate planner will be able to manually combine a set of these smaller gates into one gate of a bigger category and assign a bigger flight to this combined gate.

Although the gate planners at AAS do not have information regarding possible connecting flights of the passengers, one way they try to maximize passenger comfort is to minimize the maximum walking distance. This is achieved by putting flights with a large number of passengers at the best gates. Since we are assigning entire gate plans to gates now, we have less flexibility, but we can use the same principle. Gates that are closer to the beginning of the pier are considered to be better gates. So also in this case the number of passengers will be an important factor in the decision: when there are more gate plans to choose from, the one with the largest number of departing passengers will be assigned to the best gate.

The full assignment problem of the second phase can be decomposed into a number of smaller assignment problems for each type of gate. Most of these subproblems can be solved independently, but some are dependent, since they involve gates that have a neighboring or opposite gate of a different type. The dependent subproblems need more attention. To determine the benefit of assigning a flight to a certain physical gate, the gate planners at AAS use a number of different rules.

Presumably, the best option is to present the gate planners with the results from the first phase and have them assign the gate plans to the physical gates. This can be done *manually* since the size of these problems is rather small; generally the maximum number of gates within one gate type is around eight. Only for remote stands this number is higher, but the flexibility for assigning gate plans to these remote stands is really high. Finally, sometimes it may turn out to be beneficial to swap two flights from two gate plans, resulting in a better solution.

4.2.5 Directly assigning flights to gates

The two phases of first assigning flights to gate plans and then assigning the gate plans to gates can be integrated by modelling each single physical gate as a separate type and including all the constraints concerning specific physical gates.

As a first step, we defined in our original ILP formulation all the gates except for the remote stands as separate types. The reason we do not consider each remote stand as a separate type is that there does not exist any significant difference between these gates (e.g. they do not have waiting rooms and they all require a bus to transport passengers to and from the terminal building). Our computational experiments reveal that considering all gates as separate types is still computationally feasible. Furthermore, it decreases the amount of work in the second phase, because it limits the second phase to swapping of some gate plans between physical gates and if necessary including unassigned flights and swapping flights between gate plans.

However, the model for assigning flights to gate plans does not yet include all constraints for the case we have a separate type for each gate. The constraint that flights from the United States need extra facilities can directly be included in the definition of which flight is allowed on which type of gates. For minimizing the maximum walking distance we would need to adapt the objective function or introduce preference constraints. Moreover, preventing simultaneous push backs would also require specific constraints. To find out which constraints should preferably be included in the model and which constraints can better be handled manually by the gate planners, is a matter of further research. Presumably, there will always be need for some kind of second phase especially if it is necessary to manually violate some of the constraints to accommodate all flights.

4.3 Computational experiments

We have implemented a model and its solution in C++ and performed extensive computational experiments. All of the tests were conducted on a Pentium 4, 2.8 GHz with 1GB of RAM. For solving the LP problems and for solving the resulting ILP problem by branch-and-bound the Concert Technology interface of CPLEX 9.1.2 (ILOG, 2005) was used.

AAS has provided us with six different sets of data, three of which contain the flights on busy high season days (HS), and three on low season days (LS). From each data set we derived two instances, one instance where we group gates with equal properties into a type and one instance where, except for the remote stands, each individual gate forms a separate type. The sizes of the different data sets are given in Table 4.1. For a data set X, instance X-GG denotes the

Instance	Flights	Gate types	Total gates
HS-1-GG	699	40	128
HS-2-GG	680	40	128
HS-3-GG	688	40	128
LS-1-GG	602	40	128
LS-2-GG	608	40	128
LS-3-GG	593	40	128
HS-1-SG	699	94	128
HS-2-SG	680	94	128
HS-3-SG	688	94	128
LS-1-SG	602	94	128
LS-2-SG	608	94	128
LS-3-SG	593	94	128

Table 4.1: Sizes of the provided instances. LS is low season, HS is high season.

instance with grouped gates and X-SG the instance with a gate type for each individual gate.

During the column generation process we applied the dual simplex method for solving the LPs, which is default in CPLEX, as well as the primal simplex method. In our experiments this hardly made any difference and we report results on the dual simplex method only. To limit the size of the intermediate linear programs, we take out columns with large positive reduced cost, since these are unlikely to improve the current solution. After every given number of iterations we remove all columns from the model with reduced cost above a certain threshold. This threshold is determined by taking the average reduced cost of the columns added in the previous iteration and multiply this average with -0.75.

These removed columns are put in a special repository and are added again when we solve the ILP. We tested different frequencies of taking out columns: from every 40 iterations to every 110 iterations with step size 10, or no take out. In Table 4.2 the results are given for 40, 80, and no take out. We report on the time needed to solve the LP-relaxation, the number of iterations and the number of columns generated.

It is clear that the time needed for solving the LP is considerably bigger for the high-season data sets than for the low-season data sets. Taking out columns

	Take out freq 40			Take out freq 80			No take out		
Instance	sec.	iters.	cols.	sec.	iters.	cols.	sec.	iters.	cols.
HS-1-GG	139	693	14157	139	634	12892	195	557	12236
HS-2-GG	126	625	13622	128	580	12627	165	582	11971
HS-3-GG	110	642	13629	118	595	12628	158	537	12287
LS-1-GG	*	*	*	61	685	9910	82	640	9198
LS-2-GG	62	681	10861	67	599	9944	89	562	9640
LS-3-GG	62	693	10881	67	651	10124	82	597	9783
HS-1-SG	331	765	33004	310	652	29976	447	572	29116
HS-2-SG	300	631	31628	257	582	28410	505	589	28958
HS-3-SG	257	641	31796	236	631	28163	385	573	28423
LS-1-SG	470	1238	27975	160	700	20895	169	641	19968
LS-2-SG	137	673	24041	146	625	21960	161	562	22169
LS-3-SG	127	668	23650	122	621	22119	137	590	22276

* Take out frequency 40 resulted in an unsolvable LP

Table 4.2: Running time, number of iterations, and number of columns generated for various take-out frequency values.

every 40 iterations is faster in many cases. However, in some cases the computation time is strongly enlarged, while in one case the problem could not even be solved at all. The reason for this is that CPLEX gets stuck in a cycle of generating new columns which need some time before they become useful and before they do become useful, they are taken out and CPLEX has to regenerate them again. Our experiments indicate that a *frequency of 80 is the best* although, there is not a large difference with neighboring values.

Recall that in order to solve the resulting ILP in a reasonable time, we need to add the columns from the column pool. Moreover, we add the columns that were taken out and put in a repository. To decrease the size of the branch-and-bound tree we *gamble* for a small integrality gap. In CPLEX we manually initialize an upper bound on the best integral solution at a value of 0.35 percent above the optimal value of the LP-relaxation, which implies that all nodes with a larger lower bound are pruned. This turns out to significantly decrease the solution time. Moreover, we experimented with different settings of CPLEX, such as more elaborate preprocessing and more aggressive cut generation. In Table 4.3 we show the running times for solving the ILP for the default settings of CPLEX and for the enhanced settings. For each instance we give the average running time over the different take-out strategies and the primal and dual simplex

Instance	ILP default (s)	ILP enhanced (s)
HS-1-GG	7	6
HS-2-GG	18	9
HS-3-GG	9	8
LS-1-GG	118*	52*
LS-2-GG	101	24
LS-3-GG	58	24
HS-1-SG	21	19
HS-2-SG	48	24
HS-3-SG	26	15
LS-1-SG	168	120
LS-2-SG	73§	126
LS-3-SG	94	113

* Take out freq 40 resulted in one unsolvable LP.

§ Default CPLEX was not able to solve 3 instances within 60 minutes.

Table 4.3: Running times for solving the ILP with and without enhancements, averaged over three different values for the take-out frequency.

method for solving the LPs during column generation. In the enhanced settings, after 90 seconds, the solution process is aborted and more aggressive parameters settings are used for CPLEX, resulting in more time spent on preprocessing the problem which led to smaller running times for solving the restricted ILP. Moreover, using the enhanced settings we were able to solve more instances.

We will refer to the combination of using a take-out frequency of 80, dual simplex, and the above enhancements for the ILP as the *best variant*.

Finally, we report more details for the best variant. In Table 4.4, the running times, the number of iterations, number of generated columns, the number of columns in the column pool, together with the integrality gap are given. Here Convert denotes the time required to add all unique columns from the pool and repository to the model plus the time needed to convert the LP into an ILP. Again, a clear difference can be seen between the high season data sets and the low season data sets for both the number of generated columns and the number of columns in the column pool. The same holds for the groups of gates and single gates.

Our results indicate that the integrality gap is very small. This implies that

Instance	LP (s)	Convert (s)	ILP (s)	Total (s)
HS-1-GG	139	13	6	160
HS-2-GG	128	12	6	148
HS-3-GG	118	13	5	138
LS-1-GG	61	8	21	91
LS-2-GG	67	8	23	100
LS-3-GG	67	8	5	81
HS-1-SG	310	30	12	355
HS-2-SG	257	28	13	300
HS-3-SG	236	28	13	279
LS-1-SG	160	16	161	340
LS-2-SG	146	18	21	187
LS-3-SG	122	17	196	339

Table 4.4: Time needed for the different phases for best variant.

Instance	Iterations	Columns	Poolsize	Gap (%)
HS-1-GG	634	12892	85225	0.00
HS-2-GG	580	12627	82558	0.00
HS-3-GG	595	12628	84036	0.00
LS-1-GG	685	9910	58374	0.19
LS-2-GG	599	9944	59527	0.21
LS-3-GG	651	10124	59099	0.09
HS-1-SG	652	29976	185970	0.00
HS-2-SG	582	28410	173453	0.00
HS-3-SG	631	28163	172979	0.00
LS-1-SG	700	20895	113413	0.18
LS-2-SG	625	21960	122092	0.21
LS-3-SG	621	22119	118663	0.09

Table 4.5: Number of iterations, number of columns generated, size of the column pool, and integrality gap for best variant.

our method is able to find *practically optimal solutions* for real-life instances in about 10 minutes. Moreover, the results indicate it is feasible to consider single gates as a group. Although this does not make the second phase unnecessary, it makes it easier.

4.4 Conclusion and further research

We have investigated the gate assignment problem at AAS and developed a two phase solution approach. In the first phase we assign the flights to gate plans, while in the second phase we assign the gate plans to the actual gates. It turns out that the second phase boils down to a set of small problems that can be solved manually. For the first phase, we have presented a different way of formulating the gate assignment problem as an ILP. Our model includes all realistic constraints that are actually used for solving the gate assignment problem at AAS. Furthermore, we have described a method to find an approximate solution for this new ILP formulation. We have implemented it in C++, where we use CPLEX 9.1.2 for solving the (I)LPs.

Our implementation was tested with real life input data, provided by AAS. These instances could be solved in a matter of minutes to near-optimality and sometimes even to optimality. The planning software currently in use at AAS is based on a greedy algorithm that assigns flights to gates on the basis of an optimal point score per flight. This score includes different issues, such as preferences of airlines and ground handlers, but is does not contain any robustness measure. Our advanced LP-based algorithm and the planning software of AAS using a greedy algorithm have comparable running times. Experiences from the gate planners at AAS reveal that currently a considerable amount of time (a few hours) has to be spent on replanning the computer generated schedule in order to make it more robust, because the current planning software does not consider robustness at all. Since our algorithm focuses explicitly on robustness, the gate planners have more time for solving the conflicts that arise during the actual day. Clearly, extensive simulations and testing is required to analyse the quality of our solutions compared to the solutions presented by the current planning software.

As a first step towards the integration of the first and second phase, we defined in our first phase model every individual gate as a separate type, except for

the platform stands. Our experiments show that the problem then still can be solved within a few minutes. By increasing the number of types and defining the gate types more restrictively, we could include more constraints and preferences in the first phase. Further research possibilities are to model and implement more of the rules that the gate planners use to assign the flights to gates.

Finally, our algorithm allows parallelization, e.g. solving in parallel the pricing problems for different gate type and determining in parallel different columns for the column pool by omitting a single flight. This might be of interest when the algorithm is applied for replanning during the day of operation, because then speed is a crucial factor.

CHAPTER

FIVE

Solving the bus planning problem

5.1 Introduction

At Amsterdam Airport Schiphol (AAS) one can distinguish two types of stands where aircraft can be assigned to. There are the regular gates that are equipped with a passenger air bridge, and there are the so-called remote stands which do not have a bridge connection to the terminal. When an aircraft is assigned to a stand with a passenger air bridge, the passengers can (dis)embark the aircraft via this bridge. On the other hand, when an aircraft is assigned to a remote stand, the passengers will need to be transported to or from the terminal building. Since it is not an option that passengers just walk from the remote stand to the terminal building or vice versa, buses are used to transport them.

Since there are a limited number of buses available we need to determine which bus will drive from where to where at which time. This problem we will refer to as the *bus planning problem.*

When we look at the characteristics of the bus planning problem, one can

see similarities with some other problems. One related problem is the vehicle scheduling problem of buses in public transport. In both problems we need to determine which buses will drive what rides. A difference between the two problems is the fact that with bus planning at AAS only short rides (in the order of minutes) need to be assigned to a bus, while in the case of public transport the rides are complete lines from beginning to end and they take much longer (generally in the order of one hour or more). Furthermore, at an airport the distances are very limited and generally the time needed to drive from the end point of one ride to the start point of the next ride is very little. This means that we can be very flexible with regards to creating combinations of rides that need to be driven by one bus. This in contrast to the vehicle scheduling problem in public transport, where the time needed to drive from the end point of one ride to the start point of the next ride can be significant.

The problem also has similarities with various problems in the field of Automated Guided Vehicles (AGV). For example, the problem of routing AGV in a container terminal is related to bus planning at an airport because in both cases the goal is to assign rides to a vehicle. One major practical difference between bus planning and routing AGV is the fact that AGV have no drivers and thus there is no need for mandatory (lunch)-breaks as required by law. Le-Anh and de Koster (2006) give a review of the design and control of AGV systems. They not only discuss the online case, but also give an overview of the offline scheduling of AGV. This offline scheduling problem is similar to bus planning, since in both problems we know everything in advance and need to create a schedule subject to certain criteria. They indicate that the offline scheduling problem of AGV is similar to the Pickup and Delivery Problem with Time Windows (PDPTW).

More recently, Ropke and Cordeau (2006) proposed an algorithm for solving the PDPTW that is based on a column generation approach. Though the bus planning problem at an airport has some similarities to the PDPTW, one major difference is that at an airport, the buses must drive straight from the pickup point to the delivery point and cannot pick up more passengers in the meantime and thus the time windows are very strict. Furthermore, the same remark made earlier regarding the limitation on the distances also holds for the comparison with the PDPTW. So bus planning in this way is a special version of the PDPTW.

5.2 Problem description

As mentioned in the previous section, platform buses are used to transport the passengers to or from the aircraft standing on remote stands. For this transportation two types of buses are used at AAS. The first type of bus is called the *peak bus*, which are standard buses that are also used for city trips in public transportation. Peak buses are capable of transporting up to 50 persons at once.

The second type of bus is the *Cobus*. It is a bus that is specifically designed for use at airports. Compared to the peak bus it has several advantages, a clear one is that it is larger and can transport up to 70 persons at once. Other advantages are that the bus has a lower floor (thus giving easy access) and it has doors on both sides. This latter advantage is particularly useful, since it will not matter from which direction an aircraft at the platform is approached.

Every season the planners at AAS look at the planned flights and estimate the number of buses needed in each 15 minute time interval of a day for the complete upcoming season. This information is then sent to the Haagse Tram Maatschappij (HTM) bus company and they will create *shifts* for buses and drivers such that they fulfill the capacity requirements for each of the 15 minute intervals. The shifts have four attributes, namely a *starting time*, an *ending time*, a *type of bus* that drives the shift, and the *number* of buses within the shift. Each bus within a shift is driven by one driver.

The planners at AAS don't have a direct influence on these shifts and they must use the shift information as input for their planning. Looking at the type of bus driving the shifts, one can see that the majority of the shifts at AAS are driven by the Cobuses.

Considering passengers, we can see that for departing flights the passengers enter the bus at the bus gate at the terminal. The bus then drives to the aircraft at the platform after which the passengers disembark the bus to board the aircraft. The combination of these three events we call a *trip*. For arriving flights one can likewise define trips consisting of these three phases, but then the origin is the aircraft and the destination is the bus gate. Furthermore, the time needed for the passengers to embark or disembark the bus is set to five minutes, independent of the number of passengers and the type of bus.

Since the number of passengers of each flight is known (the airlines should provide this to AAS), one can determine the number of trips that is needed to transport all passengers to or from the aircraft. Due to the fact that the number of passengers that can be transported with a Cobus differs from the number of passengers that can be transported with a peak bus, sometimes not all trips actually need to be served. An example of such a situation would be the transportation of 60 passengers: we either need two peak buses for this, or just one Cobus. If we create two trips but one of these trips is driven by a Cobus, the second trip is redundant because all passengers already could be transported with the one Cobus.

After running preliminary experiments we decided that in the case where there are trips for a departing flight that are possibly redundant, depending on whether the other trips for the departing flight are driven by Cobuses, we will only allow Cobuses to drive the trips for this departing flight. The reason for deciding this is that the majority of the buses that are available at AAS are Cobuses. Furthermore, this situation does not occur often. One explanation for this is a preference at AAS to send a Cobus as the first bus to an aircraft in case of a departing flight. This means that for departing flights only situations where the number of passengers is between 120 (one Cobus and one peak bus) and 140 (two Cobuses) would result in a possibly redundant trip because we need to send at least two buses, but possibly three. These numbers of passengers are relatively high and such flights are seldomly assigned to remote stands.

The way the trips are generated is prescribed by AAS and is different for arriving and departing aircraft. For an arriving aircraft it is required that there is exactly the number of buses needed to transport all passengers that need to disembark after the aircraft has come to a full stop at the remote stand.

For a departing aircraft the creation of trips is different. One major difference is that regardless of the number of passengers there must always be at least two trips for a departing flight. Furthermore, in contrast to the trips for arriving aircraft, not all trips start at the same time. The last trip must be finished with disembarking the passengers five minutes before the actual departure of the flight. The trip before the last trip must be finished with disembarking the passengers 10 minutes before the actual departure of the flight, etc. Since we know the time it takes for embarking and disembarking the passengers in the bus and the time it takes to drive from the bus station to the remote stand, we can calculate at what time a trip should start, given its end time.

Our goal is to find a planning for the buses that is as *robust* as possible, where robust means that having a small change in arrival or departure times of flights during the actual day itself does not imply a lot of replanning. A possible measure is to look at the *idle time* between pairs of trips that are consecutively assigned to the same bus. The idle time between two trips t and t' that are assigned consecutively to the same bus with trip t being the first trip, is defined as follows:

$$\text{idle time}(t, t') = T_{t'}^{\text{start}} - T_t^{\text{end}} - \text{driving time}(t, t'),$$

where T_t^{start} and T_t^{end} denote the start and the end time respectively of a trip t. By subtracting the "driving time" from the end location of the first trip to the start location of the second trip, we clearly obtain a nett idle time between the two trips.

To achieve a robust schedule we want to ensure that all idle times between pairs of trips that are consecutively assigned to the same bus are as big as possible. To model this within a cost function we chose to use the cost function c^B based on the one we used in Chapter 4 for the gate assignment problem:

$$c^B(\text{idle}) = 1000(\arctan(-0.21 \times \text{idle}) + \frac{\pi}{2}),$$

where idle is the idle time between two trips.

Driving times are provided by AAS in minutes and are based on reference points. In this system, each of the platforms is represented by one reference point. This means that the driving time from one end of a platform to another end of the same platform would take zero minutes of driving time, since the whole platform uses the same reference point. To overcome this problem of zero minute driving times and to build in a bit of robustness in the short driving times, all driving times are set to at least five minutes. These five minutes are sufficient to deal also with the fact that buses are not allowed to drive freely everywhere, but must give right to aircraft and sometimes must wait till they can drive behind a just parked aircraft, till engines are switched off, etc.

Using the above cost function we ensure that very short idle times get penalized very strongly, while bigger idle times get a lower cost. We did not make use of a so-called *cut-off value* (i.e. a threshold beyond which the cost will not decrease anymore if the idle time becomes bigger), since using such a cut-off value would introduce symmetry in the model. This is not preferable because it makes the resulting ILP problems more difficult to solve.

From discussions with the planners at AAS, we learned that in the case of a departing flight, there is a preference to let the first trip to the aircraft be driven by a Cobus. This is not a mandatory rule, but in case of choice the planners prefer this. To model this, we use an additional penalty cost in case the first trip to a departing aircraft is driven by a peak bus.

5.3 Problem formulation

Similar to the gate assignment problem in Chapter 4 we have multiple ways of formulating the bus planning problem as an Integer Linear Program (ILP). One standard approach is to introduce binary variables indicating whether a bus serves one trip after another trip (this order is needed to calculate the cost because it is based on the time between two trips) but, as for gate assignment, this approach would result in a model that is way too large to handle.

To cope with this problem we will use an ILP formulation similar to the formulation we used in Chapter 4 for the gate assignment problem.

With the gate assignment problem we created gate plans that consist of flights that are to be assigned to the same physical gate. Now we will introduce *bus plans*, which consist of a set of trips that will be driven by one bus. Similar to letting gate plans correspond to gate types, we let bus plans correspond to shifts. For a given bus plan to be feasible we require that:

- all trips present in the bus plan can be realized within the start and end times of the associated shift;
- no two trips are conflicting (i.e. there must be enough time to drive from the destination of one trip to the origin of the next trip).

For the shifts that last longer than 4.5 hours (so-called *long shifts*), we need an extra constraint on the set of trips being driven. Whenever a driver has a shift that lasts more than 4.5 hours, he is required by law to have a 45 minute break that does not start within 1.5 hours of the shift start time and does not end within 1.5 hours of the shift end time. So somewhere around the middle of the shift, a 45 minute period of time should be reserved for a break. These 45

minutes do not include the time it takes the driver to drive from the last trip before the break to the canteen and from the canteen to the start of the first trip after the break.

One problem that is introduced by arriving flights with many passengers is that several trips are needed, all with exactly the same origin and destination. This results in symmetry in the model that we would rather not have. To resolve this situation we introduce so-called *supertrips*. A supertrip is like a normal trip, except that instead of having to be driven by exactly one bus, it must be driven by q buses, where q is determined by the number of passengers to be transported. So instead of having q normal trips that all need to be driven once, we now have one supertrip that needs to be driven exactly q times.

As for the gate assignment problem in Chapter 4 we formulate the bus planning problem as an ILP that uses this notion of bus plans in the following way. Suppose that we are given the complete set of all possible bus plans. To solve the problem, we have to select a subset from this complete set such that each trip is present in exactly one of the selected bus plans. We must ensure that we have as many bus plans selected for a given shift as there are buses within that shift. Furthermore, we can determine the cost c_j^B of a bus plan j by summing the cost of all successive pairs of trips in the bus plan. The objective now is to minimize the total cost of the set of the selected bus plans.

Introduce a binary variable y_j for each bus plan j as follows:

$$y_j = \begin{cases} 1 & \text{if bus plan } j \text{ is selected;} \\ 0 & \text{otherwise.} \end{cases}$$

To ensure each trip is driven the correct number of times, we add the following constraints:

$$\text{UAT}_t + \sum_{j=1}^{M} h_{tj} y_j = S_t, \tag{5.1}$$

where

$$h_{tj} = \begin{cases} 1 & \text{if trip } t \text{ is present in bus plan } j; \\ 0 & \text{otherwise,} \end{cases}$$

M denotes the total number of available bus plans, and S_t denotes the number of times trip t needs to be covered, which is equal to one for all normal trips and for a supertrip it is equal to the number of trips that the supertrip is composed

of. Finally, UAT_t (unassigned trip) allows for trip t being unassigned in case not enough buses are present to drive all trips at a certain time. To ensure trips are assigned whenever enough buses are present, the UAT_t variable for each trip t gets a very high cost coefficient R_t in the objective function. The value of this cost coefficient has been determined through preliminary experiments.

To ensure we select as many bus plans for each shift as there are buses present for that shift we add the following constraint for each shift b:

$$\sum_{j=1}^{M} f_{jb} y_j = T_b, \tag{5.2}$$

where

$$f_{jb} = \begin{cases} 1 & \text{if bus plan } j \text{ corresponds to shift } b; \\ 0 & \text{otherwise,} \end{cases}$$

and T_b denotes the number of buses shift b consists of.

Now the complete model is as follows:

$$\text{Minimize} \sum_{j=1}^{M} c_j^B y_j + \sum_{t=1}^{T} R_t \text{UAT}_t$$

subject to:

$$\begin{aligned} \text{UAT}_t + \sum_{j=1}^{M} h_{tj} y_j &= S_t \quad \text{for } t = 1, \ldots, T && (5.3) \\ \sum_{j=1}^{M} f_{jb} y_j &= T_b \quad \text{for } b = 1, \ldots, B && (5.4) \\ y_j &\in \{0, 1\} \quad \text{for } j = 1, \ldots, n. && (5.5) \end{aligned}$$

5.3.1 Pricing problem

We solve the model for the bus planning problem by means of column generation. Since the total number of possible bus plans is enormous, we work with a small subset of the possible columns. For this subset we solve the above model, which gives us a dual multiplier ϕ_t for the constraint 5.3 corresponding to trip t and a

dual multiplier ω_b for the constraint 5.4 corresponding to shift b. The reduced cost of a bus plan j for shift b is equal to

$$c_j - \sum_{t=1}^{T} h_{tj}\phi_t - \omega_b.$$

After solving the above model with the subset of the bus plans, we need to determine whether there exists a bus plan with negative reduced costs. It is not possible to check the reduced cost of all bus plans, again because the total number of possible bus plans is enormous. In the pricing problem we want to find the bus plan with minimum reduced cost. As soon as we have found the minimum there are two options: the bus plan has negative reduced cost, meaning that adding this bus plan to the model might decrease the objective value. Or, the bus plan has reduced cost ≥ 0, meaning that adding this bus plan will not decrease the objective value and thus the optimum is found.

To solve the pricing problem we introduce a Directed Acyclic Graph (DAG) $G_b = (V, E)$ for every shift b. We add a vertex to G_b if trip t is within the start and end time of shift b. We add an edge between two vertices if the two corresponding trips are not conflicting. Now a path through the graph G_b represents a feasible bus plan for shift b and vice versa, all feasible bus plans for shift b can be represented by a path through the graph G_b.

The above only holds for shifts which do not require a mandatory break. In case we have a long shift, we have to provide a way of putting a break somewhere in the middle of the shift. We can do this by adding *break vertices* to the graphs corresponding to shifts which last longer than 4.5 hours in the following way:

- Add a break vertex between two trips which are far away in time, such that there is time for a break of at least 45 minutes and the driving time to and from the break location,
- Add a break vertex between the source vertex and any vertex of which the corresponding trip has a start time that is at least 135 minutes (i.e. 90 minutes buffer and 45 minutes break) later than the start time of the shift. This indicates a driver starts his shift and does nothing and then starts his break,

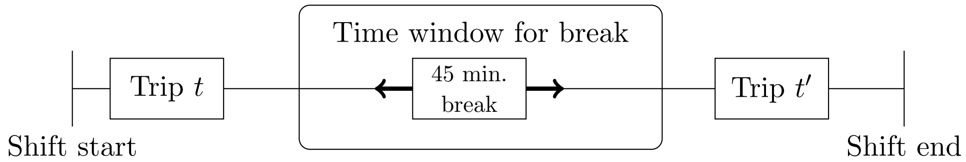

Figure 5.1: A time window within which a break must be held.

- Add a break vertex between any vertex of which the corresponding trip has an end time that is least 135 minutes earlier than the end time of the shift and the sink vertex. This indicates a drivers has his break and after that does not drive any trips anymore for the duration of his shift.

Finally, we also add the extreme situation in which a driver starts the shift, and is not assigned to any trip for the complete duration of the shift and only is assigned to have a break within the given limits. This is accomplished by adding a break vertex between the source and sink vertex, with an edge from the source to the break and one from the break to the sink.

With these break vertices added we can represent any feasible bus plan for a long-shift b with a path through the graph G_b, but now not every path through the graph corresponds to a feasible bus plan (i.e. paths that contain no break vertex or more than one break vertex do not correspond to feasible bus plans). We will address this issue while solving the pricing problem and ensure that only paths corresponding to feasible bus plans are considered.

One note to be made regarding the breaks is that with the above approach we do not plan the exact start time and end time of a break but we give a *time window* in which the break must be held. An example of this time window can be seen in Figure 5.1 where the 45 minute break must take place somewhere in the depicted time window.

We now want to put cost on each of the edges in such a way that a path through the graph does not only correspond to a feasible bus plan, but also that the total cost of the path is equal to the reduced cost of the corresponding bus plan. When looking at the reduced cost of a bus plan, the contribution of trip t if it is followed by trip t' is exactly:

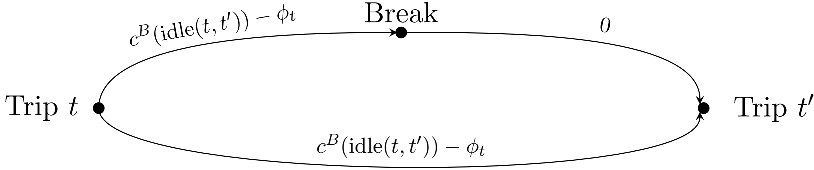

Figure 5.2: Example of possible break between two trips.

$$c^B(\text{idle}(t,t')) - \phi_t,$$

where ϕ_t is the dual multiplier corresponding to the trip constraint (5.1) corresponding to trip t. In case of a possible break between trip t and t', denoted by the presence of a secondary path from t to t' through a break vertex the cost of the edge from vertex t to the break vertex will be equal to the cost of the edge from vertex t to vertex t'. The edge from the break vertex to vertex t' will have cost 0. This cost distribution is depicted in Figure 5.2.

Furthermore, the cost of all outgoing edges from the source vertex will get cost $-\omega_b$, where ω_b is the dual multiplier corresponding to the constraint (5.2) for shift b.

In case of a shift without a break, a path through the graph corresponds to a feasible bus plan and the total cost of the path is equal to the reduced cost of the bus plan. In case of a shift that needs a mandatory break, any path through the graph that contains exactly one break vertex also corresponds to a feasible bus plan and again the total cost of the path is equal to the reduced cost of the bus plan. Finding the bus plan with minimum reduced cost for a given shift b now comes down to finding the shortest source-to-sink path in the graph G_b. For the shifts that have a duration of less than 4.5 hours, solving the shortest path problem is straight-forward. Since we have a DAG with topological order this can be done in $\mathcal{O}(V+E)$ time (cf. Cormen et al. (2001)). For the shifts that have a duration longer than 4.5 hours, and therefore require a mandatory break, determining the bus plan with minimum reduced cost takes a bit more work. Instead of having a dynamic programming problem with only one state variable per vertex for holding the cost of the minimum cost path to this vertex, we also need a second state variable for holding the cost of the minimum cost path with a break to this vertex. Furthermore, we also need to keep two references to the predecessors: one for the case a break has been taken already and one for the

case no break has been taken.

Additionally, we must make two minor modifications to the behavior of updating the cost of the successor vertices in the dynamic programming algorithm:

- Do not update the minimum cost value with a break of any successor that is a break vertex because it is not possible to have more than one break vertex in a path.
- When the current vertex is a break vertex, only update the minimum cost value with a break of the successors with the minimum cost value without a break of the current vertex. Recall from Figure 5.2 that the cost of the outgoing edges of a break vertex is set to 0.

The required modifications will not change the complexity of the algorithm because we only add a constant number of additional operations per vertex. So the running time of the algorithm for finding the shortest path with breaks also is $\mathcal{O}(V + E)$.

If our shortest path algorithm cannot find a new bus plan with negative reduced costs for none of the bus shifts, the value of the LP-relaxation cannot be improved anymore and this means that the LP-relaxation has been solved to optimality.

5.3.2 Solving the ILP

Since the solution to the LP-relaxation might be fractional, we have to ensure that we end up with an integral solution. We achieve this by reinserting the integrality constraints and solve the problem with the set of columns generated during the column generation as an ILP. It turns out that using only the set of columns generated during the column generation is often too restrictive for the ILP, in the sense that this results in quite bad integral solutions, which come at the expense of big running times.

As in the previous chapter we resolve this difficulty by creating a set of extra columns during the column generation process. These extra columns are put in a column pool and are generated in the following way:

- For each trip-vertex belonging to the path with minimum cost:
  - Take the vertex out of the graph
  - Solve the shortest path problem again
  - Put the bus plan corresponding to the new shortest path in the column pool
  - Reinsert the vertex again in the graph.

Note that when solving the shortest path after one of the vertices has been removed, we do not require the total cost of the shortest path to be negative (i.e. we do not require negative reduced cost for these additional columns).

By generating the columns this way they are not completely random, but quite related to the columns needed for solving the LP-relaxation to optimality.

After the column generation is finished, all unique columns from the column pool are added as candidates to the ILP-formulation of the problem. After these columns have been added, the problem has become considerably larger in number of variables. Fortunately, the set of columns is now less restrictive, enabling the solver to find feasible, good integral solutions easier.

Our preliminary experiments showed that the integrality gap between the solution value of the LP-relaxation and the final integral solution was really small. Similar to solving the final ILP for the gate assignment problem in Chapter 4, we can solve the present ILP in two stages. First we assume a small integrality gap by supplying an upper bound to the solver which will prune all nodes with a relaxation value above it (i.e. it behaves as if an integral solution with the given value has been found already). Due to this tight upper bound, we can also have the solver put more emphasis on finding integral solutions, instead of proving the optimum. Setting the emphasis on integral solutions will result in different branching schemes and node selections being used by the solver.

If this tight upper bound leads to an infeasible ILP problem, we will remove the upper bound and solve the problem again. After preliminary experiments, the upper bound was set such that for all instances the ILP problem was still feasible.

Instance	# Flights	# trips	# supertrips	# Shifts	# buses
1	198	299	49	14	50
2	230	346	41	21	63
3	248	372	41	23	61
4	252	379	31	24	64
5	253	380	31	19	64
6	202	305	17	23	62
7	227	343	43	20	62
8	192	292	25	28	113
9	261	391	37	24	61
10	249	378	23	24	64
11	250	378	28	21	64
12	249	377	35	26	62
13	254	382	54	21	62
14	199	303	36	13	50
15	222	332	32	21	63
16	251	378	40	22	61
17	260	390	21	24	64
18	253	382	29	23	64
19	263	394	35	23	62
20	253	382	58	20	62
21	192	290	50	14	50
22	214	318	42	21	63
23	245	369	41	26	61
24	230	344	16	25	64
25	250	373	22	21	64
26	252	380	33	24	62
27	257	387	43	22	62
28	192	287	38	15	50
29	199	302	24	20	63
30	228	342	27	26	61

Table 5.1: Instance sizes.

5.4 Computational experiments

For testing our solution method for the bus planning problem we have implemented the algorithm in C++. The computer on which we ran the experiments was equipped with an Intel Pentium 4 3.00 GHz processor and 1 GB of RAM. For solving the LPs and ILPs we made use of the Concert Technology interface to CPLEX 9.1.3 (ILOG, 2005).

For testing our algorithm, AAS provided us with all data regarding buses and flights for one complete month. These data concerned the real arrival and departure times of the flights as they were known at the end of the day. Hence, we cannot compare our results fully to the results of the approach currently in use at AAS, which uses the expected data. Another complicating factor is that

the current approach works by planning everything with a rolling time horizon of three hours ahead, while our solution method considers the whole day.

In Table 5.1 the sizes of the instances provided by AAS are given. The column with the number of supertrips gives the number out of the total number of trips which represent supertrips (i.e. trips that need to be covered more than once). In the column with the number of trips the supertrips are counted as one trip. Almost all of the supertrips consist of trips that need to be covered just twice.

The effect of the improvements separately

To avoid symmetry in the model, we introduced supertrips. To investigate the effect of using supertrips, we solved the instances not only with supertrips enabled, but also with supertrips disabled (i.e. every supertrip t is replaced by S_t identical separate trips). Furthermore, we also investigate the effect of *column deletion* (CD), which is removing columns from the model every given number of iterations.

In Table 5.2 and Table 5.3 we present the different results regarding solving the LP in case supertrips are disabled and enabled respectively.

As in solving the gate assignment problem in Chapter 4 we also make use of column deletion while solving the bus planning problem. We experimented with different values for the parameters of the column deletion and based on these experiments we used the following values for the parameters: every 30 iterations we determine the average of the reduced cost of the column added in the previous iteration and remove all columns from the model that have reduced cost larger than -0.75 times this average.

For supertrips disabled and enabled, we solved the problem both without and with column deletion and, we present for all cases the number of iterations it took for the column generation to find the optimum, as well as the number of columns that were present in the model when the column generation was finished.

We can see that that the effect of using column deletion on the number of iterations needed goes both ways: there are some instances for which the use

of column deletion decreases the total number of iterations needed, but there are also instances where the use of column deletion increases the number of iterations.

The use of column deletion does yield a significant reduction in the number of variables present in the model when the optimum is reached; reductions up to 80% can be seen. This massive reduction results in a shorter time needed for re-solving the restricted master problem after adding the new columns in each iteration of the column generation process.

The fact that the models in each iteration are smaller in size, meaning they take less time to be solved, results in the decrease in running times as seen in Table 5.2 and Table 5.3. Even in the case where more column generation iterations are needed to solve the LP-relaxation, the total time needed for solving the LP-relaxation decreases due to the size of the models in each iteration.

Default CPLEX versus CPLEX with improvements

In Table 5.4 the results of solving the problem with both the use of supertrips and the use of column deletion is compared to solving the problem without these two improvements and is created by combining Table 5.2 and Table 5.3. We refer to case with the two improvements as *Enhanced* and the case without them as *Default*.

When we look at Table 5.4 we can see that the combination of using column deletion and using supertrips yields a significant speedup. The minimum improvement encountered is a speedup of a factor 1.49, while the average speedup is 2.39. Furthermore, when looking at the speedup achieved compared to the size of the running time of the original problem, we can see that there is a trend of the speedup increasing with increasing running times of the original problems.

In Table 5.5 we present information regarding the resulting ILP problems. We compare the time needed for solving the ILP both without and with the upper bound and we can see that although guessing a relative tight upper bound on the ILP solution does not always yield a decrease in the time needed for solving the ILP problem, it enables us to solve all the ILP models within two minutes. Without using the tight upper bound approach two of the instances are not

Instance	Iterations		Columns in model		Time (s) for LP	
	No CD	With CD	No CD	With CD	No CD	With CD
1	568	621	6023	1444	103.6	79.5
2	459	428	7664	1739	172.4	106.0
3	534	559	8975	1872	241.7	158.2
4	549	598	9279	2220	246.2	163.2
5	610	684	8794	1956	236.2	160.8
6	306	301	5431	1152	51.6	38.2
7	525	560	7609	1804	168.6	111.2
8	375	420	7833	2604	68.0	63.9
10	584	570	9431	2091	292.5	173.5
11	469	489	8780	1816	214.4	143.2
12	594	563	9321	1884	311.0	139.6
13	535	486	9008	1883	319.5	145.7
14	685	635	9671	2007	439.4	196.3
15	663	709	6178	1406	97.6	72.2
16	345	373	6041	1434	100.1	76.9
17	542	538	8391	1927	212.2	155.1
18	630	589	10243	1900	305.0	167.6
19	475	511	9020	2128	246.1	152.3
20	529	640	9157	1749	304.1	206.2
21	586	585	8842	1808	301.8	167.7
22	568	602	5778	1359	79.1	61.2
23	421	425	7296	1531	122.8	79.8
24	509	518	9701	1913	318.2	157.6
25	449	478	8157	1830	145.0	87.3
26	578	536	9272	1933	267.2	132.4
27	527	544	9375	1900	281.6	164.3
28	670	738	9637	1926	575.9	237.9
29	586	606	6246	1503	88.2	62.7
30	435	456	6462	1708	81.0	56.6
31	427	418	7763	1797	130.7	85.2

Table 5.2: Overview results LP without supertrips, both without and with column deletion.

solvable within a set time-limit of one hour.

Looking at the instances that are solved by both the default and the enhanced settings, we can see that the time needed by the default settings often is less than the time needed when the enhanced settings are used. This can be explained by the fact that with the enhanced settings a considerable amount of time is spent by CPLEX on pre-processing the root-node of the branch-and-bound tree. Due to the aggressive settings supplied to CPLEX for cut-generation, solving the LP-relaxation of the root-node and adding the cuts takes quite some time.

After the rootnode has been solved, the upper bound limit is obtained by multiplying the value of the solved rootnode with the required maximum integrality

Instance	Iterations		Columns in model		Time (s) for LP	
	No CD	With CD	No CD	With CD	No CD	With CD
1	523	586	5325	1050	57.6	49.2
2	377	404	6334	1521	90.9	69.8
3	501	481	8187	1648	175.2	99.0
4	521	518	8779	1959	171.0	111.5
5	533	545	7752	1787	152.0	105.4
6	271	329	4927	1136	37.3	32.5
7	482	490	6834	1438	107.1	70.1
8	360	371	7317	2157	49.9	45.8
10	527	523	8352	1778	208.9	119.4
11	411	483	7794	1807	148.7	114.9
12	528	530	8353	1788	182.3	111.3
13	516	463	8127	1678	300.2	109.4
14	588	577	8163	1533	302.7	116.2
15	639	655	5750	1121	68.6	48.6
16	336	377	5600	1273	69.6	58.6
17	465	516	7458	1701	145.6	107.4
18	604	564	9771	1700	266.0	140.3
19	445	462	8135	1873	170.8	116.1
20	508	509	8363	1670	211.0	127.2
21	500	539	7624	1487	171.7	101.1
22	483	514	4843	1100	43.1	33.9
23	364	367	6149	1251	67.1	50.4
24	476	470	8923	1577	228.0	103.2
25	446	461	7937	1615	125.8	71.3
26	529	532	8620	1851	257.3	109.3
27	524	493	8887	1584	237.1	121.4
28	581	599	8317	1573	283.3	123.2
29	512	554	5453	1295	53.0	41.3
30	419	426	6115	1375	61.2	42.4
31	409	372	7337	1568	98.2	62.9

Table 5.3: Overview results LP with supertrips, both with and without column deletion.

gap and the settings for cut-generation are set to default again.

Furthermore, we set the emphasis for the solver to finding integral solutions because all integral solutions will fall within the given tight integrality gap.

In Table 5.6 we present all information regarding the best settings for solving the problem. We can see from the table that the time needed for solving the pricing problems is about 50% of the total time needed for solving the LP-relaxation.

In the time needed for solving the pricing problems also the time needed for creating the additional bus plans for the column pool is incorporated. The creation of these additional bus plans accounts for roughly 70% of the time

Instance	Iterations		Columns in model		Time (s) for LP	
	Default	Enhanced	Default	Enhanced	Default	Enhanced
1	568	586	6023	1050	103.6	49.2
2	459	404	7664	1521	172.4	69.8
3	534	481	8975	1648	241.7	99.0
4	549	518	9279	1959	246.2	111.5
5	610	545	8794	1787	236.2	105.4
6	306	329	5431	1136	51.6	32.5
7	525	490	7609	1438	168.6	70.1
8	375	371	7833	2157	68.0	45.8
10	584	523	9431	1778	292.5	119.4
11	469	483	8780	1807	214.4	114.9
12	594	530	9321	1788	311.0	111.3
13	535	463	9008	1678	319.5	109.4
14	685	577	9671	1533	439.4	116.2
15	663	655	6178	1121	97.6	48.6
16	345	377	6041	1273	100.1	58.6
17	542	516	8391	1701	212.2	107.4
18	630	564	10243	1700	305.0	140.3
19	475	462	9020	1873	246.1	116.1
20	529	509	9157	1670	304.1	127.2
21	586	539	8842	1487	301.8	101.1
22	568	514	5778	1100	79.1	33.9
23	421	367	7296	1251	122.8	50.4
24	509	470	9701	1577	318.2	103.2
25	449	461	8157	1615	145.0	71.3
26	578	532	9272	1851	267.2	109.3
27	527	493	9375	1584	281.6	121.4
28	670	599	9637	1573	575.9	123.2
29	586	554	6246	1295	88.2	41.3
30	435	426	6462	1375	81.0	42.4
31	427	372	7763	1568	130.7	62.9

Table 5.4: Overview results LP of default approach and enhanced approach (i.e. with supertrips and column deletion).

needed for solving the pricing problems. Due to its structure the creation of these additional columns could be separated over multiple computers or threads because their creation only depends on the graph and a set of dual multipliers.

Since in most cases the time needed for solving the LP-relaxation is larger than the time needed for solving the resulting ILP, a considerable improvement of the total running times can be achieved by parallelizing the solving of the pricing problems and also the creation of the additional columns.

To investigate the effect on the time needed for solving both the LP-relaxation and the ILP, we plot the respective running times against the number of trips.

Instance	Number of rows	Number of columns	Percentage Non-Zero	Time (s) ILP Default	Time (s) ILP Enhanced
1	313	40615	3.38	51.9	31.1
2	367	49384	2.98	5.6	13.0
3	395	58404	2.68	70.3	32.3
4	403	67855	2.65	5.4	17.9
5	399	58095	2.77	6.5	10.5
6	328	36047	3.1	1.2	3.9
7	363	51173	3.03	3.5	8.4
8	320	46378	2.69	1.8	4.6
10	415	62555	2.63	8.6	22.9
11	402	65530	2.69	*	54.7
12	399	64603	2.76	4.6	11.5
13	403	61269	2.73	5.1	9.2
14	403	61517	2.76	4.3	9.0
15	316	41331	3.19	3.4	9.4
16	353	47087	3	*	101.7
17	400	57743	2.7	39.5	35.5
18	414	71978	2.64	12.0	27.5
19	405	68003	2.77	7.7	18.7
20	417	65201	2.69	4.7	11.1
21	402	58176	2.79	2.6	6.2
22	304	33664	3.21	4.0	15.4
23	339	44892	3.06	3.1	5.2
24	395	65459	2.67	6.3	9.7
25	369	54702	2.73	2.2	5.9
26	394	65128	2.75	6.5	13.1
27	404	67885	2.71	5.0	9.7
28	409	62702	2.71	6.2	14.7
29	302	38544	3.25	2.3	7.0
30	322	42764	3	2.9	6.8
31	368	49768	2.69	2.7	6.5

*: Not Solvable within one hour

Table 5.5: The ILP information for both the default settings and enhanced settings with a tight upper bound.

In Figure 5.3 we have plotted the time needed for solving the LP-relaxation against the number of trips. We can see that the use of supertrips only incurs an improvement on the solution time for the LP-relaxation due to the decreased number of trips in the model. It can be seen that the solution times needed when solving the LP-relaxation without supertrips are in line with what would be expected when solving problem with larger amount of trips.

When we take a look at Figure 5.4, where we have plotted the time needed for solving the ILP against the number of trips, we can see that the time needed for solving the resulting ILP behaves a bit more erratic compared to the running times needed for the LP. We can see though that generally the time needed for

Instance	Number of iterations	Size of pool	Columns taken out	Time (s) needed for solving			
				Pricing	LP	ILP	Total
1	586	50189	4800	25.9	49.2	31.1	87.2
2	404	59287	5205	37.8	69.8	13.0	93.3
3	481	68237	6631	44.7	99.0	32.3	146.2
4	518	79710	7465	58.6	111.5	17.9	146.3
5	545	71700	6376	56.3	105.4	10.5	131.1
6	329	41757	4233	17.1	32.5	3.9	43.7
7	490	62626	5734	37.7	70.1	8.4	91.0
8	371	45355	5746	26.5	45.8	4.6	59.0
10	523	75863	6963	58.2	119.4	22.9	158.9
11	483	75460	7190	58.3	114.9	54.7	186.0
12	530	77342	6996	60.3	111.3	11.5	138.8
13	463	71605	6586	53.9	109.4	9.2	135.1
14	577	75736	6897	57.9	116.2	9.0	141.2
15	655	51075	5161	25.1	48.6	9.4	65.0
16	377	53986	5220	30.7	58.6	101.7	169.8
17	516	70275	6529	53.7	107.4	35.5	158.2
18	564	85648	8189	71.0	140.3	27.5	185.9
19	462	79072	6877	59.6	116.1	18.7	151.8
20	509	78166	7110	67.0	127.2	11.1	156.6
21	539	72740	6441	54.6	101.1	6.2	123.1
22	514	42103	4168	18.4	33.9	15.4	55.4
23	367	52971	5177	26.5	50.4	5.2	64.5
24	470	78872	7729	47.3	103.2	9.7	129.2
25	461	64228	6808	37.8	71.3	5.9	90.3
26	532	76408	6978	56.0	109.3	13.1	138.6
27	493	78797	7530	61.2	121.4	9.7	148.0
28	599	76563	6946	60.4	123.2	14.7	154.5
29	554	47765	4695	20.3	41.3	7.0	54.5
30	426	47560	4954	21.4	42.4	6.8	57.2
31	372	56164	5847	29.4	62.9	6.5	81.6

Table 5.6: All information with regards to the optimal approach.

solving the ILP is influenced in a positive way when the symmetry is removed with using the supertrips.

The effect of using supertrips simplifies the problem in two ways: first of all the symmetry is removed, secondly the number of trips is decreased. While solving the LP-relaxation benefits more from the decrease in the number of trips, solving the resulting ILP benefits more from the fact that the symmetry is removed.

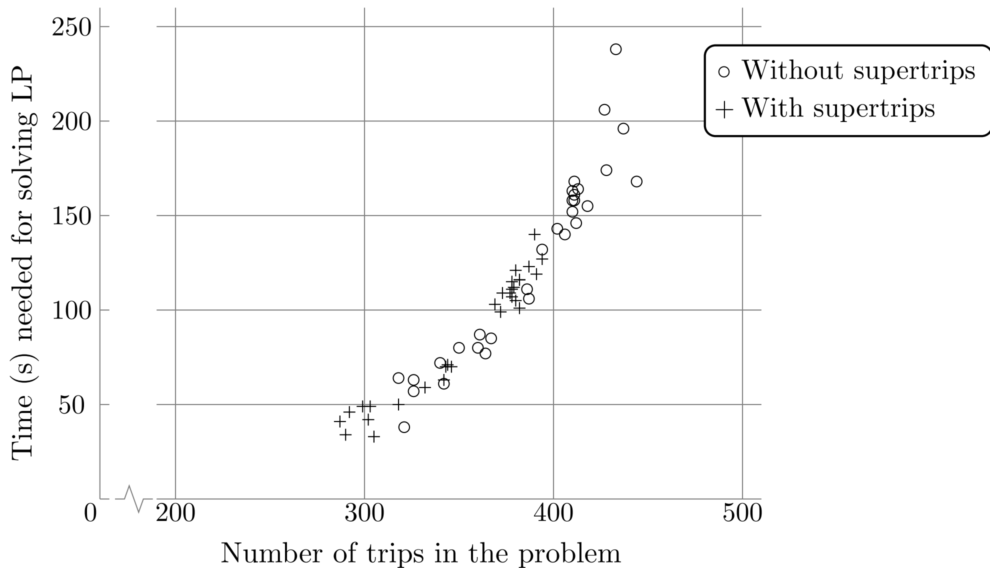

Figure 5.3: Time needed to solve the LP-relaxation in relation to number of trips.

5.5 Conclusion

We have presented an ILP representation for the bus planning problem at AAS and developed a new method for solving the problem. We have also written an implementation to test our algorithm and we were able to solve the schedule for any given day in a matter of minutes. Contrary to the current implementation at AAS, which makes use of a rolling horizon of 3 hours, we are able to plan the complete day, which enables us to have a much better overview while planning for example the mandatory breaks. Furthermore, considering the full day enables us to investigate the combination of the bus planning problem with the gate planning problem, which we will consider in Chapter 6.

In this chapter we looked at creating a robust schedule for a complete day of the bus planning problem. We assumed the arrival time and departure time of all flights to be static. One interesting idea that arose is to investigate the possi-

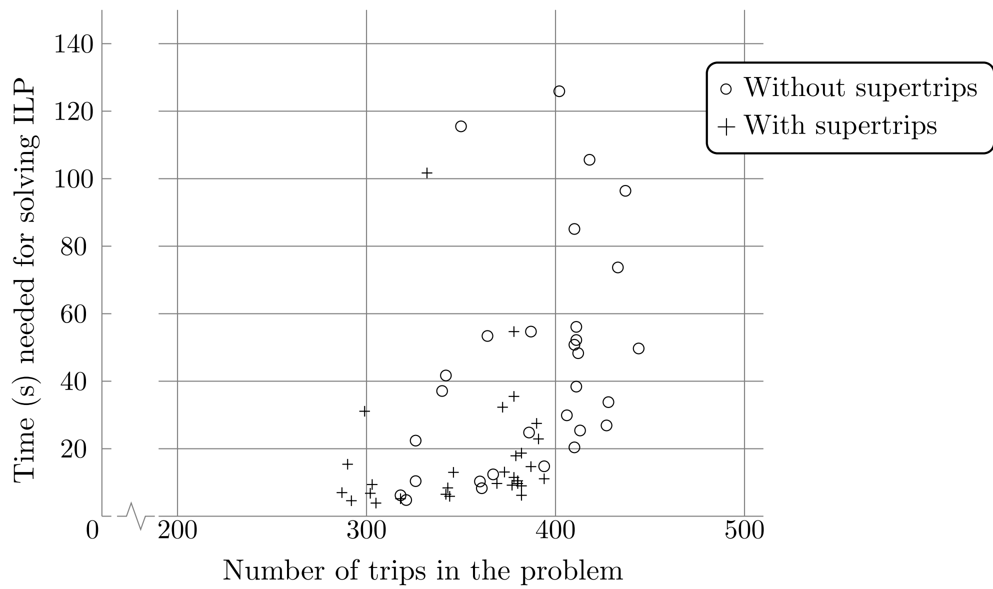

Figure 5.4: Time needed to solve the ILP in relation to number of trips.

bilities of also using this approach for the operational planning when changes in the arrival times and departure times occur. In the case of operational planning the schedule could be solved again every time a conflict introducing change (i.e. a situation in which one bus would have to drive two trips at the same time) in the data is presented by the Central Information System Schiphol (CISS).

In operational planning there is less time available for solving the bus planning problem than in day-ahead planning. The running times are a bit too large for our approach to be used as is for the operational planning. Hence, it does not allow for making a complete planning from scratch every time a conflict introducing change is presented. We may reduce the total time needed for solving the problem by making use of parallel programming, but this alone will not be sufficient.

In the case of a flight having a delay, we do not want to have to solve the problem from scratch again; we would like to reuse as much information from previous solutions as possible. One possibility is to keep all the bus plans in memory and when a flight gets a delay, update the trips corresponding to this flight in all bus plans to reflect the delay. After this we can remove all bus plans that became invalid due to trips overlapping with breaks or the end of a shift. In case a bus plan became invalid because of two trips overlapping after the delay of one of them, we could also create two new bus plans from this one invalid bus plan. Each of the two new bus plans would contain only one of the two trips that caused the original bus plan to become invalid. This way, the whole process of re-solving the problem would have a hot-start instead of starting all the way from scratch. The idea is that this would allow the problem to be solved in a much shorter time than when solved from scratch.

Currently we are investigating this possibility by comparing the suggested approach with a simple first-come-first-serve heuristic (which forms also the basis of the program currently in use at AAS for solving the bus planning problem) in a simulation study.

CHAPTER

SIX

Solving the integrated airport planning problem

6.1 Introduction

In Chapter 4 we have presented an algorithm for the assignment of aircraft to gates that is as robust as possible. The algorithm was designed to prevent that small deviations from the scheduled arrival and departure times result in lots of rescheduling.

Furthermore, in Chapter 5 we have given a similar type of algorithm to create a robust schedule for the buses transporting the passengers to and from aircraft on so-called remote stands.

Our approach resembles the way Amsterdam Airport Schiphol (AAS) is actually solving these two problems currently. First the gate assignment problem is solved, and its solution is then used as input for the bus planning problem. Although the bus planners have the possibility to influence the gate assignment

by providing preferences, in general the two problems are solved one after the other.

In general, solving two consecutive stages of a problem in a sequential way will result in a situation that is optimal for the first stage but the profit gained by this optimal solution can be nullified by bad optimal solutions for the second stage.

In case of the planning problems at AAS this could mean that a schedule for the gate assignment results in an input for the bus planning problem that allows poor solutions only. In many cases minor changes to the original solution for the gate assignment problem allow better assignments for the buses. So, although this would mean that a sub-optimal solution for the gate assignment problem would be used, the solution for the gate and bus planning problem as a whole would improve.

In the past, due to the limitations of computing power, the only feasible way to solve such a combination of problems was to decompose them according to the different stages and solve these stages sequentially and in an iterative manner. For the gate assignment and bus planning problem this means that one first solved the gate assignment problem, after which the bus planning problem was solved with the solution of the gate assignment problem as input. Next, the results of the bus planning are analyzed and certain positive and negative effects are identified. Then the gate assignment problem could be solved again, paying attention to the effects that were preferable and non-preferable regarding the remote stands (i.e. the interesting assignments for the bus planning) and steer the solution towards a certain direction. These steps were then repeated several times.

However, with the increase of computing power, it has now become possible for some problems to abandon such a decomposition approach and solve the *integrated problem*, i.e. the problem as a whole.

In machine scheduling, some research has already been done on the integration of scheduling problems. For example Chang and Lee (2004) consider integrating the production problem and the delivery of the finished products. Another type of integration of problems that has been considered is the integration of the batch sizing and scheduling problem (Coffman et al., 1990; Cheng and Kovalyov, 2000; Shabtay and Steiner, 2007).

Furthermore, a large amount of research has been performed on the integration of real-life scheduling problems. For example, Freling et al. (2003) look at the integration of the vehicle and crew scheduling problems that arise in public transport scheduling. They present two different models and algorithms for solving the integrated version of the two problems, and compare the results to the results obtained by using the standard sequential approach. They find that solving the integrated problem often results in fewer drivers being needed compared to the sequential approach.

The integration of real-life scheduling problems is investigated a lot in the airline industry. This industry has to deal with multiple scheduling problems. There is the problem of schedule design, which is concerned with determining when and where to offer flights. Another problem is the fleet assignment problem, the aim of which is to determine which type of aircraft will be assigned to the different flights. Related is the problem to determine which actual aircraft of the type designated to a particular flight will be flying on a given day, the so called aircraft routing problem. Finally, there is also the problem of determining which crew will be present on which flight. This latter problem is called the crew scheduling problem. A standard approach is to solve these four problems in a sequential way.

The integration of different combinations of the forementioned problems has been investigated already for some time. Cordeau et al. (2001) look into integrating the aircraft routing problem with the crew scheduling problem. They propose a solution approach based on Benders decomposition and show that solving these two problems as one integrated problem yields significant cost savings. Other integrations that have been considered are schedule assignment and fleet assignment (Rexing et al., 2000; Lohatepanont and Barnhart, 2004) and of fleet assignment and crew scheduling (Gao, 2007; Clarke et al., 1996; Sandhu and Klabjan, 2004).

If we solve the two problems sequentially, the solution for the gate assignment problem is used as a static input for the bus planning problem. This means that in solving the bus planning problem, it is known exactly which aircraft are assigned to the remote stands and therefore which trips need to be driven to transport all passengers. If we look at the integrated problem, it has not yet been determined which aircraft will be assigned to the remote stands and thus are of relevance for the bus planning problem. We develop an integrated solution to the gate assignment and bus planning problem by combining the two

models used for solving these problems separately into one large model.

The current software package in use at AAS follows a rule-based approach for optimizing the gate assignment. There are rules that give positive scores to assignment situations that are preferred and negative scores to situations that are not preferred. The height of the scores induces an order on the importance of the rules and mandatory situations are modeled via rules with very high positive or negative scores. Furthermore, the software package is capable of scheduling additional processes besides the assignment of aircraft to gates. For instance, in Vancouver the same program is used and there the scheduling of the push-back trucks is also handled by the program. The main problem of the current rule based approach is that it does not support the one thing we aim for, robustness, because it does not consider idle time. This means that even if the software package would be used to solve both the gate assignment and the bus planning, the final solutions for both problems presumably would not have any robustness in them.

The outline for the rest of the chapter is as follows: In Section 6.2 we give the problem formulation and in Section 6.3 we present our solution method. In Section 6.4 we present the results of the experiments that we performed and finally, in Section 6.5 we give our conclusions.

6.2 Problem formulation

Similar to Chapter 4, the goal of assigning aircraft to gates is to create an assignment schedule that is as robust as possible, meaning that the resulting schedule is able to cope with minor disturbances as well as possible during the actual day. However, now we want to create a robust schedule for the assignment of trips to buses at the same time, like we did in Chapter 5.

Cost function

Recall that a schedule is able to cope with disturbances best if the idle times between all pairs of consecutive flights on a gate are as large as possible. Hence, we want to create a final schedule where all idle times are as large as possible.

We model this using a cost function that greatly penalizes short idle times, while giving very low cost to large, and thus favorable, idles times.

For the cost of the idle time t between two consecutive aircraft on a gate we use the same cost function as presented in Chapter 4:

$$c^G(t) = 1000(\arctan(0.21(-t)) + \frac{\pi}{2}).$$

For the idle time t between the two consecutive trips driven by one bus, we use the same cost function as defined in Chapter 5, with the exception that we significantly lower the total cost, to resemble that the gate assignment problem is still the more important problem of the two:

$$c^B(t) = 50(\arctan(0.21(-t)) + \frac{\pi}{2}).$$

By taking the sum of the total cost of the two problems, we have a measure for the quality of the robustness of a solution for the problem as a whole.

The ILP formulation

The model we use consists of the combination of the separate models presented in Chapter 4 and Chapter 5 for the gate assignment and the bus planning problem respectively, together with a set of constraints linking the two models. To allow for the linking we will work with all possible trips and determine which of these will be needed in a solution and which not. We will use the binary variables NNT_t (not needed trip) for each trip t to denote whether the trip t needs to be assigned to a bus or that the trip is irrelevant for the assignment problem.

Similar to the approach in Chapter 4 and Chapter 5 we use gate plans and bus plans, respectively, to model which flights will be assigned to the same gate and which trips will be assigned to the same bus. For each gate plan i we have a binary variable x_i defined as follows:

$$x_i = \begin{cases} 1 & \text{if gate plan } i \text{ is selected} \\ 0 & \text{otherwise,} \end{cases}$$

and for each bus plan j we have a binary variable y_j defined as follows:

$$y_j = \begin{cases} 1 & \text{if bus plan } j \text{ is selected} \\ 0 & \text{otherwise.} \end{cases}$$

Furthermore, we denote the cost of gate plan i by c_i^G. This cost is the sum of the cost of the idle times between all consecutively assigned flights v and v' within the gate plan, multiplied by the *convenience multiplier* that measures the preference of this pair of consecutively assigned flights. Similarly, c_j^B denotes the cost of bus plan j and equals the sum of the cost of all idle times between consecutively driven trips.

To deal with a possible shortage of capacity we allow flights to be not assigned to gates and trips to be not assigned to buses by introducing the penalty variables UAF_v for each flight v and UAT_t for each trip t respectively. To ensure that not all flights and trips are left unassigned, these penalty variables have a very high cost Q_v and R_t for flight v and trip t, respectively, in the objective function.

Furthermore, let V denote the number of flights, T the number of trips, A the number of gate types, S_a the number of gates of gate type a, K the number of preferences, P_k the minimum number of flights that need to be assigned to a set of gate types in preference k, B the number of shifts, T_b the number of buses in shift b, and S_t the number of times trip t needs to be covered.

Now the integrated model is as follows:

$$\min \quad \sum_{i=1}^{N} c_i^G x_i + \sum_{v=1}^{V} Q_v \text{UAF}_v + \sum_{j=1}^{M} c_j^B y_j + \sum_{t=1}^{T} R_t \text{UAT}_t$$

subject to:

$$\mathrm{UAF}_v + \sum_{i=1}^{N} g_{vi}x_i = 1 \quad \text{, for each } v = 1, \ldots, V \tag{6.1}$$

$$\sum_{i=1}^{N} e_{ia}x_i \leq S_a \quad \text{, for each } a = 1, \ldots, A \tag{6.2}$$

$$\sum_{i=1}^{N}\sum_{v=1}^{V}\sum_{a=1}^{A} p_{vak}e_{ia}g_{vi}x_i \geq P_k \quad \text{, for each } k = 1, \ldots, K \tag{6.3}$$

$$\sum_{j=1}^{M} f_{jb}y_j \leq T_b \quad \text{, for each } b = 1, \ldots, B \tag{6.4}$$

$$S_t \cdot \mathrm{NNT}_t + \mathrm{UAT}_t + \sum_{j=1}^{M} h_{tj}y_j = S_t \quad \text{, for each } t = 1, \ldots, T \tag{6.5}$$

$$\mathrm{NNT}_t + \sum_{i=1}^{N}\sum_{v=1}^{V} t_{tvi}g_{vi}x_i = 1 \quad \text{, for each } t = 1, \ldots, T \tag{6.6}$$

$$x_i \in \{0,1\} \quad \text{, for each } i = 1, \ldots, N \tag{6.7}$$

$$y_j \in \{0,1\} \quad \text{, for each } j = 1, \ldots, M \tag{6.8}$$

$$\mathrm{UAF}_v \geq 0 \quad \text{, for each } v = 1, \ldots, V \tag{6.9}$$

$$\mathrm{NNT}_t, \mathrm{UAT}_t \geq 0 \quad \text{, for each } t = 1, \ldots, T \tag{6.10}$$

where

$$g_{vi} = \begin{cases} 1 & \text{if flight } v \text{ is in gate plan } i \\ 0 & \text{otherwise,} \end{cases}$$

$$e_{ia} = \begin{cases} 1 & \text{if gate plan } i \text{ is for gate type } a \\ 0 & \text{otherwise,} \end{cases}$$

$$p_{vak} = \begin{cases} 1 & \text{if flight } v \text{ has preference on a gate of type } a \text{ in preference } k \\ 0 & \text{otherwise,} \end{cases}$$

$$f_{jb} = \begin{cases} 1 & \text{if bus plan } j \text{ is for shift } b \\ 0 & \text{otherwise,} \end{cases}$$

$$h_{tj} = \begin{cases} 1 & \text{if trip } t \text{ is in bus plan } j \\ 0 & \text{otherwise,} \end{cases}$$

$$t_{tvi} = \begin{cases} 1 & \text{if assigning flight } v \text{ to gate plan } i \text{ implies trip } t \text{ must be driven} \\ 0 & \text{otherwise,} \end{cases}$$

Constraints (6.1)-(6.3) are from the gate assignment model in Chapter 4. Constraint (6.1) ensures that all flights are either present in one of the selected gate plans, or the unassignment variable for the flight will have the value 1, resulting in a penalty in the objective function. Constraint (6.2) ensures that we select as many gate plans of a certain type as there are gates of that type and constraint (6.3) ensures that we fulfill all of the preferences that are given with regards to the gate assignment.

The constraints for the bus planning problem as given in Chapter 5 need a modification in order to deal with the fact that all possible trips are generated, but not all of them need to be driven. For all of the flights that can be assigned to a remote stand we create the trips for each of the platforms that it can be assigned to. For example, if an arriving flight requires two trips because of the number of passengers, and it can be assigned to the D/E platform as well as the B platform, then we create two trips from the D/E platform and two trips from the B platform to the terminal building. Similarly, not only different platforms, but also different destinations in the terminal building must be considered. For each possible combination we create the corresponding trips also.

Since not all of these trips actually need to be driven, we define the variable NNT_t for each trip t as follows:

$$NNT_t = \begin{cases} 1 & \text{if trip } t \text{ needs not to be driven} \\ 0 & \text{otherwise.} \end{cases}$$

Constraint (6.4) is taken without modification from the model introduced in Chapter 5 and ensures that for each bus shift, we select at most the number of buses present in that shift.

Constraint (6.5) is obtained from the original definition in Chapter 5 by adding an additional term NNT_t to the constraint. This has the effect that a trip is either not needed, or, in case it is needed, must either be assigned to a bus plan or it must be explicitly become unassigned at high cost. Recall from Chapter 5 that S_t denotes the number of times a trip t needs to be assigned, which is > 1 in case of the super-trips.

Without any further constraints on the NNT_t variables, the easiest solution of the model would be to set the value of all of the NNT_t variables to 1 and all of the trip constraints would be satisfied right away. Constraint (6.6) ensures that

this cannot happen for trips that are defined for flights assigned to the remote stands. It is also this constraint that actually links the two separate models into one large unsplit model.

6.3 Solving the problem

6.3.1 Assigning flights to gate plans and trips to bus plans

To approximate the optimum of the given ILP formulation we will first relax the integrality constraints (6.7) and (6.8) which denote whether a gate or bus plan is selected or not. After that we solve the resulting LP relaxation to optimality by making use of column generation.

Solving the pricing problem

After each iteration of the column generation process, we need to solve the pricing problem, i.e. determine whether other columns exist that might improve the value of the objective function. In our case we need to determine whether there exists a new gate plan and/or bus plan that might improve the current solution. This means that we have to solve two types of pricing problems, one for determining an additional gate plan with negative reduced cost and one for determining an additional bus plan with negative reduced cost.

The pricing problem with regards to the bus planning problem is exactly the same as the pricing problem for solving only the bus planning problem in Chapter 5. So for each shift b we have a graph G_b containing a vertex for each trip that is within the start and end time of shift b. Furthermore, we introduce an arc from vertex t to t' if it is possible to drive trips t and t' in succession. This way, a path from the source to the sink corresponds to a feasible bus plan. We now set the cost on the arcs such that the total cost of a path is equal to the reduced cost of the corresponding bus plan. Finding the bus plan with minimum reduced cost for shift b now boils down to finding the shortest path in graph G_b. The only difference is that the size of the graphs for each shift b is larger due to the increased number of trips.

When looking at the pricing problem that needs to be solved for the gate assignment part, we have to make a small modification to the pricing problem presented in Chapter 4 because we have to cope with the new constraint (6.6). For each trip t this constraint gives a dual multiplier ρ_t. Since this additional dual multiplier will only be applicable to gate plans that are for remote stands (because only then t_{tvi} can have the value 1) we can modify the cost on the outgoing arcs of a vertex corresponding to flight v to a successor flight w in the graphs corresponding to gate types that are remote stands as follows:

$$conv(v,w)c^G(T_w^{\text{arr}} - T_v^{\text{dep}}) - \pi_v - \sum_{k=1}^{K} p_{vak}\psi_k - \sum_{t=1}^{T} t_{tv}\rho_t.$$

Recall from Chapter 4 that $conv(v,w)$ is the convenience multiplier and measures the preference value of flight w succeeding flight v on a gate. Furthermore, the dual multipliers π_v for flight v and ψ_k for preference k correspond to constraint (6.1) and constraint (6.3), respectively.

Because solving all of the pricing problems in each iteration can be rather time consuming, we have tried different strategies with regard to which of the pricing problems we solve during an iteration. One possible approach is to interleave the solving of the pricing problems; one iteration we solve the pricing problems for the buses and the other iteration we solve the pricing problems for the gates.

Another possible approach is to start with solving the pricing problems for the gates only and switch after a given number of iterations to solving the pricing problems for both the gates and the buses. The idea behind this is that initially the gate assignment part of the problem is more important than the bus planning part.

However, after some preliminary tests we found that searching for both gate and bus plans with negative reduced cost from the beginning on worked better than the other possibilities.

Similar to the solution of the gate assignment and bus planning problem in Chapter 4 and Chapter 5 respectively, we create additional columns when we solve the different pricing problems. These additional columns are generated in the same way as in Chapter 4 and Chapter 5. We first solve the pricing problem, that is, we find the shortest path in each of the graphs corresponding to the gate

types and bus shifts. We then take out the nodes of the shortest path one by one and solve the shortest path problem again in each of the resulting graphs. Instead of storing these additional columns in a column pool, we first sort all the additional columns based on the reduced cost and add the first n columns with negative reduced cost to the LP model. The rest of the columns are stored in the column pool again. After some preliminary experiments, we set the maximum number of columns to be added to the LP model in each iteration to 20.000.

Dealing with degeneracy

During our preliminary experiments, we found that the LP problem tends to be very degenerate. This degeneracy appears in two ways during the column generation process. First, re-solving the restricted master problem after new columns have been added takes quite a number of iterations and second, new columns that are generated with negative reduced cost do not improve the objective function after they have been added to the restricted master problem.

We have applied two different approaches to counter this problem of degeneracy. The first approach we used is *column deletion* and consists of the removal of columns with a positive reduced cost that is too large, after every given number of iterations. This removal does not only remove some degeneracy, but also the resulting models are smaller and therefore more quickly to solve. We have seen in Chapter 4 and Chapter 5 that this approach showed promising results in decreasing the running time needed for solving the problems.

The second approach we implemented is *stabilized column generation*, which was introduced by Du Merle et al. (1999). This is based on adding surplus and slack variables to overcome degeneracy by perturbing the right-hand side of all constraints. This concept is known from non-linear programming (Gill et al., 1989) where it is applied with unbounded surplus and slack variables with a positive coefficient in the objective function. Du Merle et al. (1999) apply this same technique, but use bounded surplus and slack variables.

To illustrate this technique consider the following example taken from Du Merle et al. (1999). Consider an LP problem

$$\min \quad cx$$

subject to:

$$\begin{aligned} Ax &= b \\ x &\geq 0. \end{aligned}$$

Add a surplus and slack variable, y_- and y_+ respectively, to all constraints in the problem, with cost coefficients δ_- and δ_+ respectively. Furthermore, limit the values of the surplus variables to ϵ_- and the slack variables to ϵ_+. These additions result in the following LP problem:

$$\min \quad cx - \delta_- y_- + \delta_+ y_+$$

subject to:

$$\begin{array}{rcl} Ax - y_- + y_+ & = & b \\ y_- & \leq & \epsilon_- \\ y_+ & \leq & \epsilon_+ \\ x, y_-, y_+ & \geq & 0 \end{array} \quad .$$

In the case of the integrated model, we add such slack and surplus variables to all constraints.

Du Merle et al. (1999) provide multiple methods for updating the values of both the cost coefficients δ_- and δ_+, as well as the bounds ϵ_- and ϵ_+ on the values of the surplus and slack variables respectively are suggested. The method we decided to use after some preliminary tests was to lower the bounds ϵ_- and ϵ_+ every ten iterations by multiplying them with the values 0.40 and 0.90 respectively. For setting the values of the cost coefficients δ_- and δ_+ we used a suggested approach of (Du Merle et al., 1999) to set them to the values of the dual multipliers of the perturbed original constraints from the previous iteration. Since the reduced cost of y_+ is equal to the dual multiplier $-\delta_+$ we see by inspecting the reduced cost that in case of $y_+ = \epsilon_+$, the new value of δ_+ will be increased (clearly y_+ was priced too favorably). Similarly, in case of $y_+ = 0$, the new value of δ_+ will be decreased (clearly y_+ was priced too highly). We can look at the case of y_- in a similar way.

Solving the ILP

After the LP has been solved to optimality by means of column generation, we are finished only if the obtained solution is integral. If we do not have an integral solution, then we solve the original ILP formulation for a restricted set of variables (gate plans and bus plans). We use the variables that are present in the final instance of the LP plus the gate plans and bus plans that were generated as extra columns while solving the pricing problems and all the variables that were taken out in the column deletion step. Here we make sure that we only add unique variables.

Solving the resulting ILP turned out to be still quite difficult. In order to speed up this solution process, we added additional constraints to the problem. These constraints act as a rounding heuristic. For each flight we check in which of the selected gate plans it is contained. If all of these gate plans are of the same type, then we add the constraint that the aircraft must be assigned to a gate of this type, which implies that we know whether it will require a bus or not. Similarly, we check for each bus in which of the selected bus plans it is contained; if these all are of the same type, then we formulate this as a hard constraint. Although these constraints might cause the actual optimal integral solution to be cut off, our experiments showed that this was not the case.

6.3.2 Assigning gate and bus plans to the actual gates and buses

After solving the model from the previous section, we have for each gate type exactly the number of gate plans as there are gates of that type and for each bus shift exactly the number of bus plans as there are buses in the shift. We now have to assign each of these gate plans and bus plans to the physical gates and buses, respectively.

As with the solution of the gate assignment problem in Chapter 4 we can use other properties (e.g. presence of waiting rooms, location at the pier) of the individual gates within one gate type to assign the gate plans to the gates. Since the size of these problems is relatively small (in the order of 5 to 10 gates within one group) it is probably most effective to leave this up to the gate planner to do manually. Since there are no such additional properties for the buses, assigning the bus plans to the actual buses can be done arbitrarily.

6.4 Computational experiments

For testing our model, we wrote a prototype implementation in C++ and ran numerous experiments. All experiments were run on a Pentium 4 2.8 GHz computer equipped with 1GB of RAM. The solver we used for solving all (I)LP problems is CPLEX 9.1.3 (ILOG, 2005) via the Concert Technology interface.

AAS provided us with both data regarding the gate assignment problem, which consisted of all flight information for three high-season (HS) days and three low-season (LS) days, and data regarding the bus planning problem with all information regarding buses for one complete month.

From the supplied data for the gate assignment we created two types of instances. In one type of instance we aggregated all gates with identical properties (e.g. size, region, ground handler, pier) into groups of gates. We refer to this type of instance as Grouped Gates (GG). Furthermore, we also constructed another type of instance where we did not aggregate gates with identical properties into groups of gates, but considered every gate as a group with size one. The only exception to this are the platform gates. We refer to this type of instance as Single Gates (SG). This de-aggregation results in over twice the number of gate types, as can be seen in Table 6.1. This way we created 12 instances with regard to the gate and flight information.

To create a sufficiently large test set, we combined each of the 12 gate assignment instances with each of the 30 bus planning problem instances. Since the information provided regarding the buses considers all days of the week, while the flight information only consisted of three consecutive days of the week, this combination results in considering each gate assignment instance with different availability of bus capacity.

We can expect that the set of buses available at each given time of the day is approximately enough for driving all trips, due to the way that AAS orders the buses from the bus company. This assumes that all flights arrive and leave at their exact arrival and departure times.

From Table 6.2 we can see that there is a significant difference in size between the *High-Season* (HS) instances and the *Low-Season* (LS) instances. The column "After splitting" shows the total number of flights in the model after the extra

Instance	Gates	Gate types	Remote stands
Grouped	128	40	34
Single	128	94	34

Table 6.1: Sizes of the provided instances with regard to gates.

Instance	Flights	After splitting	Arrival/Departure events
HS-1	607	699	1095
HS-2	594	680	1068
HS-3	592	688	1065
LS-1	518	602	937
LS-2	530	608	946
LS-3	523	593	933

Table 6.2: Sizes of the provided instances regarding flights.

flights representing the split parts of long-stay flights have been created. Finally, in the last column the number of *arrival and departure events* (i.e. possible triggers of platform buses) that were generated are shown.

Table 6.3 shows the information regarding the bus instances that were provided. The column Buses shows the total number of buses that are driving around at AAS during the complete day. The number of buses is greater than the number of shifts because multiple buses can have the same shift. For whatever reason the data for bus-09 were missing, while bus-08 seems to contain almost double the amount. We decided to leave this anomaly in the data as it also will give an idea what happens when a large number of buses is available. Finally, the last column shows the number of shifts that are longer than 4.5 hours and therefore require the driver to have an intermediate break.

Results for solving the LP

In Table 6.4 we present the general results of solving the LP part of the problem. Recall that we combined each instance of the gate assignment problem with the shift information of 30 available instances of the bus planning problem. We list the average solution time of the LP over 30 of such resulting instances together with the minimum time and the maximum time. We also present the

Instance	Shifts	Buses	*Long* shifts
bus-01	14	50	10
bus-02	21	63	14
bus-03	23	61	12
bus-04	24	64	15
bus-05	19	64	12
bus-06	23	62	15
bus-07	20	62	12
bus-08	28	113	19
bus-10	24	61	13
bus-11	24	64	14
bus-12	21	64	14
bus-13	26	62	16
bus-14	21	62	13
bus-15	13	50	9
bus-16	21	63	13
bus-17	22	61	12
bus-18	24	64	15
bus-19	23	64	16
bus-20	23	62	14
bus-21	20	62	13
bus-22	14	50	10
bus-23	21	63	14
bus-24	26	61	16
bus-25	25	64	15
bus-26	21	64	14
bus-27	24	62	15
bus-28	22	62	14
bus-29	15	50	11
bus-30	20	63	13
bus-31	26	61	14

Table 6.3: Sizes of the provided instances regarding buses.

Instance	Total time LP (s) Average	Min	Max	Avg iter	Avg time (s)/iter RMP	Pricing
HS-01-GG	1129.6	967.8	1472.0	161.67	2.8	3.9
HS-01-SG	2070.1	1752.1	2657.7	171.90	4.8	6.8
HS-02-GG	973.9	864.7	1213.2	148.27	2.6	3.7
HS-02-SG	1847.4	1627.4	2337.8	163.07	4.4	6.5
HS-03-GG	1142.6	1010.4	1641.3	157.50	3.2	4.0
HS-03-SG	2575.2	2189.9	3970.3	212.77	4.6	7.2
LS-01-GG	658.5	560.3	769.3	165.17	1.1	2.7
LS-01-SG	1235.8	1094.6	1472.0	175.17	1.9	4.8
LS-02-GG	710.0	623.8	850.4	161.90	1.3	2.8
LS-02-SG	1383.4	1144.0	1661.5	175.87	2.5	5.0
LS-03-GG	595.0	474.6	775.1	141.37	1.2	2.8
LS-03-SG	1125.1	991.3	1422.4	151.70	2.2	4.9

Table 6.4: General results of solving the LP relaxation.

average number of iterations needed to solve the LP relaxation and finally, we also present the average time needed in each iteration of the column generation process to solve the pricing problem and the time needed for solving the *restricted master problem* (RMP) again.

From Table 6.4 we see that a significant amount of the time needed for solving the LP-relaxation is spent in solving all the separate pricing problems. In case of the High-Season instances this is about 55% of the time, while in the Low-Season instances at least 66% of the time is spent on solving the pricing problems. Since the pricing problems for different gate types and bus shifts can be solved completely independent from each other, we could bring down the influence of the pricing problems on the total time needed for solving the LP-relaxation by making use of parallel programming. This ultimately would scale to the point where for each gate type and bus shift we can dispatch the pricing problem to one separate computer.

To investigate the effect of column deletion and stabilized column generation on the running times needed for solving the LP-relaxation, we also ran the experiments without these two enhancements. To keep the computation time within reasonable limits, we ran each instance of the gate assignment problem with only one instance of the bus planning problem. The results of this experiment can be found in Table 6.5. It can be clearly seen that the time needed to solve

			Avg time (s)/iter	
Instance	LP time (s)	Iterations	RMP	Pricing
HS-01-GG	44088.6	824	49.9	3.3
HS-01-SG	82042.2	845	90.6	5.9
HS-02-GG	43700.1	654	63.3	3.1
HS-02-SG	48344.9	626	71.0	5.7
HS-03-GG	72742.7	585	120.7	3.2
HS-03-SG	125375.2	755	159.7	5.7
LS-01-GG	9292.6	475	16.9	2.5
LS-01-SG	10785.0	480	17.8	4.4
LS-02-GG	15888.3	819	16.7	2.5
LS-02-SG	16447.4	663	20.0	4.4
LS-03-GG	16602.0	983	14.4	2.3
LS-03-SG	18366.7	985	14.0	4.4

Table 6.5: LP results without column deletion and stabilization and maximum of 20.000 extra columns per iteration.

the LP relaxation to optimality explodes without the use of column deletion and stabilization. One reason for this huge increase in time is the large increase in the average time needed for solving one iteration of the RMP. This is because after a couple of iterations, the model quickly becomes very large due to the fact that all columns remain in the model. Another result of this is that the amount of memory needed also increases. When column deletion is enabled, we do not have this problem, since every given number of iterations gate and bus plans that do not look promising are completely removed from the model.

To test how the program would perform without column deletion and stabilization but with fewer columns added in each iteration, we decided to run all flight instances with 3 separate bus instances again, but now limiting the number of new gate and bus plans added in each iteration to both 200, resulting in a maximum of 400 extra columns added in each iteration, besides the minimum reduced cost ones.

The results of this experiment can be found in Table 6.6. It can be seen that the average time needed for solving an iteration of the RMP dropped dramatically compared to the case in which we were adding a maximum of 20.000 extra columns per iteration, but it still is more than the time needed when both column deletion and stabilization are enabled.

Instance	Total time LP (s) Average	Min	Max	Avg iter	Avg time (s)/iter RMP	Pricing
HS-01-GG	9942.9	5608.9	12661.7	841.33	8.7	2.9
HS-01-SG	10944.5	6959.1	13951.4	783.67	8.1	5.5
HS-02-GG	10306.6	6214.0	12649.5	820.33	11.2	2.8
HS-02-SG	11667.7	8759.2	13588.3	777.00	10.2	5.2
HS-03-GG	22274.2	13816.3	36026.1	873.00	23.5	2.8
HS-03-SG	22999.4	16087.7	26496.2	830.33	23.9	5.3
LS-01-GG	1982.7	1590.6	2711.7	474.33	2.1	2.2
LS-01-SG	3165.2	2855.7	3468.8	493.33	2.6	4.0
LS-02-GG	3761.0	2280.9	5715.7	863.67	2.0	2.2
LS-02-SG	5965.2	5218.1	6741.8	785.00	3.4	4.2
LS-03-GG	5203.8	3069.6	8469.9	934.33	3.1	2.2
LS-03-SG	8016.2	5368.8	11550.0	1055.00	3.4	4.0

Table 6.6: LP results without column deletion and stabilization and maximum of 400 extra columns per iteration.

Two other interesting aspects seen from Table 6.6 are that the average number of iterations needed to solve the LP-relaxation to optimality is higher without than with column deletion and stabilization enabled, while the average time needed for solving the pricing problems is lower. The increase in number of iterations needed is an example of the so-called *tailing-off effect*. In the beginning there are big improvements in each iteration, while more and more iterations are needed when the algorithm comes closer to the optimum. Using stabilized column generation has a positive effect on this tailing-off effect, as can be seen from the number of iterations needed. The reason that the average time needed for solving the pricing problems is lower is in a way due to the larger number of iterations needed. In the final iterations, solving the shortest path problems for the pricing problems requires fewer updates of the length property of each node, implying that less time is needed for solving the shortest path in general.

We can see that the combination of column deletion and stabilized column generation is responsible for a huge improvement in the time needed for solving the LP-relaxation to optimality with column generation. Interesting is the fact that the improvement seems larger when the instances are larger (see HS versus LS).

	Improvement factor with regards to		
Instance	Avg. LP time	Avg. iterations	Avg. time RMP/iter
HS-01-GG	8.80	5.20	3.11
HS-01-SG	5.29	4.56	1.69
HS-02-GG	10.58	5.53	4.31
HS-02-SG	6.32	4.76	2.32
HS-03-GG	19.49	5.54	7.34
HS-03-SG	8.93	3.90	5.20
LS-01-GG	3.01	2.87	1.91
LS-01-SG	2.56	2.82	1.37
LS-02-GG	5.30	5.33	1.54
LS-02-SG	4.31	4.46	1.36
LS-03-GG	8.75	6.61	2.58
LS-03-SG	7.12	6.95	1.55

Table 6.7: Improvements with column deletion and stabilization.

Results for solving the ILP

As mentioned in Section 6.3 we added additional constraints to the model before solving the actual ILP. The average number of constraints that was added for flights as well as for buses is shown in Table 6.8. We can see that for about 100 flights and about 58 trips we restrict the possible assignment to only one type of gate and shift respectively. This means that all gate and bus plans that do not correspond with this type or shift respectively but do contain the flight or the trip, are removed from the ILP model. Furthermore, when a flight is assigned to a specific type of gate, the value for any NNT_t variable that corresponds to this flight is known also. This results in ILP models that are a lot smaller. From the results of the experiments we found that adding these additional constraints to the model did not have a significant effect on the value of the final ILP solution. Furthermore, imposing these constraints did not result in the ILP problem becoming infeasible. A possible reason for this can be the fact that even after adding these constraints, there is a very large number of gate and bus plans left to choose from in the model. Without these additional constraints, the time needed for solving the ILP was significantly higher (i.e. five to ten minutes on average).

One other way we used to speed up the process of solving the ILP was to first

Instance	Average additional constraints		Average solution
	Flight constraints	Trip constraints	time ILP (s)
HS-01-GG	121.4	57.6	43.5
HS-01-SG	103.4	57.9	54.1
HS-02-GG	117.8	57.1	42.0
HS-02-SG	105.4	57.7	103.3
HS-03-GG	119.3	57.2	82.7
HS-03-SG	108.7	57.5	95.2
LS-01-GG	108.9	58.4	86.5
LS-01-SG	91.0	59.0	271.0
LS-02-GG	107.0	59.1	45.8
LS-02-SG	84.2	59.3	170.6
LS-03-GG	118.5	59.9	20.6
LS-03-SG	105.6	59.6	29.5

Table 6.8: General results ILP.

only solve the root node relaxation. We then added a threshold that is 0.5% above the value of the root node and instructed the ILP solver to prune any node that has a value above this threshold. Using this threshold boils down to guessing that the integrality gap will not exceed this threshold.

One possible disadvantage of using such a threshold is the fact that if the final ILP solution value would be above this threshold, it will not be found, since the value of the node relaxation will be greater than the threshold meaning the node will be pruned. To counter this problem, we have implemented an approach that will increase the value of the threshold in case the problem has become infeasible. However, setting the threshold to 0.5% above the value of the root node relaxation is rather conservative and in our experiments increasing the value of the threshold was never needed.

Furthermore, when looking at the time needed for solving the various final ILP models, we can see that for Grouped Gates instances the maximum time needed is around 86 seconds. For the Single Gates instances some more time is needed, but even these times are still within acceptable running times, considering the fact that these models provide a significant larger degree of freedom for modeling certain constraints for assigning flights to gates and contain over twice the number of gate types.

6.5 Conclusion and further research

We have investigated the integrated solution of two assignment problems that in current practice are solved sequentially. We formulated the problem by one large model for which we approximated the optimal solution by means of a column generation approach.

We implemented our algorithm and tested it with real-life data provided by AAS. The results indicate that our approach is capable of solving these real-life instances within acceptable computation times, considering the fact that this approach solves two problems within about the same time that is currently spent at AAS for the computer to generate a solution for only the gate assignment problem.

We also showed that our approach is still capable of solving the instances within acceptable running times if we create a single gate type for each separate gate, except for the remote stands. This different model leads to over twice the number of gate types which significantly increases the size of the instances, but allows for much more control over assigning flights to particular gates.

We have not compared the results obtained through the integrated approach to the results by the sequential approach. To make a meaningful comparison of this kind, we need to simulate the gate and bus assignment for an entire day to find out about the number of and impact of the necessary re-plannings. Similarly, we have not compared our plans to the plans used by AAS, since it is not possible for us to retrieve the schedule to which we would like to compare our solution, namely the initial schedule as produced by the computer for the upcoming day. We are currently performing a simulation study of the model for the planning of the platform buses to evaluate the robustness of the column generation planning compared to a kind first-come-first-serve method as used at AAS. We can clearly see that the column generation schedule is more robust, in the sense that the idle time is spread more evenly.

An interesting possibility for further investigation is to start looking at a more operational type of planning. Observe that the current approach creates a schedule from scratch for the upcoming day that is as robust as possible. For example, it would be interesting to find out how our suggested approach would perform if we do not let it create a schedule from scratch but we supply it with a schedule

and some disturbances and let the program try to re-solve this updated problem.

One of the main criteria that would have to be considered for this approach is that any new solution should not deviate too much from the currently existing solution. So when solving the problems after some parts are fixed (since they already took place) and other events have changed properties (e.g. earlier or later Estimated Time of Arrivals and Departures) the cost function would not only have to consider the robustness of the schedule, but also the similarity to the original day-ahead schedule, since too many changes in a schedule will result in a lot of confusion for the different parties that depend on the schedule.

Other possibilities for further research are the integration of gate assignment and/or bus planning with other problems such as the planning of push-back trucks and the crew scheduling problems of the ground-handlers. Furthermore, when information regarding the transfer connections of passengers is available, another interesting direction for research is optimizing passenger comfort by placing the aircraft such that the real walking distance between connecting flights is minimized.

BIBLIOGRAPHY

Akturk, M.S. and Ozdemir, D. (2001). A new dominance rule to minimize total weighted tardiness with unequal release dates, *European Journal of Operational Research* **135**: 394–412.

Baptiste, P. (1999). Polynomial time algorithms for minimizing the weighted number of late jobs on a single machine with equal processing times, *Journal of Scheduling* **2**: 245–252.

Baptiste, P. (2000). Scheduling equal-length jobs on identical parallel machines, *Discrete Applied Mathematics* **103**(1-3): 21–32.

Baptiste, P., Brucker, P., Knust, S., and Timkovsky, V. (2004). Ten notes on equal-processing-time scheduling, *4OR: Quarterly Journal of the Belgian, French and Italian Operations Research Societies* **2**: 111–127.

Baptiste, P., Le Pape, C., and Nuijten, W. (2001). *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*, Vol. 39 of *International Series in Operations Research & Management Science*, Kluwer Academic Publishers, Dordrecht, The Netherlands.

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

Bazaraa, M.S., Jarvis, J.J., and Sherali, H.D. (1990). *Linear programming and network flows (2nd ed.)*, John Wiley & Sons, Inc., New York, NY, USA.

Beldiceanu, N. and Demassey, S. (2007). Global constraint catalog. `http://www.emn.fr/x-info/sdemasse/gccat/index.html`.

Bigras, L.P., Gamache, M., and Savardan, G. (2005). Time-indexed formulations and the total weighted tardiness problem, *Technical report*, GERAD and Ecole Polytechnique de Montreal.

Bihr, R.A. (1990). A conceptual solution to the aircraft gate assignment problem using 0,1 linear programming, *Proceedings of the 12th Annual Conference on Computers and Industrial Engineering*, Pergamon Press, Inc., Elmsford, NY, USA, pp. 280–284.

Bolat, A. (2000). Procedures for providing robust gate assignments for arriving aircrafts, *European Journal of Operational Research* **120**: 63–80.

Brucker, P. and Knust, S. (2000). A linear programming and constraint propagation-based lower bound for the rcpsp., *European Journal of Operational Research* **127**: 355–362.

Brucker, P. and Knust, S. (2002). Lower bounds for scheduling a single robot in a job-shop environment., *Annals of Operations Research* **115**: 147–172.

Brucker, P. and Knust, S. (2003). Lower bounds for resource-constrained project scheduling problems., *European Journal of Operational Research* **149**: 302–313.

Brucker, P., Drexl, A., Möhring, R., Neumann, K., and Pesch, E. (1999). Resource-constrained project scheduling: Notation, classification, models, and methods, *European Journal of Operational Research* pp. 3–41.

Cesta, A., Oddi, A., and Smith, S.F. (2002). A constraint-based method for project scheduling with time windows., *Journal of Heuristics* **8**: 109–136.

Chang, Y.C. and Lee, C.Y. (2004). Machine scheduling with job delivery coordination, *European Journal of Operational Research* **127**(2): 470–487.

Cheng, T.C.E. and Kovalyov, M.Y. (2000). Parallel machine batching and scheduling with deadlines, *Journal of Scheduling* **3**(2): 109–123.

Clarke, L.W., Hane, C.A., Johnson, E.L., and Nemhauser, G.L. (1996). Maintenance and crew considerations in fleet assignment, *Transportation Science* **30**: 249–260.

Coffman, E.G., Yannakakis, M., Magazine, M.J., and Santos, C. (1990). Batch sizing and job sequencing on a single machine, *Annals of Operations Research* **26**(1): 135–147.

Cordeau, J.F., Stojkovic, G., Soumis, F., and Desrosiers, J. (2001). Benders decomposition for simultaneous aircraft routing and crew scheduling, *Transportation Science* **35**(4): 375–388.

Cormen, T.H., Leiserson, C.E., Rivest, R.L., and Stein, C. (2001). *Introduction to Algorithms, Second Edition*, The MIT Press/McGraw Hill.

Dorndorf, U., Drexl, A., Nikulin, Y., and Pesch, E. (2007). Flight gate scheduling: State-of-the-art and recent developments, *Omega* **35**: 326–334.

Du, J. and Leung, J.Y.T. (1990). Minimizing total tardiness on one machine is NP-hard, *Mathematics of Operations Research* **15**(3): 483–495.

Du Merle, O., Villeneuve, D., Desrosiers, J., and Hansen, P. (1999). Stabilized column generation, *Discrete Math.* **194**(1-3): 229–237.

Freling, R., Huisman, D., and Wagelmans, A.P.M. (2003). Models and algorithms for integration of vehicle and crew scheduling., *Journal of Scheduling* **6**(1): 63–85.

Freling, R., Wagelmans, A.P.M., and Paixao, J.M.P. (2001). Models and algorithms for single-depot vehicle scheduling, *Transportation Science* **35**(2): 165–180.

Gao, C. (2007). *Airline Integrated Planning and Operations*, PhD thesis, Georgia Institute of Technology.

Gill, P.E., Murray, W., Saunders, M.A., and Wright, M.H. (1989). Constrained nonlinear programming, *in* Nemhauser, G., Rinnooy Kan, A.H.G., and Todd, M.J. (Eds.), *Handbooks in Operations Research and Management Science*, Elsevier, pp. 171–210.

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

Haghani, A. and Chen, M.C. (1998). Optimizing gate assignments at airport terminals, *Transportation Research Part A: Policy and Practice* **32**: 437–454.

ILOG (2005). ILOG CPLEX v9.1, `http://www.ilog.fr`.

Kolliopoulos, S.G. and Steiner, G. (2004). On minimizing the total weighted tardiness on a single machine, *in* Diekert, V. and Habib, M. (Eds.), *STACS 2004, 21st Annual Symposium on Theoretical Aspects of Computer Science, Montpellier, France, March 25-27, 2004, Proceedings*, Springer, pp. 176–186.

Lawler, E.L. (1977). A pseudopolynomial algorithm for sequencing jobs to minimize total tardiness, *Annals of Discrete Mathematics* **1**: 331–342.

Le-Anh, T. and de Koster, M.B.M. (2006). A review of design and control of automated guided vehicle systems, *European Journal of Operational Research* **171**(1): 1–23.

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

Leung, J.Y.T. (2004). *Handbook of scheduling*, Chapmann & Hall/CRC, New York.

Lohatepanont, M. and Barnhart, C. (2004). Airline schedule planning: Integrated models and algorithms for schedule design and fleet assignment, *Transportation Science* **38**(1): 19–32.

Nemhauser, G.L. and Wolsey, L.A. (1988). *Integer and Combinatorial Optimization*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, New York.

Pereira Lopes, M.J. and Valério de Carvalho, J.M. (2007). A branch-and-price algorithm for scheduling parallel machines with sequence dependent setup times., *European Journal of Operational Research* **176**(3): 1508–1527.

Rexing, B., Barnhart, C., Kniker, T., Jarrah, A., and Krishnamurthy, N. (2000). Airline fleet assignment with time windows, *Transportation Science* **34**(1): 1–20.

Roos, C. (2005). Private communication.

Ropke, S. and Cordeau, J.F. (2006). Branch-and-cut-and-price for the pickup and delivery problem with time windows, *Technical Report C7PQMR PO2006-21-X*, Centre for Research on Transportation (now: CIRRECT), Université de Montréal.

Sandhu, R. and Klabjan, D. (2004). Integrated airline planning, *AGIFORS Annual Symposium 2004, Singapore*.

Shabtay, D. and Steiner, G. (2007). Single machine batch scheduling to minimize total completion time and resource consumption costs, *Journal of Scheduling* **10**(4): 255–261.

Van den Akker, J.M., Hoogeveen, J.A., and Van de Velde, S.L. (2005). Applying column generation to machine scheduling, *in* Desaulniers, G., Desrosiers, J., and Solomon, M.M. (Eds.), *Column Generation*, Springer-Verlag, pp. 303–330.

Van den Akker, J.M., Hoogeveen, J.A., and Van Kempen, J.W. (2006). Parallel machine scheduling through column generation: Minimax objective functions, *in* Azar, Y. and Erlebach, T. (Eds.), *Algorithms - ESA 2006, Proceedings 14th Annual European Symposium*, Vol. 4168 of *Lecture Notes in Computer Science*, Springer, pp. 648–659.

Van Orden, A. (2002). *Gate assignment: Methods and models*, Master's thesis, Department of Mathematics, Utrecht University.

Verma, S. and Dessouky, M. (1998). Single-machine scheduling of unit-time jobs with earliness and tardiness penalties, *Mathematics of Operations Research* **23**(4): 930–943.

Wolsey, L.A. (1998). *Integer Programming*, Wiley-Interscience Series in Discrete Mathematics and Optimization, John Wiley & Sons, New York.

Xu, J. and Bailey, T.G. (2001). The airport gate assignment problem: Mathematical model and a tabu search algorithm., *HICSS '01: Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34)-Volume 3*.

Samenvatting

In dit proefschrift bekijken wij een aantal verschillende planningsproblemen. In de basis is elk van deze planningsprobleem als volgt te formuleren: “Wij hebben één of meerdere machines en een eindige verzameling van taken die hierop moeten worden uitgevoerd. Welke taak moet door welke machine worden uitgevoerd en wat is de optimale volgorde per machine om de taken uit te voeren?” Meestal moet er ook voldaan worden aan een reeks van randvoorwaarden. Taken kunnen bijvoorbeeld deadlines hebben, en soms moeten bepaalde taken afgerond zijn voor andere kunnen beginnen. Moeten de taken niet door machines maar door mensen verricht worden, dan kunnen zelfs de normen van de Arbeidstijdenwet meespelen. In sommige gevallen mogen randvoorwaarden worden overschreden, mits een boete wordt betaald afhankelijk van de mate van overschrijding.

Dergelijke vraagstukken vormen de basis van een onderzoeksgebied dat wij ‘Planning and Scheduling’ noemen. De oorsprong van het gebied voert tenminste terug tot de eerste helft van de 20$^{\text{e}}$ eeuw toen het in toenemende mate van belang werd om productieprocessen wiskundig te begrijpen en op te lossen. Met de opkomst van computers heeft het terrein een grote vlucht genomen. Er ontstonden methoden om zelfs ‘grote’ planningsproblemen te modelleren en standaardalgoritmen om de verkregen modellen op te lossen op de computer. Spoedig bleek dat het optimaal oplossen van planningsproblemen ook voor computers een hele opgave kon zijn. Er is inmiddels veel onderzoek gedaan om de complexiteit van planningsproblemen beter te begrijpen, maar hier liggen nog veel vraagstukken open.

Vanuit theoretisch oogpunt kunnen planningsproblemen geformuleerd worden als abstracte schema’s (‘programma’s’) met vele parameters en regels (‘constraints’) die bepalen hoe die parameters worden ingeperkt en van elkaar afhan-

kelijk zijn. Enerzijds wordt het op die manier mogelijk om planningsproblemen wiskundig te analyseren en kenmerken af te leiden waaraan oplossingen moeten voldoen of zelfs te bepalen hoe ze heel snel kunnen worden opgelost. Anderzijds kunnen op die manier bestaande softwarepakketten zoals CPLEX worden toegepast en kan een probleem concreet worden opgelost danwel inzicht in de praktische eigenschappen worden verkregen. Het geeft bijvoorbeeld de mogelijkheid om bepaalde parameters te wijzigen en vervolgens te kijken welk effect elke parameter heeft op het probleem en de kwaliteit van de oplossingen. Zo kan men meer inzicht krijgen in de structuur van de problemen, en dat leidt er dan hopelijk toe dat de planningsproblemen op een efficiëntere manier kunnen worden opgelost. In dit proefschrift worden beide aspecten van planningsproblemen bestudeerd. Als toepassing bekijken wij de complexe planningsproblemen die dagelijks opgelost moeten worden op een vliegveld zoals Schiphol.

In dit proefschrift bekijken wij eerst twee theoretische planningsproblemen. In hoofdstuk 2 bestuderen wij allereerst de planningsproblemen waarbij het is toegestaan dat taken te laat klaar zijn, al brengt dit wel kosten met zich mee. Elk van de taken heeft hierbij een vroegst mogelijk starttijdstip, een nagestreefd voltooiingstijdstip en een gewicht ('belangrijkheidsfactor'). Verder nemen wij aan dat alle taken evenveel tijd nodig hebben om uitgevoerd te worden. Het doel is nu om te bepalen wanneer welke taak moet worden uitgevoerd om te bereiken dat het totaal aan 'gewogen' tijd dat taken te laat klaar zijn minimaal is. Om dit probleem op te lossen formuleren wij het planningsprobleem eerst als een geheeltallig lineair programmeringsprobleem met behulp van de tijdsgeïndexeerde formulering. Laten wij vervolgens de geheeltalligheidseis weg, dan ontstaat een in beginsel afgezwakt lineair programmeringsprobleem, de zogenaamde LP-relaxatie. Wij laten zien dat indien de starttijden, nagestreefde voltooiingstijdstippen en gewichten van elke combinatie van twee taken onderling voldoen aan bepaalde eisen, de oplossing van de LP-relaxatie altijd een geheeltallige oplossing oplevert waardoor het probleem efficiënt kan worden opgelost. Voor de gevallen waarin er taken bestaan die niet aan die eisen voldoen, laten wij zien hoe wij toch zoveel mogelijk informatie en structuur van het probleem kunnen gebruiken om betere oplossingen en dus betere beslissingen kunnen nemen zodat het probleem sneller opgelost kan worden. Tevens laten wij zien dat het probleem niet alleen voor één machine op die manier kan worden opgelost, maar ook voor willekeurig aantal van m machines.

In hoofdstuk 3 bekijken wij een tweede theoretisch vraagstuk. Hierbij bestuderen wij een zelfde type planningsproblemen maar nu is het doel om de maximale

overschrijding van de nagestreefde voltooiingstijd te minimaliseren. Bij dit probleem nemen wij aan dat elke taak weer een starttijdstip, een bewerkingsduur, een nagestreefd voltooiingstijdstip en een strikte deadline heeft, maar nu ook dat er meerdere machines kunnen zijn. Verder zijn er precedentie relaties die aangeven hoeveel tijd een gegeven taak minimaal, maximaal of exact later moet starten dan een andere taak. Als wij dit probleem direct formuleren als geheeltallig lineair programmeringsprobleem dan kost het oplossen zeer veel tijd.

Om deze tijd in te korten gaan wij eerst op een andere manier bepalen wat de minimale waarde is van de maximale tijd dat een taak te laat klaar mag zijn. De methode die wij hiervoor gebruiken is een bovengrens te zetten op de tijd die een taak maximaal te klaar mag zijn. Het zetten van deze bovengrens resulteert in deadlines voor alle taken. Met behulp van de methode van kolomgeneratie bepalen wij vervolgens hoeveel machines er nodig zijn om ervoor te zorgen alle taken op hun deadline volledig klaar zijn. Door dit benodigde aantal machines te vergelijken met het aantal beschikbare machines weten wij of de tijd die een taak maximaal te laat klaar mocht zijn te scherp, of voldoende ruim gekozen was. Wanneer wij de kleinste waarde hebben gevonden waarvan we niet kunnen aantonen dat hij te scherp gekozen is, gebruiken wij deze informatie in het oorspronkelijke geheeltallig lineair programmeringsprobleem om de oplossingstijd te verkleinen. Verder laten wij zien hoe verschillende uitbreidingen in het model verwerkt kunnen worden, zodanig dat wij dezelfde oplosmethode kunnen blijven gebruiken. De resultaten van onze experimenten laten zien dat door eerst een minimale waarde te bepalen en deze informatie daarna te gebruiken bij het oplossen van het probleem, wij in staat zijn om grote problemen binnen acceptabele doorlooptijden op te lossen. Voor één van de probleemtypes blijkt zelfs dat de minimale waarde tijdens alle experimenten overeenkomt met de optimale waarde.

Vervolgens bekijken wij in dit proefschrift planningsproblemen die uit de praktijk komen. Wij bestuderen in het bijzonder enkele van de vele planningsproblemen die een rol spelen op vliegvelden. Wij behandelen het gatetoewijzingsprobleem en het plannen van de platformbussen die worden gebruikt om passagiers van of naar de vliegtuigen te vervoeren. In de hoofdstukken 4 en 5 bekijken wij beide problemen eerst afzonderlijk; daarna laten wij in hoofdstuk 6 zien hoe wij deze twee problemen tegelijkertijd kunnen beschouwen en deze gezamenlijk, als een groot 'geïntegreerd' probleem kunnen oplossen. In dit deel van ons onderzoek hebben wij veel medewerking gehad van de planners van de luchthaven Schiphol en onderzoekers bij het Nationaal Lucht- en Ruimtevaart Laboratorium.

Een eenvoudige omschrijving van het gatetoewijzingsprobleem luidt: “Bepaal voor elke vlucht die op een dag aankomt bij welke gate zij moet komen te staan, zodra het betreffende vliegtuig op het vliegveld is geland”. Er zijn twee aspecten die het gatetoewijzingsprobleem moeilijk maken. Ten eerste zijn er randvoorwaarden betreffende o.a. de minimale en maximale toegestane grootte van een vliegtuig voor elke gate en randvoorwaarden met betrekking tot de veiligheid. Ten tweede is het aantal binnenkomende vluchten per dag ‘groot’. De combinatie van deze twee aspecten maakt het toewijzingsprobleem algoritmisch complex. Op Schiphol streeft men naar robuuste oplossingen. Dit houdt in dat een kleine wijziging in de aankomst- of vertrektijd van een vlucht er niet toe mag leiden dat voor een groot aantal binnenkomende vluchten nieuwe gates moeten worden gevonden. Dit alles maakt het gatetoewijzingsprobleem een ingewikkeld planningsprobleem, waarvoor algoritmisch onderzoek vereist is om tot een goede oplossing te komen.

Om het gatetoewijzingsprobleem op te lossen formuleren wij het probleem eerst als een geheeltallig lineair programmeringsprobleem. Hiervoor maken wij gebruik van een formulering die gebaseerd is op zogenaamde gateplannen. Een gateplan bestaat uit een reeks van vliegtuigen die aan dezelfde gate wordt toegewezen. Een oplossing voor het gatetoewijzingsprobleem bestaat nu uit een verzameling van gateplannen die wij selecteren en waarvoor geldt dat elk vliegtuig in precies één van de geselecteerde gate plannen is opgenomen en er precies evenveel gateplannen zijn geselecteerd als er gates zijn. Omdat het aantal mogelijke gateplannen echter gigantisch is, gebruiken wij de methode van kolomgeneratie om alleen die gateplannen te genereren, die waarschijnlijk interessant zijn voor de optimale oplossing. Om deze aanpak te toetsen, hebben wij onze methode geïmplementeerd en uitvoerig getest met data die door Schiphol ter beschikking waren gesteld. Uit de toetsing blijkt dat de implementatie ons in staat stelt het gehele dagelijkse gatetoewijzingsprobleem van de luchthaven Schiphol in een beperkt aantal minuten op te lossen.

Op Schiphol zijn sommige ‘gates’ (opstelplaatsen) niet vast aan het luchthavengebouw verbonden door middel van een loopbrug. Aangezien het niet de bedoeling is dat passagiers zomaar tussen de vliegtuigen over het platform rondlopen, worden zij met bussen van (of naar) de vliegtuigen vervoerd. Het aantal beschikbare bussen is echter beperkt. Dit leidt tot het tweede planningsprobleem dat wij in dit proefschrift bestuderen: het busplanningsprobleem. Hierin is de vraag om te bepalen welke bus welke rit van (of naar) een vliegtuig rijdt om passagiers te vervoeren, dit alles met inachtneming van de bekende randvoor-

waarden qua tijden en het totaal aantal inzetbare bussen op elk tijdstip. Ook voor dit probleem proberen wij een oplossing te vinden die robuust is zodat wij niet alles opnieuw moeten plannen op het moment dat een vlucht eerder of later dan gepland aankomt. De oplossing voor het gatetoewijzingsprobleem die precies aangeeft waar welk vliegtuig zal staan, kan als invoer worden gebruikt bij het bepalen welke bus welke rit zal rijden. Op soortgelijke wijze als het gatetoewijzingsprobleem kunnen wij het busplanningsprobleem oplossen. Nu maken wij gebruik van het idee van busplannen, waarbij een busplan de reeks van ritten is die door een zelfde bus zal worden gereden. In dit deel van het proefschrift laten wij zien hoe belangrijke verschillen in randvoorwaarden met het gatetoewijzingsprobleem kunnen worden opgevangen in de oplosmethode (bijvoorbeeld moet er rekening worden gehouden met de verplichte rustpauzes voor de buschauffeurs).

Thans worden de genoemde twee planningsproblemen op Schiphol dagelijks als een keten opgelost: eerst wordt het gatetoewijzingsprobleem opgelost en vervolgens wordt de uitkomst daarvan gebruikt als invoer voor het busplanningsprobleem. Deze aanpak kan resulteren in een situatie waarin een geweldige oplossing voor het gatetoewijzingsprobleem gevonden wordt, maar waarin deze oplossing alleen maar slechte oplossingen voor het busplanningsprobleem toestaat. Indien er tijdens het oplossen van het gatetoewijzingsprobleem rekening gehouden wordt met bepaalde wensen of informatie vanuit het busplanningsprobleem, kan de oplossing voor de twee problemen als geheel een stuk verbeteren. Om dit te kunnen doen moeten de twee modellen worden geïntegreerd. In hoofdstuk 6 laten wij zien hoe dit kan. Onze geavanceerde methode voor een geïntegreerde aanpak hebben wij wederom geïmplementeerd en uitvoerig getoetst. De tests laten zien dat met onze aanpak de combinatie van het gatetoewijzings- en busplanningsprobleem in ongeveer een kwartier kan worden opgelost, veel sneller dan tot nu toe haalbaar werd geacht. Het laat zien hoe men met behulp van kolomgeneratie voor grote en geïntegreerde planningsproblemen, zoals die voorkomen op een groot vliegveld als Schiphol, zeer goede oplossingen kan vinden en wel binnen de redelijke tijd van een aantal minuten tot een kwartier op normale computerapparatuur.

Curriculum vitae

Guido Diepen was born on May 26th 1980 in Bergen op Zoom, the Netherlands. In 1998 he received his VWO-diploma from the St. Willibrord College in Goes. From 1998 to 2003 he studied computer science ('Informatica') at Utrecht University. His master's thesis was entitled "Solving the Gate Assignment Problem using Column Generation". This thesis was supervised by Dr.ir. Marjan van den Akker and Dr. Han Hoogeveen. In 2004 he started as a Ph.D. student in the Department of Information and Computing Sciences at Utrecht University in the group of Prof.dr. Jan van Leeuwen.

Colofon

This thesis was typeset with $\LaTeX\,2_\varepsilon$.

The design of the cover was done by Sjoerd Diepen.

ISBN: 978-90-393-4859-8

© 2008 by Guido Diepen (guido@guidodiepen.nl). All rights reserved.

Notes