Thesis for the degree of Doctor of Philosophy

The Tail Assignment Problem

Mattias Grönkvist

CHALMERS | GÖTEBORG UNIVERSITY

Department of Computer Science and Engineering
Chalmers University of Technology
and Göteborg University
SE-412 96 Göteborg, Sweden

Göteborg, August 2005

The Tail Assignment Problem
Mattias Grönkvist
ISBN 91-7291-645-1

Doktorsavhandlingar vid Chalmers Tekniska Högskola
Ny serie nr 2327
ISSN 0346-718X

Technical Report 8D
Department of Computer Science and Engineering
Algorithms Research Group

Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University
SE-412 96 Göteborg
Sweden
Telephone +46 (0)31-772 1000

Cover: An Iran Air Boeing 747SP-86 (tail number EP-IAA, cn 20998/275) blowing up a cloud of snow on take off.
Landvetter Airport, Göteborg, Sweden, December 12, 1998.
Photo: © Jan Mogren – AirPixPro.com, jm@airpixpro.com.

Chalmers Reproservice
Göteborg, Sweden, 2005

Till mamma

The Tail Assignment Problem
Mattias Grönkvist
Department of Computer Science and Engineering
Chalmers University of Technology and Göteborg University

Abstract

The *Aircraft Assignment problem* is the problem of assigning flights to aircraft in such a way that some operational constraints are satisfied, and possibly that some objective function is optimized.

We propose an approach to aircraft assignment which captures all operational constraints, including minimum connection times, airport curfews, maintenance, and preassigned activities. It also allows multiple fleet optimization, and can model various types of objective functions, which sets it apart from most other aircraft assignment approaches. The name we have chosen for our approach is the *Tail Assignment problem.* The tail assignment problem is general in the sense that it can model the entire aircraft planning process, from fleet assignment to the day of operation.

We develop solution methods based on mathematical programming and constraint programming, and also show how the methods can be combined. The resulting hybrid solution method is general in the sense that it can be used both to quickly find initial solutions and to find close to optimal solutions, if longer running times are allowed.

We present a mathematical programming approach based on column generation, conduct thorough computational experiments to show the impact of various implementation choices on running time and convergence, and present heuristics to find integer solutions.

We show how constraint programming can be used stand-alone to quickly produce solutions to the tail assignment problem, and to substantially improve the computational performance of the column generation algorithm. Preprocessing algorithms based on constraint programming are presented that can reduce the size of the problem substantially more than standard balance-based preprocessing, resulting in major speedups and increased solution quality. Our complete solution approach combines column generation, constraint programming and local search.

Finally, as proof of concept, we present modeling examples and computational results for a selection of real-world tail assignment instances, demonstrating how our model and solution methods can be used to increase operational robustness, enforce equal aircraft utilization, and decrease aircraft leasing costs. Our tail assignment system is currently in use at two medium-sized airlines.

Keywords: *fleet planning, airline optimization, column generation, constraint programming, hybrid optimization, tail assignment, aircraft routing.*

Acknowledgments

I would like to start by thanking my supervisor during my years as a Ph.D. student, Dag Wedelin, who has been a great help and motivation. His deep insights in algorithms as well as the real-world problems I have studied have really been useful. The other members of my Ph.D. advisory committee, Devdatt Dubhashi and Michael Patriksson, have been a great help in directing and supervising my work.

I would also like to thank the people at the Department of Computing Science at Chalmers for providing a stimulating and fruitful working environment. I would especially like to thank Birgit Grohe and Erik Kilborn for the interesting discussions, mostly about cost propagation and constraint programming, over the years.

Since the work presented in this thesis has been done in collaboration with, and is partially funded by, Carmen Systems AB, I would like to extend my gratitude to the people at Carmen Systems who gave me the opportunity to embark on the long journey of Ph.D. studies. Special thanks to Erik Andersson for continued support and encouragement.

The one person, except myself, who has had the most impact on the work presented in this thesis, is Sami Gabteni. Sami has always had great confidence in my work, and has never failed to support and help me. I would also like to thank Darren Dejun Hang and Magnus Larsson, members of the original Carmen fleet optimization team, for the stimulating cooperation.

I also want to thank Curt Hjorring, Rastislav 'Rastjo' Galia and Stefan Karisch for excellent cooperation, help and continued encouragement. The Carmen Matador team, consisting of Hamid Kharraziha, Marek Ozana, Johan Andersson, Geoff Bache, Claes Verdoes, Sami Spjuth and Jens Kjerrström, have provided a truly remarkable working environment. Working together with the matador team is never boring. Special thanks to Hans Eriksson, Oscar Rodriguez, Christopher Scott and Karine Cousineau-Ouimet, who have put my work to the ultimate test – reality.

Finally, I thank my family. You have always supported me and believed in me. I owe you much more than words can ever express. Jessica and Nova, you are the light of my life. Everything, including writing a doctoral thesis, is much easier with you.

Nova – kan du säga 'doktorsavhandling'?

Mattias Grönkvist
Göteborg, August 2005

Contents

List of Figures

List of Tables

List of Algorithms

List of Models

Part I

Introduction to Airline Planning and Tail Assignment

One

Introduction

The topic of this thesis is aircraft planning, and more specifically the problem of determining which individual aircraft should operate which flight in a given schedule. All flights must be operated by exactly one aircraft, and the aircraft assignments must satisfy a number of constraints, such as making sure that aircraft can be maintained at regular intervals. Often, but not always, the aircraft assignments should be planned so as to optimize some *objective function.*[1] We will use the name *Aircraft Assignment* to denote the general problem of assigning aircraft to flights. *Tail Assignment* is the name we have chosen for our specific approach to aircraft assignment. The name tail assignment comes from the fact that individual aircraft are identified by their *tail numbers*, and that the tail assignment problem considers individual aircraft.

The aircraft assignment problem is only one of the large planning problems solved by commercial airlines on a regular basis. Airlines are in fact faced with some of the largest and most difficult planning problems known. For example, one major European carrier operates and plans about 1400 flights per day to over 150 cities in 76 countries, using about 350 aircraft of 11 different types, and about 3400 cockpit, 14000 cabin, and 8300 ground crew.

The most important resources to plan for an airline are the aircraft and flight crew. Fuel consumption, other aircraft expenses and flight crew salaries typically represent the largest expenses for an airline. Using the best suited aircraft on each flight is thus crucial to be able to transport as many high-paying passengers as possible. Similarly, to reduce the crew costs, the crew must be utilized as efficiently as possible, without violating security and union regulations. Airlines also need to plan utilization of other resource types, e.g. ground crews and departure slots, but the planning of these is often less crucial

[1]An objective function specifies a value for each possible solution. If the objective function specifies the cost of the solution, the goal is to find a solution which minimizes this cost, while if the objective function specifies a quality value, the goal is to find a solution which maximizes the quality.

for the total profitability.

In this thesis, we present two separate approaches to solve the tail assignment problem, based on mathematical programming and constraint programming respectively, and proceed to present a hybrid mathematical programming, constraint programming and local search approach. Finally, computational tests for a number of real-world modeling examples are presented.

The work has been conducted in collaboration with, and has been partially funded by, Carmen Systems AB. Carmen Systems' crew planning software is used by most of the major European airlines, including British Airways, Lufthansa, KLM, SAS, Alitalia, Iberia, Air France, as well as a few American and Asian airlines, and a few railway companies. Carmen Systems has its headquarters in Gothenburg, Sweden.

Some of the work described in this thesis is joint work together with the optimization group at Carmen Systems AB, but the implementations described are all the work of the author. The tail assignment planning system which is presented in this thesis is currently used in production at two medium sized airlines.

1.1 Modeling the Aircraft Assignment Problem

Normally, the airline planning[2] process is divided into several sequentially executed steps. The first step is to decide which airports should be served, and at which time flights should be scheduled. Then the aircraft type to use for each flight is decided. Given the aircraft type, the number of cockpit and cabin crew members necessary to operate the aircraft is known, and the next step is then to create working schedules for the crew. The last part of the planning process is to decide which individual aircraft should operate each flight. We use the general name aircraft assignment to denote this last problem.

While the processes for planning aircraft types and scheduling crew are similar at most airlines, the aircraft assignment process differs quite a lot between airlines. Some airlines plan the aircraft assignments very close to flight departure, while others plan them longer in advance. Roughly speaking, airlines with very regular schedules over a normal week and fairly large fleets of similar aircraft types can manage to plan aircraft assignments very late, while airlines with more irregular schedules and many different aircraft types might need to plan aircraft assignments longer in advance. Some airlines create preliminary aircraft assignments very early in the planning process, to

[2]Sometimes, *airline scheduling* is used as a synonym for airline planning. This can be somewhat confusing, since planning normally means the process of allocating resources to fixed time activities, while scheduling means deciding times for activities. To be consistent, we will used the term airline planning throughout this thesis, except of course where scheduling is appropriate.

use as input data to the crew planning process, assign aircraft to the most profitable multi-leg itineraries, or do long-term maintenance planning.

The many possible variations of aircraft assignment means that many different solution approaches have been proposed in the literature. The most common type of approach creates aircraft routes, respecting generic maintenance constraints and maximizing the possibilities for passengers to use the same aircraft on multi-leg itineraries, but typically does not consider individual operational constraints. Other approaches are focused only on maintenance feasibility, but only very few approaches considering individual aircraft constraints, or more general optimization criteria, have been proposed.

Because of the different scopes of the proposed approaches to aircraft assignment, it is not completely clear whether the aircraft assignment problem should be considered to be an optimization problem or a pure feasibility problem, i.e. whether an optimal solution, given some objective function, is necessary or not. The aircraft assignment approaches maximizing same-aircraft connections for passengers are clearly optimization problems, while pure maintenance planning problems are normally treated as feasibility problems. On the other hand, airlines which plan aircraft assignments only a few days prior to flight departure normally consider aircraft assignment to be a feasibility problem.

The Tail Assignment Problem

In the tail assignment problem, which is what we call our approach to aircraft assignment, all individual operational constraints are considered, including maintenance and restriction constraints. The problem is solved for a fixed time period, e.g. one month, and considers preassigned activities as well as time-dependent constraints and restrictions. Unlike most presented approaches to aircraft assignment, tail assignment handles many different types of constraints, as well as a wide range of objective function types.

Treating the aircraft assignment problem as an optimization problem, rather than as a feasibility problem, has several advantages. As already mentioned, it makes it possible to assign the same aircraft to the most profitable multi-leg itineraries, but many other aspects of the aircraft assignments can also be optimized, such as various types of robustness in case of operational disruptions. In our approach, it is also possible to quickly re-optimize fleet assignments close to flight departure, e.g. in case the expected passenger load has changed since the long-term fleet assignment was performed. In this case, revenue data can be used to model a true cost-focused objective function, making optimization crucial.

While most literature on fleet planning and aircraft assignment only considers a limited scope of the complete fleet planning process, our approach is flexible enough to be used in several stages of the planning process. While the main focus of the tail assignment problem is to aircraft assignment, it is also flexible enough to consider aspects which are traditionally not handled within aircraft assignment, such as the re-fleeting mentioned above. As far as we know, no aircraft assignment or fleet planning approach as general as our tail assignment approach has been proposed before.

It should be noted that it is not our intention that tail assignment should fit perfectly into the *current* planning process at all airlines. It is rather a proposed planning step which has many benefits compared to current practice at many airlines, and which can be used directly by others. However, given a planning process in which tail assignment fits, our models and solution methods should be customizable enough to fit the needs of any airline in the world. This is a very important goal – almost all published research on airline planning, especially for aircraft assignment, focuses on the major North American carriers. This means that the networks of flights often have a very pronounced hub-and-spoke structure,[3] and that the regulations for crew and aircraft are quite homogeneous, at least for the domestic flights. In other parts of the world, conditions can be quite different, with more distributed flight networks, and different regulations in different countries. Some airlines even have aircraft registered in different countries, to make it easier to adhere to national restrictions. One thing that seems particularly common for U.S. airlines is the creation of a single repeating route, which all aircraft should fly. Tail assignment cannot directly be used to find such a route, because it does not make sense when solving for a fixed time period.

1.2 Solving the Tail Assignment Problem

Since the aircraft assignment problem can be seen either as an optimization problem or as a feasibility problem, our goal is to develop solution methods which can both obtain feasible solutions quickly, and close to optimal solutions, when more time is available. This consideration has lead to the investigation of solution methods combining Mathematical Programming and Constraint Programming. While mathematical programming techniques are very common in airline planning applications, the use of constraint programming is rare. None of the previously proposed solution approaches for aircraft assignment uses constraint programming.

Mathematical Programming and Constraint Programming

Mathematical programming is the study of so-called mathematical programs, where the goal is to find a solution minimizing or maximizing some objective function, while satisfying a number of constraints. In the most general form, the objective function and the constraints can be of any kind. Linear,

[3]Hub-and-spoke networks are the most common types of flight networks, and will be described further in Section 2.1.

non-linear, integer and stochastic programming are common special cases of mathematical programming. Mathematical programming techniques have a vast number of application areas, and are often used to solve problems arising in Operations Research (OR). Operations research is the research field of analyzing, modeling and optimizing real-world processes.

The field of constraint programming (CP) is much younger than mathematical programming, and more specialized. Constraint programming is a problem solving technique which uses general search techniques mixed with logical deductions. It has its roots in Artificial Intelligence and Computational Logic, and has since it was established in the 1970s evolved to a candidate technique for solving large-scale feasibility problems. One of the drawbacks of constraint programming, however, is that it is currently not particularly suitable for heavy-duty optimization, since it is focused only on feasibility. In mathematical programming, the situation is often the opposite – optimization is the main focus, and feasibility is a secondary priority. In other words, in mathematical programming it is typically considered more important to find the optimal solution than to quickly show that a solution exists at all.[4]

Clearly, these drawbacks and strengths of the two fields complement each other well, indicating that integration might be fruitful. Today, a lot of effort is indeed spent on integrating mathematical and constraint programming [41, 44, 86, 110, 176, 186], both for various applications, and on a more theoretical level. Since we want to be able to solve the tail assignment problem both as a feasibility problem and as an optimization problem, this research direction is clearly very suitable for us. Much of the work described in this thesis is thus a contribution to this rather new research direction, and parts of our work could possibly be used in other application areas as well.

The Mathematical Programming Approach

In mathematical terms, the tail assignment problem can be modeled as a so-called *integer multi-commodity flow problem with side constraints*. The multi-commodity flow problem is a very well studied problem, since many real-world problems can be modeled as multi-commodity flows. Because of the nature of the side constraints, as well as some other considerations, the most suitable solution technique in our case is *column generation*. Column generation is closely linked to *Dantzig-Wolfe decomposition* [61], which was proposed as a general solution method for linear programs with special structure already in 1960. However, due to its computational requirements, it is not until recently that it has come to significant practical use. One field where column generation has really become widely used is airline crew planning.

[4]Obviously, in case no solution exists, no optimal solution exists. However, many techniques for finding optimal solutions are not suitable, in terms of performance, for determining feasibility.

The basic implementation of column generation for tail assignment is fairly straightforward. However, column generation is known to converge slowly, and we therefore investigate, implement, and compare various acceleration techniques to obtain acceptable performance for practical use. Some of these acceleration techniques are documented in the literature already, while others are more unique.

The standard column generation algorithm only solves the linear programming relaxation of our integer multi-commodity flow problem, forcing us to investigate special algorithms for finding integer solutions. We investigate and implement a number of heuristics, which add heuristic fixing to the column generation algorithm, gradually restricting variables to take more and more integer values, and re-generating columns, until an integer solution is found. Using heuristics rather than exact methods provides a reasonable compromise between solution quality and running time performance. The heuristics are also versatile, and can be tuned to either be very aggressive, i.e. fast but not providing very high quality solutions, or vice versa. Lower bounds, obtained either via a network flow relaxation of the multi-commodity flow problem, or via the linear programming relaxation, provide control of the solution quality.

Our computational tests indicate that provided enough running time, our fixing heuristics will find very high quality solutions. Unfortunately, even the most aggressive fixing heuristics cannot provide solutions quickly enough to be useful when solutions are required instantly, for reasonably sized instances.

The Constraint Programming Approach

As an alternative solution method, especially for obtaining solutions quickly, constraint programming is used. We present a constraint programming model for the tail assignment problem that mixes standard and specialized constraints and propagation algorithms. The constraint programming model reuses the column generation pricing module, which captures the maintenance constraints. This is a significant advantage in practice, as it means that the complicated maintenance constraints only need to be handled by one separate module in our system. The maintenance constraint implementation can therefore be improved and extended in various ways without re-doing the same work twice.

Our constraint programming model provides solutions very quickly even for large tail assignment instances. However, another practical use of the constraint model is as a general preprocessing tool. Instead of asking the model for a solution to a problem, we ask it to apply the constraints repeatedly to make logical deductions which can be used to remove redundant input data, leaving a smaller problem for the optimizer. This kind of advanced preprocessing can often substantially reduce the problem size, and we show that it has a huge impact on e.g. column generation performance, both in terms of running time and solution quality.

As an extension of the preprocessing idea, the constraint model can also be used during the column generation fixing heuristics to ensure that no conflicting fixations, which can never lead to a solution, are done. Using the constraint model in this way leads to the possibility of more aggressive fixing methods, which can almost rival the constraint model itself in terms of running times.

We thus investigate how constraint programming can be used to accelerate our column generation based solution method, and also how parts of our column generation implementation can be re-used in our constraint model.

Combining the Approaches

Finally, we present a way to wrap the column generation and constraint programming approaches into a local search framework. We propose a method which uses our constraint programming approach to find an initial solution, and proceed to gradually improve this solution by locally re-optimizing it using column generation. The local improvement algorithm is applied iteratively until no more improvement is possible, the current solution is good enough, or a time limit is reached. Network flow or linear programming lower bounds can again be used to provide solution quality control.

This solution approach is very appealing for practical purposes. Since it usually finds the initial solution quickly, this solution can immediately be investigated by a user of the system to get some idea of the structure of the final solution, check that all constraints could be satisfied, and so on. Meanwhile, the optimizer keeps working in the background to find solutions of gradually improving quality. This kind of manual intervention is often necessary even for well tuned systems, as flight data and constraints gradually change, and new scenarios are tested. On the other hand, a system which only produces an optimal solution will make it very cumbersome and time consuming to try different planning scenarios.

This indicates that it is not only in cases where no objective function is used that it is interesting to obtain solutions quickly – this ability is in fact a very valuable feature even for situations when an objective function does exist. Another example of such a situation is if the tail assignment problem is solved very close to flight departure, when immediate proposals for re-planning are required.

1.3 An Engineering Perspective

As discussed above, we will present both solution methods which can find close to optimal solutions, and solution methods which focus only on quickly finding a solution regardless of its quality. This means that running times will often be used to judge the performance of our algorithms. However, the running time of an algorithm is a somewhat subjective measure – whether it is considered long or short is decided mainly by expectations, or by practical considerations, such as how soon a solution is required. When heuristic solution methods are used, which is almost always the case for large, difficult problems of any kind, increased running times often means higher solution quality. This further complicates the analysis of running times, by forcing one to weigh running time versus solution quality.

Unfortunately, it is very difficult to avoid this situation. One way to avoid it, which is often used in academic work, is to completely ignore either the solution quality, or more commonly the running time. Another common approach is to simplify the problem to the point where optimal solutions can be found with reasonable computational effort.

However, for a practical application it is very difficult to justify either of these approaches – the correct problem must be solved, and the algorithms must be fast and provide good solutions. Instead, in practical applications one is often prepared to accept that an algorithm does not guarantee to find an optimal solution, if this means it will instead be faster. Most of the solution methods we present are heuristic, and thus cannot guarantee to find optimal solutions. For some instances, the lower bounds which we calculate can however be used to prove optimality.

When we analyze running times and solutions in this thesis, we do this from a user's perspective. For example, if we claim that a certain running time is unacceptably high, we mean that a typical user of our system would probably not accept this running time. With our gradual improvement solution process, weighing running time versus solution quality is ultimately up the the user of the system.

It is also important to note that for most of the test instances used in the thesis, the most important objective is to assign all activities. Often, the objective function only measures the solution quality in non-monetary terms. This means that a solution can often be considered acceptable even if its objective function value is far above the optimum value, as long as all activities are assigned, and the solution can be obtained quickly enough. Of course, this does not mean that the objective function value is unimportant – it is always the goal to find a solution as close to optimality as possible!

Computational Tests

Throughout this thesis, computational testing will be performed for each individual algorithm component. Since much of the thesis material is collected from articles produced in different stages of a developing project, different test instances are used in different chapters. The characteristics of the test instances are presented before they are first used.

While the tests performed throughout the thesis indicate the computational performance of our solution methods, Chapter 13 presents final computational results for our combined approach, and shows how a number of example instances can be modeled and solved with our approach.

All computational tests have been run on a dual Intel Xeon 2.2 GHz computer with 512kB L2 cache and 2048MB main memory, running Red Hat 7.2 or Enterprise Linux 3, and using ILOG Solver 5.2 [122], ILOG CPLEX 7.5 [121], and Xpress v. 14 [64]. ILOG Solver and ILOG CPLEX are trademarks of ILOG Inc., and Xpress-MP is a trademark of Dash Associates.

All programs are compiled with GCC (g++) version 3.2.3. Reported running times are CPU running times, and have been rounded to full seconds.

1.4 Main Contributions

The main contributions of our work are the following:

The definition of a general tail assignment approach. Most proposed approaches for aircraft assignment type problems cover only a limited scope of the aircraft planning process, and can only handle a limited number of predefined constraints and objective functions, often in an approximative way. We present an approach which models individual constraints and costs exactly. It can capture many different kinds of constraints and objective functions, making it usable for different modeling scenarios. Our approach is also general enough to be used in several stages of the planning process, and has been designed to be flexible enough to be used by almost any airline. Our tail assignment problem is discussed in detail in Chapter 3, and in this technical report:

- Sami Gabteni and Mattias Grönkvist. A Unified Model for Airline Fleet Management. Technical Report, Carmen Systems AB, Gothenburg, Sweden, 2005.

The use of constraint programming in aircraft assignment. Prior to our work, no full solution approaches to aircraft assignment type problems using constraint programming has been presented, as far as we know. We present a constraint programming approach which can quickly obtain solutions to real-world tail assignment instances. The approach uses traditional efficient constraints as well as specialized propagation algorithms, and further shows that constraint programming can benefit from interaction with a column generation pricing module. The material related to our constraint model is presented in Chapter 9, and has been published in the following article:

- Mattias Grönkvist. A Constraint Programming Model for Tail Assignment. In J.-C. Régin and M. Rueher, editors, *Proceedings of CP-AI-OR'04: International Conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Prob-*

lems, vol. 3011 of Lecture Notes in Computer Science, pages 142–156. Springer-Verlag, 2004.

The integration of constraint programming and column generation. We present a novel approach to integrating constraint programming and column generation. By using our constraint programming model in several different ways during the column generation and the subsequent fixing heuristics, we show how constraint programming can be used to improve both column generation running time performance and convergence, and the performance of the fixing heuristics. The material most related to constraint programming and column generation integration can be found in Chapters 10–11, and can be found in the following articles and technical reports:

- Mattias Grönkvist. Accelerating Column Generation for Aircraft Scheduling Using Constraint Propagation. To appear in *Computers & Operations Research*.
- Sami Gabteni and Mattias Grönkvist. A Hybrid Column Generation and Constraint Programming Optimizer for the Tail Assignment Problem. Technical Report, Carmen Systems AB, Gothenburg, Sweden, 2005.

The extensive computational testing on real-world instances. Our tail assignment system is currently used in production by two medium sized airlines. Using data from these and other airlines, we provide extensive computational testing of all aspects of our solution methods, demonstrating the practical usefulness of our tail assignment approach. All our articles and technical reports contribute to this.

Finally, the following article gives an overview of our tail assignment approach, focusing especially on the column generation approach:

- Jens Kjerrström and Mattias Grönkvist. Tail Assignment in Practice. In *Proceedings of OR2004*, Springer-Verlag, 2005.

1.5 Outline of the Thesis

The thesis is divided into four parts:

Part I – Introduction to Airline Planning and Tail Assignment
The first part, consisting of Chapters 1 through 3, describes the airline planning process and gives our definition of the tail assignment problem. It contains general literature surveys of various parts of the airline planning process, as well as a more in-depth survey of literature related to aircraft assignment.

Part II – A Mathematical Programming Approach
In the second part, consisting of Chapters 4–7, mathematical models and solution methods for the tail assignment problem are presented. An overview of the literature related to similar mathematical models is included, as well as thorough computational testing of the selected solution methods.

Part III – A Constraint Programming Approach
The third part, consisting of Chapters 8–11, introduces constraint programming concepts, and formulates and solves constraint programming models of the tail assignment problem. It is also shown how elements of the constraint programming models can be used to improve the computational performance of the mathematical models described in Part II.

Part IV – The Complete Approach and Final Results
In the final part, consisting of Chapters 12 through 14, the complete system, embedding the mathematical and constraint programming models in local search, is presented. We also briefly present some of the components of our tail assignment system which are not related to optimization. Computational results for the complete system are presented. A number of modeling examples from real-world applications and the literature are presented, solved and analyzed to show the flexibility and usefulness of the tail assignment problem and our solution approach. Finally, we summarize and discuss our findings.

Two

Airline Planning

In this chapter we will give a brief introduction to the airline planning process. We will focus on the crew and fleet planning, as these are the parts most related to this thesis. We will here only give an overview of the aircraft assignment problem, of which our tail assignment problem is a special case. A detailed discussion of the tail assignment problem, as well as related aircraft assignment approaches, follows in Chapter 3.

The airline planning process is normally divided into several steps, representing different resource areas, as depicted in Figure 2.1. The sequential process is the traditional way that the planning is performed, but sometimes feedback between the stages can occur. The first step is to model the market behavior to construct a timetable. The next step is to decide which aircraft type (*fleet*) is going to operate each flight in the constructed timetable. This makes sure that it is possible to actually operate the timetable, that there are no imbalances between the flights and that there is sufficient turn time. This aircraft type assignment must be done before any other resource planning can be done, because it determines the input for the other steps. Depending on which aircraft type is used, a flight will e.g. require different numbers of cockpit and cabin crew, with different qualifications. Throughout the entire planning process, the number of seats to allocate to each fare class on each flight is dynamically updated to be consistent with the current demand.

Once the aircraft type for each flight has been decided, the (flight) crew planning is most often done. Each crew member must get an individual work schedule (called a *roster*) well in advance of the day of operation. The crew planning step is very complex when it comes to rules and regulations. There is a lot of safety and union rules that must be fulfilled by each crew roster, such as restricting the working time and ensuring enough rest time. Then, routes for the individual aircraft are constructed, ensuring that each aircraft gets enough maintenance, and that all operational constraints are fulfilled. Finally, from the end of the long-term planning phase to the day of operation, various types of recovery options must be evaluated to account for unexpected events such as sick crew members, delays, and airport closures. Excellent overviews of airline planning problems, and the use of operations research models to solve them, can be found in recent survey articles by Gopalan and Talluri [103] and Barnhart et al. [25].

In the rest of this chapter we will briefly discuss the different planning steps. We will also discuss integrated planning, as this is a field that is gaining increasing attention. Also, one of the motivations behind solving the tail assignment problem the way we will propose is that it facilitates integration between crew, fleet and aircraft planning.

2.1 Airline Network Types

Since the type of *flight network* an airline uses has a dominant impact on many of the planning problems, we will start by describing the common network types. Flight network is an informal name for the geographical network created by the flights operated by an airline timetable. There are three types [156] of airline networks – *linear networks*, *point-to-point networks* and *hub-and-spoke networks*.

Linear networks are networks in which all airports are connected by a single tour. Point-to-point networks connect all airport pairs, to form a *complete graph*. In a point-to-point network it is thus possible to fly from any airport to any other airport with a single flight. Finally, in hub-and-spoke networks all flights connect to and from a central hub. The non-hub airports are called *outlying* airports, or *spoke* airports. To fly between two outlying airports, one must thus first fly to the hub, change flights, and continue to the destination airport.

The network types described above are the pure definitions. In reality it is seldom the case that an airline has a pure point-to-point or hub-and-spoke network. Most airlines have some sort of hub, housing their main maintenance facilities and crew headquarters. And in most hub-and-spoke networks multiple hubs exist, as well as some direct flights between outlying airports.

Traditionally, hub-and-spoke networks have been dominant for U.S. airline networks, while point-to-point type networks have been more common in the rest of the world. Today, this is only partially true – more and more airlines are moving toward hub-and-spoke type networks, as it makes cost savings possible due to centralized operations. By arranging arriving and departing flights at the hubs into groups, so-called *banks*, hub-and-spoke networks also make it easy to provide flights to/from a large combination of airports, with short connection times, without resorting to a point-to-point network. Today, pure point-to-point networks are only operated by low-cost airlines, which see them as possibilities to avoid crowded and expensive large airports, and are prepared to take the extra risk of large disruptions due to local operational

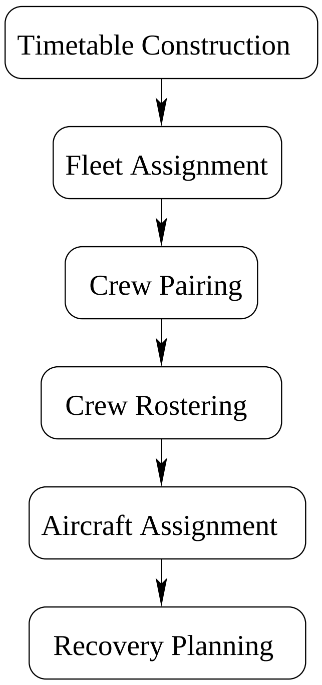


Figure 2.1 The typical airline planning process.

problems. Linear networks are not very common, but variants of them can sometimes be found e.g. at charter airlines.

2.2 Timetable Construction and Yield Management

The very first planning step is to construct a timetable. This is done using market models to try to estimate how much demand there is for flights between different places at different times. The most crucial factors are commercial aspects such as which markets to operate, and which financial margins are acceptable for each market. Other factors, such as timetable synchronization within airline alliances, can also be taken into consideration. The timetable planning problem is not studied much in operations research literature, except in connection to the fleet assignment problem (see Section 2.3).

A vastly more studied area related to market modeling is *Yield Management*. Yield management, which is sometimes also referred to as *Revenue Management*, is the problem of allocating capacity to the different fare classes in order to maximize revenue. Obviously, airlines want to sell as many tickets as possible to high-paying business travelers, who often book their tickets close to the departure date. It is therefore important to make sure that there is capacity left in the aircraft close to departure. To decide how to price tickets to make sure that enough economy tickets are sold, while reserving seats for late-booking business travelers, requires careful modeling of market demand and behavior, as well as advanced mathematical models that can react to changes. The yield management process is continued throughout the planning process, until the day of operation, to make sure that the seat allocation always reflects the demand as exactly as possible. Yield management provides important input to the fleet assignment solution, so these two problems are closely linked.

The importance of yield management is indicated by the fact that American Airlines in 1992 reported a 5% revenue increase (about \$1.4 billion over three years) due to the introduction of a new yield management system [203]. Recent literature about yield management includes [47, 179, 206]. An excellent research overview is provided by McGill and Van Ryzin [168].

2.3 Fleet Assignment

The most crucial planning problem an airline faces is probably the *Fleet Assignment problem*. Once a suitable timetable has been created, suitable aircraft types (*fleets*) must be assigned to the flight legs. This assignment, which is often called a *fleeting*, is very important, since it has a direct impact on operating costs and passenger revenues. Using a small aircraft costs less in terms of fuel costs and landing fees than using a large one, but on the other hand cannot transport as many passengers. But the fleet assignment, along with the number of booked passengers, also determines how many flight and ground crew members are necessary to operate the flight, and thus influences crew costs as well. As an example of the importance of fleet assignment, Delta Air Lines in 1994 estimated that the use of a newly developed fleet assignment system would yield savings of up to \$300 million over the following three years [207].

In its most basic form, the fleet assignment is very simple to state: *which fleet should be assigned to each flight in order for the total profit to be maximized, while making sure that aircraft balance is maintained throughout the flight network, and no more than the available number of aircraft is used?*

The aircraft balance condition simply states that the aircraft that lands at a certain airport must take off from the same airport. It might seem like an obvious condition, but for a mathematical model it must of course be stated, or practically infeasible fleet assignments would be constructed. Since fleet assignment must be solved well in advance of the day of operation, estimates of passenger demand must be used. These estimates are often based on historical data, but should also consider known future events which will influence demand, such as e.g. Olympic Games.

The profit of an airline is the passenger revenue minus the operating cost. The operating cost is the fuel, crew costs and air traffic controller (ATC) and landing fees, and in fleet assignment it often also includes a term for *spill cost*. Spill cost is a 'goodwill' cost, accounting for passengers who wanted to fly, but who where unable to fit in the aircraft, because demand exceeded the aircraft capacity. Sometimes, the operating cost in fleet assignment also accounts for spilled passengers who take another flight in the same airline, and are *recaptured*.

In fleet assignment one often solves a *daily* problem, i.e. a solution is found for a typical day, and then rolled out and adjusted for variations over weekdays. Since demand and timetables can vary over the days of the week, and even over e.g. Mondays in different weeks, this is obviously an approximation. However, since timetables, at least for major U.S. airlines (to which most of the published research applies), are often similar over weekdays, and varies more only during weekends and holidays, this is considered a reasonable approximation. Gopalan and Talluri [103] also note that having the same aircraft type serve a flight every day of the week simplifies administration of e.g. maintenance and gates. They also claim that while the demand variation over the week can be important, it is often system-wide high/low on certain days rather than highly non-uniform.

The basic fleet assignment problem has been studied thoroughly. One of the first mathematical models is due to Abara [2], but this models has several modeling and computational limitations. In [112], Hane et al. propose a model and solution approach based on interior-point linear programming and branching, which has become the standard for fleet assignment. Hane et al. also present network preprocessing techniques which have since been used extensively, and [108] presents more details of the approach, including complexity analysis. Subramanian et al. [207] presents the fleet assignment model used by Delta Air Lines, and Rushmeier and Kontogiorgis [194] give an overview of recent advances in fleet assignment. In [78], El Sakkout proposes a fleet assignment solution approach using constraint programming, and Götz et al. [105] propose an approach based on simulated annealing.

Bélanger et al. [33] solve a weekly fleet assignment problem for Air Canada. Since the problem is weekly, they explicitly add *homogeneity* constraints to force flights to be covered by the same aircraft type every day. They present one solution technique based on directly solving the resulting integer program with branch-and-bound, and one heuristic two phase approach. Results show increased profits with the homogeneity constraints, and it can also be shown that even more can be gained if these constraints are not included.

Kohl et al. [143] present an variant of fleet assignment applied to locomotive assignment. Their model does not consider passenger revenues, but only tries to minimize operating costs. Their solution approach uses a Lagrangian heuristic to decompose the problem into minimum cost network flow problems, and a primal heuristic to construct primal feasible solutions.

Fleet Assignment Extensions

One drawback with the standard approach of first deciding the timetable and then the fleet assignments, is that it can lead to suboptimal solutions. An integrated model would decide both the departure and arrival times of the flights, and the aircraft type to operate them. Obviously, this model would be very complicated to model and solve, not least because the demand for a certain flight cannot be considered constant when its departure time changes. While a fully integrated model has still not been proposed for large-scale instances, Rexing et al. [185] propose a fleet assignment model with time windows, i.e. where the flight departure times can vary slightly around a preferred time. It is shown how this can significantly decrease the operating cost compared to standard fixed-time fleet assignment, or be used to tighten the timetable to potentially save aircraft. Ioachim et al. [123] present an approach to fleet assignment and routing with time windows and synchronization constraints, to synchronize departure times of the same flight on different weekdays.

Bélanger et al. [32] extend the model presented in [33] to handle time windows and time-dependent revenues. In a similar effort, Kliewer [139] discusses a simulated annealing approach to combined fleet assignment and market modeling. Recently, Lohatepanont and Barnhart [156] presented a combined scheduling and fleet assignment approach. They use the IFAS model described in [29] and wrap it in a layer controlling changes to an old timetable. They consider the fact that removed flights change the demand for the remaining flights, and can show excellent results for data from a major U.S. airline. On the test instances, several hundred thousand U.S. dollars can be saved every day, primarily by operating fewer flights.

Another serious drawback of standard fleet assignment is that it only considers passenger demand on single flights, while in reality passengers often fly several consecutive flights, connecting at hub airports, to get from their origin to their destination. To understand why the resulting 'network effects' are important to consider, imagine trying to decide whether to spill 5 economy-class passengers from flight A, or 10 economy-class passengers from flight B. Clearly, without any further information, spilling 5 passengers from flight A seems like the best idea. But if we would have looked at the entire itinerary, we would have seen that the 5 spilled passengers would have connected to an expensive transatlantic flight, while the 10 captured passengers only fly a single leg. Clearly, by spilling these 5 passengers, the airline has missed some possible revenue. On the other hand, considering itineraries obviously makes the fleet assignment problem considerably more difficult to solve. The pioneering efforts in itinerary-based fleet assignments are due to Farkas [88], and more recently to Kniker et al., who in three papers have discussed the drawback with conventional fleet assignment [140], and proposed an itinerary-based model [29] which combines a traditional fleet assignment model with the *Passenger Mix Model* [141].

Much effort has been spent on integrating fleet assignment with other planning processes. Integration is especially interesting for fleet assignment, as it serves as input to most other crew and aircraft planning problems. Barnhart et al. [26] present a model that attempts to combine fleet assignment and aircraft routing (see Section 2.5) by using so-called *strings* of maintenance-feasible flight legs. Clarke et al. [52] describe how aircraft maintenance and crew overnight considerations can be included in the fleet assignment model, and Desaulniers et al. [70] present a daily combined fleet assignment and routing approach. Ahuja et al. [5] present a combined fleet and through assignment model, and also present a multi-criteria extension of the model considering crew and ground staff planning aspects [6]. The through assignment and aircraft routing problems will be discussed further in Section 2.5.

Once the long-term planning has been completed, it can happen that the demand for certain flights changes dramatically compared to the estimates used in fleet assignment. If this happens, it might be necessary to perform so-called *re-fleeting*, i.e. to change aircraft types of some flights to increase profitability. It is important to note that re-fleeting sometimes requires involvement of several resource types, e.g. crew, since their published schedules might change due to the re-fleeting. One of the first approaches to re-fleeting is due to Berge and Hopperstad [36], who present the 'Demand Driven Dispatch' concept, which re-solves the fleet assignment problem with increasingly more accurate demand information as the day of operation approaches. Talluri [209] presents a heuristic swap-based approach to re-fleeting, while Jarrah et al. [125] demonstrate how conventional fleet assignment models can be used efficiently for re-fleeting. Rosenberger et al. [188] present a fleet assignment model incorporating consideration of operational robustness. They define *short cycles* as sequences containing few flights, starting at the hub. By making sure that many short cycles are present, it is possible to cancel entire cycles when disruptions occur, and thus change very few flights. Similarly, using cycles decreases hub connectivity, making it easier to isolate disruptions within a hub.

Winterer [225] solves the *flight swapping with retiming problem*, i.e. the problem of retiming and swapping flights between fleets close to the day of operation, as a special case of a general *requested resource reallocation with retiming problem* (R^4P). A specialized algorithm is presented, which can produce multiple solutions, and is shown to be more efficient than a MIP model for the tested instances. The instances come from two European airlines, and contain 20-107 aircraft and 95-644 flights, for 2-8 fleets. Aircraft home bases and passenger demand data were randomly generated. Simons [202] describes how constraint programming can be used to solve the aircraft swapping problem, without re-timing, when a limited number of flights need to be upgraded to larger aircraft.

2.4 Crew Pairing and Rostering

Since the main focus of this thesis is on fleet planning, we will only give a very brief introduction to crew planning, and provide references to the interested reader. The process of planning the crew is divided into two steps. First, the so-called *Crew Pairing* problem is solved. This is the problem of creating anonymous working *pairings* in such a way that the flights are covered by as many crew members as necessary, while minimizing costs. A pairing is a sequence of flights operated by the same anonymous crew member, starting and ending at the same crew base. The pairings must respect a large number of safety and union rules. One must also limit the number of pairings from each base to respect the number of crew available at this base. The optimization objective in crew pairing is to minimize costs, such as salaries and overnight expenses.

In the next step, called *Crew Rostering*, the objective is to assign the pairings to individual crew members, respecting their qualifications, vacation periods, training assignments, and perhaps also their individual wishes, while minimizing a cost function. A system which can handle wishes (*bids*) from crew members is often referred to as a *preferential bidding* system. Especially in the U.S., a rostering process known as *bidlines* is common. Bidlines are anonymous rosters for which the crew members can bid. The bidding is often performed in seniority order, making sure that the most senior crew members get their bids satisfied. In the crew rostering step, the optimization objective is related to robustness, or 'quality of life', rather than actual monetary costs. The airline crew planning process has been subject to extensive research, see e.g. [14, 66, 69, 71, 96, 97, 106, 109, 117, 118, 119, 136, 137, 138, 142, 151, 162, 172, 223].

2.5 Aircraft Assignment

The *Aircraft Assignment problem* is the problem of deciding which individual aircraft should be assigned to each flight leg. The flight legs assigned to an aircraft represent a *route*, i.e. a set of consecutive flights with pair-wise matching arrival and departure airports, to be flown in sequence by the aircraft. Since the aircraft type for each flight is already decided by the fleet assignment solution, one separate aircraft assignment problem per aircraft type must be solved. The main constraints to consider when assigning aircraft to flight legs are maintenance constraints and restrictions on which aircraft can operate which flight. Maintenance constraints ensure that all aircraft are subject to various kinds of maintenance checks with regular intervals, and flight restriction constraints ensure that various kinds of restrictions, due to e.g. noise levels and aircraft equipment, are satisfied. Observe that when aircraft assignments are created for single aircraft types, restrictions depending only on the aircraft type are automatically modeled.

While the fleet assignment and crew planning problems are modeled similarly for most airlines, the way the aircraft assignment problem is modeled and solved differs a lot between airlines. In fact, the term 'aircraft assignment' is seldom used in practice. We use it here only as a common name for a set of problems which all assign aircraft to flights, but which have different objectives and scopes. Some airlines plan the actual aircraft assignments very close to flight departure, while others do it longer in advance. Airlines with very regular schedules, large fleets of aircraft, and a lot of maintenance hangar capacity can typically manage to decide which aircraft should operate which flight very late, while airlines which do not meet these requirements must often plan their aircraft assignments longer in advance. However, the fact that some airlines plan aircraft assignments close to flight departure does not mean that they could not benefit from more long-term planning.

Airlines which plan aircraft assignments close to flight departure often use very simplistic procedures, such as *first-in first-out* (FIFO) or *last-in first-out* (LIFO) queuing, to decide which arriving aircraft should be assigned to which departing flight. Often, however, even these airlines do preliminary long-term aircraft assignment planning, for several different reasons. Since crew can connect between two flight legs in less time if they do not have to switch aircraft, the crew pairing process must get some preliminary aircraft routes as input data. However, using preliminary aircraft routes as input has its problems – the lack of integration between crew and aircraft planning might lead to situations where it becomes impossible to construct aircraft routes satisfying the assumptions made during crew planning. For example, Cohn and Barnhart [55] note that using too approximate aircraft assignments during crew planning can lead to maintenance infeasibility for the aircraft. Preliminary aircraft assignment problems are sometimes also solved to maximize so-called *through values*, or to do long-term maintenance planning. Through values represent the desirability of one-stop service, i.e. a multi-leg flight without aircraft change, between two airports. Maximizing through values thus has the effect of trying to make sure that there is no aircraft change for these desirable connections, since passengers are often willing to pay an additional fee for this.

So, whether or not the true aircraft assignments are planned early or late, most airlines solve some kind or long-term aircraft assignment problem. The most common variant of aircraft assignment is *Aircraft Routing*,[1] indicating that routes must be created for all aircraft. Occasionally the term *Aircraft Rotation problem* is used to denote the same problem.

Many approaches to the aircraft routing problem, especially those for U.S.

[1]Aircraft Routing sometimes also denotes the problem of routing aircraft through a restricted airspace, see e.g. [30]. That problem is not related to the type of aircraft routing which we discuss here.

airlines, try to create one single, repeating, route which all aircraft fly.[2] This has the benefit that all aircraft are subject to the same weather conditions, and that all aircraft visit all airports. The drawback is that constraints on individual aircraft are most often ignored with such an approach, making it difficult to use when many individual constraints should be satisfied. The central issues in most studies of aircraft routing are maintenance constraints and through values. The maintenance constraints are most often rather simplistic, such as requiring a visit to a maintenance station every X days, and are typically homogeneous over the aircraft in each fleet. Most approaches to aircraft routing do not consider individual restriction constraints. The focus on maintenance and through values has coined the terms *Maintenance Routing* and *Through Assignment* to denote the problems where maintenance and through values are considered separately.

As we have already discussed in Chapter 1, we define the *Tail Assignment problem* as an aircraft assignment problem which handles *all* operational constraints, such as maintenance and restriction constraints. The constraints are individual for each aircraft, and for each aircraft the individual requirements, such as preassigned maintenance activities, individual restriction constraints, and possibly even temporary constraints, are modeled. Also, in tail assignment we allow several aircraft types to be planned simultaneously. The problem is solved for a fixed time period, normally a few weeks, and requires historical data about e.g. previous maintenance checks. The idea is that the solution to the tail assignment problem should be possible to implement directly, without further manual intervention. Instead of focusing on a specific problem, such as maximizing through values, the tail assignment problem is flexible enough to model many different kinds of optimization criteria, for example through values, but also criteria related to operational stability.

The tail assignment problem does not create one single repeating route, simply because it does not make sense when solving the problem for a fixed time period. It can therefore be difficult to directly compare our tail assignment approach to aircraft routing and aircraft rotation approaches. On the other hand, the tail assignment problem will consider all constraints to create a truly feasible solution.

One of the earliest studies of aircraft routing is due to Daskin and Panayotopoulos [65], who solve the aircraft routing problems in hub-and-spoke networks. More recent approaches to aircraft routing are due to Kabbani and Patty [130], Clarke et al. [53], Cordeau et al. [57], Barnhart et al. [26], Desaulniers et al. [70] and Elf et al. [79]. The maintenance routing problem is covered by Sarac et al. [196] and Gopalan and Talluri [104], while Ahuja et al. [5] look at a combined through and fleet assignment problem. Feo and

[2]Observe that this can be seen as not satisfying our definition of the aircraft assignment problem as deciding which *individual* aircraft should operate which flight. However, this should not create too much confusion, since even in this case it is decided which flight to flight connections the aircraft should use.

Bard [89] did an early study of scheduling combined with maintenance base planning. A more detailed literature survey of this area, as well as a more detailed description of the tail assignment problem, is given in Chapter 3.

2.6 Recovery Planning

Once the long-term planning is completed, the planned schedules are *published* to the crew. From this point and until the day of operation, the planned crew and fleet schedules must be maintained to account for late changes. For example, demand might change dramatically for a certain flight, making it necessary to re-optimize the fleet assignments. Sick crew members might make it necessary to re-optimize the crew pairings and/or crew rosters, and faulty aircraft might trigger changed tail assignments. In case these disruptions occur long before the day of operation, they can most often be solved by slightly modified variants of the long-term planning processes. This phase is sometimes called the *tracking* or *maintenance* phase. However, when the changes occur very close to the day of operations, a completely different solution approach is often necessary. This is called the *recovery* phase. With flight departure only hours away, it is not possible to launch optimizing planning software which might take several hours to find a solution. Also, the interdependence between crew, aircraft and passengers becomes a very important factor this close to take-off. A recovery solution that is very good from a crew perspective might be unacceptable for the passengers, as many of them will miss their connecting flight. Or an aircraft might miss a crucial maintenance stop, effectively grounding the aircraft for a long period of time.

Recovery planning models must therefore be designed to interact much more to quickly find recovery alternatives which are good for crew, aircraft and passengers. Since the day of operation situation is very volatile, it is also good to provide an operations controller with several different alternative recovery solutions, to make it possible to manually select a solution to implement. Recovery planning is not very extensively studied in the literature, probably due to its complexity, and the fact that it is difficult to get hold of data and validate solution approaches. A good starting point to understand the complexity of an operations control center can be found in Rakshit et al. [180], where a decision support system used at United Airlines is presented.

Early studies of airline recovery problems are e.g. Teodorovic and Guberinic [213] and Jarrah et al. [127]. Cao and Kanafani [43] show how runway time slots influences the flight recovery. Some approaches, such as [3, 153, 169, 205, 229], focus only on the problem of rescheduling crew. But the most recent trend is definitely to incorporate all resource areas (crew, aircraft and passengers) in the same model. The most complete, but mainly theoretically interesting, such approach is Lettovsky [152], which attempts to model the entire recovery problem and solve it with Bender's decomposition. Kohl et al.

[144] present a recent approach to recovery planning which has been developed in cooperation with British Airways' operations control center, as part of the EU project *Descartes*. In this approach, aircraft, crew and passengers are considered by dedicated sub-models, and coordinated by an integration layer. The system can quickly produce many solution alternatives to various disruption scenarios, and has been shown to produce better or equally good solutions as operations controllers at BA. Clarke and Laporte [54] integrate some crew and passenger considerations into an aircraft re-routing model. Finally, the Ph.D. thesis of Rosenberger [187] contains a comprehensive list of references to airline recovery literature.

2.7 Integrated Planning

As described above, the planning steps shown in Figure 2.1 are performed sequentially, without any feedback. Traditionally, this picture is accurate. For administrative as well as computational reasons, the planning steps are normally performed in sequence, possibly with some very limited feedback. Today, more and more effort is being put into integrating the different planning procedures. In Section 2.3, we discussed how fleet assignment could be integrated with timetabling, aircraft routing, and crew planning. Integration can be done in several ways. Either by fully integrating steps, e.g. doing simultaneous fleet assignment and crew planning, by partially integrating steps, e.g. making sure there is space for maintenance activities in the fleet assignment step, or by iterating between the steps.

The first option, to plan several resources at once, is becoming more and more popular as computing power increases. Ioachim et al. [123] combine fleet assignment and routing, and also include re-timing considerations. The solution approach is Dantzig-Wolfe decomposition embedded in a branch-and-bound search to find integer solutions. Desaulniers et al. [70] solve a daily combined aircraft routing and scheduling problem using Dantzig-Wolfe decomposition. Cordeau et al. [57] use Bender's decomposition to simultaneously solve the aircraft routing and crew planning problems. They are able to show significant improvements compared to sequential planning on instances with up to 500 flight legs over 3 days. Mercier et al. [170] generalize the implementation of Cordeau et al. by adding constraints which increase robustness. Mercier at al. also use Bender's decomposition, but switch the order compared to Cordeau et al., using the crew pairing problem as the master problem, instead of the aircraft routing problem. They further describe procedures for adding strong Bender's cuts to improve convergence, and show that the improved model and solution approach significantly decrease computation times compared to the more basic approach. Ahuja et al. [5] solve the combined fleet and through assignment problem using very large neighborhood search, as the model becomes far too large for exact IP approaches. Substantial improvements compared to solving fleet and through assignment separately is reported.

Yan and Tseng [228] integrate flight scheduling and fleet routing for a 24-aircraft fleet of a Taiwanese airline, using Lagrangian relaxation combined with network flow algorithms. The aircraft routing model of Barnhart et al. [26] can be used for combined fleet assignment and aircraft routing. Kliewer [139] presents a model for integrating fleet assignment and demand forecasting, but presents few details. Finally, a recent paper by Sandhu and Klabjan [195] considers fleeting, routing and crew planning in the same model. Pairings are modeled explicitly, while routing issues are handled through so-called plane count constraints [138], which essentially means that the model ignores the maintenance aspect of aircraft routing. Two solution approaches are presented, based on combined Lagrangian relaxation and column generation and Bender's decomposition, respectively. Computational results are presented for a major U.S. carrier.

Partial integration of planning steps is fairly common, since it often requires little effort to include some extra constraints in a mathematical model. The most common example is the consideration of aircraft maintenance and crew overnights in fleet assignment, as was first described by Subramanian et al. [207] and Clarke et al. [52], and has since been used by e.g. Rushmeier and and Kontogiorgis [194]. When planning the fleetings, one simply tries to make sure that a certain number of opportunities for maintenance exist. A *lonely crew overnight* occurs when a crew arrives late at night at a non-base airport, and the aircraft it arrived on departs too early the next morning, leaving no other departing flight until the following morning. This incurs a pairing cost penalty, and should thus be avoided. By putting a penalty cost on lonely overnight connections, the fleet assignment model can balance the extra cost versus the fleeting cost, to decide which is the best alternative. Klabjan et al. [138] propose a model adding so-called *plane count constraints* and flexible departure times to the crew pairing problem. The plane count constraints ensure that aircraft assignments, ignoring maintenance and other operational constraints, can be found, and the flexible departure times allow the model to move flights by small amounts to improve the crew pairing solution.

In a similar effort, but instead focusing on maintenance feasibility, Cohn and Barnhart [55] propose an extended crew pairing model for integrating maintenance routing decisions in crew planning. They include additional variables in their crew pairing model representing *complete solutions* to the maintenance routing problem, add show how only some critical such variables must be included to guarantee optimality of the integrated model. Also, the variables representing maintenance routing solutions do not need to have integrality constraints, which means that the integrated model only has as many integer variables as the original crew pairing model. Unfortunately, no computational results are provided.

In [6], Ahuja et al. extend their fleet and through assignment model [5] to handle costs related to flight crew and ground staff planning. The proposed solution method is large-scale neighborhood search. Higle and Johnson [115] present a flight scheduling approach which also considers maintenance. By combining flight scheduling and maintenance feasibility models, they obtain schedules which are 'maintenance feasible', i.e. which contain enough maintenance opportunities. They present results for a number of randomly generated instances with 15-90 aircraft, and use an aircraft routing model similar to that described by Barnhart et al. [26] to validate that there are enough maintenance opportunities. They finally propose a number of extensions to their model, such as random maintenance times and multiple maintenance types.

As a third option it is possible to iteratively solve the problems, giving feedback from a later step back to an earlier, and then re-solve. This type of integrated planning is attractive in practice since it can often be done using existing planning models, without much development work. Provided that the existing models can handle the necessary constraints and costs, most of the work is then put into communicating between them. An example of this kind of iterative approach has been developed at Carmen Systems using the tail assignment system described in this thesis, and Carmen Systems' crew pairing optimizer [116]. In the crew pairing step it is necessary to know about the aircraft routes, in order to correctly model rules and costs related to aircraft changes for the crew. By iteratively solving crew pairing and tail assignment, and in the tail assignment step try to get the aircraft to follow as many *tight connections* as possible, the number of aircraft changes for crew can be decreased dramatically. In Chapter 13 we will take a closer look at this integration.

However, there are not only computational reasons why integrated planning is not used all the time at airlines today. Another reason is that the planning steps are often assigned to different departments, with different focus and responsibility. Therefore, changing the airline planning process to become more integrated is not simply a matter of showing that it is possible from a computational point of view.

Three

Tail Assignment

The purpose of this chapter is to introduce our definition of the tail assignment problem, and present the types of constraints and objective functions which we will handle. We will *not* give a mathematical model of the problem in this chapter, but will instead thoroughly explain the real-world problem which we aim at solving, and a few crucial modeling decisions which are necessary for our definition of the tail assignment problem. Mathematical models and possible solution methods will be discussed in subsequent chapters.

Our tail assignment problem captures all aspects of aircraft assignment, from aircraft routing to maintenance routing and through assignment, and even touching fleet assignment and recovery planning. We define the tail assignment problem as the problem of creating routes for a set of individual aircraft, covering a set of flights in a timetable, such that various operational constraints are satisfied, while minimizing some cost function. Since the fleet assignment step partitions the flights by fleet type, the aircraft assignment problem is often solved for a single homogeneous fleet of aircraft. However, we choose to be more general in our definition of the tail assignment problem, by allowing aircraft from different fleets to be solved simultaneously. Since fleet assignment normally does not consider operational constraints, modeling multiple fleets is sometimes necessary to ensure feasibility of the tail assignment problem. In some cases, considering multiple fleets can also help reduce fleet assignment suboptimality, which can otherwise occur since operational constraints are not considered in fleet assignment.

The activities that we plan can be individual flight legs, manually defined sequences of flights, or maintenance and other ground activities. We will therefore refer to all these as *activities* rather than flights from now on.

To be able to handle all operational constraints, activities depending on exact dates, e.g. preassigned maintenance activities, must be considered. This means that we are only interested in solving *dated* problems, i.e. problems in a fixed time period. In contrast, solving cyclic problems, i.e. problems with a time horizon connecting back to the beginning, is common especially in U.S. fleet and crew planning approaches [53, 112, 207]. In such an approach a typical week (or day) is solved in such a way that each aircraft is assigned a cyclic route, connecting from the end back to the beginning. This cyclic plan is then rolled out and adjusted to cover an actual (dated) period of time. U.S. timetables are typically very regular over a week, which makes it easy to roll out a cyclic solution to a dated one. But since all individual constraints have not been considered, the rolled out solution might still not be possible to operate. Since fleet assignment models are most often solved as daily cyclic problems, the fact that tail assignment requires a dated approach can of course makes it difficult to use directly for traditional fleet assignment. The tail assignment problem also cannot create a single repeating route, as discussed in the previous chapter, because it does not make sense in a fixed time period.

The tail assignment problem does not put any limitation on the period of time that we solve for, but we have aimed at being able to solve about one month at a time. Some of the motivations for planning aircraft routes long in advance are:

- integration with crew planning requires it, since crew planning must be performed well in advance in order for the crew to get their working shifts early;
- to properly consider individual long-term maintenance constraints;
- to be able to satisfy some individual operational constraints it is sometimes *necessary* to consider long periods;
- to do re-fleeting with multiple aircraft types, as discussed in the previous chapter, and further in Section 3.6, it is necessary to consider longer periods.

The tail assignment problem can thus be used to describe the entire fleet planning problem, from fleet assignment to aircraft assignment close to departure. As far as we know, no aircraft assignment or fleet planning model as general as tail assignment has been proposed before.

In the rest of this chapter, we will first discuss the constraint types and objective functions which are possible in tail assignment. In the second half of the chapter, we will take a detailed look at the literature related to the various variants of aircraft assignment.

3.1 Connection Constraints

The most basic constraints are the *connection constraints*. These constraints specify whether it is possible to connect between two activities, i.e. operate them in sequence. The connection constraints can be very simple, by specifying only a minimum allowed time between an arriving activity and a departing activity. The minimum allowed time should account for the time it takes to clean, refuel and prepare the aircraft for the next departure, as well as change crew and passengers. Many of the aircraft assignment models proposed in the literature only consider this simple type of connection constraint.

At busy airports, the connections times must also account for aircraft tow times and transfer times between gates and terminals. Often, connection times must also be longer between international flights than between domestic ones. Similarly, in the European Union, different connection time constraints must sometimes be used for flights to and from countries who have signed the Schengen Convention, and countries who have not signed the Convention.[1] Connection time constraints obviously also differ between aircraft of different types, since the refueling and preparation times differ, and they can also differ depending on the time of day.

We have chosen to model the flight network with a so-called *connection network* [26, 70], where each individual activity-to-activity connection is an arc between two activity nodes. A connection which is present in the connection network is called *legal*, while connections not in the network are *illegal*. A possible (legal) route may only contain legal connections.

The use of a connection network gives us the possibility to forbid individual connections, so even if a connection seems possible, i.e. provides sufficient connection time, it might be deemed illegal due to other reasons. Normally, a legal connection exists between two activities if the second activity departs from the same airport as the first one arrives at, and there is sufficient turn time, typically about 30-45 minutes, between arrival and departure times. In a connection network it is also possible to put costs on individual connections, which is necessary for representing the cost function discussed in the beginning of this chapter. It is also trivial to introduce so-called *forced turns* [193]. Forced turns are connections that are locked for some external reason, e.g. to protect a known good through value. Since we solve a dated problem, the connection network is acyclic, as it is obviously impossible to connect backward in time.

Another very common flight network representation is the time-line network [26, 52, 53, 112, 154, 185, 207]. The time-line network has a time-line for each station, with nodes placed at arriving and departing flights, and the activities represented by vertical arcs between arrival and departure nodes. To account for a minimum connection time, the arrival time of a flight is adjusted to the *ready time* of the aircraft, i.e. the time when it has unloaded all passengers and is ready to board new passengers. Also, an arrival or departure node can be aggregated with an immediately following departure node, as described

[1]The Schengen Convention abolished the checks at borders between the signatory States, and created a single external frontier [84].

Flight	From	To	Departure	Arrival
SK100	CDG	ARN	08:00	09:30
SK101	CDG	ARN	09:00	10:30
SK102	ARN	LHR	11:00	12:00
SK103	ARN	LHR	12:30	13:30
SK104	FRA	ARN	14:00	15:15
SK105	ARN	CPH	15:45	16:25

Table 3.1 Example timetable for airport ARN. (ARN = Stockholm Arlanda, CDG = Paris Roissy Charles de Gaulle, CPH = Copenhagen, FRA = Frankfurt, LHR = London Heathrow.)

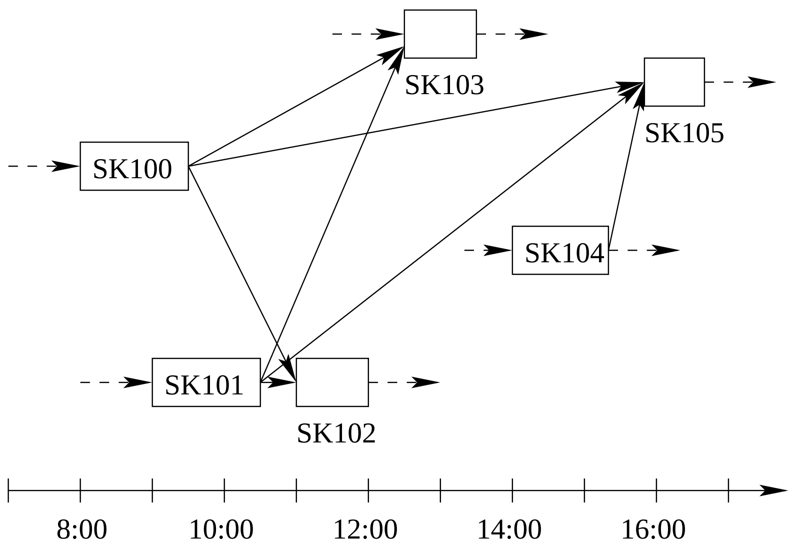


Figure 3.1 A connection network.

in [112] and [154], to reduce the network size. The time-line representation is more compact than the connection network, but on the other hand does not distinguish between individual connections, and is thus a relaxation of the connection network. For example, it is not uncommon that the minimum connection time between domestic flights is shorter than the minimum connection time between a domestic and an international flight. This cannot be modeled correctly in a time-line network.

Table 3.1 shows a tiny example timetable at airport ARN, and Figures 3.1 and 3.2 shows the resulting connection and time-line networks, assuming a 30 minutes minimum connection time. The dotted arcs in Figure 3.1 represent connections to flights which are not present in the example. In Figure 3.2, the dotted arcs arc ground arcs. The two morning arrivals from CDG have

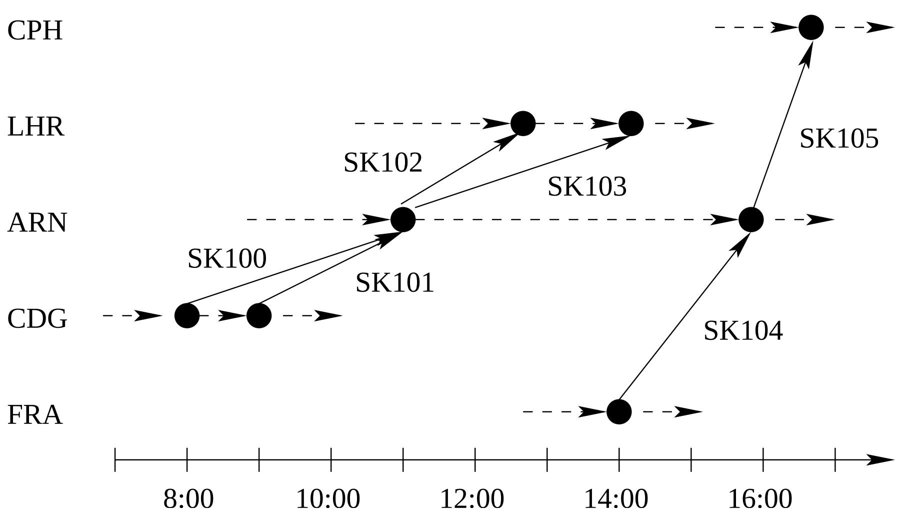


Figure 3.2 A time-line network.

been aggregated with the two departures to LHR. The nodes at LHR and CDG have not been aggregate simply because not all arrivals and departures at these airports are shown in the figure. Observe that even though we stated above that the time-line network is more compact than the connection network, in this tiny example the connection network actually seems more compact.

We have chosen to use a single connection network for all aircraft in the problem, rather than constructing a specific network for each aircraft, or aircraft type. But since constraints and costs can differ between the aircraft in the problem, we need to store extra information in the network. For each connection and activity we store information about costs and legality for individual aircraft. The reason for not using individual networks is the memory consumption – each individual network can be quite large, and since the main part is common for all aircraft, storing specific information on one single network is more memory efficient.

3.2 Maintenance Constraints

The most important of the operational constraints are the maintenance constraints. Maintenance regulations are often decided by national agencies, like the Federal Aviation Administration (FAA) in the U.S., or Luftfartsverket in Sweden. Aircraft manufacturers such as Boeing and Airbus also provide *maintenance programs*, such as ATA's[2] MSG and EMSG programs [10], which decide which types of maintenance should be performed. Airlines often use

[2]ATA is the Air Transport Association (of America), which currently has 25 member airlines [18].

their own constraints, which are stricter than the national regulations, to make solutions more robust in case of disruptions.

For our purposes, the maintenance constraints can be divided into two main categories – preassigned maintenance stops, and minor maintenance. The preassigned maintenance stops are maintenance activities that are fixed to certain aircraft, as a result of long-term maintenance planning. This type of planning must be performed with a very long-term time horizon, since the maintenance activities are performed very seldom, and take a lot of time. The resulting preassigned activities are thus quite long, and simply require that a certain aircraft is on ground at a particular airport (or *maintenance station*) during a particular period in time.

The minor maintenance checks are normally performed during nights, with quite short intervals. These checks are not initially fixed in time like the preassigned activities. They can be of different kinds, depending on the way in which the maximum interval between consecutive checks is defined. Normally, they are defined as a maximum number of flying hours, or a maximum number of landings (often referred to as *cycles*).

But there can also be situations where they are defined as a maximum number of calendar days, regardless of how much of this time is spent flying. This is e.g. the case for very long-term maintenance types. Another example could be a kind of robustness constraint, where one wants each aircraft to return to the main base with regular intervals. Then if an aircraft is slightly defect in some way, it might be possible to just wait a few days until it gets back to base and repair it then, rather than reroute several aircraft over many days to get it back. In our approach, actual maintenance constraints and robustness-related constraints are handled in the same way. It is important to note that we might not have to explicitly consider all maintenance types when solving a problem. Some types of maintenance checks include other checks, and forcing each aircraft to return to base with regular intervals will often make it possible to perform certain types of maintenance regularly, without explicitly planning when to perform them. In fact, some approaches to maintenance routing [104, 210] focus only on making sure that the aircraft regularly return to base.

Table 3.2 shows some typical maintenance regulations for a certain type of Boeing aircraft, for an example airline. The aircraft must do the 100H check at least every 100 flight hours, the A check at least every 500 flight hours, the C check at least every 6000 flight hours *or* every 547 days, and so on. In this table the checks with longer intervals are preassigned checks as mentioned above, and the shorter ones, such as 100H, A, and 2A, are minor checks. These constraints might vary from airline to airline, depending on the implemented maintenance program, or even between different versions of the aircraft. Gopalan and Talluri [103] describe the maintenance regulations mandated by the FAA in the U.S., and descriptions of the MSG programs can be found in [10].

Name	Period	Type	Name	Period	Type
100H	100	Flight hours	3C	18000	Flight hours
A	500	Flight hours	3C	1642	Days
2A	1000	Flight hours	E4	9000	Cycles
6A	3000	Flight hours	E4	1642	Days
C	6000	Flight hours	E5	12000	Cycles
C	547	Days	E5	2190	Days
2C	12000	Flight hours	L1	1000	Flight hours
2C	1095	Days	L2	3000	Flight hours

Table 3.2 Maintenance constraints for a certain type of Boeing aircraft for an example airline. Observe that all checks are not included here, and the types and intervals might differ between airlines, and also depending on which maintenance program is implemented.

Of course, airport hangar capacity limits the number of aircraft that can simultaneously be maintained. We will model this by creating activities representing *maintenance possibilities*. For example, if only 5 maintenance hangars are available at an airport, we can create 5 maintenance possibility activities each night at this airport. The maintenance constraints have to be specified such a way that maintenance can only be performed using these activities. Obviously, this does not capture all maintenance possibilities, as we have to specify the start and end times of the maintenance beforehand, but often it is a good enough approximation in practice, since normally there exists a period each night when very few aircraft are active.

3.3 Flight Restriction Constraints

Another common type of individual aircraft constraints are so-called *curfews*, or destination restrictions. These constraints forbid certain aircraft from flying to certain destinations, possibly only at certain times or dates. There can be several reasons for the restrictions, for example that the aircraft in question is too noisy to land at the destination, or that its reverse thrusters are defective, so it cannot land on short landing strips.

There can also be restrictions related to the flights, e.g. that an aircraft has too few fuel tanks to fly more than a certain number of hours, or that it lacks the in-flight entertainment system required for long flights. Since both restrictions with respect to destinations and with respect to flights can be modeled by forbidding certain aircraft from operating certain flights, we will refer to them all as flight restriction constraints. Observe that flight restrictions are aircraft specific. While some curfews depend only on the fleet type, many other restrictions vary for aircraft within the same fleet.

3.4 Other Constraints

While connection time, maintenance and flight restriction constraints are the most common constraints, there can also be other constraints. For example constraints ensuring that aircraft routes do not make it impossible to find solutions to the crew planning problems. An aircraft can turn, i.e. arrive and depart again from an airport, in less time than it takes to switch crews. Therefore, if an aircraft route consists of more than, say, eight hours of consecutive short connections, it will be impossible to find a solution for the crew, as a single crew member cannot fly eight consecutive hours without rest. Including constraints of this type in tail assignment is an example of integrated planning of the second type, as discussed in Section 2.7.

It is often also desirable that the aircraft get about equal usage. The reason is that it is good if all the aircraft in a fleet age at the same rate in terms of flying time. A constraint ensuring that all aircraft get about the same amount of flying time, or landings, in a solution can therefore sometimes be present. If the aircraft are not of the same age, ‘equal usage’ can be replaced by ‘target usage’, indicating that each aircraft should meet a target usage appropriate for its age.

Sometimes, it can be desirable to limit the number of landings at a certain airport, e.g. to limit the the number of visits to delay-critical airports per day [79]. Like the maintenance, consecutive short connections and equal usage constraints, this can be modeled by a general *cumulative constraint*.[3] A cumulative constraint is a constraint which cumulates some resource, e.g. flying time, calendar days or landings at some airports, and has an upper limit on how much of the resource can be consumed. In mathematical programming terminology, this type of constraint is called a *resource constraint*. We will allow any cumulative constraint, not only maintenance constraints, to be present. Even soft cumulative constraints, i.e. constraints for which there is a cost penalty for consuming more than the limit, are allowed.

One type of constraints which our model does not consider is general ‘global constraints’, i.e. general constraints involving more than one aircraft route.[4] It does allow some specific global constraints to be modeled, such as maintenance capacity and equal aircraft utilization. One global constraint which could become interesting in the future, since e.g. the European Union have started to demand that airlines increase their environmental responsibilities, is a constraint limiting the total amount of exhaust fumes produced. Like the equal aircraft utilization, this constraint can also be approximated

[3]Not to be confused with the `cumulative` constraint in constraint programming [4].

[4]Not to be confused with the concept of global constraints in constraint programming [34, 171].

by cumulative constraints. However, more general constraints cannot easily be modeled. Fortunately, such constraints are not very common in aircraft planning, unlike the situation for crew planning.

3.5 Optimization Criteria

The optimization criterion in tail assignment is often related to the robustness or quality of the solution, rather than real monetary costs. As an example, a typical objective function (*cost function*) rewards short and very long connections between successive activities, but penalizes medium length connections. The reason is that a medium length connection, i.e. a connection lasting between, say, two and three hours, causes an aircraft to be unavailable during a period of time. The aircraft cannot be used for anything while it is waiting, and perhaps it even has to wait at the gate, incurring a cost. On the other hand, if the connection is longer it is possible to use the aircraft as *standby*, i.e. letting it perform extra activities in the event of disruptions. Connections longer than some limit, for example two hours, are sometimes called *standby connections*. The standby time limit might vary from airport to airport, and also between different times of the day. Aircraft *availability* can loosely be defined as the proportion of non-flying time spent on standby connections.

An example of a cost function is depicted in Figure 3.3, which plots the connection time versus cost. The minimum connection time is denoted t_{min}, so there are in fact no connections with connection time less than t_{min}. Connections close to the minimum connection time are penalized, up to an optimal connection time denoted t_{start}. Connections between t_{start} and t_{max} in duration are penalized with a linearly increasing penalty, and connections longer than t_{max} are penalized with a linearly decreasing penalty.

An alternative objective is to maximize the through values, i.e. the desirability of one-stop service between a pair of cities [26, 53]. For the interaction with crew pairing discussed in Section 2.7, a cost function rewarding the use of tight crew connections is used. Finally, in case the tail assignment problem is re-solved close to the day of operation, it might be desirable to obtain a new solution which differs as little as possible from the currently published solution, but which e.g. satisfies additional constraints. This can easily be modeled by adding penalty costs for all connections which are not in the current solution.

Since cost functions can differ between different applications and airlines, we will allow different kinds of cost functions. However, the solution approaches we will use limit the costs to be defined on connection (or activity) level. Unlike the situation in crew planning, this is not a major problem in fleet planning, as most costs are related to single connections or activities anyway. However, using soft cumulative constraints it is possible to model some cost components on sequences of activities. In fact, most of the extensions in terms of cost functions to the fleet and aircraft assignment problems discussed

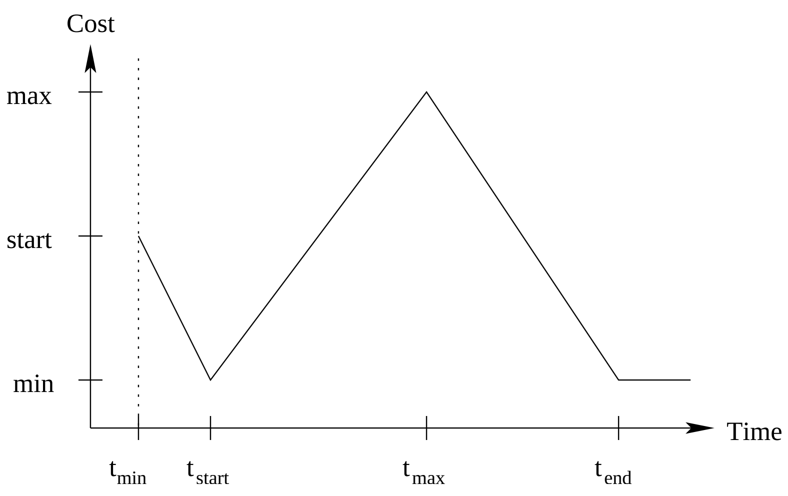


Figure 3.3 An example cost function.

in Chapter 2 can be captured by our model. In Chapter 13 we will look at a few special cost functions in more detail.

3.6 Multiple Fleets

As mentioned above, the tail assignment problem is traditionally solved for one homogeneous fleet at a time. However, there are several reasons why it might in fact be a good idea to mix fleets in the tail assignment step.

One reason is that normally, fleet assignment does not guarantee that the resulting fleeting is feasible for tail assignment, as it does not consider individual operational constraints. It might therefore be necessary to undo some of the fleet assignment decisions in order to find feasible tail assignments. The airline planning departments should have experience enough not to produce such solutions, and the fleet assignment solvers take more and more operational constraints into consideration, see e.g. [26, 52], but this problem still cannot be ignored in practice.

Further, it is possible that the estimates about passenger demand, that the fleet assignment solution is based on, changes as the departure day approaches. It might therefore be desirable to change fleets for some flights, to capture more passengers or downgrade the aircraft size, and thus the operating cost. This process is known as *re-fleeting*, and has been studied by some airlines. In [36], Berge and Hopperstad introduce D^3, Demand Driven Dispatch, which is in use at American Airlines, and in [125] Jarrah et al. discuss an algorithm for re-fleeting at United Airlines. Another reason for mixing fleets could be that a set of fleets have similar cockpit configurations, making them identical in terms of crew requirements, but different in terms of passenger capacity. This is for example the case for many versions of Airbus aircraft.

In recovery situations, it is often *necessary* to mix fleet types to be able to avoid delaying or canceling flights.

Our tail assignment optimizer can easily handle multiple fleets, and can at least potentially be used even for full re-fleeting purposes. In order to capture a cost function depending on the fleet type, each connection has not only a single cost, but rather one cost for each tail. To facilitate re-fleeting it is also possible to specify a tail-specific cost of covering an activity. Since tail assignment is capable of modeling constraints and costs for individual aircraft, it can even be said to handle *re-tailing*. When solving multi-fleet instances, one can specify whether to allow an aircraft to be assigned activities 'belonging to' another fleet. It is also possible to use the individual flight restrictions discussed above to specify even more targeted restrictions.

In theory, tail assignment can solve the standard fleet assignment problem, as described e.g. in [112, 207], with the exception that tail assignment requires a fixed time period. In fact, since it models individual aircraft rather than fleets, tail assignment would be even more accurate than standard fleet assignment. In practice, the expressiveness of our model makes fleet assignment difficult, as the sizes of typical fleet assignment instances are larger than tail assignment instances in terms of number of aircraft, and thus lead to extreme memory requirements. Nowadays, fleet assignment solvers focus more and more on considering network effects on passengers, so-called itinerary-based fleet assignment (IFAS) [29]. The solution methods we will present cannot directly be used for IFAS, but could likely be extended to handle it. Of course, this would increase the memory consumption even more.

Some of the test instances used throughout this thesis are multi-fleet instances. For these instances, we do not do re-fleeting with a fleet assignment cost function, but only use the ability to mix fleets of similar types. But as explained above, full re-fleeting is possible with our model.

3.7 Related Problems and Literature Review

As we have already discussed in Section 2.5, many problems related to tail assignment have been studied before, under different names. Here, we will provide a detailed overview of these related problem formulations and the related literature.

Through Assignment

Gopalan and Talluri [103] differentiate between the through assignment and maintenance routing aspects of aircraft assignment. As described in Chapter 2, a through flight is a flight between two airports, via the hub, that does not require an aircraft change. On certain itineraries, passengers are prepared to pay extra for the convenience of not having to change aircraft. This is especially true in hub-and-spoke networks which are common in the U.S.[5] Through assignment is the problem of deciding which flight-to-flight connections are to be through flights, and thus flown by the same aircraft. Gopalan and Talluri [103] describe the considerations necessary to decide how much a through flight is worth for a particular market.

Bard and Cunningham [23] present a heuristic solution approach to the through assignment problem, and present results from American Airlines' Dallas/Fort Worth hub. They improve current (in 1987) practice by considering through values to depend on time, and claim revenue benefits of up to 60% compared to current procedures. Gopalan and Talluri [103] subdivide each day into time slices and markets, and formulate a straight-forward set packing model of the through assignment problem. Jarrah and Strehler [126] presents an approach used at United Airlines, increasing revenues by $10 million annually. The problem formulation is a network flow problem with side constraints, and it is solved using CPLEX's [121] hybrid network flow solver combined with branch-and-bound.

Ahuja at el [5] solve the combined fleet and through assignment problem using very large-scale neighborhood (VLSN) search. First a close to optimal solution to the fleet assignment problem is found, and then a through assignment solution is found. Finally, the solutions are improved using VLSN search, using so-called 'A-B swaps' [209], swapping fleet types for a sequence of flight legs. Computational results are presented for a large U.S. airline, showing substantial improvements over solutions obtained using the traditional sequential planning process. In [7] the model in extended to allow varying departure times (time windows) for flight legs, and [6] further extends it to include consideration for costs due to flight crews and ground staff.

Maintenance Routing

When through assignment and maintenance routing are considered separately, the maintenance routing part becomes a feasibility, rather than optimization, problem. Since maintenance most often takes place during nights, flight activities during days are often considered only as *lines of flying* (LOFs) [103, 104] in maintenance routing. Ignoring complicating flight restrictions, lines of flying can easily be constructed with *first-in first-out* (FIFO) or *last-in first-out* (LIFO) techniques.

Gopalan and Talluri [104] describe a system for maintenance routing implemented as USAir, where only the maintenance aspect of the problem is taken into account, and the problem is to find a solution, regardless of its cost. The problem is also simplified to require the aircraft to return to a maintenance base every three days, and assumes that the LOFs are fixed. It is shown that this problem can be solved in polynomial time, by introducing

[5]In fact, Elf et al. [79] state that through flights are not important for short-haul European instances.

artificial arcs in an activity network and finding an Euler tour in the resulting graph. The drawback is of course that the model is simplified both by assuming that the LOFs are fixed, and by treating only a single maintenance-like constraint. Talluri [210] has also shown that the three-day limit is a special case, and that finding a four-day maintenance routing is in fact an NP-hard problem.

Sriram and Haghani [204] present a model for maintenance scheduling and aircraft re-assignment. They assume that the maintenance scheduling is performed after aircraft have been assigned to routes, and must therefore introduce the possibility to re-assign aircraft to ensure maintenance feasibility. Only the FAA mandated A and B checks are modeled. They use heuristic depth-first search and random search methods to solve the resulting integer program, and show that this approach produces results which compare well to optimal results obtained with branch-and-bound. The tested instances are all randomly generated.

Combined Approaches

Most published approaches to aircraft assignment combine the maintenance routing aspect with a cost function, which is often through value based, but can also capture other aspects.

One of the very first proposals for using integer programming techniques for fleet scheduling and routing is due to Levin [154]. He discusses network flow models for various problems, such as a minimum fleet routing problem, and also proposes a model for routing with variable departure times. He also mentions Dantzig-Wolfe decomposition as a possible solution technique for this problem.

One of the first modern approaches to aircraft assignment is due to Daskin and Panayotopoulos [65]. They study a version of the aircraft assignment problem which is really a mix of fleet and aircraft assignment, in that it considers profits associated to individual aircraft assignments. Only hub-and-spoke networks with a single hub are considered, making the model quite simple in terms of connecting possibilities. Daskin and Panayotopoulos use a Lagrangian relaxation to obtain upper bounds, and combine it with heuristic methods to obtain feasible solutions. Computational results on partially simulated data indicates that good solutions can be obtained when the number of aircraft needed is less than the number of available aircraft, while the case were some flight legs must be left unassigned is more difficult.

Another early contribution is due to Feo and Bard [89]. They discuss a model for combined flight scheduling and maintenance base planning. They propose two models for the min-cost multi-commodity network flow problem – one arc-based, and one path-based. They conclude that the arc-based model is too difficult to solve, and proceed to discuss techniques for solving the path-based model, which was the model used by American Airlines in 1987.

Since they consider the problem of generating columns as being too difficult, they resort to heuristic methods to solve the model. A two-phase heuristic based on generating maintenance-infeasible routes in the first step, and using Chvátal's set covering heuristic [50] in the second phase, is described. Results are presented for American Airlines' B727 fleet of about 150 aircraft. It is a bit difficult to compare this approach to others, since maintenance bases are not fixed. However, the model only handles maintenance constraints of specific types.

Kabbani and Patty [130] model the aircraft routing problem for American Airlines as a set partitioning problem, where each column represents a week-long aircraft route. By separately generating maintenance-feasible routes and selecting which to use, it is possible to implement general maintenance (and other) constraints. The drawback is longer solution times.

In [53], Clarke et al. solve an aircraft rotation problem for Delta Air Lines, where the aim is to provide maintenance feasible routes for aircraft while maximizing through values. They require all aircraft to fly the same cyclic route (rotation), and the problem is hence formulated as a TSP with side constraints. The selected solution method is Lagrangian relaxation, using a subgradient approach where subtour elimination constraints are added dynamically. Results are presented for a problem considering 3 and 4 day maintenance checks, but it is not clear how large the instances are. One drawback with this method is that since individual aircraft are not considered, and a cyclic rather than dated problem is solved, the obtained aircraft rotations might not be possible to operate in reality. In [70], Desaulniers et al. solve a daily aircraft routing and scheduling problem (DARSP). The problem is a daily combined fleet assignment and aircraft routing problem, maximizing anticipated profit, with a scheduling aspect (departure times can vary) built in as well. The authors present two models for it, one set partitioning model and one model based on multi-commodity network flow. They demonstrate how to combine the two approaches to obtain optimal branching strategies compatible with column generation, and show results on data instances from a North American and a European carrier. The instances contain up to 383 flights and 91 aircraft.

In [26], Barnhart et al. solve a combined fleet assignment and aircraft routing problem by an approach based on maintenance feasible *strings* of activities, that are combined to create feasible routes, within a branch-and-price framework. The aim is to minimize operating costs while satisfying maintenance constraints. They show that their model can solve both combined fleet assignment and aircraft routing problems, and pure aircraft routing problems, and that maintenance constraints and equal usage-constraints can be handled. They also extend the formulation to cope with one single rotation for all aircraft. Results are presented for short-haul instances with up to 190 flights.

Cordeau et al. [57] use Bender's decomposition to simultaneously solve the aircraft routing and crew planning problems such that the combined fleet and crew costs are minimized. The master problem is the aircraft routing problem, and crew planning is handled as a subproblem. Both master and subproblems are solved with column generation and heuristic branching strategies. They are able to show significant improvements compared to sequential planning on problems with up to 500 flight legs over 3 days.

Sarac et al. [196] points out that many proposed approaches to solve the aircraft routing problem only approximate crucial maintenance constraints, and thus are of limited use in practice. To remedy this, they model and solve the *operational aircraft maintenance routing problem*, which models maintenance slots and available man-hours at maintenance stations properly. The planning horizon is one day, although it is argued that a longer time horizon might be useful. They use a column generation solution approach, and detail the steps necessary to find an initial solution as well as good integer solutions. Results are presented for instances which are a mix of real-world, published, and randomly generated airline data.

Elf et al. [79] propose an aircraft rotation planning model for minimizing delay risk. In their model, a 'delay risk' is either individual connections being too short, or consecutive visits to certain airports. Some airports are defined as *air traffic critical* due to their tendency to often cause delays, and routes which do not repeatedly visit such airports are preferred, to avoid propagation of delays. Very limited motivation is provided for this delay model, and instead an 'industrial partner' is said to have defined it. The authors carefully describe why through assignment is typically not applicable for European flight networks, and why they do not consider maintenance in their model. A solution method based on Lagrangian relaxation is proposed, and results are presented for seven fleets which show less delay risk than actually flown routes.

3.8 Rolling Stock Assignment

While this thesis is focused on airline planning and tail assignment, problems similar to tail assignment can be found in other areas as well. One of the obviously similar problems is that of assigning locomotives, and other types of rolling stock, to trains. This problem has many similarities to tail assignment, but also surprisingly many differences. Maintenance regulations and restrictions on activity and vehicle combinations exist also for locomotive assignment, but there are some crucial differences. Sometimes, several locomotives are required to cover a single train 'leg'. This can be either because extra engine power is needed to climb steep hills, or because the locomotive also transports passengers, as is the case with many high-speed trains, and a certain passenger capacity must be met. Further, extra locomotives are sometimes assigned to a leg only for re-location purposes.

Also, the locomotive problem is much more focused on the possible ways to combine incoming and outgoing legs at a station than the tail assignment problem. While at an airport most arriving and departing flight combinations are legal, provided a certain minimum connection time, the one-dimensional nature of a railway track, combined with the fact that several locomotives can be assigned to activities, puts many constraints on the possible combinations. For a two-locomotive train, the front locomotive can e.g. depart before the second locomotive in one direction, but not in the other direction. For stations with a single-direction track (a *reverse* station) care must be taken so that no locomotive is 'locked in' by another train. To fully model these things, so-called *coupling* and *decoupling* of locomotives must be considered, to re-position them at the station. Altogether, these constraints shift the focus of the problem from routes to combinations of connections. As a consequence, the tail and locomotive are essentially the same problem, but differ at a few crucial points. Sometimes, however, it is still possible to solve special instances of locomotive assignment problems using the tail assignment model we describe here. Cordeau et al. [58] give a recent survey of railway optimization problems, including the locomotive assignment problem.

Part II

A Mathematical Programming Approach

FOUR

Mathematical Models and Solution Methods

In this chapter we will present some mathematical models for the tail assignment problem as defined in Chapter 3, and discuss various solution methods for them. Our main solution method, column generation, will be thoroughly covered in Chapters 5, 6 and 7. We will start by describing two mathematical models. The first model is perhaps the most intuitive one, and the second one is more suitable in practice.

4.1 Model TAS

The objective in tail assignment is to construct routes for a set of aircraft, covering all activities, respecting all maintenance and other regulations, while minimizing some cost function. Let T be the set of aircraft (tails) to plan, and F the set of activities. Where necessary for algorithmic purposes, F will also be assumed to contain a special *sink* activity, which is the last activity of all routes. Let our binary decision variables x_{ijt} be 1 if activity j should follow immediately after activity i and these activities should be operated by aircraft t, and 0 otherwise. The decision variables thus represent connections between activities, rather than the activities themselves. Correspondingly, c_{ijt} is the cost of assigning activities i and j in sequence to aircraft t. Illegal connections are represented either by not including their corresponding variables at all, or by setting their costs to very high values. Our first mathematical model, TAS, is shown in Model 4.1.

The objective is to minimize the total cost of the connections used. Constraint (4.2) ensures that the aircraft that connects to an activity also connects from it, and constraint (4.3) makes sure that each activity is covered exactly once. Observe that we assume that the sink activity is connected back to the carry-in activities, to ensure flow balance. Constraints (4.4) and (4.5)

Model 4.1 Tas

$$\min \sum_{i\in F}\sum_{j\in F}\sum_{t\in T} c_{ijt}x_{ijt}, \tag{4.1}$$

$$\text{s.t.} \quad \sum_{j\in F} x_{jit} - \sum_{j\in F} x_{ijt} = 0, \quad \forall t \in T, \forall i \in F, \tag{4.2}$$

$$\sum_{t\in T}\sum_{j\in F} x_{ijt} = 1, \quad \forall i \in F, \tag{4.3}$$

$$\sum_{j\in F} x_{ijt} = 1, \quad \forall i \in P_t, \forall t \in T, \tag{4.4}$$

$$\sum_{j\in F} x_{ijt} = 0, \quad \forall i \in R_t, \forall t \in T, \tag{4.5}$$

$$r_{im} \leq l_m, \quad \forall i \in F, \forall m \in M, \tag{4.6}$$

$$x_{ijt} \in \{0,1\}, \quad \forall i,j \in F, \forall t \in T. \tag{4.7}$$

ensure that preassigned and restricted activities are respected. P_t is the set of activities which are preassigned to aircraft t, and R_t is the set of activities which aircraft t is not allowed to operate. Since the decision variables represent connections rather than activities, these constraints must be represented by summations over all connections to activities, rather than equality or inequality constraints for single variables. Since for preassigned activities, all aircraft except the one assigned to the activity will be restricted from being assigned, constraint (4.4) is really redundant. We have kept it in the model anyway, for clarity. Observe that the possibility to leave activities unassigned with a penalty cost can be captured by adding connections x_{iit}, and setting the corresponding cost c_{iit} to a high value.

The most complicated constraint, (4.6), ensures that each aircraft gets enough maintenance. The constraint is a so-called *resource constraint*. A resource constraint limits the consumption of some resource along a route to be within certain bounds. The resource can be virtually anything, such as the number of landings at a specific airport or the number of flying hours performed. For our maintenance constraints, the resource consumption is the number of flying hours, calendar days, or landings since the previous maintenance, or maintenance possibility. For each activity i, r_{im} specifies the resource consumption for maintenance type m from the start of the route until activity i, and l_m specifies the maximum interval between consecutive maintenance checks of type m. M is the set of maintenance types.

For notational purposes, let i' denote the unique activity preceding activity

i, i.e. such that $x_{i'it} = 1$ for some t. Now, r_{im} is defined recursively as

$$r_{im} = \begin{cases} s_{im} & \text{if maintenance of type } m \text{ can be} \\ & \text{performed between activities } i' \text{ and } i, \\ r_m^t & \text{if } i \text{ is a carry-in activity for aircraft } t, \\ r_{i'm} + s_{im} & \text{otherwise.} \end{cases} \tag{4.8}$$

s_{im} is the *resource consumption* of maintenance type m for activity i. r_m^t is the initial resource consumption for maintenance type m and aircraft t, specifying the consumption at the start of the planning period. Observe that the variables r_{im} are defined in terms of the x_{ijt} variables, via the definition of $i^{'}$, and are undefined unless these are fixed.

In Chapter 3, we stated that general cumulative constraints, and not only maintenance constraints, can be present in the tail assignment problem. If such constraints are present, they are also modeled as resource constraints, with the consumption, reset and initial values properly defined. In Chapter 3 we also mentioned the possibility to create maintenance possibility activities to model hangar capacity at airports. If such activities are present, the corresponding resource constraint is modeled to only be reset at the maintenance activities. Normally, the resource consumption is reset whenever maintenance *can* be performed. This means that model Tas normally does not place the actual maintenance checks, but just ensures that there are enough *possibilities* for checks. We will get back to this point in Section 5.3.

4.2 The Multi-Commodity Network Flow Problem

Without the resource constraints (4.6), Tas is a pure *integer multi-commodity network flow problem* [8, Chapter 17], where each aircraft represents one commodity. Since all arc capacities and demands are 1, this problem is sometimes also called the *arc disjoint path problem* [199, Chapter 70]. To transform the activity restrictions (4.4) and (4.5) into arc capacities, standard *node splitting* [8, Chapter 2] is used.

In its most general form, the (continuous and linear[1]) multi-commodity network flow problem for a network with arcs E, nodes V and commodities

[1]Throughout this thesis, we will only discuss linear flow problems, and most often the term *linear* will be skipped.

K is:

$$\min \quad \sum_{(i,j)\in E} \sum_{k\in K} c_{ij}^k x_{ij}^k, \tag{4.9}$$

$$\text{s.t.} \quad \sum_{k\in K} x_{ij}^k \leq u_{ij}, \quad \forall (i,j)\in E, \tag{4.10}$$

$$\sum_{j\in V} x_{ji}^k - \sum_{j\in V} x_{ij}^k = b_i^k, \quad \forall i \in V, \forall k \in K, \tag{4.11}$$

$$l_{ij}^k \leq x_{ij}^k \leq u_{ij}^k, \quad \forall (i,j)\in E, \forall k\in K. \tag{4.12}$$

The arcs have lower and upper bounds (l_{ij}^k, u_{ij}^k) for each commodity, and there are also bounds u_{ij} on the total flow on the arcs. Constraint (4.11) requires flow balance to hold for each node and commodity. Clearly, without constraint (4.10), the problem would decompose into one minimum cost network flow problem per commodity. A thorough theoretical treatment of the multi-commodity flow problem and its variations can be found in Schrijver [199]. Many practical problems can be modeled as variants of the multi-commodity flow problem, for example problems in communication, transportation and manufacturing systems [8].

Computational Complexity

Clearly, the continuous multi-commodity flow problem can be modeled as a linear program. Tardos [212] has shown that any linear program can be solved in strongly polynomial time, which means that polynomial time algorithms exist for the continuous multi-commodity flow problem. However, Even et al. [85] have shown that the integer multi-commodity network flow problem is NP-hard, even in the case of only two commodities and with capacities equal to 1.

As constraint (4.6) is defined over a sequence of activities, it contributes to destroying the flow structure of the problem. Prömel [99, Appendix 2] has shown that the resource constrained network flow problem (i.e. with only one commodity, but with additional constraints on routes) is NP-hard. However, fully polynomial approximation schemes exist for any fixed number of resources [222].

4.3 Solution Methods for the Multi-Commodity Network Flow Problem

Good overviews of the continuous multi-commodity flow problem and various solution algorithms can be found e.g. in [8, 17, 131, 165]. Ahuja et al. [8], Assad [17], Kennington and Shalaby [131] and McBride [165] all categorize multi-commodity network flow algorithms into three groups: *price-directive decomposition*, *resource-directive decomposition* and *partitioning methods*. McBride

[165] also adds interior-point algorithms to the list, but we will not discuss that method further here. Ali et al. [11] in 1980 reported that the performance of their implementations of price-directive and partitioning methods did not differ much, while the resource-directive decomposition method was about twice as fast, on average. However, it should be noted that some of the references above are a bit out-dated, and the performance results reported might not hold today.

We will now introduce the solution approaches for the continuous multi-commodity flow problem, before moving on to the integer multi-commodity flow problem. Observe that even though the continuous problem can be solved directly as a linear program, this approach is often not practically possible, since it often requires a huge amount of constraints and variables. The methods discussed below are often more useful in practice.

Price-Directive Decomposition

As we have already mentioned, if constraint (4.10) is removed, the multi-commodity flow problem decomposes into separate min-cost flow problems for each commodity. Since the min-cost flow problem can be solved in strongly polynomial time [211], constraint (4.10) can be said to constitute the hard part of the multi-commodity problem. In price-directive decomposition methods the hard constraints are relaxed, and put in the objective function with prices. The problem then becomes to find prices such that the single-commodity subproblems together yield an optimal solution to the multi-commodity problem.

Lagrangian relaxation

Lagrangian relaxation is an example of a price-directive decomposition strategy, which associates multipliers (prices) π_{ij} with the arc capacities:

$$L(\pi) = \min \sum_{(i,j)\in E} \sum_{k\in K} c_{ij}^k x_{ij}^k + \sum_{(i,j)\in E} \pi_{ij} \Big(\sum_{k\in K} x_{ij}^k - u_{ij} \Big) \tag{4.13}$$

$$\text{s.t.} \quad \sum_{j\in V} x_{ji}^k - \sum_{j\in V} x_{ij}^k = b_i^k, \quad \forall i \in V, \forall k \in K, \tag{4.14}$$

$$l_{ij}^k \le x_{ij}^k \le u_{ij}^k, \quad \forall (i,j) \in E, \forall k \in K. \tag{4.15}$$

When Lagrangian relaxation is applied, *subgradient optimization* is typically used to dynamically update the multipliers for the subproblems (4.13)–(4.15) in such a way that an optimal solution to the master problem is found. Lagrangian relaxation and subgradient optimization are both well studied areas, and we direct the interested reader to e.g. the textbook by Bazaraa et al. [31] for a more in-depth treatment of these topics. We will here only give a brief introduction to how subgradient optimization can be applied to the multi-commodity flow problem.

Given multipliers π_{ij}^s at iteration s, the single-commodity flow problem (4.13)–(4.15), with costs adjusted to $c_{ij}^k + \pi_{ij}^s$, is solved for each commodity $k \in K$. Using the resulting solution $\hat{x}_{ij}^k$, the multipliers are updated according to

$$\pi_{ij}^{s+1} = \max\Big\{0, \pi_{ij}^s + \theta^s\Big(\sum_{k\in K} \hat{x}_{ij}^k - u_{ij}\Big)\Big\}, \tag{4.16}$$

where θ^s is an appropriately chosen step length. Clearly, overused arcs are penalized with a higher cost, while underused arcs are rewarded. This process is iterated until it converges. To ensure theoretical convergence, the step size θ^s must be chosen carefully. However, for practical applications, careful tuning of the step size is often required to achieve satisfactory performance.

Kohl et al. [143] uses the so-called *geometric series rule*

$$\theta^s = \alpha\beta^s, \alpha > 0, 0 < \beta < 1. \tag{4.17}$$

to guarantee convergence of their subgradient algorithm for the railway fleet assignment problem. One of the problems which they encounter, and which is typical when dual methods such as subgradient optimization are used, is that primal heuristics must be developed to obtain primal solutions.

Dantzig-Wolfe Decomposition

In the context of price-directive methods for multi-commodity flow problems, Dantzig-Wolfe decomposition [61] can be seen as a method for adjusting the penalty prices π_{ij} in equation (4.13). We will postpone a more general discussion of Dantzig-Wolfe decomposition to Chapter 5. However, explaining the specific case of multi-commodity flow before explaining the general case is not as odd as it might first seem – Dantzig and Wolfe themselves actually got the idea for the general decomposition method from Ford and Fulkerson's article [91] about an algorithm for "simplex computation for an arc-chain formulation of the maximal multi-commodity network flow problem". We will here use notation more similar to that of Ford and Fulkerson than that of Dantzig and Wolfe.

Let us first transform the multi-commodity flow problem to a path formulation instead of the arc formulation above. According to the *Flow Decomposition Theorem* [8, Chapter 3], any arc flow can be transformed into a path flow, i.e. a flow on entire paths in the network.[2] Reversely, every path flow can be *uniquely* transformed to an arc flow. Let L^k be the set of possible paths for commodity k, and let p_l^k be the amount of flow to send along path $l \in L^k$. c_l^k is the per unit cost of path $l \in L^k$. Using the indicator $\Delta_{ij}(l)$, which is 1 if arc (i,j) is included in path l, and 0 otherwise, the relationships between the

[2]Assuming positive costs, or acyclic networks, we can ignore cycles that might otherwise be necessary in the path formulation.

arc and path flows, and their costs, are:

$$x_{ij}^k = \sum_{l \in L^k} \Delta_{ij}(l) p_l^k, \tag{4.18}$$

$$c_l^k = \sum_{(i,j) \in E} \Delta_{ij}(l) c_{ij}^k. \tag{4.19}$$

Using the path flow variables, the multi-commodity flow problem can be stated as

$$\min \quad \sum_{k \in K} \sum_{l \in L^k} c_l^k p_l^k, \tag{4.20}$$

$$\text{s.t.} \quad \sum_{k \in K} \sum_{l \in L^k} \Delta_{ij}(l) p_l^k \leq u_{ij}, \quad \forall (i,j) \in E, \tag{4.21}$$

$$\sum_{j \in V} \Delta_{ji}(l) p_l^k - \sum_{j \in V} \Delta_{ij}(l) p_l^k = b_i^k, \quad \forall i \in V, \forall k \in K, \tag{4.22}$$

$$l_{ij}^k \leq \Delta_{ij}(l) p_l^k \leq u_{ij}^k, \quad \forall (i,j) \in E, \forall k \in K. \tag{4.23}$$

Clearly, without any additional assumptions about the problem structure, the re-formulation does not do much good. However, if we assume that there is a single source and sink for each commodity,[3] that there is a fixed demand d^k between the source and sink of commodity k, and ignore the commodity specific bounds, the problem becomes

$$\min \quad \sum_{k \in K} \sum_{l \in L^k} c_l^k p_l^k, \tag{4.24}$$

$$\text{s.t.} \quad \sum_{k \in K} \sum_{l \in L^k} \Delta_{ij}(l) p_l^k \leq u_{ij}, \quad \forall (i,j) \in E, \tag{4.25}$$

$$\sum_{l \in L^k} p_l^k = d^k. \quad \forall k \in K. \tag{4.26}$$

This problem decomposes into one shortest path problem per commodity if constraint (4.25) is relaxed. Using Dantzig-Wolfe decomposition, the penalty prices are thus decided by solving a sequence of shortest path problems. Observe that the assumptions made above are not unrealistic; they do in fact hold for TAS.

Unfortunately, while the path formulation contains fewer constraints than the arc formulation, it contains an exponential number of variables. It is therefore often solved with a method called *column generation*, which dynamically generates variables that can improve the current solution. Column generation will be the main topic of the next chapter. Compared to Lagrangian relaxation solved with subgradient optimization, the column generation approach requires more computational effort per iteration. Observe that the

[3]This is sometimes called an origin-destination formulation [128].

subproblems of the two approaches are the same, even though in our case the Lagrangian subproblem was described as a minimum cost network flow problem, and the column generation subproblem as a shortest path problem. However, applying the assumptions above about a single source and sink, and the demand d^k between them, the minimum cost network flow problem can clearly be simplified to a shortest path problem. The additional computational effort for column generation instead comes from the fact that a linear program is solved in each iteration to calculate the current optimal prices. On the other hand, column generation in general requires fewer iterations to converge than the subgradient optimization approach.

The original column generation approach for multi-commodity flows is as already mentioned due to Ford and Fulkerson [91]. However, Ford and Fulkerson only suggested column generation as a method to solve the 'maximal multi-commodity network flow problem', i.e. the extension of the max-flow problem to multiple commodities, and not the more general minimum cost network flow problem. It was instead Tomlin [214] who noted that Ford and Fulkersson's approach could be used also for the minimum cost network flow problem. Jones et al. [128] investigate how the problem formulation affects the performance of a Dantzig-Wolfe solution approach. They compare

- the *origin-destination problem*, where each commodity has a single origin and a single destination, and for which the subproblem is a shortest path problem;

- the *destination specific problem*, where each commodity has multiple origins and a single destination, and for which the subproblem is a shortest path tree problem;

- and the *product specific problem*, where each commodity has multiple origins and destinations, and for which the subproblem is a general minimum cost network flow problem.

The problem types differ both in the number of constraints in the master problems, and in the complexity of the subproblems. In their computational tests, Jones et al. show that the path-based origin-destination problem is the computationally most efficient formulation. Two reasons are mentioned for why this formulation is the most efficient one: The simpler subproblem of this formulation, and the fact that fewer total extreme points of the underlying linear program are enumerated.

Resource-Directive Decomposition

While price-directive decomposition methods decompose the multi-commodity flow problem by relaxing and penalizing the hard constraints (4.10), resource-directive decomposition achieves the same thing by dividing the commodity-common bound u_{ij} among the commodities. By limiting the flow of commodity k to r^k_{ij}, the following problem is obtained:

$$\begin{aligned}
\min \quad & \sum_{(i,j)\in E} \sum_{k\in K} c^k_{ij} x^k_{ij}, && (4.27)\\
\text{s.t.} \quad & \sum_{k\in K} r^k_{ij} \le u_{ij}, \quad \forall (i,j)\in E, && (4.28)\\
& \sum_{j\in V} x^k_{ji} - \sum_{j\in V} x^k_{ij} = b^k_i, \quad \forall i\in V, \forall k\in K, && (4.29)\\
& l^k_{ij} \le x^k_{ij} \le r^k_{ij}, \quad \forall (i,j)\in E, \forall k\in K, && (4.30)\\
& l^k_{ij} \le r^k_{ij} \le u^k_{ij}, \quad \forall (i,j)\in E, \forall k\in K. && (4.31)
\end{aligned}$$

Constraint (4.28) makes sure that the total arc capacities are not violated, and constraints (4.30) and (4.31) relate r^k_{ij} to the original arc bounds. Now, by considering the problem as a function of the vector of limits r, and denoting by $z(r)$ the optimal value of problem (4.27)–(4.31) for limits r, a *resource allocation* problem with simple constraints and a difficult objective function is obtained:

$$\begin{aligned}
\min \quad & z(r), && (4.32)\\
\text{s.t.} \quad & \sum_{k\in K} r^k_{ij} \le u_{ij}, \quad \forall (i,j)\in E, && (4.33)\\
& l^k_{ij} \le r^k_{ij} \le u^k_{ij}, \quad \forall (i,j)\in E, \forall k\in K. && (4.34)
\end{aligned}$$

It is well known from linear programming theory [8, 175] that the optimal objective function of a linear program with parameterized right-hand side is piecewise linear and convex in the parameter, so $z(r)$ is obviously piecewise linear and convex in r. Ahuja et al. [8] present a method which can be used for resource-directive decomposition. The method uses ideas similar to subgradient optimization to adjust the limits r_{ij}, but since the objective function $z(r)$ is nondifferentiable, they do not present any exact methods. Kennington and Shalaby [131] in 1977 reported that at this time, subgradient optimization was the most efficient method for resource-directive decomposition. More recently, McBride and Mamer [167] have presented a combined partitioning and resource-directive method.

The Partitioning Method

Partitioning methods make use of the fact that any feasible basis to the multi-commodity flow problem can be partitioned into one basis for each single-commodity problem, plus some additional arcs. By performing a suitable variable transformation, a problem with a special block structure is obtained, for which a basis is made up of one basis for the hard constraints, and one basis per commodity. Using efficient spanning tree techniques, as for the single-commodity network simplex algorithm, it is possible to relatively efficiently solve the resulting linear program. Unfortunately, in practice this method has turned out to perform worse than the decomposition methods [8, Chapter 17].

Recently, McBride [166] has claimed that his EMNET [164] implementation of a basis partitioning multi-commodity flow solver is superior to other solution methods for multi-commodity flow problems. To the best of our knowledge, however, his claim has not been supported by other researchers.

Other Solution Methods

Another, less common, solution method for the multi-commodity flow problem is the dual-ascent algorithm due to Barnhart [24]. The dual-ascent method iteratively adjusts dual prices to improve the dual solution. Since the method does not produce any primal solution, Barnhart also presents a heuristic primal solution generator. Frangioni and Gallo [93] present a 'bundle type' dual-ascent approach. They use a specialized bundle algorithm to solve the Lagrangian dual, and report promising results, especially for problems with a large number of commodities.

Larsson and Yuan [150] present an augmented Lagrangian method to solve the multi-commodity flow problem. The method augments the Lagrangian relaxation with nonlinear penalties, which improves convergence, and leads to convergence to an optimal primal solution. The presented solution algorithm lacks theoretical convergence in finite time, but gives very competitive computational results for instances with up to 80000 commodities. Comparison to a Dantzig-Wolfe decomposition algorithm shows that the Dantzig-Wolfe algorithm works better on small instances, while the augmented Lagrangian algorithm is more suited to produce approximate solutions to huge instance quickly.

Awerbuch and Leighton [20] present a $(1+\epsilon)$-approximation algorithm for the multi-commodity flow problem. The algorithm runs in $\mathcal{O}(\frac{|K|\cdot L^2\cdot |E|}{\epsilon^3}\ln^3\frac{|E|}{\epsilon})$ steps, where L is the length of the longest flow path. Cleary, for small L, this is linear in the number of commodities, which is much better than most exact algorithms. Schneur and Orlin [198] present a scaling based approximation algorithm.

The natural decomposition into single-commodity network flow problems suggests that algorithms using parallel computers are interesting for the multi-commodity flow problem. Pinar and Zenios [178] and Castro and Frangioni [45] present parallel implementations of multi-commodity flow solvers. Pinar and Zenios use a linear-quadratic penalty algorithm, while Castro and Frangioni implement an interior-point algorithm specialized to multi-commodity flow problems.

The Integer Multi-Commodity Network Flow Problem

So far, we have only dealt with methods for continuous multi-commodity flow problems. As we have already mentioned, the integer multi-commodity flow problem is much more difficult to solve. Some of the approaches to the continuous multi-commodity flow problem are more suitable than others to be extended to solving integer multi-commodity flow problems. For example, the partitioning method, since already embedding a linear programming algorithm, is fairly easy to extend to solve integer problems. This possibility exists in e.g. the Ilog CPLEX network flow solver [121]. A lot of effort has also been spent on extending the column generation method to handle integrality [28, 218]. We will discuss these extensions in more detail in Chapter 7.

In [219], Verweij et al. solve an integer multi-commodity flow problem originating in the aircraft production industry. They use column generation, rounding heuristics and branch-and-bound to obtain close to optimal solutions. Barnhart et al. [27] use column generation combined with branching and cutting planes, so-called *branch-and-price-and-cut*, to solve integer multi-commodity flow problems.

Most of the column generation approaches to e.g. transportation problems solve integer multi-commodity flow problems, often with side constraints, even if it is not directly stated. Chapter 5 contains a longer list of references to such approaches.

The Multi-Commodity Network Flow Problem with Side Constraints

Holmberg and Yuan [120] present a multi-commodity flow problem with side constraints on paths. Holmberg and Yuan note that the path formulation is much more suited to path-dependent side constraints than the arc formulation, and they propose a column generation solution technique. The situation is very similar for TAS, where the maintenance side constraints (4.6) and (4.8) are easy to define with a path formulation, but difficult with the arc formulation. In cases where the side constraints are easy to model in the arc-based formulation, using an extended version of a partitioning method is a possibility. By identifying embedded networks in linear programs, and exploiting this structure, partitioning methods can sometimes be very effective. Wollmer [226] discusses resource constrained multi-commodity flow.

For problems containing particularly difficult side constraints, especially integer problems, various types of heuristics algorithms and meta-heuristic methods can be useful. These methods typically do not rely on a linear, or even a mathematical, representation of the problem, and are therefore often appealing to use for practical applications. *Local search* (LS) is a general name for heuristic methods which improve a solution to a problem by gradually applying small local changes to it. Typical local changes for routing problems are e.g. swapping connections in a route, or swapping activities between routes. A good reference to local search methods is the book by Aarts and Lenstra [1]. In the last part of this thesis, we will demonstrate how our mathematical and constraint programming models can be combined with local search to form a more scalable tail assignment system.

The standard local search algorithm, which is often called *hill-climbing*, applies all local changes which improves the current solution in a greedy fashion. The algorithm thus has the drawback that it can get stuck in suboptimal solutions. Several improvements have been proposed to remedy the suboptimality problem, the two most common ones being *Tabu Search* (TS) and *Simulated Annealing* (SA). Tabu search was first presented by Glover in 1986 [101]. A more complete, and recent, reference is the book by Glover and Laguna [102]. The main idea of tabu search is to avoid cycling in the hill-climbing algorithm by maintaining a list of the most recent local changes applied, the so-called *tabu list*, and make sure not to return to a recently visited solution. There is also an *aspiration function* by which it can be decided to override the tabu list under special circumstances.

Simulated annealing is a method originating from physics, designed to mimic the way an annealing crystal or metal finds the minimum energy state. It was first presented in 1983 by Kirkpatrick et al. [134]. Instead of applying all promising local changes, they propose that a *probability function* is used to decide the probability that a change is applied, even if the change leads to a solution worse than the current one. By gradually decreasing a virtual *temperature*, the probability of accepting worse solutions exponentially decreases. The idea is that initially a large part of the solution space is scanned more or less randomly, while as the algorithm progresses it gets more and more focused on only finding improvements.

A solution technique that has emerged as a viable candidate in recent years is Constraint Programming. We will get back to constraint programming in the second part of this thesis.

4.4 Model Path-Tas

The most commonly used exact solution techniques for large-scale multicommodity flow problems are price-directive methods, especially Lagrangian relaxation and subgradient optimization, and column generation.

For our purposes, the Lagrangian relaxation method as described in Section 4.3 has the drawback that it does not produce a primal solution. Strictly speaking, no primal solution to the linear relaxation of the master problem is necessary, since we are really only interested in integer solutions. However, access to a primal solution to the linear relaxation is very useful to methods looking for integer solutions.

We have decided to use column generation as our main solution technique,

Model 4.2 PATH-TAS

$$\min \sum_{r \in R} c_r x_r, \tag{4.35}$$

$$\text{s.t.} \sum_{r \in R} a_{fr} x_r = 1, \quad \forall f \in F, \tag{4.36}$$

$$x_r \in \{0, 1\}. \tag{4.37}$$

as this makes it easy to model the complicated resource constraints, produces primal feasible solutions, and because column generation is a well tested and documented solution method that is used extensively in airline planning [28, 73, 96, 109, 117, 151, 155]. Column generation also has a modular structure, with a route *generation* (pricing) part and a route *selection* (RMP) part, which is straightforward to extend, upgrade and maintain. In Chapter 5 we will take a closer look at column generation. Reformulating model TAS using path flows gives a path-based model, PATH-TAS, shown in Model 4.2. Observe that even though we have given the name PATH-TAS to Model 4.2 to indicate its connection to the flow decomposition theorem, we will most often refer to paths in the flight network as *routes*, as that is a more suitable name for the physical movement performed by the aircraft.

Clearly, PATH-TAS is a pure set partitioning problem. Here, R represents the set of all possible routes satisfying individual route constraints, and x_r are binary decision variables that are 1 if route $r \in R$ is used and 0 otherwise. c_r is the cost of using route r, and a_{fr} is 1 if activity f is covered by route r and 0 otherwise. Note that R contains the routes for *all* commodities, which combined with the fact that all capacities are 1 makes it unnecessary to mention the commodities at all in PATH-TAS. Constraint (4.36) makes sure that all activities are covered exactly once, and since carry-in activities belong to F, that one route per aircraft is selected. To handle unassigned activities, the set R contains columns covering single activities, at a high penalty cost. This is equivalent to adding slack variables to constraint (4.36). Maintenance possibility activities, as discussed in Section 3.2, are typically given a penalty cost of 0, since they are optional activities. Each feasible route is represented by a column in the constraint matrix of constraint (4.36), and a binary variable in PATH-TAS. When referring to PATH-TAS, the words route, column and variable can all be used to denote a feasible aircraft route, represented by a column in the constraint matrix.

Computational Complexity

We have already seen that TAS is NP-hard. Of course, reformulating the problem in terms of path flows does not change this – the set partitioning problem is NP-hard [99, Appendix 2]. To make things even worse, much of the complexity of Path-Tas is hidden in R, since all individual route constraints are implicitly defined by R, and as we will see in Chapter 5, generating routes in R is an NP-hard problem too. But the main drawback of Path-Tas compared to Tas is that we go from a model with at most $|T| \times |F|^2$ variables, to a model with exponentially many variables, which is why column generation is used as the solution method.

4.5 Mixed Fleet Models

In Chapter 3 we discuss the potential benefits of mixing fleet types in tail assignment. While we have not explicitly modeled this, nothing in models Tas or Path-Tas contradicts using multiple fleet types. In fact, since we consider each aircraft individually, it is no problem to enforce even aircraft specific route-based constraints, such as maintenance constraints or activity restriction constraints, or aircraft dependent costs. There are several reasons why costs might differ between aircraft. Obviously, aircraft belonging to different fleets have different characteristics, and might therefore differ in costs. But the costs might differ even within the same fleet, e.g. if some aircraft are leased and others owned, or if there are different contracts for aircraft leased from different lessors. Observe that model Tas does not include costs on activities, but only on connections. When solving e.g. re-fleeting problems it is necessary to have fleet dependent costs, representing revenue minus operation costs, on activities. This is easily modeled by simply including the activity cost in the connection cost in such a way that c_{ijt} is the cost of the connection between activities i and j, plus the cost of activity i, for aircraft t. Our model can thus handle both aircraft dependent constraints and aircraft dependent costs, which makes it possible to mix fleets without any additional variables or constraints.

4.6 Lower Bounds

There are lots of reasons why it is good to be able to calculate good lower bounds for optimization problems. The most important reason is the ability to measure solution quality of a feasible solution in terms of a gap between lower and upper bounds, the latter being given by the objective value of the feasible solution. In this section we will show how a lower bound can be calculated for the tail assignment problem. The process used to calculate the lower bound can also be used to detect imbalances in the input data, which is a useful side effect.

At the beginning of this chapter we concluded that model Tas is a multi-commodity network flow problem with side constraints. Considering this, a natural relaxation of the problem is to remove the side constraints and the commodity-linking constraints, and use the resulting model to calculate a

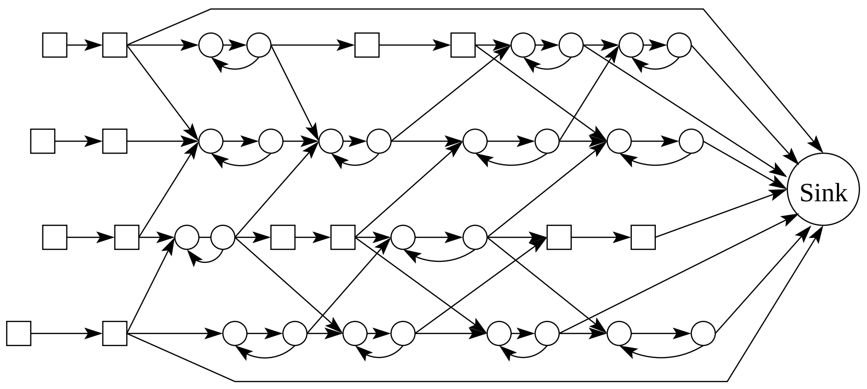


Figure 4.1 The flow network corresponding to model TAS^{relax}.

lower bound. This is very easy, as the resulting model is a network flow problem that is solvable in polynomial time [211]. Observe that in the general case the result would be one network flow problem per commodity. But since there are unit flow bounds on all arcs, the problems can be combined to a single problem. This leaves the following problem: Cover all the activities once, and use each aircraft in at most one route. Only legal connections may be used, since minimum connection time constraints and other connection-based restrictions are still present. But which individual aircraft covers a preassigned activity is not specified, it is only required that *some* aircraft covers all such activities. Also, cumulative constraints and individual flight restrictions are obviously not considered.

Since it is necessary to ensure that each activity is covered exactly once, and a network flow problem only allows bounds on arcs, activities must be represented by arcs rather than nodes. Each activity will thus be represented by an arc fixed at unit flow. All the nodes, with the exception of start nodes and a special end node (sink), have node balance 0. The start nodes have node balance 1, and the end node has node balance $-\#aircraft$. The cost of a connection arc is of course the connection cost, while the cost of an activity arc is typically 0. It does not necessarily have to be 0, but since normally all activities should be covered, it makes little sense to have costs on activities, as that would only add a fixed cost to the final objective value. In cases where fleets are mixed, aircraft dependent activity costs different from 0 might be desirable, as well as aircraft dependent connection costs. When calculating the lower bound, the least cost over all aircraft is then used as arc cost.

The easiest way to make it possible to leave activities unassigned is to set the bounds on the arcs representing the activities to $[0, 1]$ instead of $[1, 1]$. This will however not give the high cost that we want associated with unassigned activities – it will rather be beneficial to leave activities uncovered. So instead

Model 4.3 Tas^{relax}, network flow relaxation

$$\min \quad \sum_{i\in F}\sum_{j\in F} g_{ij}y_{ij} + \sum_{i\in F} h_i z_i, \tag{4.38}$$

$$\text{s.t.} \quad \sum_{j\in F} y_{ji} + z_i = 1, \quad \forall i \in F, \tag{4.39}$$

$$\sum_{j\in F} y_{ij} + z_i = 1, \quad \forall i \in F, \tag{4.40}$$

$$\sum_{i\in F} y_{is} = |T|, \tag{4.41}$$

$$0 \le z_i \le 1 - p_i, \quad \forall i \in F, \tag{4.42}$$

$$0 \le y_{ij} \le 1, \quad \forall i,j \in F. \tag{4.43}$$

an expensive backward arc is created for each activity. In the same way we also want to create the possibility not to use an aircraft. This is easily accomplished by adding an arc without cost from each start activity to the sink. The resulting network is depicted in Figure 4.1.

Let us denote by z_f the backward arc for activity $f \in F$. Since the forward arc is fixed to 1, it can be left out of the problem. The sink node is called s, and y_{ij} is the arc representing the connection between activities i and j. Let the parameter p_f be 1 if activity f is a preassigned activity, and 0 if not. Constraint (4.42) ensures that only activities which are not preassigned can be left unassigned in the model. The cost of arc (i,j) is g_{ij}, and the cost of leaving activity f unassigned is h_f. Using this notation, the network flow model Tas^{relax} is defined in Model 4.3. Since Tas^{relax} is a minimum cost network flow problem, its constraint matrix is obviously totally unimodular [8, Thm. 11.12], which means that an integer solution will be obtained even without integrality constraints, if a solution method producing a basic solution is used [8, Thm. 11.11]. Observe that this would not necessarily be true if p_f was not integer.

Model Tas^{relax} can be solved very fast indeed, even for networks of substantial size. We use standard network flow or linear programming solvers, such as CPLEX's network flow optimizer [121] or Xpress [64]. A valid question is of course how good the lower bound actually is on real instances. For instances with few, or non-constraining, aircraft specific constraints, the bound is often very good, while for more severely constrained instances, or instances with varying aircraft dependent costs, it is worse. For some instances a stronger lower bound can be obtained by calculating the sum of the cheapest routes for all aircraft.

Five

Column Generation

In [61], Dantzig and Wolfe describe a decomposition principle that is especially suitable for linear programs with a certain structure, such as block diagonal programs. It involves reformulating the original program, and since the reformulated linear program has an exponential number of variables, it often cannot be solved directly, but variables (columns) must be generated dynamically. The process of dynamically generating variables is called *column generation*.

In the rest of this chapter, we will describe how column generation can be used to solve the tail assignment problem. We will start in Section 5.1 by taking a brief look at the theoretical foundation of Dantzig-Wolfe decomposition, before describing the master and pricing problems for the specific case of tail assignment in Sections 5.2 and 5.3. In Section 5.4 we will describe our solution method for the pricing problem, and the chapter will be concluded by a brief look at the integer problem and a final summary in Sections 5.5 and 5.6.

5.1 Dantzig-Wolfe Decomposition

Assume that we want to solve the block diagonal linear program

$$\min \quad c_1^T x_1 + c_2^T x_2 + \cdots + c_p^T x_p, \tag{5.1}$$

$$(LP) \quad \text{s.t.} \quad \begin{pmatrix} A_1 & A_2 & \cdots & A_p \\ B_1 & & & \\ & B_2 & & \\ & & \ddots & \\ & & & B_p \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_p \end{pmatrix} \leq \begin{pmatrix} b_0 \\ b_1 \\ \vdots \\ b_p \end{pmatrix}, \tag{5.2}$$

$$x_i \geq 0, \quad \forall i. \tag{5.3}$$

To simplify things, let us assume that the convex polyhedra

$$X_i = \{x_i | B_i x_i \leq b_i, x_i \geq 0\}$$

are non-empty and bounded for $i = 1, \dots, p$. It is well-known from linear programming theory [175] that any $\hat{x}_i \in X_i$ can be written as a convex combination of the (finite number of) extreme points of X_i.[1] So, letting x_i^l, $l = 0 \dots L_i$, be the extreme points of X_i, any $\hat{x}_i \in X_i$ can be written as

$$\hat{x}_i = \sum_{l=0}^{L_i} \lambda_i^l x_i^l, \tag{5.4}$$

where

$$\begin{aligned} \sum_{l=0}^{L_i} \lambda_i^l &= 1, \quad i = 1 \dots p, & (5.5)\\ \lambda_i^l &\geq 0 \quad i = 1 \dots p, \quad l = 0 \dots L_i. & (5.6) \end{aligned}$$

Now, replacing x_i in problem (LP) with equation (5.4), the linear program becomes

$$(LP') \qquad \begin{aligned} \min \ & \sum_{l=0}^{L_1} (c_1^T x_1^l)\lambda_1^l + \dots + \sum_{l=0}^{L_p} (c_p^T x_p^l)\lambda_p^l, & (5.7)\\ \text{s.t.} \ & \sum_{l=0}^{L_1} (A_1 x_1^l)\lambda_1^l + \dots + \sum_{l=0}^{L_p} (A_p x_p^l)\lambda_p^l \leq b_0, & (5.8)\\ & \sum_{l=0}^{L_i} \lambda_i^l = 1, \quad \forall i, & (5.9)\\ & \lambda_i^l \geq 0 \quad \forall i, \forall l. & (5.10) \end{aligned}$$

In (LP'), the variables are the λ_i^l, rather than the x_i^l. Clearly, (LP') has only $|b_0| + p$ constraints, compared to $\sum_{i=0}^{p} |b_i|$ for (LP). However, (LP') in general has a huge number of variables. Since the number of variables is the sum of the number of extreme points of the polyhedra X_i, it is even exponential in the original number of variables. This type of decomposition was first presented by Dantzig and Wolfe [61] in 1960. It is called *the decomposition principle of linear programming*, or more commonly *Dantzig-Wolfe decomposition*. In [62], Dantzig and Wolfe detail the algorithmic part of the decomposition principle, and provide some insight concerning unboundedness and non-linear cost functions.

To solve (LP'), Dantzig and Wolfe proposed a dynamic column generation[2] algorithm, which works as follows. Let us first denote the optimal dual values

[1] In case X_i is unbounded, any $\hat{x}_i \in X_i$ can be written as a convex combination of extreme points of X_i plus a non-negative combination of extreme rays of X_i.

[2] It is sometimes simply called Dantzig-Wolfe decomposition, and sometimes *delayed* column generation [117].

corresponding to constraint (5.8) by π, and the dual values corresponding to constraint (5.9) by β. Now, the reduced cost $\overline{c}_i^l$ of variable λ_i^l is

$$\overline{c}_i^l = c_i^T x_i^l - \pi A_i x_i^l - \beta_i. \tag{5.11}$$

According to linear programming theory, variables with negative reduced cost can improve the current basic feasible solution. So, having an extreme point (basic feasible solution) of X_i for which equation (5.11) is negative could improve the current solution to (LP'). Clearly, one such candidate, if any exists, can be found by solving the linear program

$$(PP) \qquad \begin{aligned} \min \quad & (c_i^T - \pi A_i)x_i, & (5.12)\\ \text{s.t.} \quad & B_i x_i \leq b_i, & (5.13)\\ & x_i \geq 0. & (5.14) \end{aligned}$$

If the optimal objective value of (PP) is less than β_i, the corresponding extreme point will have negative reduced cost. The column generation algorithm is thus:

1. solve the *master* problem (LP') to obtain dual values π and β;

2. using the dual values, solve the *pricing* problem (PP) for each i, to find at least one negative reduced cost variable to add to the master problem. If one is found, go to step 1. If not, the current solution to (LP') is optimal.

While we have described Dantzig-Wolfe decomposition for a block structured linear program, the same reasoning can be applied to any linear program. Also, the decomposition into extreme points of the pricing problems can be a bit misleading – while the decomposition in terms of extreme points suggests the column generation algorithm, column generation can be applied directly on problem (LP), if an algorithm exists to find a negative reduced cost variable given a dual solution. The resulting variable does not have to be an extreme point of any X_i.

In fact, already Dantzig and Wolfe [61] noted that column generation is really a generalization of the revised simplex method. In the simplex algorithm [60, 227], the solution process moves between so-called basic solutions. In each step the reduced cost of the non-basic variables are checked, and when a variable is found that has negative reduced cost (in the case of minimization), it is inserted into the basis, and one of the current basic variables is removed. If no such variable is found, the current solution is optimal. The process of finding a variable with negative reduced cost is called *pricing*, and the selected variable is *priced out*. We refer the reader to any linear programming textbook, e.g. [175], for more information about the details of the simplex algorithms. The difference between the standard simplex algorithm and the revised one is that the revised one does not rely on heavy matrix calculations to obtain the reduced costs for all variables, but instead calculates them for each variable individually as needed. Column generation is a further generalization of this – it is in fact not even necessary to have all the variables available when pricing. Instead, variables can be generated dynamically, using the reduced cost as a criterion whether a variable is worth generating or not. Obviously, only variables with negative reduced cost relative to the current basic solution are eligible to enter the basis.

The problem of finding negative reduced cost variables is called the *pricing problem*, and the master program is called the *restricted master problem* (RMP), since it contains a restricted number of variables. If negative reduced cost variables are found when solving the pricing problem, it in theory suffices to add one of them, and resolve the RMP. In practice [72] it is often a good idea to find either the most negative one, or to add several variables in each step.

Abstracting away the details, we are left with a procedure where the *selection* and *generation* of variables are clearly separated. Clearly, this separation suits the purposes of Path-Tas perfectly, as it gives a way to dynamically generate and add legal routes, without having to enumerate them all. The term column generation is sometimes used not only for the exact Dantzig-Wolfe kind of column generation described here, but also for heuristic methods that iteratively solve an LP, or an ILP, and re-generate variables until some convergence criterion is met. Andersson et al. [14] provide an example of such an approach to crew pairing. Further, while some researchers use the term 'column generation' to denote only the pricing problem, we use it to denote the entire process, including the RMP, the pricing problem and the iteration between them.

Column generation has been used to solve a vast array of problems, and its theoretical and computational properties have been studied by numerous researchers over a long period of time. Examples of column generation approaches to crew and fleet planning can be found e.g. in [12, 42, 86, 96, 109, 117, 151, 192, 200], and more general theoretical studies can be found e.g. in [28, 73, 76, 77, 80, 82, 157, 191, 215]. The book [74] edited by Desaulniers et al. seems to be a good future reference for column generation.[3]

5.2 The Restricted Master Problem

Since all the constraints related to individual paths are handled by the pricing problem, and using the definitions from Chapter 4, the restricted master

[3]When writing this, [74] is not available, but it is cited by one of its editors in [76].

problem of PATH-TAS can be stated simply as

$$(RMP) \qquad \begin{aligned} \min \quad & \sum_{r \in R'} c_r x_r, && (5.15) \\ \text{s.t.} \quad & \sum_{r \in R'} a_{fr} x_r = 1, \quad \forall f \in F, && (5.16) \\ & x_r \geq 0, \quad \forall r \in R', && (5.17) \end{aligned}$$

where $R' \subseteq R$ is the set of routes currently available. The problem with this formulation is that the set of variables R is huge, and difficult to enumerate. Clearly, RMP is an LP relaxed pure set partitioning problem. Observe that if $R' = R$, or if RMP is solved by column generation, it will solve the LP relaxation of PATH-TAS (PATH-TAS^{relax}, Model 5.1), and thus provide a lower bound to PATH-TAS. Since the lower bound from model TAS^{relax} is a lower bound even to the non-integral multi-commodity network flow problem, it is in fact a lower bound to PATH-TAS^{relax}, which means that PATH-TAS^{relax} provides a better lower bound to PATH-TAS than TAS^{relax}. However, the PATH-TAS^{relax} bound is much more computationally difficult to obtain.

In order for the column generation to start, an initial feasible solution to the RMP must be provided. The most simplistic approach is to use slack variables with high penalties. The initialization process is discussed in more detail in Section 6.2. To solve the RMP we can use several different methods. We have chosen to use the primal or dual simplex methods, or the barrier methods, of CPLEX [121] or Xpress [64]. Section 6.7 discusses which method is best to use for different problems.

While we have chosen to use linear programming based methods such as barrier and simplex methods, it is also possible to use methods of other kinds to solve RMP. A key observation is that we do not need a primal feasible solution to RMP, but only dual values. To prove optimality, we must find an optimal dual solution, but in intermediate steps not even that is necessary. While the linear programming viewpoint of column generation described above makes it natural to use linear programming based methods, other methods can sometimes be useful. For example, the *Sprint* [92] and *volume* [22] algorithms have been used in a column generation context [12]. Kohl and Madsen [145] present an approach to the Vehicle Routing Problem with Time Windows (VRPTW) using Lagrangian relaxation and subgradient and bundle algorithms to obtain dual values from the master problem. Gustafsson [109] presents a column generation approach to crew pairing using a subgradient algorithm combined with probing integer heuristics.

It is sometimes claimed that basic solutions, given e.g. by simplex algorithms, are not suitable for column generation due to the fact that they are located in extreme points of the optimal face of the LP polyhedron [157]. To overcome this problem, and instead produce solutions in the relative interior of the optimal face, methods such as *analytic center* [81] have been developed.

Model 5.1 Path-Tas^{relax}, linear programming relaxation of Path-Tas

$$\min \sum_{r\in R} c_r x_r \tag{5.18}$$

$$\sum_{r\in R} a_{fr} x_r = 1 \quad \forall f \in F \tag{5.19}$$

$$x_r \geq 0 \quad \forall r \in R \tag{5.20}$$

To date, these methods have not been used extensively for practical problems.

5.3 The Pricing Problem

Since we are using standard linear programming algorithms to solve the master problem, the most complex part of the column generator for us to consider is the pricing problem solver. This is where most of the problem specific details and difficult constraints are handled. Observe that we talk about a single pricing problem here. Applying Dantzig-Wolfe decomposition to the multi-commodity network flow problem [8, Section 17.6], it is natural to think of one pricing problem per commodity (aircraft), while seeing the process as a generalization of the revised simplex algorithm makes a single pricing problem more natural. The two viewpoints are of course equivalent, since finding a negative reduced cost variable in the single pricing problem is equivalent to finding a negative reduced cost variable in *any* of the single-commodity pricing problems.

As explained above, the purpose of the pricing problem is to find variables with a negative reduced cost. In our case, variables correspond to routes for individual aircraft. We want these routes to be legal with respect to maintenance constraints, flight restrictions, and other individual constraints. For model Path-Tas, the reduced cost of route r can be written as

$$\bar{c}_r = c_r - \sum_{f\in F} a_{fr}\pi_f, \tag{5.21}$$

where c_r is the cost of variable r and π_f is the dual value for constraint f. Since a_{fr} is 1 if activity f is covered by route r and 0 otherwise, this can be simplified to

$$\bar{c}_r = c_r - \sum_{f\in F_r} \pi_f, \tag{5.22}$$

where F_r is the set of activities covered by route r. Further, if we assume that

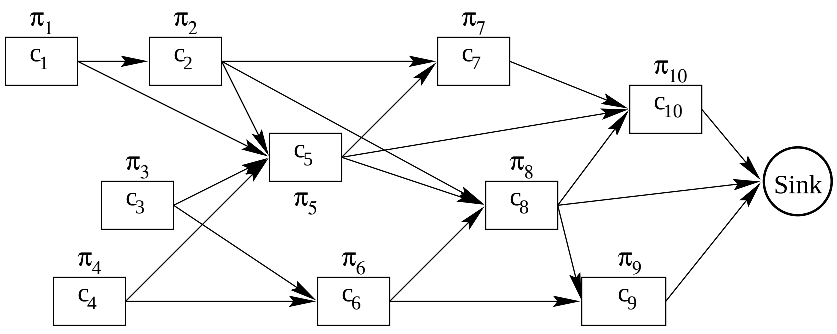


Figure 5.1 A shortest path network.

c_r is the sum of costs of activities or connections,[4] so that $c_r = \sum_{f \in F_r} c_f$, where c_f is the cost of activity f, the reduced cost of route r can be written as

$$\bar{c}_r = \sum_{f \in F_r} c_f - \sum_{f \in F_r} \pi_f = \sum_{f \in F_r} (c_f - \pi_f). \qquad (5.23)$$

Assuming that we have a network of activities, such as the connection network described in Section 3.1, the problem of finding routes with negative reduced cost is simply a matter of finding paths in the network representing legal routes, with negative reduced cost. Since the costs are defined on each node in the network, the pricing problem can be formulated as a *shortest path problem* – find the shortest path in the network with respect to the costs above: if it is negative, add it to the RMP; if not we know that no negative reduced cost path exist, and the column generation process can be stopped. Figure 5.1 shows a tiny shortest path network example.

Unfortunately, the problem is not quite as simple as this in our case, since we have some further complicating constraints. With a pure shortest path algorithm generating routes, we will get routes that potentially violate flight restrictions, maintenance constraints, and preassigned activity constraints. Obviously, we have to modify the shortest path algorithm to consider these constraints as well.

Also, it should be observed that we have already made a quite severe limitation of the way the cost function can be formulated, by restricting it to be described in terms of costs on activities or connections, as a so-called *additive* cost function. However, it is quite common to make this restriction for problems of this kind, to make the pricing problem fit with the shortest path formulation. Also, we actually already assumed this in model TAS. A column generation algorithm which can handle a non-additive cost function is presented by Hjorring and Hansen [117].

[4]It is actually more natural to have costs on connections, and put any activity costs on the connections as well. The two representations are computationally equivalent.

Preassigned Activities and Activity Restrictions

We generate routes for one aircraft at a time, to be able to enforce aircraft-specific constraints. This is in accordance with the fact that when using column generation for the multi-commodity network flow problem, one has to solve one pricing problem per commodity. When generating routes for a specific aircraft, we make sure that the aircraft is assigned to all its preassigned activities by simply solving a sequence of shortest path problems between consecutive pairs of preassigned activities. That way, we make sure that no route is generated that bypasses such activities.

The activity restrictions are handled by simply forbidding aircraft t from covering the activities in R_t. Including these two constraints is trivial with any standard shortest path algorithm.

Cumulative Constraints

The cumulative constraints are the complicating ones. In this section, we will only discuss maintenance constraints, as these are the most common types of cumulative constraints, but our pricing algorithm handles general cumulative constraints. As long as we do not consider cumulative constraints, the pricing problem is solvable in polynomial time, as it is just a normal shortest path problem, slightly modified to handle preassigned activities and activity restrictions.

To handle the cumulative constraints, we introduce *resources*. A resource is a value that is accumulated along a route, and that can never exceed a certain limit. Each resource thus corresponds to a *resource constraint*. Observe that the resources correspond exactly to r_{im} in model Tas, and the resource constraints correspond to constraint (4.6). We also allow the resources to be reset to a special *reset value* at certain points along a path. For each type of maintenance we introduce a resource, for which the upper limit is the maximum number of flying hours or landings allowed between checks for this maintenance type, and which can be reset whenever a check for this maintenance type can be performed.

The introduction of resources turns the shortest path problem into a *resource constrained shortest path problem* (RCSPP), which unfortunately is NP-hard [99, Appendix 2]. Normally, resources in RCSPPs are not reset along the path, so in that aspect our problem is special.

The default mode of our model is to only plan maintenance possibilities, rather than actual maintenance checks. This means that we will normally only make sure that there are enough possibilities for maintenance checks in all routes, and not actually specify where maintenance will be performed. The reason is that it is not immediately clear what is preferred – lots of checks, or only the least possible amount of checks. In reality, only as few checks as necessary will be performed, of course. But it might be good, for example when disruptions occur, to have more than the minimum amount of checks to choose from. However, maintenance possibility activities can be used to model hangar capacity and planned maintenance checks, as discussed in Sections 3.2 and 4.1, and the cost function can be adjusted to penalize connections representing maintenance checks. Doing so requires a re-formulation of the underlying cumulative constraint to be reset only at specific maintenance activities.

5.4 Solving the Resource Constrained Shortest Path Problem

To solve the resource constrained shortest path problem we use the standard label-setting algorithm [75] proposed by Desrochers and Soumis. The fact that the resources are reset at e.g. maintenance opportunities in our problem does not change the algorithm substantially. For each node we save a set of labels, each representing a path to the node. The algorithm works by pulling labels from all possible predecessor nodes to a node, while updating costs and resource consumption.

Since the problem is dated there is a total ordering among the nodes in the network, and by pulling labels in order of increasing activity start time, we only have to pull once for each node. If it is possible to perform a maintenance check of type m between nodes A and B, the consumption of the resource corresponding to m will be reset to 0, or another appropriate value, for labels pulled from A to B. If not, the labels will be updated with the value on the previous label plus the consumption on the arc between A and B.

For flexibility, the resource updates are controlled via the Carmen Rave modeling language, described in Section 12.5. When we pull labels, we ask Rave if it is possible to do a maintenance check for the different types between the nodes. If it is, we ask what value we should reset the resource consumption to, and if it is not, we ask how much we should update the resource. Since Rave is linked dynamically to the optimization software, it is easy to add new types of constraints without re-compiling the software.

Once the sink node has been labeled, negative reduced cost paths, if any, can be constructed by tracing the negative cost sink labels backward through the network. Observe that while the purpose of the original algorithm is to find the least reduced cost path through the network, several negative reduced cost paths can easily be obtained by constructing paths from several negative cost sink labels. However, these paths are not necessarily the least cost paths.

Label Pulling Strategy

For each node, we save a restricted number of labels. We apply *dominance*, as described e.g. in [75], to limit the number of labels saved. A label dominates another label if its cost and consumption of all resources is less than or equal to that of the other label:

Definition 5.1. *Label i* dominates *label j if*

$$\bar{c}_i \leq \bar{c}_j \text{ and } r_i^k \leq r_j^k, \quad \forall k.$$

Here, r_i^k is the consumption of resource k of label i, and $\bar{c}_i$ is the reduced cost of label i.[5] A label that is not dominated by any other labels at a node is called *efficient*. Observe that two labels dominate each other if their reduced costs and resource consumptions are equal. Only efficient labels are saved in the labeling, since optimal paths for our RCSPP cannot be dominated, and in fact no sub-path of an optimal path can be dominated [163]. In practice, it is often not possible to store all efficient labels at each node. Instead, various heuristic ways to limit the number of labels must be used.

Our label strategy is the following: For each node, we have two limits for the maximum number of labels, $\underline{\chi}$ and $\overline{\chi}$, $\underline{\chi} \leq \overline{\chi}$. As long as the number of labels is less than $\underline{\chi}$, we insert new labels if they are efficient, and remove dominated labels after insertion, thus making sure that only efficient labels are present after insertion. The labels are sorted lexicographically, i.e. first according to reduced cost, and then according to the resource consumption, for each resource in turn. Now, if the number of labels is above $\underline{\chi}$, we insert new labels if they are efficient and lexicographically less than the last label.

Definition 5.2. *Label i is* lexicographically less than *label j ($i <_{lex} j$) if*

$$\bar{c}_i < \bar{c}_j,$$

or if

$$\bar{c}_i = \bar{c}_j \text{ and } \exists l : r_i^l < r_j^l \text{ and } r_i^k = r_j^k \quad \forall k \in [0, l).$$

If the number of labels after insertion is greater than $\overline{\chi}$, the lexicographically largest labels are removed until only $\overline{\chi}$ labels remain. Inserting labels according to the lexicographical ordering is heuristic – for two labels, neither dominating the other, it is impossible to say which is better, without some information about which costs or consumptions in the labels are more important. By using the lexicographical ordering, we assume that the reduced cost is the most important attribute of the labels, and the resource consumptions are ranked according to index, so that resource n is considered more important than resource $n+1$, and so on. This measure of importance is only used when inserting labels into an already full set of labels, simply because one must

[5]Observe that the label resource consumptions r_i^k are similar, by not identical, to the path resource consumptions r_{im} used in model Tas. The r_{im} consumptions are only defined for fixed routes, while the r_i^k consumptions are defined for *potential* routes.

make some decision about which labels are better than others to prevent the number of labels from growing without limits.

To guarantee optimality of the pricing algorithm, i.e. that the least reduced cost path is always found, $\underline{\chi}$ must be chosen large enough so that all efficient labels at a node are stored. However, the optimality criterion for column generation does not require the least reduced cost path to be found, but only that *any* negative reduced cost path is found, if one exists. However, making sure that a negative reduced cost path is found, if one exists, also requires choosing $\underline{\chi}$ large enough so that all efficient labels at a node are stored, or dynamically increasing the number of labels when no negative reduced cost path is found.

Using lexicographical ordering to limit the number of labels is reasonable in practice, since we are mainly interested in finding the best paths from a reduced cost perspective, and are only considering the resource constraints as something we have to respect. Observe that the ordering is not the same as dominance. While dominance is an exact[6] mathematical property that helps remove useless labels, the lexicographical ordering is a heuristic way to rank labels when dominance is not enough to limit the number of labels. A label that is lexicographically larger than another label is not necessarily dominated, but it cannot dominate the other label.

We use a static lexicographical ordering, which always ranks the reduced cost first, and then uses the same ordering of the resources. However, one could imagine that the ordering depended on how difficult the cumulative constraints represented by the resources are to satisfy, for each aircraft. This would be suitable in cases where the constraints are difficult to satisfy, and the difficulty varies from aircraft to aircraft.

If one is looking for several paths, not only the optimal ones, applying dominance according to Definition 5.1 is not necessarily a good criterion to use to filter away labels. A label that is as good as another in terms of cost, but consuming slightly more of one resource, will be removed by the dominance relation, even though it might represent a very good path. We still use dominance as a way to filter away labels, simply because it is a nice criterion that efficiently limits the number of labels.

In [149], Larsson and Miloloza investigate how the labeling process can be adapted to minimize conflicts when using a sliding time window solution approach, as described in Chapter 12.

Soft Cumulative Constraints

In Chapter 3, we mentioned that tail assignment could include soft cumulative constraints. Soft cumulative constraints are useful to model constraints which we are not sure can be satisfied, or constraints which we want to satisfy 'as

[6]Unless one is looking for more than one path, or only store a limited number of labels.

much as possible'. For example, if we suspect that our 5-day maintenance constraint could be tightened to 4 days, we might change the upper limit of the constraint to 4, and add a quadratic penalty for each extra day without maintenance. Essentially, the optimizer will then try to satisfy the 4-day limit, but if this is not possible, it will try to minimize the violation of the constraint. By setting the cumulative constraint limit to 0, soft cumulative constraints can also be used to model some cost function components which cannot be captured by connection costs.

A cumulative constraint is made soft by simply specifying a penalty function above *or* below the limit. For example, if the penalty function $f(x)$ is specified for violations above the limit l_m, consuming x' units incurs a penalty cost of $f(x')$ if $x' > l_m$, and 0 otherwise. The function can be any non-decreasing function, and is specified as described in Section 12.5. The penalties are handled by calculating the 'violation' (below or above) of the constraint each time a label is pulled, and adjusting the cost of the label accordingly. Since we use dominance to remove labels, decreasing penalty functions might lead to removal of efficient labels, which is why only non-decreasing penalty functions are allowed.[7]

Observe that the presence of soft cumulative constraints might make the Tas^{relax} lower bound very poor, since it relaxes all cumulative constraints.

The Complete Pricing Algorithm

The complete label pulling algorithm for one aircraft, including soft constraints, is shown in Algorithm 5.1. Γ_f denotes the labels on activity f, and $\max_{lex}(\Gamma_f)$ is the lexicographically largest label in Γ_f. c_γ and $\bar{c}_\gamma$ are cost and reduced cost of label γ, respectively, and r^m_γ is the consumption of resource $m \in M$ on label γ. The consumption of resource m on the connection between activities i and j is s^m_{ij}, and the consumption on activity k is s^m_k. Ω^m_{ij} is 1 if the consumption of resource m can be reset between activities i and j, and l^t_m is the maximum consumption for resource m and aircraft t. Observe that we have here made the resource limit aircraft dependent, which is a generalization of model Tas. The cost of connecting activities i and j for aircraft t is c_{ijt}, and the current dual value for activity i is π_i. For clarity, Algorithm 5.1 only considers penalties *above* the limit for soft resource constraints. Φ^m is 1 if resource constraint m is soft, and $\Delta^m(x)$ gives the corresponding penalty for violation x. It is assumed that the carry-in activities have no predecessor activities.

Steps 5–11 of Algorithm 5.1 handle the actual resource consumption updating, and steps 12–20 handle the reduced cost updating, including penalty costs due to violated soft cumulative constraints. The remaining steps of the

[7] In fact, decreasing penalty functions can be used, but the pricing problem might then be solved suboptimally, which might in turn lead to poor performance of the column generation algorithm.

Algorithm 5.1 The label pulling algorithm for aircraft $t \in T$.

```
procedure PULLLABELS($t$)
    for all $f \in (F \setminus R_t)$, in increasing departure time order do
        for all predecessors $f'$ of $f$ do
            for all labels $\gamma_{f'} \in \Gamma_{f'}$ do
                for all $m \in M$ do
                    if $\Omega^m_{f'f} = 1$ then
                        $r^m_{\gamma_f} \leftarrow s^m_f$
                    else
                        $r^m_{\gamma_f} \leftarrow r^m_{\gamma_{f'}} + s^m_{f'f} + s^m_f$
                    end if
                end for
                if $\Phi_m = 1$ and $r^m_{\gamma_f} > l^t_m$ and $r^m_{\gamma_{f'}} > l^t_m$ then
                    $c^p \leftarrow \Delta^m(r^m_{\gamma_f}) - \Delta^m(r^m_{\gamma_{f'}})$
                else if $\Phi_m = 1$ and $r^m_{\gamma_f} > l^t_m$ then
                    $c^p \leftarrow \Delta^m(r^m_{\gamma_f})$
                else
                    $c^p \leftarrow 0$
                end if
                $\bar{c}_{\gamma_f} \leftarrow \bar{c}_{\gamma_{f'}} + c_{f'ft} - \frac{\pi_{f'}+\pi_f}{2} + c^p$
                $c_{\gamma_f} \leftarrow c_{\gamma_{f'}} + c_{f'ft} + c^p$
                if $\exists m$ s.t. $\Phi_m = 0$ and $r^m_{\gamma_f} > l^t_m$ then
                    next $\gamma_{f'}$
                end if
                if $\gamma_f$ efficient w.r.t. $\Gamma_f$ then
                    if $|\Gamma_f| < \underline{\chi}$ then
                        $\Gamma_f \leftarrow \Gamma_f \bigcup \gamma_f$
                        Remove non-efficient labels in $\Gamma_f$
                    else if $\gamma_f <_{lex} \max_{lex}(\Gamma_f)$ then
                        $\Gamma_f \leftarrow \Gamma_f \bigcup \gamma_f$
                        if $|\Gamma_f| > \overline{\chi}$ then
                            Remove $|\Gamma_f| - \overline{\chi}$ worst labels from $\Gamma_f$
                        end if
                    end if
                end if
            end for
        end for
    end for
end procedure
```

algorithm remove non-efficient labels, and make sure that at most $\overline{\chi}$ labels are stored at each node.

Other Approaches to Solve the RCSPP

One could imagine other approaches for solving the RCSPP, e.g. Lagrangian methods. Borndörfer et al. [42] uses a Lagrangian relaxation of the resource constrained shortest path problem to obtain shortest path distances which are then given to a depth first search algorithm. In [135] Kjerrström investigates the possibilities for such a solution approach to our pricing problem, and concludes that the possibility to reset resource consumptions along paths complicates the standard approach. Instead a model including explicit maintenance possibilities in the pricing network is proposed. Unfortunately, computational results indicate that this method cannot compete with the labeling method described above in terms of computational efficiency.

5.5 The Integer Problem

Once we have solved the relaxed problem using column generation, we are still left with the problem of finding an integer solution. There are several possibilities for this, e.g. various heuristic approaches, or branch-and-price. Branch-and-price [28] combines column generation and branch-and-bound, and has the benefit that it can guarantee optimality, but is on the other hand somewhat time consuming. However, most often optimal solutions are not necessary in tail assignment. It often suffices with a 'good enough' solution within a reasonable amount of running time.

Instead, three heuristic algorithms have been implemented, all based on heuristic fixing combined with column re-generation. The integer heuristics are described in more detail in Chapter 7, where we also present computational results and possible extensions of the methods.

5.6 Summary

We have now introduced a mathematical model, some possible solution methods, and described column generation, the mathematical solution method we have implemented. In the pricing problem solver, we introduced heuristic acceleration techniques. While these techniques improve the practical computational performance, they can under some circumstances make the pricing problem solver fail to find optimal solutions to the pricing problem. We therefore also discussed how the pricing algorithm can be adjusted to guarantee optimality.

Six

Solving the Linear Programming Relaxation

In Chapter 5, the basic column generation algorithm was described. However, in order to get the best possible performance, a lot of implementation details must be considered. Algorithm 6.1 shows the column generation solution process in detail. Almost every step in Algorithm 6.1 can be varied in different ways, leading to slightly different implementations of the column generation algorithm, which all vary in performance. Recent surveys of acceleration techniques for all aspects of column generation are also provided by Desaulniers et al. [73] and Lübbecke and Desrosiers [76, 157].

In this chapter we will focus on solving the linear relaxation of the RMP, and look at some of the implementation choices we have made, and the performance implications that these choices have. The preprocessing algorithms (step 3 in Algorithm 6.1) are described in Chapter 10, but we will use them already here, as they improve results substantially. We will investigate the following aspects of the column generation algorithm:

- the initialization of the RMP, (Section 6.2);
- how many columns to generate per iteration, (Section 6.3);
- how column removal affects performance, (Section 6.4);
- dual value re-evaluation to avoid similar columns, (Sections 6.5 and 6.6);
- the impact of the choice of algorithm to solve the RMP, (Section 6.7);
- stabilization of the dual values, (Section 6.8); and
- a constraint aggregation strategy to speed up convergence, (Section 6.9).

However, we will start this chapter by presenting the test instances we have used throughout this chapter, and the testing methodology we have used.

Algorithm 6.1 The column generation solution process for minimizing a linear program.

```
data
    SolveRMP: Function solving RMP, returning primal solution and objective value
    $\underline{z}$: Known lower bound for LP, or $-\infty$
    $\epsilon$: Relative optimality tolerance
    $\sigma$: Maximum number of iterations

1: procedure ColumnGeneration($\sigma$, $\underline{z}$)
2:     Read data
3:     Preprocess data
4:     Add initial columns to RMP
5:     for $i \leftarrow 1, \sigma$ do
6:         Remove ‘bad’ columns from RMP  ▷ In Section 6.4 we will explain what ‘bad’ means in this context
7:         $(\overline{x}, \overline{z}) \leftarrow$ SolveRMP
8:         Set dual values in pricing problem
9:         $C \leftarrow \emptyset$
10:        for all pricing problems do
11:            Solve using dual values from RMP
12:            $C \leftarrow C \cup$ (negative reduced cost columns generated)
13:        end for
14:        Add columns $C$ to RMP
15:        if $C = \emptyset$ then
16:            $\underline{z} \leftarrow \overline{z}$          ▷ LP optimality provides new lower bound
17:            break
18:        else if $100 \times \frac{\overline{z}-\underline{z}}{\underline{z}} < \epsilon$ then
19:            break
20:        end if
21:    end for
22:    return $(\overline{x}, \underline{z}, \overline{z})$    ▷ Return solution, lower bound and objective value
23: end procedure
```

Name	Period	Fleet	#Aircraft	#Flight legs	Maint. constr.	Flight restr.
A320_1w	1 week	A320	42	2012	No	No
B737_1w	1 week	B73d	43	1313	No	No
B757_1w	1 week	B752	39	1251	No	No
DC9_1w	1 week	DC9	17	544	Yes	Yes
DC9_1w2	1 week	DC9	17	735	Yes	Yes
DC9_10d	10 days	DC9	17	948	Yes	Yes
DC9_2w	2 weeks	DC9	17	1468	Yes	Yes
DC9_1m	1 month	DC9	17	3063	Yes	Yes
M82_1w	1 week	MD82	14	539	Yes	Yes
M82_2w	2 weeks	MD82	14	1044	Yes	Yes
M82_1m	1 month	MD82	14	2285	Yes	Yes

Table 6.1 Test instances for Chapter 6, except Sections 6.8 and 6.9.

6.1 Test Instances and Testing Methodology

Tables 6.1 and 6.2 show a set of test instances that will be used throughout this chapter. The instances in Table 6.2 are used in Sections 6.8 and 6.9, and the instances in Table 6.1 are used elsewhere. The instances come from European and American carriers, and vary in size. They are chosen to give a spread in terms of number of aircraft, fleet mixing strategies and planning period lengths. No really large instances, as solved in subsequent chapters, are included here. The reason is that the column generation algorithm alone is not able to solve them, as this chapter will show.

The instances in the top-most part of Table 6.1 come from three different airlines, and thus have quite different structures. The bottom-most instances (DC9 and downward) in Table 6.1 and the instances in Table 6.2 come from the same airline. These are the only instances that contain maintenance constraints and restrictions on destinations. The maintenance constraints are mixed 'base visit rules', forcing all aircraft to return to their home base every 6 days, and for some instances additional 'A' check and lubrication check constraints are added. The flight restriction constraints are constraints used in production, which are far to complicated, and volatile, to be explained here. For all instances, variants of the simple cost function discussed in Section 3.5 are used. In this chapter we will not analyze the objective values or solutions, only compare them.

The MDS instances mix MD82, MD83 and MD88 fleets. Solving MD82, MD83 and MD88 together is common practice for the airline in question, as the three fleets are very similar in terms of cockpit configuration and seating

Name	Period	Fleet	#Aircraft	#Flight legs	Maint. constr.	Flight restr.
DC9_1m2	1 month	DC9	17	2378	Yes	Yes
M83_1m	1 month	MD83	9	1337	Yes	Yes
M87_1m2	1 month	MD87	7	1151	Yes	Yes
DC9_1w3	1 week	DC9	17	727	Yes	Yes
MDS_1m3	1 month	MDS	31	5571	Yes	Yes
MDS_1m4	1 month	MDS	31	5127	Yes	Yes
MDS_1m5	1 month	MDS	31	5816	Yes	Yes
MDS_1m6	1 month	MDS	31	4987	Yes	Yes

Table 6.2 Test instances for Sections 6.8 and 6.9.

capabilities. Observe that Tables 6.1 and 6.2 show the number of atomic activities (including preassigned activities), often called *flight legs*, of the problems. The number of flight legs is not identical to the number of activities, as some legs are sequenced already in the input data.

Testing Methodology

In this chapter, we will only look at the column generation process to solve the linear programming relaxation of Path-Tas, i.e. Path-Tas^{relax}. Since some instances take a very long time to solve, especially with certain parameter settings, we have not always solved the instances to optimality. Instead, we have performed at most 1000 column generation iterations, which is a substantial number. Instances for which LP optimality were not reached within 1000 iterations will be marked with '*' in the tables throughout this chapter. LP optimality is reached if either the lower bound produced by Tas^{relax} is reached, or if no negative reduced cost columns are generated by the pricing problem. To guarantee LP optimality we here save a large number of labels per node in the pricing phase, and dynamically increase $\underline{\chi}$ in Algorithm 5.1 when no negative reduced cost columns are generated.

Table 6.3 shows the performance of the most basic column generation approach. Here, only one column is generated per aircraft per iteration, no columns are removed, and CPLEX's primal simplex algorithm is used to solve the RMP. We use primal simplex as the basic choice since the RMP stays primal feasible when columns are added, making it possible to warm-start the RMP from the previous solution when using primal simplex. The table shows the lower bound of each instance produced by Tas^{relax}, as well as the CPU time in seconds to complete the 1000 iterations, and the resulting objective value after 1000 iterations. In the rest of this chapter, we will describe and test various acceleration techniques which improve the performance of the basic

Instance	TAS^{relax}	Time	LP obj.
A320_1w	628145	25392	*2688760
B737_1w	4969800	10724	*4987345
B757_1w	4385060	10702	*4452041
DC9_1w	12101800	239	12101800
DC9_1w2	16662800	160	16662800
DC9_10d	21314200	847	*21315217
DC9_2w	33062400	3552	*33096871
DC9_1m	69513000	43920	*69905470
M82_1w	11984100	23	11984100
M82_2w	23628800	1004	*23693150
M82_1m	52775500	11372	*55730434

Table 6.3 Performance of the basic column generation algorithm, performing at most 1000 iterations.

column generation algorithm. This will lead to a gradually improving column generation algorithm, as we will use the best strategy from each section for the rest of the thesis, with only a few exceptions. In Section 6.10 we will summarize the strategies we have found to work best.

Throughout this chapter, we will only report results in terms of running times and LP objective values. While in some cases it might be interesting to report how many columns are generated, how many columns are used in the final solution, and other statistical data, we have decided to exclude this data from the tables. The main reason is simply to limit the amount of numerical data presented. Reporting more data would require many extra tables to be included, which would substantially increase the space required, and make the text more hard to digest. Also, we feel that it is more clear to only focus on a few key figures throughout the chapter, and the running times and objective values seemed like the best choice, given that we often will not solve the problems to optimality.

Most instances in Table 6.3 are marked with '*', meaning the optimal LP objective values were only reached for the three smallest instances with this basic approach. If the progress of the objective values is studied more closely, it becomes clear that the cause of the long running times in general is an extremely slow convergence when approaching an optimal solution. For example, for the DC9_1m instance, the objective value is about 1% from the lower bound after 750 iterations (26559 seconds), and then the remaining 40% of the running time is spent shrinking the gap to 0.56% in the last 250 iterations. This kind of behavior is typical for column generation algorithms. One reason for the slow convergence is likely to be degeneracy: the presence of several similar near-optimal solutions leads to many similar columns being generated, which tends to mean that little progress is made between iterations.

How much of the time is spent on solving the RMP and how much time is spent on solving the pricing problem differs from instance to instance. For the smallest instance, the RMP solution time constitutes about 43% of the total running time, while for the largest instance solving the RMP takes 98% of the total running time. Compared to many other column generation applications, these are rather large proportions on the running time spent on solving the RMP.

In Sections 6.8 and 6.9, a slightly different testing methodology is used. The reason is that these tests have been conducted later than the others, using other test instances. Also, in these sections a few larger instances, for which solving the LP to optimality is not possible within reasonable time, are used. Therefore, in Sections 6.8 and 6.9 we perform only 250 iterations instead of 1000 iterations.

6.2 Initialization of the RMP

As discussed in Chapter 5, the RMP needs an initial feasible solution to be able to start the column generation process. Generally, it is considered very beneficial to start from an as good set of initial columns as possible, to get the best possible convergence. Our strategy, which is fairly standard [157], is to start from an initial identity matrix, where each column has a very high cost. These columns can be seen either as illegal initial routes that have to be priced out of the basis, or as slack columns, giving us the opportunity to leave activities unassigned at a very high penalty. It is crucial that the slack penalties are set sufficiently high, or they might give tight bounds for the dual variables, i.e. it might be optimal to use some slack variables to leave some activities unassigned. Other possibilities for the initialization could be to start from a previous feasible solution, if one is known, or to initially generate a set of columns and let them form the initial solution, possibly together with the slack columns. The difficulty with starting from a previous solution is of course that it is in general a very difficult problem to find a solution.

Even when using a partial or a complete initial solution, there are good reasons to keep the slack columns. In the integer fixing heuristics, which are described in Chapter 7, fixing decisions that forces us to leave some activities unassigned might be performed. Also, the partial solution might not be possible to extend to a complete solution, which might also force the slack columns in the solution for some of the activities already covered by our initial solution to be used. So, our approach will be to always start from the identity matrix, but possibly extend it with some columns. Many researchers [157, 218] recommend tuning the slack column costs to stimulate good dual values, as the slack costs bound the dual variables. For us, this is not directly possible, since the slack columns and their costs have a physical meaning. Even so, in Section 6.8 we will investigate a dual stabilization technique that dynamically adjusts the slack costs.

To test how the presence of initial columns affect the column generation performance, we have conducted two experiments. In the first experiment we have started from a complete integer feasible solution, obtained by applying the integer heuristics presented in Chapter 7. In the second experiment we greedily generate disjoint columns until no more columns can be generated that do not cover already covered rows. The results of the runs are shown in Table 6.4. Since the effect of the initialization is really only interesting in the initial stage of the column generation, and we have seen that running to optimality can take a lot of time, we here only show the first 100 iterations of the column generation algorithm. The table shows lower bounds and objective values obtained after 100 iterations when using the normal slack variable initialization approach, and when starting from an integer feasible solution and a greedily generated set of disjoint columns.

The results indicate that both starting from a complete and a partial initial solution leads to worse convergence than starting from scratch. This is not all that surprising, as Vanderbeck [218] has also found that using integer solutions as starting solutions for column generation leads to poor performance. When starting from an initial solution, optimality is reached within 100 iterations only for the M82_1w instance, while for the other instances the LP objective values obtained are far worse than the ones obtained when starting from an identity matrix. Table 6.4 does not show running times, as the main purpose here is to demonstrate the convergence in terms of objective values. When starting from a partial solution the results are slightly better, but still far worse than the original results.

What happens in both cases is that since the initial columns are disjoint, the resulting dual solution is often such that a few of the dual variables involved in the constraints representing the column will take very large positive values, a few will take very large negative values, and the majority will be 0. This leads to the initial pricing phases generating columns which all cover the rows with large positive dual values, and completely avoids the ones with negative dual values. Of course, this extreme nature of the duals gradually decreases, as more and more columns are generated, but typically it takes many iterations to 'recover' to more reasonable dual values. Observe that in the case when the RMP is initialized with only slack columns, no duals will initially take negative values, since only one dual variable is involved in each constraint. The initial pricing phase will thus use dual values which are quite similar in size, and try to cover the rows for which it is most expensive to use the slack columns. It is probably possible to improve the poor convergence when using initial disjoint columns, e.g. by perturbations, but based on the results in Table 6.4, we have decided not to investigate this further.

Of course, an initial solution can be used in other ways than for warm-

Instance	TAS^{relax}	Only slack	Initial IP solution	Greedy initial generation
A320_1w	628145	628145	*4767216	*4068462
B737_1w	4969800	*4992880	*5809000	*11200900
B757_1w	4385060	*5435673	*11148000	*5660783
DC9_1w	12101800	12101800	*13276500	12101800
DC9_1w2	16662800	16662800	*17561600	16662800
DC9_10d	21314200	*21314728	*22699800	*21315485
DC9_2w	33062400	*33082701	*33309710	*33104691
DC9_1m	69513000	*70773639	*71377647	*73738904
M82_1w	11984100	11984100	11984100	11984100
M82_2w	23628800	*23636906	*26820300	*25123600
M82_1m	52775500	*54521648	*58781900	*56261793

Table 6.4 Performance of the basic column generation algorithm, with three different initialization strategies. At most 100 iterations have been performed, to only show the initial convergence. The 'Only slack' column thus represents the results from Table 6.3, truncated after 100 iterations.

starting the column generation algorithm. An initial solution provides an upper bound, and can e.g. be used to guide integer heuristics. However, since an initial solution is most often not available, these options have not been explored further. Also, it might be reasonable to add columns representing several solutions, to avoid getting only disjoint columns, and to hopefully avoid the extreme dual values. This has not been investigated, simply because obtaining one solution is difficult enough. Also, adding many non-disjoint initial columns very much resembles performing a few column generation iterations, and therefore seems somewhat pointless. Vanderbeck [218, Page 81] comes to the same conclusion, and states that 'as far as LP optimization is concerned, the column generation algorithm itself seems to be the best mechanism for generating appropriate columns'.

6.3 Number of Columns per Iteration

In order for column generation to work in theory, only one negative reduced cost column has to be added per iteration. However, in practice it is often a good idea to generate more than one column per iteration. Of course, since the reduced costs of these columns will be calculated with the same dual values, but only one of them can enter the basis at a time, there is no guarantee that all the columns that are generated will be inserted into the basis. In fact, since

		10 columns/aircraft		25 columns/aircraft	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
A320_1w	628145	121394	*1862249	223623	*1620948
B737_1w	4969800	28842	*4975591	60798	*4973923
B757_1w	4385060	66248	*4399422	134848	*4393914
DC9_1w	12101800	234	12101800	264	12101800
DC9_1w2	16662800	350	16662800	351	16662800
DC9_10d	21314200	3213	*21314404	5147	21314300
DC9_2w	33062400	12721	*33070554	26597	*33067257
DC9_1m	69513000	186744	*69782877	306773	*69723676
M82_1w	11984100	27	11984100	37	11984100
M82_2w	23628800	3658	*23631222	7817	23628800
M82_1m	52775500	41253	*55098990	78102	*54268843

Table 6.5 Performance when generating 10 or 25 columns per aircraft and iteration. Results when generating one column per aircraft and iteration can be found in Table 6.3. At most 1000 iterations have been performed.

many of the columns will likely be quite similar to each other, it is unlikely that this would happen. Section 6.5 addresses this problem in more detail. However, if more than one column does enter the basis, something has been gained compared to only adding a single column. But even if only one column enters the basis, the generated non-basic columns do influence the dual values.

The drawback of generating many columns is that it might take more time. Since more than one label is saved at each node in the labeling algorithm, the actual algorithm does not necessarily have to be more time consuming when more routes are generated. But there is some extra time involved in creating the routes once the labeling is finished. More importantly, more columns might slow the RMP down, simply because it gets larger. So there are definitely both good and bad points of generating many columns.

Table 6.5 shows the performance when generating 10 or 25 columns per aircraft and iteration. A comparison with Table 6.3 shows that in terms of convergence, these results are slightly better than when only generating one column per aircraft. But unfortunately, the running times are much higher. A more thorough investigation reveals that almost all the extra time is spent in the RMP. In many cases the pricing time is even decreased when generating more columns, primarily because less iterations are needed. It should be observed that since the dominance criterion is used in the labeling, it is not possible to guarantee that the lowest-cost columns are always generated. As we have to settle for a trade-off between running time and solution quality, we have chosen to generate 10 columns per aircraft as our default option.

6.4 Column Deletion

In column generation, columns that are promising from a dual point of view are generated and added to the RMP. However, even columns that initially seem promising, i.e. that have a negative reduced cost, can turn out to be useless when more columns have been generated. In this section we will investigate how to deal with columns that no longer have a negative reduced cost. We will only investigate whether a column is promising or not from a linear programming perspective. Columns which are useless in the LP phase might be important in an integer solution, but that is a different discussion, which we will return to when we discuss our integer heuristics, in Chapter 7.

Of course, if non-negative reduced cost columns are deleted, the RMP can probably be solved faster, since it contains fewer columns. We have already seen in Section 6.3 that generating too many columns increases running times substantially. On the other hand, non-basic columns also contribute with dual information. Loosely speaking, if the basic columns provide information about what is good, the non-basic columns provide information about what is bad. Hence, it might be a good idea to keep even non-basic columns in the problem. Of course, there is a middle way – only a few of the worst columns, based on reduced costs, could be deleted. To test how column deletion affects column generation performance, we have tested deleting columns with reduced costs above a certain threshold. We delete such columns after each column generation iteration. Clearly, while the purpose of generating many columns per iterations is to improve the convergence, column deletion is focused only on reducing running times. Deleting columns will not help convergence, it will likely even make it worse. But it should help reduce running times, which can in turn make it possible to perform more iterations in the same amount of time.

Table 6.6 shows the effect of removing columns with reduced cost $\bar{c}_r$ higher than 100 or 10^6. These almost naively simple column deletion rules have been chosen only to illustrate column deletion for this fixed set of instances. In all cases but three, the running times are lower when removing more columns (using a lower reduced cost threshold), as is expected. For comparison, results for the case when no columns are removed can be found in Table 6.5. For the DC9_1w and DC9_1w2 instances, removing more columns takes more time simply because it takes more iterations to converge. Since the RMP solution time is not as dominating for these problems as for the larger ones, the overall effect of removing columns is that the total running times increase. For the B737_1w instance the running time difference is not substantial, considering the total times. However, in all cases where LP optimality is not reached within 1000 iterations, better objective values are obtained when deleting columns with

		Delete $\bar{c}_r > 100$		Delete $\bar{c}_r > 10^6$	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
A320_1w	628145	83932	*2236148	93763	*1788236
B737_1w	4969800	25415	*4983869	23735	*4977138
B757_1w	4385060	28715	*4417893	39761	*4405065
DC9_1w	12101800	331	12101800	247	12101800
DC9_1w2	16662800	300	16662800	259	16662800
DC9_10d	21314200	2174	*21314634	3163	*21314417
DC9_2w	33062400	7710	*33089942	12166	*33070401
DC9_1m	69513000	123983	*69999511	150766	*69790194
M82_1w	11984100	28	11984100	32	11984100
M82_2w	23628800	2319	*23655652	2971	*23635204
M82_1m	52775500	22481	*55660470	36186	*55184233

Table 6.6 Performance with different column deletion strategies. At most 1000 iterations have been performed.

reduced costs above 10^6, than when deleting columns with reduced costs above 100. Our strategy will be to use a threshold of 10^6 for these instances, as that gives a good compromise between running time and solution quality.

For a more general collection of instances, other strategies for removing non-basic columns could be used. For example, a certain percentage of the non-basic columns could be removed, instead of all columns with reduced costs above a certain threshold. Another option could be to dynamically adapt the reduced cost threshold during the solution process. However, since the simple strategy described above works well for our instances, we have not investigated other methods further.

6.5 Dual Value Re-Evaluation

As illustrated by steps 10–13 of Algorithm 6.1, columns are generated for all aircraft using the same dual values in each column generation iteration. This means that the activities that are good from a dual point of view will be covered by many of the aircraft, even though ultimately only one aircraft can cover the activity. Observe that this similarity problem might occur even for columns generated for a single aircraft. Similar columns being generated is also related to degeneracy, since many of the similar columns will have the same reduced costs, so the method we propose below to generate less similar columns can be seen as an effort to decrease the effect of degeneracy.

In order to avoid that similar columns are generated, we have implemented a dual value re-evaluation scheme, to simulate what happens when a column is inserted into the master problem basis. The purpose is to make it possible to generate a larger amount of useful columns in each column generation iteration, without having to re-solve the RMP to obtain new dual values.

Imagine a column with reduced cost $\overline{c}_r = c_r - \sum_{f \in F_r} \pi_f$, where F_r is the set of activities covered by column r. If this column were to be inserted into the basis of the master problem, LP theory tells us that the dual values π_f would be re-calculated in such a way that $\overline{c}_r = 0$. This dual re-calculation can be simulated by penalizing the dual values of the activities in the column according to

$$\pi_f^{new} = \pi_f^{old} + \frac{\overline{c}_r}{\xi \times |F_r|}, \tag{6.1}$$

where ξ is a smoothing parameter. The new, simulated, reduced cost is now

$$\begin{aligned} \overline{c}_r^{new} &= c_r - \sum_{f \in F_r} \pi_f^{new} = c_r - \sum_{f \in F_r} \pi_f^{old} - \frac{\overline{c}_r^{old}}{\xi \times |F_r|} \sum_{f \in F_r} 1 = \\ &= c_r - \sum_{f \in F_r} \pi_f^{old} - \frac{\overline{c}_r^{old}}{\xi}. \end{aligned} \tag{6.2}$$

Let us first assume that $\xi = 1.0$. Then the reduced cost of the column will be evenly spread over the activities it covers, so that

$$\overline{c}_r^{new} = c_r - \sum_{f \in F_r} \pi_f^{old} - \overline{c}_r^{old} = 0. \tag{6.3}$$

By penalizing the dual values in this way the reduced cost of the column has been set to 0. Of course, the re-evaluation does not constitute a valid simplex pivot, so the new dual values do not form a feasible dual solution, but that does not matter here. The desired effect is that the activities in this column are penalized during the rest of the current pricing iteration, to avoid that similar columns are generated. Observe that if only one negative reduced cost column exists, it will always be generated, since re-evaluation is applied only after a negative reduced cost column is found. This is necessary to guarantee that the column generation algorithm converges to an optimal solution.

Adjusting ξ, it is possible to tune the behavior of the re-evaluation. A value of 1.0 sets the reduced cost to 0, as equation (6.3) demonstrates. However, using other values, we can simulate various kinds of behavior. For example, using a positive value very close to 0, we can simulate a greedy behavior. The dual values will be severely penalized, so that once an activity is covered by a column, it will not be covered by *any* other column in the rest of the pricing iteration. The effect is similar to the *disjoint column method* proposed by [95] for the crew rostering problem. It is also how the greedy initial generation in Section 6.2 was done. To reward overlapping columns, a negative ξ could be used, but that does not seem very useful.

6.6 Ordering of Pricing Subproblems

Remember that in each pricing iteration, one resource constrained shortest path subproblem is solved for each aircraft. As an effect of the dual value re-evaluation scheme, the order in which the individual RCSPPs is solved influences the result. While the problem which is solved first will use the true dual values from the RMP, the subsequent problems will use increasingly distorted dual values. As a result of this, it will be difficult to generate useful columns for the RCSPPs solved last. The columns generated by these problems will always be heavily influenced by which columns have been generated by the previously solved problems. To counter this, the order in which the problems is solved must be changed dynamically, so that the order of the problems gets fairly distributed over the total column generation process.

Re-ordering the RCSPPs can be done in several ways. One way could be to simply *swap* the order, so that the last problem ends up being solved first, and the first one last, in the next pricing iteration. However, this would mean that some problems get stuck in the middle all the time, never using the true dual values. Another strategy could be to *rotate* the problems, moving the front one to the end after each iteration. The drawback with this strategy is that it takes too long for some problems to be solved first. Instead, we propose a more randomized[1] re-ordering scheme. Each RCSPP, indexed by the aircraft t it represents, is assigned a rank $rank_t$, and the problems are solved in order of decreasing rank. The initial ranks are set to $1 \dots |T|$, in order of the aircraft's tail number, and after each pricing iteration the ranks are re-calculated according to

$$rank_t^{new} = rank_t^{old} + (pos_t^{old} + 1)^2, \tag{6.4}$$

where $pos_t \in 1 \dots |T|$ is the position of problem t in the ordering. For example, a problem which has a rank of 50, and is solved in position 3, will get the new rank $50 + (3 + 1)^2 = 66$. The thought behind the ranking is that a position close to the end of the ordering will give a high rank, and thus a position closer to the front in the next iteration. What happens is typically that a few of the problems in the end are moved to the front, and the remaining ones are shuffled in a seemingly random fashion. It should be emphasized that we have not put a great deal of effort into investigating different, more advanced, randomization schemes, but rather taken the first one that worked well enough for our purposes.

Table 6.7 shows results of test runs with $\xi = 10.0$ and $\xi = 100.0$, using the 'random' re-ordering strategy. Comparison with Table 6.6 clearly shows that the re-evaluation along with the re-ordering has a substantial effect on the convergence. In fact, LP optimality is reached within 1000 iterations for all but

[1]The word 'randomized' is abused here. We do not mean true randomization, or even pseudo randomization, but rather something that seems at least somewhat random.

		$\xi = 10.0$		$\xi = 100.0$	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
A320_1w	628145	60087	628145	258187	628145
B737_1w	4969800	2905	4973650	4763	4973650
B757_1w	4385060	19642	4388950	57963	4388950
DC9_1w	12101800	42	12101800	72	12101800
DC9_1w2	16662800	43	16662800	80	16662800
DC9_10d	21314200	360	21314300	835	21314300
DC9_2w	33062400	25218	33062500	42024	33062500
DC9_1m	69513000	703341	*69514105	437230	*69523384
M82_1w	11984100	4	11984100	11	11984100
M82_2w	23628800	1110	23628800	2039	23628800
M82_1m	52775500	286928	*52787226	142503	*52880339

Table 6.7 Performance with different re-evaluation factors. At most 1000 iterations have been performed.

two instances when using $\xi = 10.0$ and $\xi = 100.0$. When it comes to running times, the situation is less positive. For some of the instances the running times increase dramatically when using re-evaluation. The increased running times are caused by the RMP approaching optimality, and hence taking longer to solve due to slow convergence of the primal simplex algorithm.

According to the tests, the best strategy is to use a value for ξ around 10.0. It should be observed that the re-evaluation is applied not after generating each individual column, but after generating all columns for an aircraft. The reason is that all columns for an individual aircraft are generated in the same labeling sweep. The effect is that the similarity between columns for different aircraft is reduced, but the columns generated for each aircraft might still be similar to each other.

It is interesting to note that Rousseau [190] recently reported about convergence problems in so-called *constraint programming based column generation* [86, 129, 192], caused by similar columns being generated by the constraint programming based pricing algorithm. He compared standard constraint programming search with *limited discrepancy search* (LDS) [114] and an interior point stabilization technique [191], and found that LDS decreases the number of column generation iterations necessary to reach optimality, due to more diverse columns being generated.

6.7 RMP Algorithm Choice

As mentioned in Section 5.2, the simplex and barrier LP algorithms of CPLEX [121] or Xpress [64] are used to obtain dual values from the restricted master problem. However, which of the methods to use is not obvious, as they all have their pros and cons. The most obvious choice is primal simplex, since it is very closely related to the way column generation is described in Chapter 5. Primal simplex has been used for all tests in the previous sections in this chapter. One advantage of using the primal simplex method is that it is very well suited to re-optimization when adding columns. A given optimal primal solution is still primal feasible when additional columns are added, which might speed up re-optimization.

The dual simplex algorithm cannot be warm-started from a primal feasible solution, but it can on the other hand be expected to work better with the rather degenerate problems we are facing [73, 121, 157]. Barrier, or interior-point, LP algorithms have received a lot of attention in recent years, and have established themselves as the best choice for solving large airline crew rostering problems [106]. Unfortunately, barrier algorithms have the drawback that they cannot be warm-started from an initial solution. Further, barrier algorithms are typically not recommended when dealing with high density constraint matrices, especially high-density columns, as the factorization algorithms they use then tend to produce too dense Cholesky factorizations [121]. Since we are dealing with matrices with relatively high density, this does not seem to indicate barrier algorithms as the perfect choice for us. Assuming there are 20 aircraft in an instance, the average aircraft route will cover about 5% of the activities, some probably up to around 10%. Since crew work less than aircraft, the corresponding numbers for crew will be much lower, typically less than 1%.

Typically, barrier algorithms perform a *cross-over* operation in the end, to transform the possibly interior-point solution to a basic one. However, this is not necessary here, as a basic, or even primal, solution is not required during the column generation process ($\overline{x}$ is not used in Algorithm 6.1). Therefore, the cross-over phase can be skipped altogether. This saves a lot of time on some of our problems, where the cross-over operation can in fact be more time-consuming than actually finding the interior-point solution in the first place. Also, solution quality might be another reason why skipping crossover is a good idea. Since solutions obtained by a barrier algorithm might be interior-point solutions, they are often claimed [157, 191] to produce less extreme dual values than basic solutions, in the sense described in Section 6.2. More specialized methods, such as analytic center methods [81], have been proposed to remedy extreme dual values by forcing dual values to lie in the relative interior of the optimal face. Preliminary tests with such methods for our RMP indicated that such methods could be useful, but did not show any real advantage compared to the standard linear programming methods.

		Dual simplex		Barrier	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
A320_1w	628145	–	–	2049	628145
B737_1w	4969800	–	–	581	4973650
B757_1w	4385060	–	–	919	4388950
DC9_1w	12101800	262	12101800	28	12101800
DC9_1w2	16662800	343	16662800	46	16662800
DC9_10d	21314200	1454	21314300	128	21314300
DC9_2w	33062400	–	–	1066	33062500
DC9_1m	69513000	–	–	58679	69513100
M82_1w	11984100	12	11984100	6	11984100
M82_2w	23628800	2103	23628800	138	23628800
M82_1m	52775500	–	–	16831	52775500

Table 6.8 Performance when using dual simplex or the barrier algorithm to solve the RMP. At most 1000 iterations have been performed.

Table 6.8 shows results of using CPLEX's dual simplex and barrier algorithms without crossover on our test instances.[2] Corresponding results using primal simplex can be found in Table 6.7. Dual simplex performs extremely poorly even on the smallest instances, and for the larger instance it took so long that we did not even bother to wait for termination. On the other hand, the barrier algorithm without crossover clearly outperforms both primal and dual simplex.

The reason why the barrier algorithm is better is a combination of it simply solving the RMPs faster, and better convergence of the column generation algorithm. Observe that even though the RMP is solved to optimality in each iteration, the dual values produced by the different methods are different, which results in different convergence. Table 6.9 shows how many column generation iterations it takes to reach LP optimality using the different LP algorithms. In all cases, the barrier algorithm requires substantially less iterations than the simplex methods, in some cases less than half that of the others. When using the barrier algorithm, all instances can be solved to LP optimality within at most 567 iterations. This seems to indicate that the general assumption that the barrier algorithm produces more stable dual values than simplex methods is correct for our instances. We will use the barrier algorithm without cross-over to solve the RMP.

While it is not clear from the tables, most of the running time for almost all instances is spent solving the RMP. For the larger problems, the RMP

[2]In this chapter, we will only test CPLEX's LP algorithms.

Instance	Primal	Dual	Barrier
A320_1w	178	–	103
B737_1w	88	–	73
B757_1w	160	–	94
DC9_1w	63	141	48
DC9_1w2	76	87	67
DC9_10d	141	141	82
DC9_2w	494	–	159
DC9_1m	–	–	567
M82_1w	40	39	30
M82_2w	219	216	86
M82_1m	–	–	398

Table 6.9 Number of column generation iterations required to reach LP optimality with different RMP algorithms.

time completely dominates over the pricing time, accounting for up to 90% of the running time. The reason is the high column density for our instances, as discussed above. To reduce the RMP solution times with the present LP solution methods, attempts have been made to stop the simplex and barrier algorithms prior to reaching optimality in each iteration. For simplex algorithms, the stopping criterion is the reduced cost tolerance, while for the barrier algorithm the complementarity tolerance controls convergence [64, 121]. We have tested simply relaxing the tolerances in the initial column generation iterations, and then gradually tighten them. Unfortunately, no conclusive benefit could be found. While the first column generation iterations were faster, the total convergence was worse. As a further complication, it turned out to be very difficult find strategies to adjust the tolerance dynamically.

A key observation for column generation is that it is not necessary to find a primal solution to the RMP in each iteration. This means that many algorithms, such as subgradient algorithms, could potentially be used to provide a dual solution to the RMP. We have not investigated such algorithms further, but have instead focused our attention on the primal and dual simplex methods, and the barrier algorithm without crossover. One of the main reasons for not investigating methods more focused on duality is that while primal solutions are not necessary in the relaxed column generation phase, they are necessary for our integer fixing heuristics.

6.8 Dual Stabilization

Dual stabilization is often considered to be one of the most important techniques to improve column generation performance [35, 77, 157, 161, 201]. The idea is to stop the dual values from oscillating between iterations, which is considered very bad for convergence.

In [201], Sigurd and Ryan present a simplified version of the technique proposed by du Merle et al. [77], specialized for set partitioning problems. The approach bears some resemblance to the classical `Boxstep` method of Marsten et al. [161]. Sigurd and Ryan show examples of how dual values can vary for a typical unstabilized column generation run, and demonstrate how most of the columns used in the optimal solution are generated in the very last iterations. The first iterations are thus only useful for gradually improving the dual quality. Generating a large number of bad columns leads to longer generation time as well as longer RMP solution time. The goal of dual stabilization is therefore to generate good columns early, by stabilizing the dual values around some initial estimate, and as a consequence improve the column generation convergence.

Consider the linear program

$$
\begin{aligned}
& & \min\ & c^T x, & \\
(P) & & \text{s.t.}\ & Ax = 1, & (6.5)\\
& & & x \geq 0. & (6.6)
\end{aligned}
$$

Now let us add slack and surplus variables, and linear penalties for using these:

$$
\begin{aligned}
& & \min\ & c^T x + \delta_- y_- + \delta_+ y_+, & \\
(P') & & \text{s.t.}\ & Ax - y_- + y_+ = 1, & (6.7)\\
& & & x \geq 0, & (6.8)\\
& & & y_- \geq 0, & (6.9)\\
& & & y_+ \geq 0. & (6.10)
\end{aligned}
$$

Clearly, (P') is a relaxation of (P), and the optimal value of (P') will be less than that of (P) only if some slack or surplus variable is non-zero (and the corresponding δ is negative). Now, the dual of (P') is:

$$
\begin{aligned}
& & \min\ & b^T \pi, & \\
(D') & & \text{s.t.}\ & A^T \pi \leq c, & (6.11)\\
& & & -\delta_- \leq \pi \leq \delta_+. & (6.12)
\end{aligned}
$$

From (D'), it is clear that the slack costs δ_- and δ_+ bound the dual variables. δ_- and δ_+ will thus force the dual variables to take values within a bounding box in each iteration. The only step that has to be added to the standard column generation algorithm to include the dual bounding is that whenever a slack or surplus variable becomes non-zero, the corresponding δ_-^i or δ_+^i is increased. When no more negative reduced cost columns can be generated, and all slack and surplus variables are 0, an optimal solution to (P'), and by the reasoning above also to (P), has been found.

So, how are δ_- and δ_+ initialized, and how are they updated? Sigurd and Ryan [201] do not elaborate much on the initialization, as that is typically a problem-dependent decision. They propose that earlier runs of the problem are used, or some approximation of the solution, but they do not provide many details. They do present more detailed proposals on how to update δ_- and δ_+, though. A dual variable can hit the bound (6.12) either because the optimal value is really outside the bounds, or because not enough columns have been generated. Therefore, it is proposed that if a dual variable π_i hits one of its bounds for several iterations in a row, the corresponding δ_-^i or δ_+^i should be increased. However, the value of the corresponding slack or surplus variable should also be taken into account. If it is close to 0, it may not be necessary to increase the dual variable bound, as the problem is almost primal feasible.

Here, we use the following method to initialize the dual variable bounds: first we use model $\text{T}\textsc{as}^{relax}$ to calculate a relaxed solution to the problem. Using the flow decomposition theorem [8, Chapter 3] we can then obtain a set of columns representing routes in model $\text{T}\textsc{as}^{relax}$, which are potentially illegal, but which form an integer 'solution' to the problem. Given column $x_r^{'}$ from the network flow relaxation, the dual values of the constraints covered by $x_r^{'}$ are approximated by $\pi^{'} = c_r^{'}/|F_r|$, where $c_r^{'}$ is the cost of $x_r^{'}$. The initial dual variable bounds are then set to $[\pi^{'} - \zeta, \pi^{'} + \zeta]$, where ζ is some appropriate value. We have experimented with different values for ζ, as the tests below show.

We use a very simple updating process, which simply increases δ_-^i or δ_+^i by a factor p each time a corresponding slack or surplus variable is non-zero. We have experimented with different choices for p. We have further chosen to run a maximum of 250 column generation iterations on a set of instances which are not the same as the others in this chapter. While 250 iterations is not enough to reach LP optimality for most of these instances, it will still show the initial convergence. Tables 6.10 and 6.11 show the performance of dual stabilization, using $\zeta = 0, p = 10$, compared to no stabilization, when using primal simplex and barrier algorithms for the RMP. It is clear that the stabilization has a large effect when using primal simplex, as it gives a better objective in less time. The speed-up is especially large on DC9_1m2, which is heavily affected by degeneracy without stabilization. However, when using the barrier algorithm there is little or no benefit using stabilization. This is not very surprising, as the barrier algorithm is not affected by degeneracy, and in fact using interior point dual values is in itself considered a stabilizing factor [191].

Tables 6.12 and 6.13 show results when using primal simplex and different

		No stabilization		$\zeta = 0, p = 10$	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
DC9_1m2	69579700	107007	*69782318	18397	*69745411
M83_1m	52911800	2042	*53183087	1035	*53173328
M87_1m2	48009600	1536	*48083240	683	*48078267
DC9_1w3	16558500	61	16558500	41	16558500

Table 6.10 Dual stabilization using primal simplex. At most 250 iterations have been performed.

		No stabilization		$\zeta = 0, p = 10$	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
DC9_1m2	69579700	7175	*69776308	8894	*69754493
M83_1m	52911800	2363	*53086752	2578	*53072311
M87_1m2	48009600	2047	*48060736	2741	*48052383
DC9_1w3	16558500	26	16558500	22	16558500

Table 6.11 Dual stabilization using the barrier algorithm. At most 250 iterations have been performed.

	$\zeta = 5$		$\zeta = 10$		$\zeta = 20$	
Instance	Time	LP obj.	Time	LP obj.	Time	LP obj.
DC9_1m2	22293	*69762277	16957	*69732984	17215	*69747735
M83_1m	907	*53144210	1029	*53170336	909	*53160248
M87_1m2	534	*48077633	647	*48075446	658	*48077633
DC9_1w3	38	16558500	36	16558500	42	16558500

Table 6.12 Dual stabilization using primal simplex, $p = 10$, and varying ζ. At most 250 iterations have been performed.

settings for ζ and p. There is very little difference between the tested settings, and the best overall setting, with respect to both running time and solution quality, seems to be $\zeta = 5, p = 5$. However, since no positive effect of dual stabilization was detected when using the barrier algorithm, which is the best LP algorithm for our instances, we will not use dual stabilization.

	$p = 5$		$p = 20$		$p = 50$	
Instance	Time	LP obj.	Time	LP obj.	Time	LP obj.
DC9_1m2	18007	*69715450	17026	*69761860	22304	*69762277
M83_1m	963	*53157279	900	*53156738	823	*53164189
M87_1m2	548	*48071952	505	*48087570	508	*48091216
DC9_1w3	43	16558500	38	16558500	33	16558500

Table 6.13 Dual stabilization using primal simplex, $\zeta = 5$, and varying p. At most 250 iterations have been performed.

6.9 Constraint Aggregation

In [80], Elhallaoui et al. present an algorithm for *dynamic constraint aggregation* column generation. The idea is to aggregate constraints to obtain a smaller RMP, and then dynamically update the aggregation when necessary during the column generation process. We will start by thoroughly describing the algorithm of Elhallaoui et al., and then describe how the algorithm had to be modified to suit our purpose.

Let us call the set of rows in the original problem W (referred to in [80] as 'tasks'). Given a set of columns C, rows $w_1 \in W$ and $w_2 \in W$ are *equivalent* with respect to C if every column in C covers either both rows, or none of them. Now, it is possible to partition the rows into a set L of *equivalence classes* such that all rows in a certain class are equivalent. The set of rows in class $l \in L$ is W_l, and $Q = \{W_l : l \in L\}$ forms a *partitioning* of the original rows. A column x_k is said to be *compatible* with an equivalence class l if $|W_l \cap A_k| = 0$ or if $|W_l \cap A_k| = |W_l|$, where A_k is the set of rows covered by column x_k.

Now, the idea of the algorithm is to maintain an RMP for some partitioning, where each equivalence class is represented by a row. This RMP is called the *aggregated restricted master problem* (ARMP). Since each equivalence class contains at least one original row, ARMP contains at most as many rows as the original RMP. The pricing problem is unchanged, and to calculate the dual value of an original row from the dual value of a row in ARMP, a special algorithm, described below, is used. In case it would be beneficial to break some equivalence class, e.g. because it would make it possible to add an attractive column, the partitioning is *disaggregated*. Also, the partitioning can be further aggregated to avoid that it is split into singleton rows. The constraint aggregation algorithm of Elhallaoui et al. is shown in Algorithm 6.2.

First, a set of columns that defines an initial partitioning are chosen. In each *inner iteration* (steps 5–10) the current ARMP is solved, dual values for the original rows are calculated as described below, and new columns

Algorithm 6.2 The constraint aggregation algorithm of Elhallaoui et al. [80].

```
 1: procedure ConstraintAggregationColumnGeneration
 2:     $C \leftarrow$ some original columns
 3:     Create partitioning from $C$
 4:     repeat
 5:         repeat
 6:             Solve ARMP
 7:             Compute original dual values
 8:             Generate columns
 9:             Add compatible columns to ARMP
10:         until no column generated or modify()
11:         $I \leftarrow$ nonempty set of incompatible columns
12:         $B \leftarrow$ current non-zero basic columns
13:         if stalling or using slack columns then
14:             Create partitioning from $C \cup L$
15:         else
16:             Create partitioning from $B \cup L$
17:         end if
18:         Refine ARMP to be compatible with new partitioning
19:     until $I$ empty
20: end procedure
```

are generated and added. This is repeated until no negative reduced cost columns are generated, or the `modify()` predicate tells us to stop. We will not discuss here how `modify()` can be implemented. An *outer iteration* starts in step 4 and ends in step 19. Whenever an inner iteration is completed, a number of incompatible columns are picked, which have been stored in step 9, and the partitioning is either disaggregated or aggregated. Disaggregation (enlargement) of the partitioning, by adding the chosen incompatible columns to the current partitioning, is performed if the process is stalling, i.e. if the objective value is the same as in the previous outer iteration, or if some slack column is used in the current solution. If not, the partitioning is aggregated further by creating a new partitioning from the current non-zero basic columns and the chosen incompatible columns.

The most crucial part of the algorithm is the computation of dual values for the original rows from the partitioning dual values. To describe how this is done, denote by $\hat{\alpha}_l$ the ARMP dual variable of partition l. To obtain dual values α_w for the original rows, the linear system

$$\sum_{w \in W_l} \alpha_w = \hat{\alpha}_l, \quad \forall l \in L, \tag{6.13}$$

must be solved. This system has an infinite number of solutions whenever at

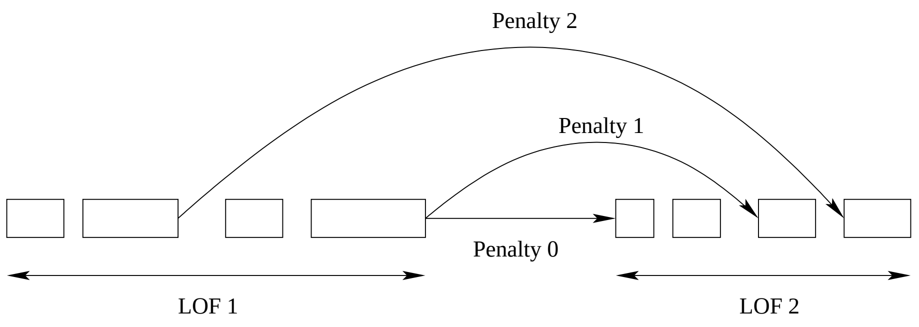


Figure 6.1 Constraint aggregation pricing penalties.

least two rows are aggregated, and a simple solution is

$$\alpha_w = \hat{\alpha}_l / |W_l|, \quad \forall l \in L, \forall w \in W_l, \tag{6.14}$$

However, a better choice would be to let the dual values be such that not only the current basic columns, but also some of the incompatible columns, were priced out. This way, a new set of incompatible columns would be generated in the next iteration. This would diversify the set of generated incompatible columns, which we will later use to re-define the partitioning. So, it is desirable that the dual variables also satisfy

$$\sum_{w \in W} a_{wp} \alpha_w \leq c_p, \quad \forall p \in P, \tag{6.15}$$

where P is a set of incompatible columns. Unfortunately, solving both (6.13) and (6.15) could be very difficult, in fact as difficult as solving the original problem. What Elhallaoui et al. [80] propose instead is to categorize the incompatible columns in six categories. For five of the categories, the problem of finding dual values satisfying equations (6.13) and (6.15) can then be formulated as a shortest path problem. [80] contains a proof that the dynamic constraint aggregation algorithm converges to the optimal solution to the original linear program.

Constraint Aggregation for Path-Tas

Unfortunately, the constraint aggregation algorithm described above does not work well at all for our instances. Very few of the columns which are generated are compatible with the initial partitioning, even when starting from an initial partitioning which is known to contain the optimal solution. Furthermore, very few of the incompatible columns can be categorized as belonging to one of the five ‘nice’ categories, which means that the dual quality is very low, and as a result the convergence is very poor.

Instead, we had more success with a constraint aggregation algorithm that also considers the pricing problem. This way it is possible to generate attractive columns for the current partitioning, rather than generating mostly incompatible columns, and being forced to disaggregate the partitioning right away. First, day-long sequences, similar to LOFs [103, 104], are formed. This can be done either manually or by some automatic method. We have used a method which will be described in Chapter 9 to obtain a relaxed integer solution, which does not satisfy all constraints. Other options could be to use model $\textsc{Tas}^{relax}$, or an existing solution, should one be available.

Remember that our pricing problem is a resource constrained shortest path problem, where activities are nodes, and possible connections between activities are arcs. Here, the network is adjusted in such a way that connections which break the partitioning defined by the LOFs are given a *violation penalty*. Figure 6.1 illustrates the penalty, which is such that connections going to *or* from the middle of a LOF are given a penalty of 1, while connections going from the middle of a LOF to the middle of another LOF are given a penalty of 2. Additionally, a cumulative constraint counting the violation of each generated path is added. In normal pricing iterations no violation arcs may be used, so the constraint will have an upper limit of 0. This will ensure that only columns that are compatible with the current partitioning are generated. To calculate dual values for the original columns, the simplistic solution of equation (6.14) is used. When no more compatible columns can be generated, or the column generation algorithm shows little progress, a very high upper limit is set for the cumulative constraint, and some incompatible columns are generated. A subset of these columns is then selected, and the partitioning is disaggregated to be compatible with these. Dominance combined with the penalty definitions will help ensure that the chosen incompatible columns violate the current partitioning as little as possible. This modified constraint aggregation algorithm is shown in Algorithm 6.3.

The main difference between Algorithms 6.3 and 6.2 is that our algorithm forces the pricing problem to generate compatible columns if such exist. We also use a more simplistic way to calculate the original dual values, and our algorithm currently never aggregates the partitioning. Since the primal variables often take very fractional values for our instances, it is not efficient to use an aggregation scheme like the one proposed in [80], which uses all non-zero basic columns to refine the partitioning. But it would probably be possible to use a procedure similar to *connection fixing*, see e.g. [28], to gradually aggregate the partitioning. However, this has not been tested.

Tables 6.14 and 6.15 show that the constraint aggregation gives speed-ups for most instances both when using primal simplex and when using the barrier algorithm. The solutions obtained when using constraint aggregation are a bit worse in terms of quality, but on the other hand they are obtained quicker. The exceptions are the tiny DC9_1w3 instance, where the overhead of disaggregation, along with the improved number of iterations required to

Algorithm 6.3 Our constraint aggregation algorithm.

```
1: procedure PathTasConstraintAggregation
2:     Create LOFs
3:     Create partitioning from LOFs
4:     Set pricing penalty costs from LOFs
5:     repeat
6:         repeat
7:             Solve ARMP
8:             Calculate original dual values according to equation (6.14)
9:             Set penalty limit to 0
10:            Generate columns
11:            Add columns to ARMP
12:        until no columns generated
13:        Set penalty limit to some large value $M$
14:        Generate columns $I$
15:        Choose a few incompatible columns $I' \subseteq I$
16:        Refine partitioning to be compatible with $I'$
17:        Refine ARMP to be compatible with new partitioning
18:    until $I$ empty
19: end procedure
```

		Normal		Dyn. Constr. Aggr.	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
DC9_1m2	69579700	107007	*69782318	256892	*70107601
M83_1m	52911800	2042	*53183087	1140	*53340246
M87_1m2	48009600	1536	*48083240	678	*48161987
DC9_1w3	16558500	61	16558500	138	16558500

Table 6.14 Dynamic constraint aggregation using primal simplex. At most 250 iterations have been performed.

		Normal		Dyn. Constr. Aggr.	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
DC9_1m2	69579700	7175	*69776308	5932	*69845912
M83_1m	52911800	2363	*53086752	1158	*53073234
M87_1m2	48009600	2047	*48060736	957	*48162556
DC9_1w3	16558500	26	16558500	57	16558500

Table 6.15 Dynamic constraint aggregation using the barrier algorithm. At most 250 iterations have been performed.

reach LP optimality, makes constraint aggregation substantially slower than the original algorithm, and the DC9_1m2 instance, which stalls and is affected by a lot of degeneracy when solved with primal simplex.

The effect of constraint aggregation is typically large at the beginning of the process, and gradually decreases as the problem gets closer to optimality, when more and more disaggregation becomes necessary. To illustrate this, Table 6.16 shows the time it takes to get within 1% of the lower bound, when using the barrier algorithm. It is clear that especially for the larger two instances, the effect of constraint aggregation is significant in speeding up the initial column generation convergence. To further illustrate the improved convergence on large instances, Table 6.17 shows the effect of constraint aggregation when performing at most 100 iterations on a set of larger instances. Clearly, constraint aggregation gives better solutions much faster than before for these instances. It should be noted that the extremely large gaps for these problems are partly due to poor convergence, and partly due to the lower bounds being of poor quality.

However, due to the poor convergence properties when optimality is approached, and the fact that constraint aggregation cannot be trivially combined with the integer fixing heuristics we will use, we have selected not to use constraint aggregation as a default option in our column generation implementation.

6.10 Summary

In this chapter we have presented various details and acceleration strategies for our column generation implementation. Some of the strategies have a huge impact on performance, while others influence less. We have shown that starting from an initial integer solution does not give an immediate benefit, as the dual values tend to become quite extreme when initialized with a feasible integer solution. Obtaining more stable dual values takes quite a few iterations, and many times the benefit of starting from an initial feasible solution is then

	Normal	Dyn. Constr. Aggr.
Instance	Time	Time
DC9_1m2	3016	51
M83_1m	993	202
M87_1m2	196	105
DC9_1w3	7	15

Table 6.16 Time to get within 1% of the lower bound, using the barrier algorithm.

		Normal		Dyn. Constr. Aggr.	
Instance	TAS^{relax}	Time	LP obj.	Time	LP obj.
MDS_1m3	103359000	5439	*7608918659	594	*371490571
MDS_1m4	93679500	4653	*5432467096	468	*349372913
MDS_1m5	83044500	4438	*4825090366	969	*1701088365
MDS_1m6	48630700	4646	*4222444533	768	*719844441

Table 6.17 Dynamic constraint aggregation using the barrier algorithm, larger instances. At most 100 iterations have been performed.

already gone.

We have investigated how to improve the column generation convergence by generating many columns per aircraft and iteration, and how the resulting running time increase can be tackled by removing non-basic columns from the RMP. Removing columns gives slightly worse convergence, but considering the running time improvements, the overall effect is still positive. We have also shown how the use of a dual value re-evaluation scheme can help us to generate less incompatible columns, and as a consequence substantially accelerate convergence as well as decrease running times.

We also tested different LP algorithms to solve the RMP, and found that the barrier algorithm without crossover often outperforms primal simplex on large instances. Also, dual simplex turned out to be a really bad choice, even though much literature indicates it should work well for problems of the kind we are dealing with. Comparing Tables 6.3 and 6.8 it is clear that the improvements we have made to the column generation convergence in this chapter are substantial. With the most basic approach, LP optimality could only be reached within 1000 iterations for three of the test instances. Using all the acceleration techniques, except dual stabilization and constraint aggregation, we are able to solve all test instances to LP optimality within at most 567 column generation iterations.

A dual stabilization algorithm was tested. The algorithm attempts to control the oscillation of the dual values during the process by adding dynamic bounds to them, with the aim of improving convergence. In our tests, column generation using primal simplex was improved both in terms of running time, and in terms of solution quality, when using dual stabilization. Unfortunately, when using a barrier algorithm to solve the RMP, which is much faster for the test instances, no positive effect at all can be observed when using dual stabilization.

Finally, a constraint aggregation algorithm was presented and tested. The idea of this algorithm, which is a specialization of the algorithm presented by Elhallaoui et al. [80], is to aggregate several constraints into a single constraint, to speed up the RMP solution, and dynamically refine the aggregation to make sure that the optimal solution is found. Our tests indicate that this can be a very powerful method, especially to improve the initial convergence. For some of the larger instances tested, the first 100 column generation iterations were up to 10 times faster, while obtaining substantially better solution quality. However, convergence problems closer to optimality, and the difficulties in combining constraint aggregation with our integer fixing heuristics, have forced us not to use constraint aggregation as a default option.

To summarize, the default column generation strategy, with respect to the techniques described in this chapter, is the following:

- the RMP is initialized only with slack columns;
- in each column generation iteration, at most 10 columns per aircraft is generated;
- for the test instances used here, all columns with reduced cost $> 10^6$ will be removed in each iteration. For other types of instances, this default value should be be tuned properly;
- we use dual value re-evaluation to reduce the problem of generating identical columns, using $\xi = 10.0$;
- we use the barrier algorithm to solve the RMP;
- we do not use dual stabilization or constraint aggregation, as they do not work well overall, and are difficult to use together with our integer heuristics.

We have in this chapter only used column generation to solve model Path-Tas^{relax}, the LP relaxation of model Path-Tas. However, since the column generation algorithm is not at all focused on integrality, an optimal LP solution is in fact of limited usefulness for the purpose of finding integer solutions. The columns used in the relaxed solution might be completely different from

Instance	TAS^{relax}	Time	LP obj.
A320_1w	628145	1966	628145
B737_1w	4969800	557	4973650
B757_1w	4385060	919	4388950
DC9_1w	12101800	28	12101800
DC9_1w2	16662800	46	16662800
DC9_10d	21314200	128	21314300
DC9_2w	33062400	429	33062500
DC9_1m	69513000	1900	*71170621
M82_1w	11984100	6	11984100
M82_2w	23628800	138	23628800
M82_1m	52775500	1193	*55025956

Table 6.18 Performance of the column generation algorithm when performing at most 100 iterations.

the columns needed to obtain an integer solution. However, one thing we do gain by solving the problem to LP optimality is a lower bound. Since model TAS^{relax} is a lower bound even to the non-integer multi-commodity flow problem, it is a lower bound to the optimal LP value of PATH-TAS. This means that the optimal solution to PATH-TAS^{relax} might provide a stronger lower bound than the network flow relaxation. From Table 6.8, we see that the optimal LP objective value slightly improves the network flow lower bound in five of the test cases.

The price, as we have seen, is substantially longer running times. In fact, even with the most improved version (Table 6.8) it takes an unreasonably long time to reach LP optimality for the largest instances. Since in most cases, an LP lower bound is not required, we will normally not run to LP optimality, but instead stop after at most 100 column generation iterations. Table 6.18 shows the running times and LP objectives reached when at most 100 iterations are used. The running times are a lot more reasonable than when running to optimality, while the LP objectives are still quite good. And as we will see in the next chapter, we will still be able to find integer solutions of reasonable quality. In fact, with the integer fixing heuristics we use, nothing prevents us from finding integer solutions with objective values better than the LP objective we reach using the column generation algorithm.

There are many possible acceleration techniques and parameter tunings that are not covered here, e.g. how many labels to save in the pricing problem, how to use dominance, and so on. Many of these things have in fact been investigated, but have not been found important enough to include here.

Seven

Solving the Integer Program

Once a good enough LP solution has been found by the column generation algorithm, the problem of finding an integer solution remains. One of the pioneers in using column generation for integer programming was Appelgren [15], who combined column generation with integer heuristics to find solutions to ship routing problems. Unfortunately, the early efforts by Appelgren seem to have been largely forgotten, and most recent work on using column generation for integer programming can instead be traced to the work of Barnhart et al. [28] or Vanderbeck [218], who propose the names *branch-and-price* and *IP column generation*, respectively, for the combination of column generation and branch-and-bound. While the two approaches are essentially the same, we will use the name branch-and-price.

Branch-and-bound is a systematic search algorithm for integer programs, which recursively partitions the search space (using so-called *branching*), and uses some bound, typically obtained from the linear programming relaxation, to avoid searching parts of the search space. The recursive search space partitioning is often visualized as a *search tree*, where each branching represents a partitioning of the search space. The basic branch-and-bound algorithm, which is often attributed to Land and Doig [148], has in recent years been improved in various ways [40], and is available in all commercial IP solvers [64, 121]. As Lübbecke and Desrosiers [157] observe, the branch-and-price algorithm is really a branch-and-bound algorithm in which column generation is used to calculate bounds during the search.

However, branch-and-price is quite complex to implement, for example requiring careful management of the generated columns, since not all columns are valid in all branches of the search tree. Further, branch-and-price is likely too slow for our purposes, especially considering that we most often do not need a proven optimal solution. Instead, for our purposes, being able to find good solutions within a reasonable amount of time for large instances is more important than optimality.

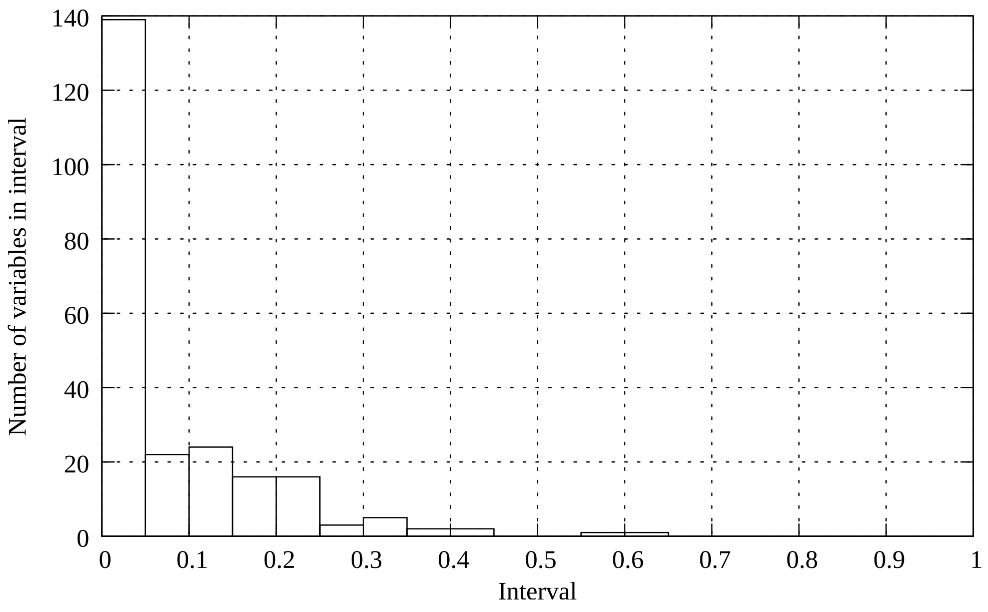


Figure 7.1 Spread of the primal variable values in an optimal LP solution produced by our column generation algorithm for a typical instance. For clarity, variables taking the value 0 are *not* included in the figure.

The easiest way, in terms of implementation complexity, to obtain an integer solution, is to directly run an integer programming solver on the resulting LP after the column generation algorithm. Not surprisingly, this has turned out not to work well at all for our instances. For large instances, it is often difficult to find *any* integer feasible solution covering all activities using only the columns of the LP generated by the column generation algorithm. The fundamental reason for this is that there is no guarantee that the columns generated are enough, or even useful, for finding an integer solution [119, 146, 157, 160, 218]. This well-known phenomenon is very clear for our instances, which due to their combinatorial nature normally contain very few variables with values close to 1 in the optimal LP solutions. Figure 7.1 shows how the variable values are typically spread in an optimal LP solution.

Instead of using branch-and-price, we have chosen to use a heuristic algorithm. Our heuristic iteratively restricts the search space and re-generates columns, until an integer solution is found. If an infeasible branch or a branch for which the lower bound is poor is detected, limited heuristic backtracking can be performed, but the idea of using a heuristic is of course that backtracking should be avoided. The general heuristic algorithm, described independently of the type of search space restriction used, is shown in Algorithm 7.1. As we will discuss below, search space restriction is achieved by *fixing* routes

Algorithm 7.1 The general integer fixing heuristic.

data

Solve(m): A function that solves model m, returning the optimal objective function value

RestrictAndEnforce(x): A function that restricts the solution space, and enforces the restrictions in the RMP and in the pricing problem

ColumnGeneration: Algorithm 6.1, returns a relaxed solution, possibly an improved lower bound, and an objective function value

Integral(x): Returns **true** if x is integral, **false** otherwise

$\sigma_{initial}$: Number of column generation iterations to perform before starting heuristic

σ: Number of column generation iterations to perform within each restriction iteration

ϵ: Relative optimality tolerance

z_u: Known upper bound on integer objective, or ∞

δ: Number of restrictions to undo when backtracking

```
1:  procedure IntegerFixingHeuristic
2:      $\underline{z} \leftarrow$ Solve(Tas$^{relax}$)          ▷ $\underline{z}$ is lower bound of LP solution $\overline{x}$
3:      $(\overline{x}, \underline{z}, \overline{z}) \leftarrow$ ColumnGeneration($\sigma_{initial}, \underline{z}$)
4:      loop
5:          RestrictAndEnforce($\overline{x}$)
6:          $\underline{z}' \leftarrow$ Solve(Tas$^{relax}$)
7:          $(\overline{x}, \underline{z}', \overline{z}) \leftarrow$ ColumnGeneration($\sigma, \underline{z}'$)
8:          if Integral($\overline{x}$) and $\overline{z} < z_u$ and $100 \times \frac{\overline{z}-\underline{z}}{\underline{z}} < \epsilon$ then
9:              return $(\overline{x}, \underline{z}, \overline{z})$
10:         else if Integral($\overline{x}$) or $\overline{z}' \geq z_u$ then
11:             Undo the last $\delta$ solution space restrictions
12:         end if
13:     end loop
14: end procedure
```

or connections which must be used in a solution. We will therefore call our heuristic a *fixing heuristic*, and use the term *fixations* to denote the search space restrictions. Each main loop in Algorithm 7.1, steps 4–13, will be called a *fixing iteration*.

Observe that the only thing that differentiates Algorithm 7.1 from branch-and-price is the fact the the solution space is *restricted* rather than *partitioned* in step 5. The fact that our search space restriction is heuristically constructed means that backtracking cannot be performed in an exact fashion. Instead, we use heuristic backtracking combined with random perturbations to avoid re-exploring the same part of the search tree several times. The backtracking step will be discussed further in Section 7.4. Also note that we do not necessarily have to start the fixing heuristic from an optimal LP solution to Path-Tas. As discussed in the previous chapter, to get acceptable running times we normally only perform a fixed number of column generation iterations. In step 6 of Algorithm 7.1, a *local* lower bound is calculated to the restricted problem, which is used to monitor the convergence of the column generation subprocess.

Just as for the column generation process for the relaxed problem, there are a lot of crucial algorithmic choices available in Algorithm 7.1. The most obvious question is of course how to restrict the solution space. This will be the main topic of this chapter – in the following three sections, we will discuss fixing heuristic variants that give different behavior in terms of running time and solution quality. Since we use the column generation algorithm as a subroutine, all the parameters for standard column generation will also have an impact on the integer heuristic performance. The criteria for when to backtrack, and the backtracking itself, will be discussed in Section 7.4. Finally, in Section 7.5, we summarize this chapter.

Testing Methodology

In this chapter, we have used $\sigma_{initial} = 100$, i.e. 100 column generation iterations have been performed before the fixing heuristic is started. The overall focus of the tests have been to find solutions which are as good as possible within as short time as possible. The most important requirement is that no activities are allowed to be left unassigned. No backtracking has been used, to show the ability of the fixing heuristic variants to find solutions with no unassigned activities without backtracking.

The result tables show the Tas^{relax} lower bounds, running times in seconds, IP objectives obtained, and the total slack costs included in the IP objectives. The slack costs indicate whether any activities have been left unassigned, so only slack costs of 0 are considered acceptable here.

The test instances are the same as those used in Chapter 6, and are presented in Table 6.1.

Algorithm 7.2 RESTRICTANDENFORCE for the variable fixing heuristic.

```
data
    Recall that $F_r$ is the set of rows covered by column $r$.

1: procedure VARIABLEFIXRESTRICTANDENFORCE($\overline{x}$)
2:     Choose the variable $\overline{x}_i$ closest to 1, and fix it to 1
3:     Remove all columns $j$ s.t. $F_i \cap F_j \neq \emptyset$ from RMP
4:     $\overline{x}_i \leftarrow 1.0$
5:     Remove arcs from/to activities in $F_i$ from pricing network
6: end procedure
```

7.1 A Variable Fixing Heuristic

Perhaps the most obvious integer fixing technique is a rounding, or variable fixing, heuristic. Such a heuristic works similarly to standard variable branching in branch-and-bound, and fixes variables to integer values until an integer solution is found. Since the fixing is performed in a greedy fashion, and backtracking should be avoided, it is a good idea to fix variables which take on close to integral values. This is contrary to common practice for branch-and-bound, where balanced search trees are formed by branching on variables which take values far from integrality.

Another difference to branch-and-bound is that we do not want to fix variables to 0. Firstly, this is not a very limiting solution space restriction, and secondly it is often difficult to enforce such a restriction in the pricing problem. The fixations we make must be possible to enforce in the pricing problem, or columns conflicting with previous fixations might be generated. By simply removing connections in the pricing network, it is easy to fix a variable to 1 in the pricing problem. On the other hand, fixing a variable to 0 is very difficult. It must be made sure that the route representing the variable in the pricing network is never generated, while other routes, consisting of parts of the route representing the variable, are allowed. While this is possible by introducing extra resources in the pricing problem, it becomes extremely difficult to combine it with properties needed for the pricing problem, such as guaranteeing that a negative reduced cost path will always be generated if one exists. The performance of the pricing problem would likely also be severely deteriorated. In fact, this difficulty makes variable branching practically unusable in a branch-and-price context [28]. Since all variables in PATH-TAS are restricted to the interval $[0, 1]$, procedure RESTRICTANDENFORCE in Algorithm 7.1 is implemented as illustrated in Algorithm 7.2 for the variable fixing heuristic.

It should be observed that there are other fixing criteria than the primal value criterion in Algorithm 7.2. Several criteria, such as minimum reduced

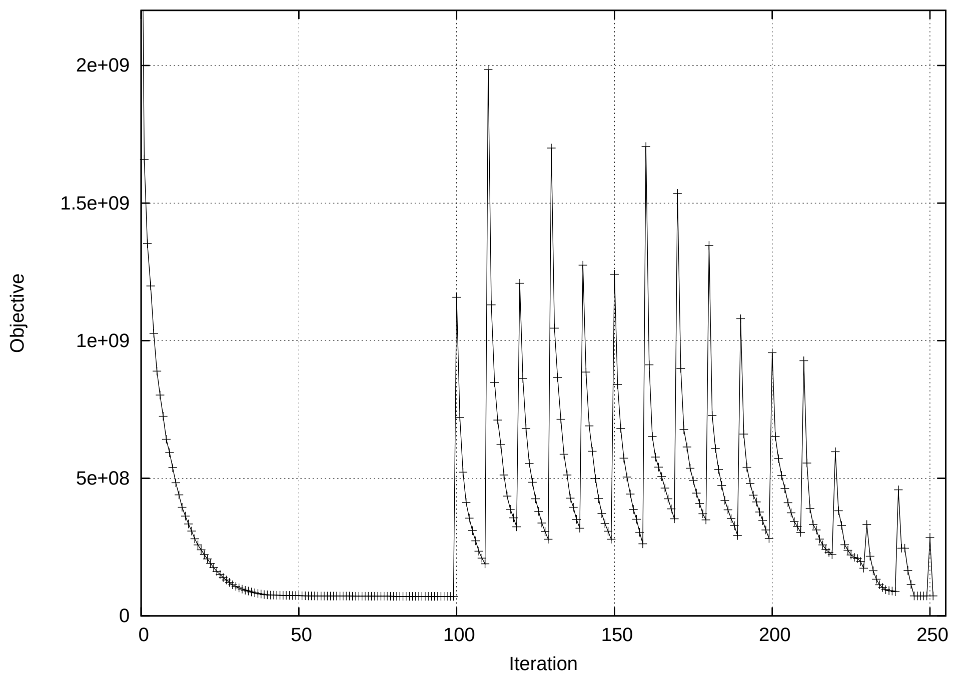


Figure 7.2 The convergence of the variable fixing heuristic, for a typical run.

cost, minimum cost, or maximum or minimum row coverage, have been tested but worked too poorly to be included here [107].

Performance

The main problem with the variable fixing heuristic is the performance. Obviously, the performance depends on many factors apart from the fixing criterion, the most important one being how many column generation iterations (σ) are performed in each fixing iteration. The linear programs generated by the column generation algorithm are, as mentioned above, very fractional. Often only a few variables have values above 0.5 in the optimal LP solution. Fixing one variable therefore makes many other variables incompatible, as they cover many of the rows which the fixed variable covers. The objective function thus makes a jump when a variable is fixed, as depicted in Figure 7.2, which it takes many iterations to recover from. Using a reasonable number of column generation iterations in each fixing iteration, so as not to increase running times enormously, it is very difficult to find good solutions with the variable fixing heuristic without backtracking. Table 7.1 shows the performance of the heuristic on the test instances, using $\sigma = 10$ and $\sigma = 30$, when no backtracking is performed.

The table shows lower bounds, running times, objective values, and slack

		$\sigma = 10$		
Instance	TAS^{relax}	Time	IP obj.	Slack cost
A320_1w	628145	4288	657847	0
B737_1w	4969800	1688	17166350	12000000
B757_1w	4385060	1691	4446950	0
DC9_1w	12101800	62	12101800	0
DC9_1w2	16662800	76	16662800	0
DC9_10d	21314200	224	21741200	0
DC9_2w	33062400	602	42419200	4000000
DC9_1m	69513000	2464	251802400	156000000
M82_1w	11984100	10	11984100	0
M82_2w	23628800	188	23699700	0
M82_1m	52775500	1291	107215900	43000000
		$\sigma = 30$		
Instance	TAS^{relax}	Time	IP obj.	Slack cost
A320_1w	628145	4899	628145	0
B737_1w	4969800	2604	4973650	0
B757_1w	4385060	1806	4388950	0
DC9_1w	12101800	71	12101800	0
DC9_1w2	16662800	82	16662800	0
DC9_10d	21314200	345	21314500	0
DC9_2w	33062400	1310	33749400	0
DC9_1m	69513000	4347	84658000	12000000
M82_1w	11984100	10	11984100	0
M82_2w	23628800	282	23628800	0
M82_1m	52775500	2149	64930600	6000000

Table 7.1 Performance of the variable fixing heuristic, using $\sigma = 10$ and $\sigma = 30$.

costs. The slack costs are included in the IP objectives. It is clear that for some of the instances, all activities could not be assigned even when many columns were generated in each fixing iteration. Since solutions using no slack columns exist for all instances, using slack columns is not acceptable. In the following sections we will look at how the variable fixing heuristic can be used despite its poor performance. Often, the type of variable fixing heuristic we have described here is used to initially fix variables which take on values close to integer. This approach is e.g. used by Gamache and Soumis [95] to solve the crew rostering problem. Borndörfer et al. [42] use a *probing* heuristic, which tentatively applies fixations and explores their consequences, to solve duty planning problems in public transportation. Their method is more advanced than the one described here, and uses complex scoring and probing techniques similar to *strong branching* [121] to select candidates for fixing.

7.2 A Connection Fixing Heuristic

Branching on connections, or follow-ons, has been proposed quite frequently in recent years for many kinds of airline planning problems, both in the crew and fleet areas [26, 28, 87, 157, 218], and has been successful in most cases. Our main argument for fixing connections rather than variables is that it is a less aggressive method, and therefore less inclined to cause infeasibilities during the search. A simple way to explain why this is the case is to observe that fixing a variable is identical to fixing all the connections in the route corresponding to the variable. So, if the route contains e.g. 52 activities, fixing the variable corresponds to simultaneously fixing all 51 connections in the route. Clearly, to instead fixate them in several steps and generate new columns in between, decreases the risk of causing infeasibility.

Observe that fixing connections in model Path-Tas is equivalent to fixing variables in model Tas. Model Tas is our so-called *compact representation*, while Path-Tas is an *extensive representation* [157]. In this sense, the connection fixing heuristic is also a variable fixing heuristic; it just fixates variables in the compact representation. Algorithm 7.3 shows procedure RestrictAndEnforce for the connection fixing heuristic.

Fixed connections are enforced in the master problem by removing all the columns that only cover one of the two activities, and all columns that cover both activities, but having other activities in between. In the pricing network, all connections from the incoming activity except the one to be fixed are removed, and correspondingly for the outgoing activity. Unlike the variable fixing heuristic, it is in this case simple to enforce both a connection fixed to 1 and one fixed to 0 in the master and pricing problems, which makes connection branching a better choice than variable branching in a branch-and-price algorithm [28, 218].

Algorithm 7.3 RestrictAndEnforce for the connection fixing heuristic.

```
1: procedure ConnFixRestrictAndEnforce($\overline{x}$)
2:     Select for fixing connections between activities represented by rows $s$
           and $t$ for which $\sum_{j \in J} \overline{x}_j \geq 1.0 - \epsilon$ holds for some small $\epsilon$, where
           $J = \{j | a_{sj} = 1 \text{ and } a_{tj} = 1\}$.
3:     Remove from RMP all columns $k$ s.t. $(a_{sk} = 1 \text{ and } a_{tk} = 0)$ or $(a_{sk} = 0$
           and $a_{tk} = 1)$ for any selected pair $(s, t)$
4:     Remove from the pricing network all arcs from $s$ except the one to $t$,
           and all arcs to $t$ except the one from $s$
5: end procedure
```

Proofs of Existence

We have now outlined the connection fixing procedure, but we have not proven that there always exist connections which are candidates to be fixed. The following theorem proves that for any non-integral solution, rows that fulfill the necessary conditions for being fixed exist:

Theorem 7.1. *For a non-integral solution to an LP relaxed set partitioning problem*

$$\begin{aligned} &\min\ c^T x, \\ &s.t. \quad Ax = \mathbf{1}, \\ &\qquad\quad x \in \{0,1\}, \end{aligned}$$

there always exist rows s and t such that

$$0 < \sum_{j \in J} x_j^{'} < 1,$$

where x' is the optimal relaxed solution, and

$$J = \{j | A_{sj} = 1 \text{ and } A_{tj} = 1\}.$$

Proof. See Barnhart et al. [28]. □

In [106], Grönkvist shows how the theorem can easily be extended to the set covering problem. The following theorem shows that in our case, there always exist two rows s and t, as in Theorem 7.1, with a legal connection between them:

Theorem 7.2. *For any non-integral solution to a relaxed set partitioning problem, where each column corresponds to a path in a network and each row corresponds to a node, there exist two rows s and t such that*

$$0 < \sum_{j \in J} x_j^{'} < 1,$$

where x' is the optimal relaxed solution,

$$J = \{j | A_{sj} = 1 \text{ and } A_{tj} = 1\},$$

and such that there exists an arc in the network between nodes s and t.

Proof. The proof is similar to that of the general case ([28]). Pick a basic column which has a non-integral optimal primal value, say column p. Let s' be any row covered by column p. Since p is non-integral and the basis cannot contain two identical columns, there exists another basic column $p' \not\equiv p$ that also covers row s'. Since both columns cover row s', i.e. their respective paths contain node s', but the paths are non-identical, there must exist nodes s and t such that node s is included in both paths, and node t is adjacent to s in only one of them. This means that

$$1 = \sum_k A_{s'k} x_k^{'} = \sum_{k: A_{s'k}=1} x_k^{'} > \sum_{k: A_{sk}=1, A_{tk}=1} x_k^{'},$$

where $x^{'}$ is the optimal relaxed solution. □

Since an arc in the connection network in our case always corresponds to a legal connection, Theorem 7.2 states that there exists a legal connection between activities s and t. This means that as long as the relaxed solution is non-integral, there exists at least one connection that can be fixed. In the rest of this section, we will show results of using the connection fixing heuristic both statically, i.e. fixing as much as possible with a static limit, and as a dynamic process, where the limit is adjusted when no good candidates for fixing can be found.

Connection Fixing with a Static Limit

In [106], Grönkvist presents a fixing heuristic for set partitioning and covering problems from crew pairing and crew rostering. The heuristic is based on a generalization of the connection fixing idea, where not only fixations between follow-ons are allowed, but between arbitrary activities possibly not in sequence. Grönkvist uses a static limit, and the problem is iteratively re-solved and fixed until nothing more can be fixed. In that context no column re-generation is performed, since the problem is a static integer programming problem. However, this heuristic inspired us to test a fixed small $\epsilon > 0$ for the connection fixing heuristic. The strategy is thus to fix all connections satisfying

$$\sum_{j \in J} \overline{x}_j \geq 1.0 - \epsilon \tag{7.1}$$

for the fixed value of ϵ, re-generate columns and re-solve the LP until no more connections satisfy equation (7.1). The variable fixing heuristic is then used to finish the search.

Since this heuristic leaves the crucial final fixations to the variable fixing heuristic, it does not perform very well. Previous results, which can be found in [107], show that while this heuristic gives better solutions than the variable fixing heuristic, it still cannot solve all test instances without using slack columns.

Connection Fixing with a Dynamic Limit

To be able to use only the connection fixing heuristic, and not having to rely on the variable fixing heuristic for the last crucial fixations, a dynamic limit can be used. The dynamic limit is decreased when no candidate for fixing can be found within the current limit. Observe that Theorem 7.2 shows that it is possible to fix connections until an integer solution is found. This section presents such a heuristic, where ϵ is increased by a fixed factor p each time no candidate for fixing can be found with the current value of ϵ. Sometimes it is necessary to increase the value of ϵ quite a lot in order to reach integrality, and then an additional complication must be considered. When $\epsilon \geq 0.5$, it is possible to find several fixing candidate connections from, or to, the same activity, thus causing a conflict. To make sure this does not happen, the algorithm is restricted to fixing at most one connection at a time when $\epsilon \geq 0.5$. Algorithm 7.4 shows the necessary adjustments of Algorithm 7.1.

This connection fixing heuristic is less sensitive to the number of column generation iterations in each fixing iteration than the variable fixing heuristic. The problem is that it is difficult, if not impossible, to estimate the computational effort required to reach integrality. It is difficult to say how many connections need to be fixed before an integer solution is obtained. Of course, the number of connections which can be fixed is bounded by the number of activities, since at most one connection from each activity can be fixed. But even so, it is difficult to know how many *fixing iterations* will be required to reach an integer solution, as that depends on how many connections are fixed in each iteration.

Table 7.2 shows the performance of the connection fixing heuristic, with $\sigma = 5$, $\epsilon_{start} = 0.05$, and $p = 1.1$. Apart from the lower bound, running time and objective value, Table 7.2 shows the number of fixing iterations required to reach integrality. Comparing the results with those in Table 7.1 it is obvious that the connection fixing heuristic performs a lot better than the variable fixing heuristic in terms of solution quality. Solutions very close to lower bounds, without any active slack columns, are found for all instances. In fact, comparing to LP lower bounds in Table 6.8, it is clear that proven optimal solutions have been found for seven of the eleven instances. However, the running times are generally higher than for the other methods, especially for instances where many fixing iterations are performed. Results when using different values of ϵ_{start} and p can be found in [107]. Figure 7.3 shows how many connections are fixed per iteration for a typical run. It is clear that in

Algorithm 7.4 RestrictAndEnforce for the connection fixing heuristic with a dynamic choice of ϵ.

```
1: procedure DynamicConnFixRestrictAndEnforce($\overline{x}$)
2:     $q \leftarrow \infty$
3:     $\epsilon \leftarrow \epsilon_{start}$
4:     loop
5:         Select for fixing at most $q$ connections between activities represented by rows $s$ and $t$ for which $\sum_{j \in J} \overline{x}_j \geq 1.0 - \epsilon$ holds, where $J = \{j | a_{sj} = 1 \text{ and } a_{tj} = 1\}$.
6:         if no connections were selected then
7:             $\epsilon \leftarrow \epsilon * p$, for a suitable value of $p$
8:             if $\epsilon \geq 0.5$ then $q \leftarrow 1$
9:             end if
10:        else break
11:        end if
12:    end loop
13:    Remove from RMP all columns $k$ s.t. $(a_{sk} = 1$ and $a_{tk} = 0)$ or $(a_{sk} = 0$ and $a_{tk} = 1)$ for any selected pair $(s, t)$
14:    Remove from the pricing network all arcs from $s$ except the one to $t$, and all arcs to $t$ except the one from $s$
15: end procedure
```

Instance	TAS^{relax}	#Fixing iter's	Time	IP obj.	Slack cost
A320_1w	628145	85	2550	628145	0
B737_1w	4969800	70	1980	4978250	0
B757_1w	4385060	83	1196	4388950	0
DC9_1w	12101800	32	55	12101800	0
DC9_1w2	16662800	35	59	16662800	0
DC9_10d	21314200	67	200	21314300	0
DC9_2w	33062400	70	1289	33062700	0
DC9_1m	69513000	110	12933	70404300	0
MD82_1w	11984100	24	10	11984100	0
MD82_2w	23628800	36	156	23628800	0
MD82_1m	52775500	117	7241	52791700	0

Table 7.2 Performance of the connection fixing heuristic.

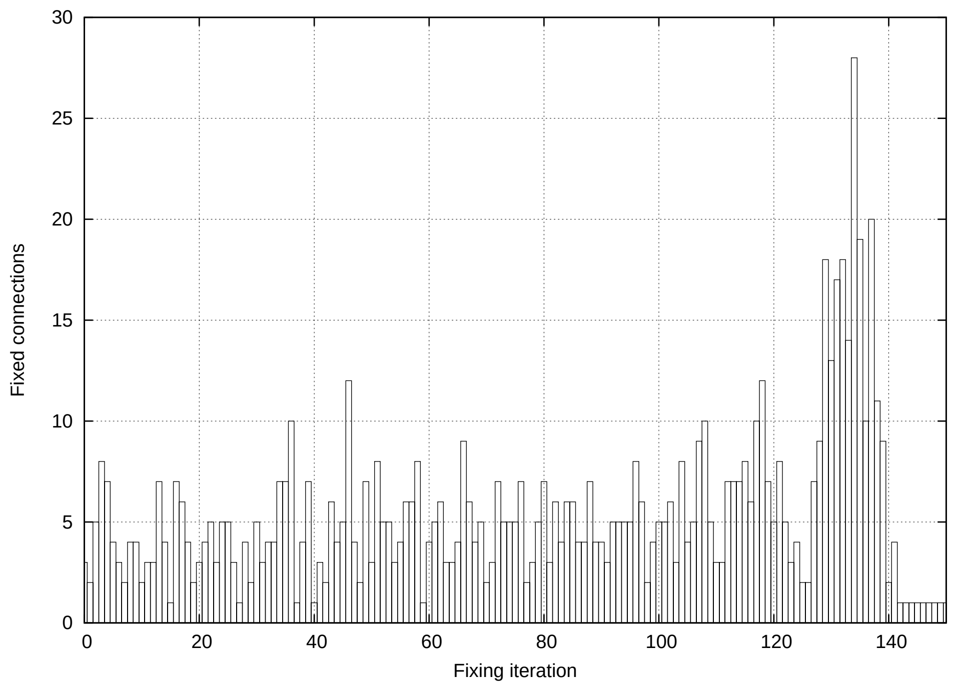


Figure 7.3 Number of fixed connections per iteration for a typical run of the connection fixing heuristic.

many iterations, very few connections are fixed, which as mentioned above is one reason why connection fixing is a lot slower than variable fixing.

Fixing a Relative Amount of Connections

Another very natural strategy is not to use a limit on the sum of the primal values to decide which connections to fix, as in the two strategies above, but to instead fix a certain percentage of the connections at each step, as Algorithm 7.5 shows. With such a strategy, the primal variable values would still be used to guide which connections to fix first. A benefit of such a strategy is that it would give more control over how many iterations are required to reach an integer solution. However, this fixing strategy requires more computational effort than the previously described strategies, since it requires the values for all connections to be calculated and sorted. Because of the added computational effort, we have decided not to test this fixing strategy further.

7.3 A Hybrid Fixing Heuristic

From the previous sections it is clear that the variable fixing heuristic is rather fast, but often too aggressive, and that the connection fixing heuristic delivers

Algorithm 7.5 RestrictAndEnforce for the relative connection fixing heuristic.

```
data
    $r$: Percentage of connections to fix in each iteration

1: procedure RelativeConnFixRestrictAndEnforce($\overline{x}$)
2:     Select for fixing the $r\%$ of the connections for which
           $\sum_{j \in J} \overline{x}_j, \quad J = \{j | a_{sj} = 1 \text{ and } a_{tj} = 1\}$ is largest
3:     Remove from RMP all columns $k$ s.t. $(a_{sk} = 1$ and $a_{tk} = 0)$ or $(a_{sk} = 0$
           and $a_{tk} = 1)$ for any selected pair $(s, t)$
4:     Remove from the pricing network all arcs from $s$ except the one to $t$,
           and all arcs to $t$ except the one from $s$
5: end procedure
```

Algorithm 7.6 RestrictAndEnforce for the hybrid fixing heuristic.

```
data
    VariableFixRestrictAndEnforce: Algorithm 7.2
    DynamicConnFixRestrictAndEnforce: Algorithm 7.4
    $V$: Number of variables to fix before starting connection fixing

1: procedure HybridRestrictAndEnforce($\overline{x}$)
2:     static $totalCalls \leftarrow 0$
3:     if $totalCalls \leq V$ then
4:         VariableFixRestrictAndEnforce($\overline{x}$)
5:     else
6:         DynamicConnFixRestrictAndEnforce($\overline{x}$)
7:     end if
8:     $totalCalls \leftarrow totalCalls + 1$
9: end procedure
```

high quality solutions, but results in long running times for the large instances in which we are most interested. This raises the question whether it could not be possible to combine them, to obtain a heuristic that is both fast *and* provides high quality solutions.

The hybrid heuristic we propose (Algorithm 7.6) is to first use variable fixing to quickly fix a rather large portion of the problem, and then continue with connection fixing. The only additional parameter to this heuristic, apart from the parameters to the variable- and connection fixing subroutines, is a specification V of how many variables should be fixed before switching to connection fixing.

Table 7.3 shows results when running the hybrid heuristic, fixing 20 and 60% of the variables before starting connection fixing, and using $\sigma = 5$. For the connection fixing, the same parameters as in Table 7.2 have been used. Clearly, the results are almost as good or better than pure connection fixing when fixing only a few variables, but get worse as the variable fixing becomes more dominating. It is interesting to note that the objective value of the A320_1w instance when fixing 60% of the variables is very much larger than the optimal value, even though no activities are left unassigned. This is an example of the phenomenon discussed in Section 1.3, that solutions can be acceptable even if they are bad in terms of objective value, as long as all activities are assigned, and the running time is low. Deeper analysis of the instance shows that the reason for the large objective value is that the objective function is very unstable, in the sense that small differences in connection times can lead to very large differences in objective values. Also, in this case the running time is not low enough, compared to when only fixing 20%, to make the deterioration in objective value acceptable. In general, the running times for the hybrid heuristic are lower than with connection fixing for the large instances, but higher than variable fixing times. Typically, the running times decrease when more variables are fixed, as expected.

7.4 Backtracking

So far, we have not used backtracking at all in the fixing heuristics. Still, we have been able to find integer solutions of reasonable quality. But this might not always be possible when aggressive fixing strategies are used, as Tables 7.1 and 7.3 show, so backtracking is necessary in some cases. In standard branch-and-bound, backtracking is performed when the lower bound at a node is greater than or equal to the objective value of the best known integer solution.

As indicated in Algorithm 7.1, backtracking is performed either when an integer solution is found which is not good enough, or when $\overline{z}' \geq z_u$. $\overline{z}'$ is either the lower bound from TAS^{relax}, or the LP lower bound, if COLUMN-GENERATION reaches LP optimality. Observe that the heuristic as described in Algorithm 7.1 returns the first good enough integer solution which is found. It is trivial to modify it to search for improving solutions by replacing z_u with the best integer solution objective value found so far.

Backtracking is performed by simply undoing the last δ search space restrictions, i.e. re-inserting the last δ removed pricing network arcs. The removed RMP columns are not re-inserted, so as not to repeat the same search space restrictions when continuing. Also, simple perturbation techniques, not included in Algorithm 7.1, are used to make sure that the heuristic does not get stuck in an infinite loop of fixing and backtracking. In Chapter 11 we will discuss a fixing strategy for which backtracking is crucial to get acceptable performance.

		Fixing 20% of the variables		
Instance	Tas^{relax}	Time	IP obj.	Slack cost
A320_1w	628145	3424	628145	0
B737_1w	4969800	1665	4973800	0
B757_1w	4385060	1248	4388950	0
DC9_1w	12101800	52	12101800	0
DC9_1w2	16662800	78	16662800	0
DC9_10d	21314200	239	21314500	0
DC9_2w	33062400	1661	33129600	0
DC9_1m	69513000	6410	71665600	0
MD82_1w	11984100	9	11984100	0
MD82_2w	23628800	176	23628800	0
MD82_1m	52775500	4976	54294600	0
		Fixing 60% of the variables		
Instance	Tas^{relax}	Time	IP obj.	Slack cost
A320_1w	628145	3248	1858794	0
B737_1w	4969800	1404	7060700	2000000
B757_1w	4385060	1448	4489650	0
DC9_1w	12101800	51	12148300	0
DC9_1w2	16662800	68	17131000	0
DC9_10d	21314200	202	21831300	0
DC9_2w	33062400	575	36210700	2000000
DC9_1m	69513000	2881	216785100	142000000
MD82_1w	11984100	10	11984200	0
MD82_2w	23628800	181	25703900	2000000
MD82_1m	52775500	1182	65713900	8000000

Table 7.3 Performance of the hybrid fixing heuristic, fixing 20 and 60% of the variables before starting connection fixing.

7.5 Summary

In this chapter we have presented three variants of our general integer fixing heuristic, with rather different performance. A very nice property of the fixing heuristics we propose is that neither of them requires adding any additional constraints to the master problem, but can be enforced directly in the master and pricing problems. This is good, as additional constraints not only make the master problem more difficult to solve, but also complicate the pricing problem by adding dual variables, that must be integrated dynamically into the pricing algorithm.

With the simplest heuristic, the variable fixing heuristic, it is difficult to find solutions covering all activities without using backtracking, as the heuristic is often too aggressive. The reason for the poor performance of this heuristic is that the linear programs generated by the column generation algorithm are normally not very close to integrality, which means that their primal variable values are not very good indications of good integer values.

The best heuristic in terms of solution quality is the connection fixing heuristic with a dynamic limit. It finds solutions without unassigned activities, and very competitive objective values, for all our tests instances. Further testing, which is not reported here, has verified that the connection fixing heuristic almost never needs to use backtracking for any instance. However, since finding an integer solution requires fixing a lot of connections, this method is inherently slow for large instances. We finally show how shorter running times can be obtained without leaving activities unassigned by combining the variable and connection fixing heuristics into a hybrid heuristic.

Part III

A Constraint Programming Approach

Eight

Introduction to Constraint Programming

In this chapter will we give a brief introduction to constraint programming (CP). We will focus most of our attention on consistency and propagation techniques, but we will also discuss search techniques and backjumping. We recommend the books by Marriott and Stuckey [159], Apt [16] and Dechter [68] for more in-depth coverage of advanced constraint programming topics.

One of the first systems to use constraint technology was Sutherland's SketchPad system [208], which used constraints to model relations between graphical objects. Waltz [221] was the first to use constraints in artificial intelligence, and his work was followed up by Montanari [174], who introduced the concepts of constraint graphs and path consistency. It is interesting to note that all these three early contributions were done in the field of computer graphics. In [158], Mackworth defined node and arc consistency, and introduced the first basic algorithms to achieve consistency. Kumar [147] has written an excellent survey paper.

8.1 Variables and Domains

The first consideration when defining a constraint programming problem are the *variables*. Variables most often represent decisions or alternatives for some object, such as how many cars to manufacture, or which color to paint a surface. Each variable x_i has a *domain* $D(x_i)$ containing the possible values the variable can take. The domain can be of several types, such as a set of colors, a finite set of integer values, or a real-valued interval. We will here limit our discussion to finite integer domains, i.e. domains which only contain a finite number of integer values.

8.2 Constraints

Obviously, constraints play a central role in constraint programming. While mathematical programming or linear programming restrict constraints to take the form of (linear) equations and inequalities, constraint programming has a very general view of constraints. A constraint is simply a relation defined on a set of variables, limiting the mutual variable assignments to a subset of the Cartesian product of the domains. That is, if $S_i = \{x_1, \ldots, x_n\}$ is the set of variables included in constraint C_i, the variable assignments are restricted to a subset of $D(x_1) \times \ldots \times D(x_n)$. Which subset the assignments are restricted to depends on C_i, of course.

The *arity* of a constraint is the number of variables included in the constraint, i.e. the cardinality of S_i. A *unary* constraint includes a single variable, while *binary* and *ternary* constraints include two and three variables, respectively. In the early days of constraint programming almost only binary constraints were used, but today constraints with higher arity are becoming more and more common, partly due to the introduction of so-called *global constraints* (see Section 8.4). However, it is easy to show [68, page 31-32] that all constraints can in principle be transformed to a system of binary constraints.

8.3 The Constraint Satisfaction Problem

The finite-domain *Constraint Satisfaction Problem* (CSP) is the problem of finding value assignments to a set of variables such that a set of constraints is satisfied:

$$CSP(X, D, C) \qquad \begin{array}{l}\text{For all variables } x_j \in X \text{, find values } v_j \in D(x_j) \\ \text{such that } C = C_1 \wedge C_2 \wedge \ldots \wedge C_m \text{ is satisfied.}\end{array} \tag{8.1}$$

Here, $x_1 \ldots x_n$ are decision variables with domains $D(x_1) \ldots D(x_n)$, and the constraints to satisfy are $C_1 \ldots C_m$. The fact that we want all constraints to be satisfied is expressed by taking the conjunction over them. The entire CSP can thus be represented by a single constraint, which is rather different from mathematical programming notation. In its general form, finding a solution to a CSP is an NP-hard problem. This can easily be seen from the fact that the satisfiability problem SAT, which is clearly NP-hard [56], is an instance of CSP, where variables are `TRUE` or `FALSE`, and constraints are boolean clauses.

In [174], Montanari introduced the concept of a *constraint graph*. Constraint graphs are very useful for visualizing CSPs, and for describing various properties of them. A constraint graph for a *binary* CSP, i.e. a CSP with only binary constraints, has one node per variable, and arcs between nodes that are included in a common constraint. Figure 8.1 shows the constraint graph for a simple CSP. For CSPs with non-binary constraints, the graph is extended to

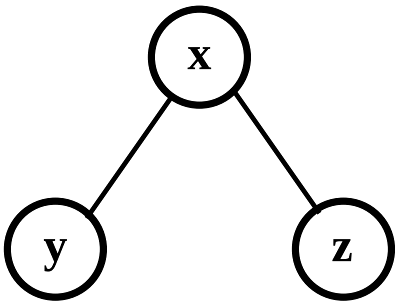


Figure 8.1 Constraint graph for the CSP $x, y, z \in \{0, \ldots, 10\}$, $x > y$ and $x + z = 8$.

a *hyper-graph*, i.e. a graph where arcs can connect to more than two nodes. In the rest of this thesis, the term *constraint graph* is used to denote constraint graphs for CSPs containing binary or non-binary constraints. Observe that while constraint graphs are very useful in certain contexts, they do not capture all aspects of a CSP. For example, the variable domains are not shown in constraint graphs.

8.4 Consistency and Propagation

The main idea of constraint programming is to use logical deductions from constraints and variables to reduce the search space and efficiently find solutions to CSPs.

Imagine a CSP with three variables x, y, z, all with domains $\{0, \ldots, 10\}$, and constraints $x > y$ and $x + z = 8$ (see Figure 8.1). The second constraint immediately implies $x \leq 8, z \leq 8$, and the first constraint implies $x > 0$. Using the reduced domain of x, we can use the second constraint again to deduce that $z < 8$, and the first constraint to deduce that $y < 8$. So by simple logical deductions, we have immediately removed many impossible variable assignments from the domains. Removing impossible assignments from the domain of a variable is called *domain reduction*, and iteratively applying domain reduction is called *constraint propagation*, as it propagates information about domain changes via constraints to other domains, for further domain reductions.

Domain reduction and constraint propagation are very central in constraint programming, and can be formalized by the concept of *consistency* [158]. Since consistency is most often defined in terms of binary constraints, we will for the time being restrict ourselves to the binary case. There are different kinds of consistency, depending on how many variables are simultaneously considered.

Definition 8.1. *A unary constraint C is* ***node consistent*** *with respect to a variable x_i iff all $a \in D(x_i)$ satisfy C.*

Definition 8.2. *A binary constraint C is* ***arc consistent*** *with respect to variables x_i and x_j iff*

$$\forall a \in D(x_i), \exists b \in D(x_j) \text{ s.t. } C(a,b) \text{ is satisfied,}$$

and

$$\forall b \in D(x_j), \exists a \in D(x_i) \text{ s.t. } C(a,b) \text{ is satisfied.}$$

In other words, arc consistency means that all values for x_i and x_j can be extended to a solution for the considered constraint.

Definition 8.3. *A CSP is node/arc consistent iff all its unary/binary constraints are node/arc consistent.*

A weaker version of arc consistency is *directional arc consistency*:

Definition 8.4. *A binary constraint C is* ***directionally arc consistent*** *relative to an ordering $d = (x_1, \ldots, x_n)$ of the variables, iff all values for x_i can be extended to a solution to C using only values of x_j, for $i < j$.*

Observe that the ordering of the variables is important in the definition of directional arc consistency – all variable instantiations must be possible to extended to solutions using *only* values of variables *succeeding* it in the ordering.

Clearly, the names node and arc consistency come from the constraint graph representation. Looking at the constraint graph, it is also obvious that the shortcoming of node and arc consistency is that they only look at single unary and binary constraints, and thus cannot find logical implications including more than two variables. Path consistency is an extension that looks at several variables. Observe that since we are only considering binary constraints at the moment, path consistency is defined over several constraints.

Definition 8.5. *A pair of variables (x_i, x_j) are* ***path consistent*** *with respect to a third variable x_k iff for all assignments $(x_i = a, x_j = b)$ satisfying the binary constraint between x_i and x_j, there exists a value $c \in D(x_k)$ such that the assignments $(x_i = a, x_k = c)$ and $(x_j = b, x_k = c)$ satisfy their respective constraints.*

It is clear that path consistency is a generalization of arc consistency and node consistency. While arc consistency requires that it is possible to extend one variable assignment with one more variable in such a way that a binary constraint is satisfied, path consistency requires that any two-variable assignment is possible to extend with a third variable. The most general form of consistency is *k-consistency*:

Definition 8.6. *A set of k variables is* ***k-consistent*** *iff for any assignment to any $k-1$ of the variables, satisfying all involved constraints, it is possible to find a value for the kth variable satisfying all involved constraints. A CSP is k-consistent iff any subset consisting of k of its variables is k-consistent, and* ***strongly k-consistent*** *if it is l-consistent for all $l \leq k$. A strongly n-consistent CSP with n variables is called* ***globally consistent****. Also,* ***directional k-consistency*** *is defined by extending Definition 8.4 to k variables.*

Observe that Definition 8.6 holds for constraints of any arity. For binary constraints, 1-consistency is equivalent to node consistency, 2-consistency to arc consistency, and 3-consistency to path consistency. For a globally consistent CSP, it is clear from Definition 8.6 that a solution can be found without any search.

Propagation Algorithms for Consistency

Obtaining node consistency is trivial, as it is just a matter of checking which values in the domain satisfy the constraint, and remove them. In fact, since node consistency is so simple, unary constraints most often do not play an important role when solving CSPs.

Obtaining arc consistency is a bit more complicated, but not much. In [158], Mackworth presents the AC-1, AC-2 and AC-3 algorithms for obtaining arc consistency for a CSP. The idea of all the algorithms is to simply check all value combinations for all variable pairs, and remove inconsistent values (Algorithm 8.1). AC-1 is the simplest algorithm for arc consistency, and AC-2 and AC-3 are more computationally efficient implementations of the same idea, which iteratively remove inconsistent values but only check constraints for which the included variable domains have changed. AC-3 has complexity $\mathcal{O}(ed^3)$, where e is the number of binary constraints, and the domain sizes are bounded by d. Several even more efficient algorithms for arc consistency exists, such as AC-4 to AC-8 [37, 38, 48, 173, 216], and AC-2000 and AC-2001 [39]. Most of these algorithms are more space and running-time efficient versions of the basic AC-3 idea, tailored to fit the implementations of modern constraint solvers.

Path consistency can be obtained by algorithms very similar to the arc consistency algorithms by just extending them to check three variables at a time. The PC-1 and PC-2 implementations for path consistency are due to Mackworth [158], the PC-2 algorithm having time complexity $\mathcal{O}(n^3d^5)$, where n is the number of variables.

Obtaining strong k-consistency, or even k-consistency, is computationally very expensive. The fact that CSP in general is NP-hard, and that global consistency leads to a backtrack-free search, tells us that obtaining global consistency is an NP-hard problem. Dechter [68, Section 3.4] has shown that obtaining k-consistency has complexity exponential in k.

Algorithm 8.1 The AC-3 algorithm for arc consistency. While it is stated here in terms of variable domains, it is easy to re-formulate it to the equivalent form of repeatedly propagating constraints for which the included variable domains have changed.

```
data
    Queue $q$ of variable pairs, initially empty.
procedure AC-3
    for all variable pairs $(x, y)$ do
        $q \leftarrow q \cup (x, y)$
        $q \leftarrow q \cup (y, x)$
    end for
    while $q \neq \emptyset$ do
        $(x, y) \leftarrow pop(q)$
        Remove all inconsistent values in $D(x)$ w.r.t. $y$
        if $D(x)$ changed then
            for all variable pairs $(z, x), z \neq y$ do
                $q \leftarrow q \cup (z, x)$
            end for
        end if
    end while
end procedure
```

Bounds Consistency

Bounds consistency, sometimes also called *interval consistency*, is a relaxed form of arc consistency, that is sometimes computationally more efficient than full arc consistency. Instead of requiring that all values in a domain are possible to extend to a solution of a constraint, bounds consistency only requires that the smallest and the largest value in the domain have this property. Let $min\_value(D(x))$ and $max\_value(D(x))$ be the minimum and maximum values in $D(x)$, respectively.

Definition 8.7. *A constraint C of any arity is* ***bounds consistent*** *with respect to a variable x_i iff $min\_value(D(x_i))$ and $max\_value(D(x_i))$ are possible to extend to a solution to C.*

Obtaining bounds consistency is typically computationally less expensive than obtaining arc consistency if a domain contains a large number of values. Also, for certain kinds of constraints, e.g. linear constraints, bounds consistency is the most natural type of consistency.

Generalized Arc Consistency

With the notable exceptions of k-consistency and bounds consistency, we have so far only considered binary constraints. Since many constraints are not binary, a consistency definition for constraints of any arity is necessary. *Generalized arc consistency* simply extends the definition of arc consistency accordingly:

Definition 8.8. *A constraint C of any arity is **(generalized) arc consistent** with respect to a variable x_i, iff for any value $a \in D(x_i)$ there exists simultaneous assignments for all other variables in C such that C is satisfied.*

Henceforth, 'arc consistency' will be used to denote both standard arc consistency and generalized arc consistency, depending on the context.

Global Constraints

One of the reasons of the recent upswing for constraint programming techniques is the introduction of so-called *global constraints*. Global constraints are constraints which in a single constraint captures the semantics of many smaller constraints, and often come with a specialized algorithm to obtain (generalized) arc consistency, or an approximation thereof. Many global constraints use operations research techniques such as network and optimization algorithms to efficiently obtain arc consistency. The most famous example is probably the `all_different` constraint. `all_different` constrains a set of variables to all take different values, and has widespread use in e.g. scheduling applications. While a quadratic number of not-equal constraints has the same semantics as `all_different`, the propagation is much stronger for `all_different`, which has a global view of all the included variables. Using matching algorithms and strongly connected components, Régin [181] has developed an algorithm that obtains generalized arc consistency for `all_different` in polynomial time. It is easy to construct tiny examples for which the binary representation cannot detect inconsistency, but Régin's algorithm can.

Other notable examples of efficient global constraints are the `cumulative` [4], `generalized cardinality` [182] and `sum` [184] constraints. Milano et al. [171] provide an interesting look at the use of integer programming techniques in global constraints, and Beldiceanu and Contjean [34] describe the implementation of global constraints in the CHIP constraint programming solver.

8.5 Searching for Solutions

Most CSPs cannot be solved by propagation alone. In such cases, it is necessary to perform search to find a solution. The classical tree search algorithm, sometimes called the *backtracking* algorithm, selects a variable, and a value

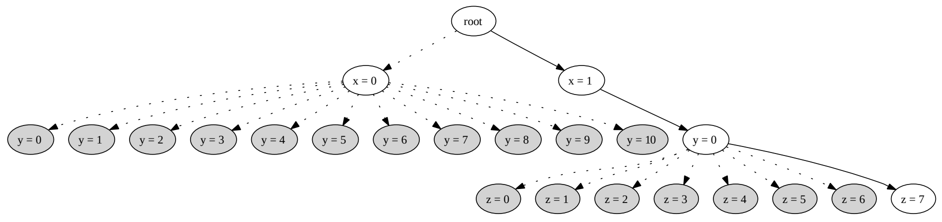


Figure 8.2 Backtracking search for the CSP $x, y, z \in \{0, \dots, 10\}$, $x > y$ and $x + z = 8$. Shaded nodes represent nodes where inconsistency has been detected.

in the domain of the variable, and *instantiates* the variable to this value. If this instantiation is consistent with all constraints, a new variable and value pair is selected, and the algorithm is applied recursively until all variables are instantiated. If an instantiation is inconsistent, another value is tried, and when all values for a certain variable have been tested and found inconsistent, another variable is selected for instantiation. The term 'backtracking' thus comes from the fact that once all values have been tried for a variable, one must move one step upward/back in the tree to select a new variable. This process is often depicted as a search tree, as demonstrated in Figure 8.2.

Look-Ahead

In the most trivial search algorithms, constraint satisfaction is checked either for individual constraints when all variables included in the constraint have been instantiated, or for all constraints when all variables have been instantiated. This means that if a bad instantiation is made early on in the search, a lot of backtracking and testing of infeasible solutions must be done before a feasible solution is obtained. Obviously, one way to improve this situation is to use constraint propagation throughout the search, once a variable has been instantiated. This is normally called *look-ahead*, as it looks at the effect of a variable instantiation for the other, not yet instantiated, variables. Look-ahead will check constraint satisfaction dynamically in the search tree rather than only once all variables are instantiated, and it will also propagate domain reductions to decrease the search space. The effect of applying look-ahead is demonstrated in Figure 8.3, which shows the search tree for the same problem as in Figure 8.2 when using propagation based on bounds consistency. Since initial propagation removes 0 from the domain of x, the first instantiation we do is $x = 1$. The constraints then force y to be 0, and z to be 7, making only one search node necessary. While this is a trivial example, it does illustrate the effect propagation can have in reducing the search space.

From figures 8.2 and 8.3, it is clear that constraint propagation can help reduce the search space. But what is not clear is what effect it has on the actual

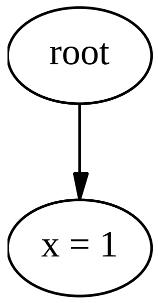


Figure 8.3 Backtracking search with bounds consistency propagation for the CSP $x, y, z \in \{0, \ldots, 10\}$, $x > y$ and $x + z = 8$.

running time of the search. While propagation helps to search fewer nodes, it has in fact put more effort in reducing the domains, and as we have already discussed, this can be a complex task. In fact, adding constraint propagation can only be considered useful if applying the propagation algorithm is more efficient than searching all the nodes removed by the propagation. This is why efficient propagation algorithms are crucial for improving search performance in constraint programming.

To tune constraint programming search performance, several versions of look-ahead exist. *Forward checking* means that arc consistency is obtained in each search node for all the constraints in which the instantiated variable is included. *Partial look-ahead* establishes directional arc consistency with respect to some ordering among the not yet instantiated variables, and *full look-ahead* does one pass of arc consistency over *all* variables. *Really full look-ahead*, or *maintaining arc consistency (MAC)*, establishes full arc consistency for the CSP after each variable instantiation. Clearly, forward checking is the least expensive of these look-ahead methods in terms of computational complexity, and really full look-ahead is the most expensive. Which degree of look-ahead to use depends on the propagation algorithms for the constraints, and the variable domains. Loosely speaking, if propagation can remove many values it might pay off to run even an expensive propagation algorithm, while if few values can be removed it is probably better to use less propagation. In the early days of constraint programming, Haralick and Elliot [113] concluded that forward-checking was the overall most efficient look-ahead scheme for the N-queens problem as well for randomly generated CSPs. Later studies, e.g. [94], have shown that the stronger propagation methods are often superior to forward checking.

Variable and Value Ordering

Two very important considerations for maximizing search performance are the variable and value ordering strategies. In the description of tree search above, we only said that in each search node a variable and a value are selected for instantiation. In practice, the order in which we choose to instantiate the variables and the order in which we try the values have a huge impact on the search performance.

The order in which variables are instantiated is most often not statically set, but depends on the search performed so far. To detect inconsistencies as early as possible, and thus limit the search as much as possible, it is considered good to start by instantiating variables which are likely to lead to inconsistencies. This strategy, which is known as *fail-first*, can be implemented e.g. by first instantiating variables with a small domain, or variables included in as many constraints as possible. When it comes to value ordering, it is however good to first try values which are *unlikely* to lead to inconsistencies. While this might seem counterintuitive at first, a straightforward explanation of why it is good is to remember that *all variables must be instantiated, but all values do not have to be tested.*

Common general strategies for value ordering are e.g. *min conflict*, that gives higher priority to values for which look-ahead reduces the domains of other variables the least, and *max domain size*, which starts with values which lead to the largest minimum domain size among the remaining variables. The *estimated solutions* strategy attempts to estimate the number of solutions a certain value can lead to. Observe that all these strategies require extra computation to evaluate the effect of instantiations. Sometimes it is also possible to come up with good variable and value instantiation orderings using problem specific information.

Thrashing

The most common reason for poor search performance is *thrashing*. Thrashing denotes the repeated search of similar subtrees, all leading to conflict due to a poor instantiation higher up in the search tree. Thrashing might sound relatively harmless, but in practice it can be devastating to performance. Imagine a CSP with an average of 10 values per domain, and that a poor instantiation, which always leads to conflict further down the tree, has been made. If this conflict cannot be detected until e.g. 10 instantiations later, one might be forced to search the entire subtree containing 10^{10} nodes! For practical problems this number can be even larger, making it impossible for even the fastest computers to cope with thrashing. Thrashing can of course also occur for similar, seemingly independent, subtrees in the same search. Often, thrashing occurs when a problem contains complicated constraints for which no efficient propagation algorithms exists.

Backjumping

The most straight-forward way to avoid thrashing is to design variable and value ordering strategies in such a way that a minimal amount of backtracking is necessary, and to implement powerful propagation algorithms. But this is not always possible, and in such cases more direct action must be taken.

If the search algorithm is designed to handle it, one way to avoid excessive thrashing can be to completely switch context, i.e. to jump to a completely different part of the search tree, when symptoms of thrashing are detected. We can think of the search as consisting of a set of ’frontier nodes’, which have not yet been examined. When we have selected a variable to instantiate, each possible instantiation adds one frontier node, and the value ordering determines which frontier node to examine first. In the depth-first case we can only choose from the frontier nodes of the current variable, but one can imagine a search algorithm implementation that allows us to select any frontier node for examination, allowing ’jumps’ in the search tree e.g. to avoid thrashing.

Intelligent backtracking, or *backjumping*, is a general term denoting methods to decide where to jump in case of conflicts. Ideally, one would of course like to be able to decide which instantiation is the culprit of the thrashing, and jump to the corresponding node, but this is most often very difficult by the very definition of thrashing – if we knew which instantiation caused thrashing, we would not make it, and thus no thrashing would occur. However, by clever examination of the search tree, some conclusions can be drawn which make backjumping possible. The original work on backjumping was made by Gaschnig [100], but the current authority is Dechter [68]. Her book includes a chapter [68, Chapter 6] which thoroughly covers the theory and practice of backjumping, including recent learning backjumping algorithms, and provides a large number of bibliographical references. As an interesting contrast, the book by Apt [16] does not even mention backjumping, and just briefly describes what intelligent backtracking is, so clearly the importance of backjumping is not undisputed. Our own practical experience tells us that many times when thrashing occurs it is more worth-while to focus on improving the variable and value ordering heuristics than to implement a complex backjumping strategy.

8.6 Constraint Optimization

So far in this chapter we have only discussed constraint satisfaction, i.e. finding any solution satisfying a number of constraints. But as we have seen in previous chapters, we are really interested in optimization in this thesis: finding the best solution, according to some cost function, which satisfies all constraints.

Optimization is seldom considered in constraint programming, and when it is, it is treated in a very simplistic, almost naive, way. The most often recommended method [16, 68, 159] of finding optimal solutions in constraint programming is to find a solution the normal way, and then add a constraint constraining the cost to be better than for the solution just found, and restart/continue the search. Clearly, this completely ignores the cost function as a guide to the search, and simply limits its use to that of a constraint. For difficult large-scale optimization problems, this method has found little practical use.

Dechter [68, Chapter 13] gives an overview of some other *constraint optimization* techniques, such as Russian Doll Search and Bucket Elimination. In practice these methods have also not been used much. However, since the integration of operations research and constraint programming techniques is a very active research field currently, much work is being put into merging optimization and constraint programming.

Approaches merging the fundamental ideas of integer programming and constraint programming on a 'low level' have been proposed. For example, the Ph.D. thesis of Ottosson [176] proposes mixed IP/CP modeling, combining CP's domain reductions and logical inference with linear programming relaxation. Bockmayr and Kasper [41] presents a framework for unifying IP and CP which they call Branch-and-Infer. They compare both the declarative powers of IP and CP and their respective computational properties, and proceed to suggest how the techniques can be integrated. Rodošek et al. [186] present an approach to IP/CP integration using automatic transformation of disjunctions to binary 0/1 variables. The ECL^{i}PS^e [46] constraint solver is combined with the CPLEX [121] linear programming solver to solve e.g. progressive party problems which cannot be solved by IP or CP alone in reasonable computation time.

Haijan et al. [110] describes a method for integrating IP and CP by letting a CP model find a starting solution for a simplex-based integer programming algorithm. Jain and Grossmann [124] develop a method for integrating IP and CP for problems containing only a subset of costed variables, and show how the hybrid solver can improve running times by several orders of magnitude. Darby-Dowman and Little [63] analyze the properties of combinatorial optimization problems, and show how these properties affect the computational performance for IP and CP models. Drawing conclusions from this, they also propose ideas for IP/CP integration. Wedelin [224] has introduced *Cost Propagation*, which generalizes constraint propagation to optimization problems. The work is an extension of [223], where a competitive heuristic algorithm for the airline crew pairing problem was developed.

A slightly different research direction focuses on the use of operations research techniques for implementing efficient propagation algorithms for global constraints. Milano et al. [171] gives a thorough overview of this research area, and present case studies for a few example constraints. Focacci et al. [90] present global constraints incorporating domain reduction based on lower bounds and reduced costs. Practical examples of global constraints that use operations research techniques are e.g. `all_different` [181], `GCC` (Global Cardinality Constraint) [182], `costGCC` (Global Cardinality Constraint with Costs) [183]. We will re-visit these constraints in the following chapters. Further, `costGCC` is an example of a constraint that actually tries to embed the cost function within a constraint propagation framework, to make constraint optimization more efficient.

Several approaches to practical problems using hybrid IP/CP methods have been presented. Focacci et al. [90] solve the Traveling Salesperson Problem (TSP) using a hybrid solver, with a strong emphasis on optimization-oriented global constraints. Rousseau et al. [192] and Junker et al. [129] solve Vehicle Routing Problems (VRP) and crew planning problems respectively, using column generation combined with constraint programming. They both use constraint programming techniques to solve the complex pricing problems, and standard linear programming for the RMP. Caprara et al. [44] present a crew planning application for an Italian railway company using a specialized hybrid IP/CP solver, and de Silva [67] propose a similar approach for bus driver scheduling. El Sakkout et al. [78, 110] use the starting-solution hybrid approach mentioned above to solve the fleet assignment problem for British Airways.

Finally, it is interesting to note that the well-known branch-and-bound algorithm in integer programming has many similarities to CSP tree search. It usually instantiates variables in the same way, but it does not perform any propagation. And even if some propagation is performed, the theoretical foundation of linear programming does not allow non-convexity, and thus cannot handle removal of intermediate domain values, or additional deduced non-linear constraints. In integer programming, this can be overcome by re-defining the variables to be 0/1, but this has the disadvantage of increasing the number of variables dramatically. However, branch-and-bound has something which CSP tree search has not - the underlying relaxed linear program, and efficient algorithms for solving it. This is used to guide the search, and cut off branches which are guaranteed to lead to inferior solutions. So in some sense the underlying linear programming solver can be said to act as a combined variable and value ordering oracle, and global constraint.

8.7 A Constraint Solver Implementation

In the subsequent chapters of this thesis, we will describe how constraint programming techniques can be used to solve the Tail Assignment problem. Initially, we started by using the ILOG Solver [122] constraint solver to model and solve CSPs. But as our models became increasingly complex, it was necessary to develop our own constraint solver to facilitate speed-up and ease of implementation. We therefore implemented our own constraint solving engine in the C++ programming language. The CSP engine contains classes for domains, constraints and search, and various controller and callback classes. The engine is general enough to model and solve just about any CSP, but we have only implemented the constraints which have been necessary.

Constraint Propagation

The central classes for modeling CSPs in our constraint solver are `CSPDomain` and `CSPConstraint`. A `CSPDomain` is simply a variable with a name and a domain of integer values.[1] The `CSPDomain` class contains various useful methods for accessing the domain, as well as removing values and instantiating the variable. `CSPConstraint` is the base class for all constraints. When a constraint is created, it *registers* itself with each of the domains included in it. Each `CSPDomain` thus knows which `CSPConstraints` it is included in, and will when modified tell all these constraint about the change, to facilitate propagation.

A somewhat simplified listing of the `CSPConstraint` class is shown in Algorithm 8.2. The public methods are only for internal use by the search and propagation code. `CSPConstraint::remove` and `CSPConstraint::insert` notifies the constraint about removals and insertions in domains. The domain changes are stored in `CSPConstraint::changesM`, and if the constraint is not already about to be propagated, it is pushed on a global constraint queue. Once a constraint is at the top of the constraint queue, `CSPConstraint::doPropagate` is called. This is just a wrapper around `CSPConstraint::propagate`, which performs the actual propagation. Observe that `CSPConstraint::propagate` is the only method which should be implemented by subclasses implementing new constraints.[2] Normally, `CSPConstraint::propagate` checks the changes in `CSPConstraint::changesM`, updates its internal representation of the constraint, and performs propagation. If the propagation changes any `CSPDomain`, the domain will notify its constraints, which will in turn be pushed on the constraint queue. If a conflict is detected, either by the constraint finding some inconsistency, or by a `CSPDomain` becoming empty, a `CSPConflictFound` exception is thrown. During propagation, all domain changes and constraint postings are stored in a `CSPUndoStack`, to make future backtracking possible.

Search Control

The class `CSPEngine` implements the actual search. At the moment, only depth-first search is supported. Not having a more general search model makes the implementation quite simple, and for our purposes depth-first search has turned out to be enough. `CSPEngine` maintains a queue of the search nodes from the root of the search tree to the current node. To instantiate a new variable, it asks a `CSPSearchController` which `CSPDomain` to instantiate,

[1]Currently, only finite integer domains are supported.
[2]Along with some additional methods not included here.

Algorithm 8.2 The `CSPConstraint` class, somewhat simplified.

```
class CSPConstraint
{
public:
  CSPConstraint(CSPDomainList& domains,
                CSPUndoStack* undoStack,
                CSPConstraintQueue* queue);
  virtual ~CSPConstraint();

  void remove(CSPDomain* domain, int value,
              CSPConstraintQueue* queue);
  void insert(CSPDomain* domain, int value,
              CSPConstraintQueue* queue);

  void doPropagate(CSPUndoStack* stack,
                   CSPConstraintQueue* queue)
    throw (CSPConflictFound);

protected:
  virtual void propagate(CSPUndoStack* stack,
                         CSPConstraintQueue* queue)
    throw (CSPConflictFound) {}

  // Changes done, not yet registered.
  std::list<CSPDomainChangeEvent> changesM;
};
```

and in which order its values should be tried. It then instantiates the variable to the first value in the list, and creates a search node. The search node (`CSPSearchNode`) contains information not only about which value the variable was instantiated to, but also about which actions must be taken to undo the node, i.e. to backtrack from the node. It therefore contains a stack of all domain changes and constraints posted by instantiation and propagation at this node.

Once a variable has been instantiated, propagation is performed as described above. At the moment, only AC-3 type propagation is implemented. The constraints in which the instantiated variable is included are notified about the domain change, perform propagation, which might trigger further constraints to be queued for propagation. This is repeated until the constraint queue is empty. When propagating, the undo stack and constraint queues from the current search node are passed to the constraint's `propagate` method, to track domain changes, and make it possible to add old constraints for which domains have changed as well as to post new constraints dynamically.

Conflicts are detected either by a constraint propagation algorithm, which throws a `CSPConflictFound` exception, or by a `CSPDomain` becoming empty and throwing a `CSPDomainEmpty` exception. When a conflict if detected, backtracking is performed. First, `CSPSearchNode::undo()` is called to undo everything done in the current node. Then, the next value to try for this variable is retrieved, and a new node is created. If all values have been tried, backtracking is performed for the parent node.

If at any point we must backtrack past the root node, the search has failed. Otherwise, we proceed until all variables have been instantiated, or a search limit is reached. Currently, there are six types of search limits, which can be set in `CSPSearchController`:

- a maximum number of search nodes explored;
- a maximum number of backtracks performed;
- a maximum search depth;
- a maximum number of CPU seconds;
- a maximum memory consumption;
- a maximum number for backtracks/depth. This is to detect thrashing, as e.g. performing 100 backtracks on depth 10 indicates that we are exploring large parts of the search tree.

Since the code is object oriented, a user of the engine should not change any of the classes, but instead subclass them to create new constraints and implement new search strategies. To simplify debugging, there is also a `CSPTracer` class which when turned on can print diagnostic information about just about anything that goes on in the search, as well as draw graphical search trees,[3] such as the ones in figures 8.2 and 8.3.

Example

As a simple example of how to use our CSP solver, Algorithm 8.3 shows how to model and solve the CSP

$$\begin{aligned} x &> y, \\ x + y + z &= 8, \\ x, y, z &\in \{0, \dots, 10\}. \end{aligned}$$

[3]This is done using the Graphviz graph-drawing package [98].

Algorithm 8.3 An example of how to use the CSP solver.

```
// Create domains
CSPDomain x("x", 0, 10);
CSPDomain y("y", 0, 10);
CSPDomain z("z", 0, 10);

// Create constraint x - y > 0
CSPLinearConstraint c1(&x, CSPLinearConstraint::Greater, &y);

// Create constraint x + y + z = 8
CSPDomainList domains;
domains.push_back(&x);domains.push_back(&y);
domains.push_back(&z);
CSPLinearConstraint c2(domains, CSPLinearConstraint::Equal, 8);

// Create search controller and engine, and solve
CSPConstraintList constraints;
constraints.push_back(&c1);constraints.push_back(&c2);
CSPSearchController controller(domains);
CSPEngine engine(domains, constraints, controller);
engine.instantiate();

// Print some statistics, and the solution
engine.outputStatistics(std::cout);
std::cout << x << ‘‘, ‘‘ << y << ‘‘, ‘‘ << z << std::endl;
```

The result is:

```
------------------- CSP Search Statistics -------------------

  Total search time (s).........:    0.0
  Propagation time (s)..........:    0.0
  Solutions found...............:      1
  Last solution on depth........:      1
  Search nodes explored.........:      2
  Search nodes created..........:      2
  Max search depth..............:      1
  Number of backtracks..........:      0
  Constraints posted............:      2

x : [ 1 ], y : [ 0 ], z : [ 7 ]
```

Nine

Constraint Programming Models

In previous chapters, we have described a mathematical programming model and solution technique for the tail assignment problem. One problem with the mathematical programming model is that it is not directly suited to finding solutions quickly. It rather puts optimization in focus, and is therefore very suitable indeed for the longer-term planning, but less suitable when a quick solution is desired. As already mentioned, it is sometimes crucial to quickly obtain solutions. A quick method for finding solutions could be used for feasibility checking, but also to provide the initial solution to the column generator for further improvement, and thus make it possible to first provide an initial solutions very quickly, and then more optimized solutions after some more time. Since constraint programming is a technique that is well suited to solving difficult feasibility problems, we have focused on constraint programming as a means to quickly find initial solutions, regardless of solution quality.

In this chapter we describe the development of new constraints and ordering heuristics to build a complete constraint model for tail assignment. We start in Section 9.1 by doing a literature survey of constraint programming approaches to problems similar to tail assignment. In Section 9.2 we describe the basic constraint model, and some simple extensions of it. In Sections 9.3–9.6 we describe the new constraints and ordering heuristics introduced to complete and solve the full model. Section 9.7 presents an alternative way to handle cumulative constraints, and Section 9.8 shows computational results on real-world tail assignment instances. In Section 9.9 we summarize this chapter.

Notation

Constraint programming notation is less standardized than mathematical programming notation. Here, we will use the following notation: Variables and constraints are typeset in `typewriter` font. Arrays of variables use a sub-

Algorithm 9.1 An example of basic constraint propagation notation.

```
$D(\mathtt{x}) == \{a\} \Rightarrow$ POST $\mathtt{y} = \mathtt{z}$
```

script, such as $\mathtt{variable}_{index}$. As in Chapter 8, $D(\mathtt{x})$ denotes the domain of variable $\mathtt{x}$, which contains a set of variables.

Basic propagation algorithms and semantics for simple constraints are described as in the small example in Algorithm 9.1. This notation simply means that whenever the domain of $\mathtt{x}$ contains the single value a, the equality constraint $\mathtt{y} = \mathtt{z}$ should be *posted*, i.e. added to the set of constraints at the current point in the search tree. Dynamically posted constraints are removed if the search is backtracked past the point where they were posted.

More complex propagation algorithms, which require additional internal data structures and variables, are described in a more conventional algorithmic pseudo-code style.

9.1 Literature Review

While there is plenty of constraint programming literature available, very little has been published related to airline planning problems. Slightly more common are constraint programming approaches to the Traveling Salesperson Problem (TSP) and the Vehicle Routing Problem (VRP), which have some similarities to tail assignment.

Halatsis et al. [111] discuss a complete crew planning suite, including both crew pairing and crew rostering, which uses constraint programming. Unfortunately, very few details about the models are presented. Instead, the problems are described, and the authors just state that CP is used to solve them. The system has been tested, and found satisfactory, by Olympic Airways.

Christodoulou and Stamatopoulos [49] describes what seems to be the crew rostering component of the complete approach of Halatsis et al. Christodoulou and Stamatopoulos use redundant variables, where the pairings assigned to each crew, as well as the crew selected for each pairing, are represented. They only describe a selection of the constraints, which link the redundant variables, and describe various working time rules. They use flight time fairness as their optimization criterion, and the optimization is done in the standard CP way, by just adding an additional constraint forcing the next solution to be better than the best found so far.

Curtis et al. [59] show how constraint programming can be used to create bus driver schedules. This problem is a little different from airline crew planning problems, due to the shorter transportation times and simpler working time rules. For each *piece*, i.e. fixed sequence of trips, a variable with the rosters covering the piece as possible values, is added. This results in a standard set partitioning problem, and Curtis et al. the proceed to describe how this can be solved using constraint programming. The set partitioning problem is very simple to state in terms of constraints, so most of the effort is instead spent describing various preprocessing steps and redundant constraints. They also discuss how the LP relaxed solution of the set partitioning problem can be used to guide the CP search.

Caprara et al. [44] present an integrated CP/OR solution approach to the crew rostering problem at the Italian Railway Company. The constraint model has successor variables, similar to the ones we will use, and various other variables to keep track of properties of duties. To model all the complicated rules, a large number of specialized constraints are added. Redundant variables and special ordering heuristics are also described. Operations research techniques are used for bounding, and the computational results of the integrated approach compares well to other pure OR approaches. The authors especially point out that the ease of implementation of the CP model is a big advantage. Both de Silva [67] and Fahle et al. [86] present hybrid CP/OR approaches to the bus driver scheduling and crew rostering problems, respectively, but since we are here mainly interested in pure CP approaches, we will not describe these approaches in more detail.

El Sakkout [78] presents a constraint programming approach to the daily cyclic fleet assignment problem. Each leg has a fleet type variable, and for each airport there is a variable keeping track of the number of overnighting aircraft at the airport. The straightforward constraints limit the number of overnighting aircraft at each airport, as well as the number of aircraft of each fleet. There are also constraints ensuring that the flow of aircraft is maintained, and that a certain fraction of the aircraft overnight at a maintenance station. Redundant constraint are added to speed up the search, but for optimization purposes the search is still far to slow. As a consequence, El Sakkout presents a local improvement heuristic which can quickly improve an existing solution.

In [202], Simons describes how constraint programming can be used to solve the aircraft swapping problem, i.e. the problem of changing aircraft types close to departure due to increased demand. The presented approach, which is developed in collaboration between British Airways and IC-PARC, can solve problems for which a few flights have changed demand. Very few technical details are presented in the article, but the system was tested by British Airways' operations controllers during three months, and performed well.

Pesant et al. [177] present a model for the traveling salesperson problem with time windows (TSPTW). The model without time windows contains only successor variables, a quadratic number of disequality constraints stating that all successors must be different, and subtour elimination constraints. The time windows are modeled by simply adding bounded time variables, and combining these with the successor variables to set the arrival time of each fixed successor relative to its predecessor. Redundant constraints are added to increase propagation for the time variables, and in this process redundant predecessor variables are also added. The approach is shown to improve the best known solutions on a few instances in the well-known Solomon test suite.

De Backer et al. [21] present a constraint programming and meta-heuristic solution approach to the VRPTW. Their model uses successor variables, and Ilog Solver's `path` constraint [122] is used to accumulate starting times and vehicle capacities along the routes. De Backer et al. also discuss how practical constraints, such as length constraints, or individual vehicle restrictions, can be modeled by propagating a *vehicle tag* along the constructed routes. Since large-scale VRPs cannot be solved efficiently by CP search, the authors instead use local improvement heuristics to search for solutions. The constraint model is used only for propagation, and to check the feasibility of solutions. Two main local improvement heuristics, *tabu search* [102] and *guided local search* [220], are tested, and computational results are reported for the Solomon test suite.

Rousseau et al. [192] present a combined CP and column generation approach to the VRPTW. In an approach which was initially presented by Fahle et al. [86] for crew rostering, constraint programming is used to solve the pricing problem within the column generation algorithm. The pricing problem model proposed by Rousseau et al. uses successor and predecessor variables, as well as variables for arrival times and vehicle capacity. The basic constraints are `all_different` constraints over the successors, constraints maintaining consistency between successors and predecessors, and time and capacity propagation constraints. However, most of the article is spent describing two new, redundant, constraint types, which are useful to improve the computational performance. One of the constraints, the so-called `CanBeConnected` constraint, makes sure that subchains arising during the search can be connected. The other constraint type is a family of arc elimination constraints which use filtering based on optimization to remove arcs which can never be used in optimal solutions.

9.2 Model Csp-Tas^{relax}

The most basic tail assignment constraint model has three sets of variables. As in Chapter 4, let F denote the set of activities and T the set of aircraft. The first set of variables are `successor` variables for all activities $f \in F$, the domains of which contain the possible successors of activity f. For optional activities, e.g. maintenance possibility activities (Section 3.2), f is added to $D(\texttt{successor}_f)$, to make it possible to simulate leaving activities unassigned by assigning them to be their own successors.

There are also `vehicle` variables for all activities, initially containing the

Model 9.1 CSP-TAS^{relax}. A basic constraint model without maintenance constraints.

$$\forall f \in F: \quad D(\texttt{successor}_f) = \bigcup_{f' \in F} (f' \text{ possible follow-on of } f)$$

$$\forall f \in F: \quad D(\texttt{predecessor}_f) = \bigcup_{f' \in F} (f \text{ possible follow-on of } f')$$

$$\forall f \in F: \quad D(\texttt{vehicle}_f) = \bigcup_{t \in T} (t \text{ can be assigned to } f)$$

$$\texttt{inverse}(\texttt{successor, predecessor}) \tag{9.1}$$

$$\wedge$$

$$\texttt{all\_different}(\texttt{successor}_1, \ldots, \texttt{successor}_{|F|}) \tag{9.2}$$

$$\wedge$$

$$\texttt{all\_different(vehicle}_{G(f)}) \quad \forall f \text{ s.t. } |T_f| \neq |\mathrm{T}| \tag{9.3}$$

$$G(f): \text{Activities overlapping the start time of activity } f$$

$$\wedge$$

$$\texttt{tunneling}(\texttt{successor, predecessor, vehicle}) \tag{9.4}$$

aircraft in T that are allowed to operate the activity, T_f. These variables model preassigned activities as well as flight restriction constraints. Observe that since connection constraints are captured by the `successor` variables, and flight restriction constraints are captured by the `vehicle` variables, a lot of the tail assignment constraints will not be necessary to explicitly add to the model.

Further, `predecessor` variables are added, representing the possible predecessor activities of an activity. While predecessor variables are redundant for describing the problem, they can help the search by improving propagation. As a simple example, considering predecessors explicitly has the benefit of finding activities that only has one predecessor, but whose predecessor has multiple possible successors. Our basic constraint model, CSP-TAS^{relax}, is shown in Model 9.1.

To keep the `successor` and `predecessor` variables consistent, the `inverse` constraint (9.1) is added. Our `inverse` constraint, which is described in Algorithm 9.2, ensures that f exists as a predecessor to f' if and only if f' exists as a successor to f. The constraint has been implemented as a single constraint rather than as many binary constraints to reduce overhead caused by large numbers of constraints having to be processed.

Algorithm 9.2 The `inverse` constraint.

```
inverse(A, B):

        $D(\mathtt{A}_i)$ contains $j \iff D(\mathtt{B}_j)$ contains $i$
```

Since all activities must have unique successors to form disjoint routes through the network, all `successor` and `predecessor` variables must take unique values. Therefore the `all_different` [181] constraint (9.2) over all `successors` is added. Since the `predecessor` variables are kept consistent with the `successor` variables via the `inverse` constraint, no `all_different` constraint is added for them. Activities which can be the last activity in a route reconnect back to the carry-in activities, by including the carry-in activities in their `successor` domain.

Constraint (9.3) adds `all_different` constraints over some `vehicle` variables. Since activities are fixed in time, activities which overlap in time can never be operated by the same aircraft. Therefore, `all_different` constraints are added for the `vehicle` variables of all activities which pass a certain point in time. Observe that adding an `all_different` for *all* activities overlapping another activity is not a good idea, as an aircraft operating an activity overlapping only the beginning of the activity might be able to also operate an activity that overlaps only the end. The problem with adding `all_different` constraints is thus to decide for which fixed time to add them. Our strategy is to initially add one `all_different` for the start time of each activity which has some restriction, i.e. which cannot be operated by all aircraft.

Finally, to maintain consistency between the `successor` and `vehicle` variables, the specialized `tunneling` constraint (9.4) is added. Observe that once completely instantiated, the `successor`, `predecessor` and `vehicle` variables all describe the solution completely. The `successor` or `predecessor` variables obviously give a direct route for an aircraft, and the `vehicle` variables specify which activities are assigned to a certain aircraft. Since the activities are fixed in time, simply sorting all activities assigned to an aircraft gives the route for this aircraft. The tunneling constraint is implemented as a single constraint that is propagated each time a variable is fixed, and takes appropriate action to keep the variables consistent, as shown in Algorithm 9.3.

The standard `element(b,A,c)` constraint forces the `b`:th variable in variable array `A` to take value `c`, i.e. $\mathtt{A}_\mathtt{b} = \mathtt{c}$. The first implication in Algorithm 9.3 thus states that whenever a `vehicle` variable is fixed, the not yet fixed successor and predecessor of the corresponding activity must be assigned to the same vehicle. The second implication states that fixed successors and predecessors must be assigned to the same vehicles. Finally, the last implication states that overlapping activities cannot be assigned to the same vehicle. In

Algorithm 9.3 Full `tunneling` constraint propagation. The constraint is propagated each time one of its variables gets instantiated.

```
$D(\texttt{vehicle}_f) == \{t\}$ $\Rightarrow$ POST element(successor$_f$,vehicle,t)
                                POST element(predecessor$_f$,vehicle,t)
$D(\texttt{successor}_f) == \{f'\}$ $\Rightarrow$ POST $\texttt{vehicle}_f = \texttt{vehicle}_{f'}$
$D(\texttt{predecessor}_f) == \{f'\}$ $\Rightarrow$ POST $\texttt{vehicle}_f = \texttt{vehicle}_{f'}$
$D(\texttt{vehicle}_f) == \{t\}$
        $\wedge$
overlap_time($f', f$) $\Rightarrow$ POST $\texttt{vehicle}_f \neq \texttt{vehicle}_{f'}$
```

light of this implication, the `all_different` constraints (9.3) are redundant. In reality, constraints (9.3) are only added to improve propagation, since the propagation for `all_different` is very powerful. We have also experimented with dynamically posting `all_different` constraints in the `tunneling` constraint, but that turned out to post too many constraints, causing poor search performance.

The basic constraint model, without the `predecessor` variables and the `all_different` constraints over `vehicle` variables, was first proposed by Kilborn in [133].

To get good search performance for this model it is crucial to define good variable and value ordering heuristics. Only `successor` variables are instantiated. The reason is that these variables are propagated much more than the others, thanks to the strong consistency algorithm for `all_different` due to Régin [181]. The `successor` variables are fixed in order of increasing domain size, i.e. using the well-known *first-fail* ordering, with the exception of preassigned activities, which are fixed first. Values are chosen in increasing connection time order, except for long connections. If a connection is long, for example over night, connecting it to the next *preassigned* activity is tested first.

The result is a model that captures all tail assignment constraints except the maintenance constraints. The model works very well for problems without many flight restriction constraints, showing almost linear time behavior for problems of increasing size. For more results of the basic model we refer the reader to Kilborn's paper [133]. Since the model does not capture maintenance constraints, it is not useful stand-alone in practice. However, as we will see in later chapters, CSP-TAS^{relax} is still useful e.g. for preprocessing and column generation integration.

9.3 Model Csp-Tas

There are two main problems with model Csp-Tas^{relax}. Firstly, it does not handle maintenance constraints, or other cumulative constraints, at all. Since these constraints are vital to obtain feasible solutions, this makes Csp-Tas^{relax} unusable for stand-alone solution of the tail assignment problem. Previous unsuccessful attempts have been made [83] to adjust the model to handle maintenance constraints.

The second problem with Csp-Tas^{relax} is that it works poorly in the presence of complicated flight restriction constraints. Flight restriction constraints are modeled by Csp-Tas^{relax}, but the propagation of the `vehicle` variables in Csp-Tas^{relax} is too weak in practice for practical flight restrictions.

In the next two sections we will describe how these two serious drawbacks are solved. The resulting model, Csp-Tas, which fully models the tail assignment problem, is presented in Model 9.2. Compared to model Csp-Tas^{relax}, two things are added:

- the `reachability` propagation algorithm (9.9) improves the propagation for the `vehicle` variables, to handle instances with many flight restriction constraints;
- the `pricing` constraint, which handles the maintenance constraints, and is also used to define the variable and value ordering to solve Csp-Tas.

Clearly, model Csp-Tas^{relax} is a relaxation of model Csp-Tas, which explains its name. In Sections 9.4 and 9.5 we will present the `reachability` propagation algorithm and the `pricing` constraint, respectively. Section 9.6 will present the variable and value ordering heuristics.

9.4 Reachability Propagation: Reducing Thrashing in the Presence of Multiple Flight Restrictions

As we have already discussed, the flight restriction constraints are in fact modeled already in the basic model, but when several restrictions are present, the performance of the search deteriorates. Since the propagation for the `vehicle` variables is poor, the search algorithm ends up instantiating partial routes, which do not fit together because of the restrictions, resulting in excessive thrashing. One way to deal with this problem is improved propagation for the `vehicle` variables. The introduction of `all_different` constraints over `vehicle` variables in the previous section is an attempt at improving this, but it is not enough.

Our proposal is instead to add a propagation algorithm that for each activity keeps track of the number of predecessor and successor activities which can be reached by each aircraft. That an activity is *reachable* by aircraft t means that a route from the carry-in of t to the activity, satisfying all flight

Model 9.2 CSP-TAS

$$\forall f \in F: \quad D(\texttt{successor}_f) = \bigcup_{f' \in F} (f^{'} \text{ possible follow-on of } f)$$

$$\forall f \in F: \quad D(\texttt{predecessor}_f) = \bigcup_{f' \in F} (f \text{ possible follow-on of } f')$$

$$\forall f \in F: \quad D(\texttt{vehicle}_f) = \bigcup_{t \in T} (t \text{ can be assigned to } f)$$

$$\texttt{inverse(successor, predecessor)} \tag{9.5}$$

$$\wedge$$

$$\texttt{all\_different}(\texttt{successor}_1, \ldots, \texttt{successor}_{|F|}) \tag{9.6}$$

$$\wedge$$

$$\texttt{all\_different}(\texttt{vehicle}_{G(f)}) \quad \forall f \text{ s.t. } |T_f| \neq |\texttt{T}| \tag{9.7}$$

$G(f)$: Activities overlapping the start time of activity f

$$\wedge$$

$$\texttt{tunneling(successor, predecessor, vehicle)} \tag{9.8}$$

$$\wedge$$

$$\texttt{reachability(successor, predecessor, vehicle)} \tag{9.9}$$

$$\wedge$$

$$\texttt{pricing(successor, predecessor, vehicle)} \tag{9.10}$$

restrictions, exists. An activity can be reached in the forward direction, via `successors`, and in the backward direction, via `predecessors`. If no predecessors, or no successors, can be reached by an aircraft, this aircraft can be removed from the `vehicle` domain of the activity, since it is clearly impossible to reach the activity itself.

We will only describe the propagation algorithm of `reachability`, and not the general semantics of the constraint, and we therefore refer to it as a propagation algorithm, rather than as a constraint. However, it is simple to think of `reachability` as a constraint ensuring that each activity is reachable in both the forward and backward direction by at least one aircraft.

To keep track of the number of reachable predecessors and successors, our propagation algorithm will maintain two *labels* per activity and aircraft. The first label, called the *forward label*, counts how many predecessors of each activity that have a non-zero forward label for the aircraft, i.e. how many predecessors that can be reached, in the forward direction, by the aircraft.

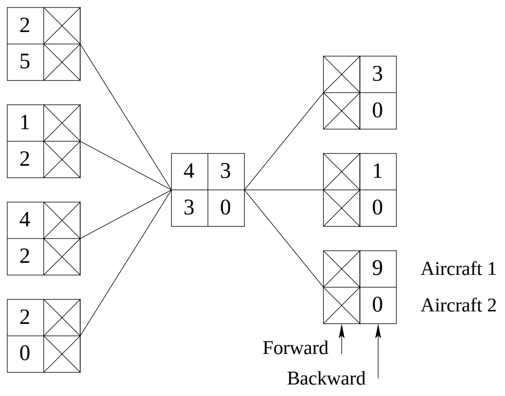


Figure 9.1 An example of forward and backward labels with only two vehicles. Labels which depend on activities not in the figure have not been filled in.

The other (the *backward label*) counts how many successor activities have a non-zero backward label for the vehicle, i.e. how many successors can be reached, in the backward direction, by the aircraft.

In the example in Figure 9.1, all of the predecessors have non-zero forward labels for aircraft 1, and the forward label is thus 4. Since none of the successors have non-zero backward labels for aircraft 2, the backward label is 0, and so on. As a consequence, the activity cannot be covered by aircraft 2.

To explain the `reachability` propagation algorithm, some additional notation is required. The initial possible successors and predecessors of activity $f \in F$ are denoted by S_f and P_f, respectively. The forward label of activity f for aircraft t is $\texttt{forward\_label}_f^t$ and the backward label $\texttt{backward\_label}_f^t$. Let us further denote the set of carry-in activities by F_c, and let c_f^t be 1 if f is the carry-in activity of aircraft t and 0 otherwise. Now, the labels have the following relationship:

$$\texttt{forward\_label}_f^t = c_f^t, \quad \forall f \in F_c, \forall t \in T, \tag{9.11}$$

$$\texttt{backward\_label}_f^t = c_f^t, \quad \forall f \in F_c, \forall t \in T, \tag{9.12}$$

$$\texttt{forward\_label}_f^t = \sum_{\substack{f' \in P_f \& \\ \texttt{forward\_label}_{f'}^t > 0}} 1, \quad \forall f \notin F_c, \forall t \in T, \tag{9.13}$$

$$\texttt{backward\_label}_f^t = \sum_{\substack{f' \in S_f \& \\ \texttt{backward\_label}_{f'}^t > 0}} 1, \quad \forall f \notin F_c, \forall t \in T. \tag{9.14}$$

Maintaining the Labels During Search

To initially set the forward and backward labels to their correct values is simple: Just set the forward labels by counting labels for predecessors, starting with the carry-ins and ending with the activity with the latest departure time. The backward labels are set in the opposite direction. The carry-in activities always have one forward and one backward label, and aircraft restricted from flying an activity of course always get forward and backward labels 0. Maintaining the correct labels once `successor`, `predecessor` and `vehicle` domains are changed during search is more complicated.

Let us first look at the case when f' is removed from $D(\texttt{successor}_f)$. In case f' is a carry-in activity, nothing happens, as carry-ins always have one single label. If f' is not a carry-in activity, the procedure SuccessorRemoval(f, f') in Algorithm 9.4 is applied.

What happens is that first the forward label on activity f' is decreased by 1 if both f and f' have non-zero forward labels, as one of the routes to activity f' is now removed. If the decrement makes the forward label 0, all successors of f' will potentially be affected, and ForwardRemoveAircraft must be executed. ForwardRemoveAircraft iteratively decreases forward labels as long as some label becomes 0 when decreased. The use of a queue makes sure that the labels are processed in the proper order, i.e. all predecessors of f are processed before f is processed. Upon termination of procedure SuccessorRemoval, all forward labels are correct with respect to the variables. Backward labels are treated analogously. Algorithm 9.5 shows the algorithm to apply if aircraft t is removed from $D(\texttt{vehicle}_f)$, and should be self-explanatory in light of the discussion above.

Value insertions are treated much the same way as the removals, except that instead of updating forward/backward when the label becomes 0, the label is updated when it is increased from 0 to 1.

In terms of complexity, the worst case is when all labels for all activities must be updated each time a value is removed or inserted. If there are $|T|$ aircraft, $|F|$ activities and each activity has m successors in average, the worst-case complexity of each successor removal is thus $\mathcal{O}(|T||F|m)$, which is not that good. However, in practice this behavior will not occur. The reason is that in the initial stage of the search, most activities can be reached via several routes, so most labels will have high values. Labels will thus seldom become 0, which means that only one or a few activities have to update at each removal. As the problem gets more fixed, more labels take values close to 0. But on the other hand, each activity does not have as many possible successors at this stage, which means only few labels have to be updated anyway. So in practice, this propagation algorithm works well, as Section 9.8 will show.

It should be observed that this propagation algorithm could probably have been stated in terms of a number of simpler constraints rather than as a global propagation algorithm. The propagation algorithm does not provide extra

Algorithm 9.4 Successor removal propagation. Procedure SuccessorRemoval is executed when f' is removed from $D(\texttt{successor}_f)$.

```
procedure SuccessorRemoval($f, f'$)
    for all aircraft $t$ do
        if $\texttt{forward\_label}^t_f = 0$ or $\texttt{forward\_label}^t_{f'} = 0$ then
            do nothing
        else
            $\texttt{forward\_label}^t_{f'} \leftarrow \texttt{forward\_label}^t_{f'} - 1$
            if $\texttt{forward\_label}^t_{f'} = 0$ then
                ForwardRemoveAircraft($t, f'$)
            end if
        end if
    end for
end procedure

procedure ForwardRemoveAircraft($t, f$)
    $q \leftarrow$ empty queue
    $q$.push($f$)
    while $q$ not empty do
        $i = q$.pop()
        for all successors $j$ of $i$ do
            if $\texttt{forward\_label}^t_j \neq 0$ and $j$ not carry-in then
                $\texttt{forward\_label}^t_j \leftarrow \texttt{forward\_label}^t_j - 1$
                if $\texttt{forward\_label}^t_j = 0$ then
                    $D(\texttt{vehicle}_j) \leftarrow D(\texttt{vehicle}_j) \setminus t$
                    $q$.push($j$)
                end if
            end if
        end for
    end while
end procedure
```

Algorithm 9.5 Aircraft removal propagation. This procedure is executed when t is removed from $D(\texttt{vehicle}_f)$.

```
procedure AircraftRemoval($t, f$)
    if $\texttt{forward\_label}^t_f > 0$ then
        $\texttt{forward\_label}^t_f \leftarrow 0$
        ForwardRemoveAircraft($t, f$)
    end if
end procedure
                                        ▷ And analogously for the backward label
```

propagation because of the fact that it is implemented as a global algorithm. The decision to model it this way is rather motivated by performance, since posting too many constraints adds an administrative overhead to the search algorithm. Also, the algorithm has been implemented as a constraint in the sense that it reacts to domain changes as a constraint, but it really only consists of the propagation algorithm.

9.5 The `pricing` Constraint: Handling the Cumulative Constraints

In Sections 5.3 and 5.4 we described the column generation pricing algorithm, which given a set of dual costs can find a set of minimum cost routes,[1] satisfying all flight restriction and maintenance constraints. In the pricing algorithm, the maintenance constraints are modeled as *resource constraints*, making the pricing problem a resource constrained shortest path problem.

Now, since we already have an implementation of the cumulative constraints in the pricing algorithm, and would like to avoid implementing the same thing twice, it seems reasonable to try to re-use the pricing algorithm for the constraint model as well. The pricing algorithm implementation is highly tuned and general, as it is customizable via the domain-specific Rave language [13], making it possible to model any kind of cumulative constraint. Completely re-implementing this functionality in a constraint model would be cumbersome indeed. However, as the pricing algorithm is labeling-based [75], it can easily allow us to check feasibility by checking whether

1. Each activity is reachable in the forward direction, i.e. there is at least one route arriving at each activity.

2. There exists at least one legal route per aircraft.

Unfortunately, this check is *incomplete*, in the sense that a partially instantiated network can pass the feasibility check even if it is infeasible, e.g. if an activity is the only possible successor of several activities. However, for a fully instantiated network it is complete, so the feasibility check will never allow solutions violating cumulative constraints to be created.

By adding a constraint that tunnels the changes in the variable domains to the internal structures used in the pricing algorithm, and performs the feasibility check described above, we have created a constraint that makes sure that a solution must be feasible with respect to cumulative constraints. The constraint (which we call the `pricing` constraint) is not very strong in terms of propagation, but fortunately practical experience tells us that common cumulative constraints, like maintenance constraints, are seldom very tight,

[1] A least cost column is found, along with a predefined number of other low-cost columns, but not necessarily the least cost ones.

so this is often not a problem in practice. The only propagation performed by the `pricing` constraint is that in case no route for a certain aircraft exists to activity f, this aircraft is removed from $D(\texttt{vehicle}_f)$.

One major concern with this constraint is the fact that it is fairly expensive to check. Since the column generation pricing algorithm is an external module to the constraint solver, it is difficult to customize it perfectly to the needs in the constraint solver case. We can customize it to a large extent by deciding on the number of labels, number of generated routes etc, but checking the `pricing` constraint still takes too long for it to be performed at every domain change. Instead, we have to settle for checking/propagating the constraint at some search nodes during the search, and at the very end. However, as the next section will show, this fits rather well into the full picture.

9.6 Ordering Heuristics for Csp-Tas

Model Csp-Tas is a complete model for the tail assignment problem. However, to practically solve tail assignment instances, well-working variable and value ordering heuristics must also be defined. Unfortunately, the Csp-Tas^{relax} variable ordering heuristic of instantiating preassigned activities first, and then instantiate according to increasing `successor` domain size, does not work well at all for Csp-Tas. Instead, we must again take a route-oriented view of the problem. Instead of instantiating single variables based on their local properties, e.g. domain size, an entire route is created, which is known to satisfy all flight restrictions and maintenance constraints. The route is then used to decide the order in which the `successor` variables are instantiated, by simply instantiating the activities in order of increasing start times. Once all `successor`s in a route have been instantiated, another route is created, and so on, until all aircraft have been assigned routes.

This goes hand-in-hand with the feasibility check performed in the `pricing` constraint, as the `pricing` constraint can provide the necessary routes. When checking feasibility using the pricing algorithm, a labeling sweep is performed which results in routes for all aircraft, if such exist. This suggests the following strategy for variable ordering and `pricing` constraint propagation:

- whenever all `successor`s in a previous route have been instantiated;
  - check feasibility of the `pricing` constraint;
  - if infeasible, backtrack;
  - if feasible, take one of the generated routes as the next route to instantiate.

The main question is which route to choose for instantiation. Our strategy is to instantiate routes in order of decreasing number of flight restrictions.

That is, to avoid the aircraft getting ‘trapped’ when most of the network is instantiated, routes for the aircraft which have the largest number of flight restrictions are instantiated before routes for the other aircraft. The aircraft with the largest number of flight restrictions is the aircraft t for which $\sum_f f_t$, where f_t is 1 if activity f can be operated by aircraft t and 0 otherwise, is minimal. Also, to avoid the aircraft of the last few routes which are instantiated getting stuck because there are no maintenance possibilities left, as few maintenance possibilities as possible should be used each time a route is instantiated. This can be at least approximately achieved by adjusting the pricing network costs for the `pricing` constraint. Remember that the pricing algorithm finds a set of least-cost routes with respect to the provided duals.[2] Since long, typically overnight, connections are used to perform maintenance, the use of such connections is penalized in the `pricing` constraint. This is done by setting the dual cost of all activities to the negation of the sum of the connection times to all successors.

Finally, to speed up the search in case backtracking is necessary, the entire route which is currently being instantiated is uninstantiated, instead of just the last instantiation. This makes the search incomplete, but is only done to avoid excessive backtracking, and excessive checking of the `pricing` constraint. Backtracking in single steps could potentially mean that the `pricing` constraint would have to be checked far too many times to be efficient, if a conflict is found half-way through instantiating a route. Instead, heuristic backjumping to the node where the route currently being instantiated starts is performed. As the variable ordering strategy above indicates, at this node the `pricing` constraint has been checked, so re-checking it is not necessary. To avoid the same route from being re-generated, a penalty is added to the ‘dual cost’ of all activities for which the `successor` variable is instantiated.

As we will see in Section 9.8, the ordering heuristic described above, together with `pricing` constraint propagation, works very well for our test instances. Before we present computational results, however, we will look at an alternative way of handling the cumulative constraints.

9.7 An Alternative Way to Handle Cumulative Constraints

Since the maintenance handling described in the previous section has very weak propagation, it does not work well with very tight cumulative constraints. In cases where only a few reset possibilities (e.g. night connections at maintenance stations) exist, the variable ordering can also be problematic. Imagine that one route has been instantiated, which uses a reset possibility needed by another, not yet instantiated, aircraft. Now, if this reset possibility is in

[2]In fact with respect to the reduced cost. But if the real costs are set to 0, the cost will only be minimized using the duals.

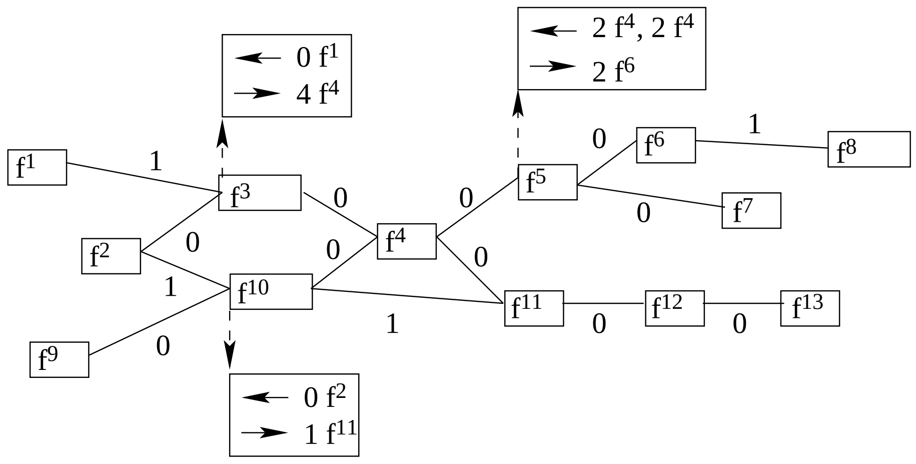


Figure 9.2 A tiny example of a cumulative constraint shortest path tree. The connections are marked 0/1 depending on whether the single cumulative constraint can be reset, and the activities are marked f^1-f^{13}. For activities f^3, f^5 and f^{10} shortest path labels are shown.

the middle of the instantiated route, many instantiations must be undone to make it possible to generate a route for the other aircraft. As mentioned in the previous section, most cumulative constraints used in practice are not that tight, but that does not mean that the cases where they are can be completely ignored.

As an alternative, we have therefore investigated another way to handle the maintenance constraints. This approach is similar to the `reachability` propagation algorithm in the sense that it keeps track of path information for each activity. For each activity, shortest path trees are maintained to previous and following reset possibilities, as well as the distances to the closest reset possibility in each direction, for each cumulative constraint type. The distance measure is of course resource consumption rather than actual distance. Clearly, there must exist routes passing through each activity for which the distances to the previous reset possibilities plus the distance to the next reset possibilities is less than or equal to the cumulative constraint limit. Therefore, whenever the sum of the shortest path from a previous reset possibility and the shortest path to a future reset possibility exceeds the cumulative constraint limit, a conflict is detected. Also, successors and predecessors via which the total shortest distance between reset possibilities is too long can be removed from the `successor` and `predecessor` domains.

Figure 9.2 shows a tiny example of a cumulative constraint shortest path tree. In the example it is assumed that only a single cumulative constraint exists, which counts the number of landings since the last reset. The arcs in

Figure 9.2 represent connections between activities, and are marked with 1 if the cumulative constraint consumption can be reset at this connection, and 0 otherwise. For simplicity, we do not care about the possibility to reset the consumption on activities. The activities are marked f^1-f^{13}. For activities f^3, f^5 and f^{10}, the shortest path distances to the closest reset possibility in both directions are shown. In both directions, the next/previous activity in the path to the closest reset, along with the actual distance, are shown. For activity f^5, two different paths with distance 2 exist in the backward direction, both passing through activity f^4.

Now, if the cumulative constraint limit is greater than or equal to 4, the problem is feasible, since the shortest paths through all activities is less than or equal to 4. However, if the limit is 3, the problem is clearly infeasible – no paths exist through activities f^3 and f^5 which 'consume' less than 4 landings. Also, if the connection between activities f^1 and f^3 is removed, e.g. because the successor of f^1 is instantiated to some other activity, not present in the figure, the best path through activity f^3 will consume at least 5 landings between consecutive resets (4 in the forward direction, and at least 1 in the backward direction, via activity f^2). From this example, it should be clear that maintaining shortest path trees makes it possible to detect cumulative constraint conflicts early in the search.

Unfortunately, setting up and maintaining the shortest path trees during search is rather complicated. Let us first introduce some necessary notation. Let $\texttt{backward\_dist}_{f,m}$ be the resource consumption from the closest earlier reset possibility for constraint type m to activity f, and $\texttt{forward\_dist}_{f,m}$ be the resource consumption to the closest future reset possibility. Further, let $P^{sp}_{f,m}$ and $S^{sp}_{f,m}$ form the shortest path trees. $P^{sp}_{f,m}$ is the set of predecessor activities of f in the 'backward' shortest path tree for constraint type m, via which the distance to the closest reset possibility is $\texttt{backward\_dist}_{f,m}$, and $S^{sp}_{f,m}$ are the successor activities in the 'forward' shortest path tree.

Algorithm 9.6 shows the basic procedures for calculating the distance (resource consumption) to the closest reset possibility in the backward and forward direction via a certain connection. These procedures constitute the only interface to the cumulative constraint logic. The idea is to simply check if the consumption can be reset on the connection, and if so set the shortest distance to the distance to/from the reset, depending on whether one is looking in the forward or backward direction. If the consumption cannot be reset, the connection consumption plus the consumption of the predecessor/successor is returned.

Algorithm 9.7 shows how the shortest path trees are initialized. The idea is simply to scan the predecessors/successors of each activity, and use the procedures in Algorithm 9.6 to compute the resource consumptions.

Algorithms 9.8 to 9.10 show the necessary algorithms to keep the shortest path trees consistent when a value is removed from a `successor` domain. When a successor is removed, SuccessorRemoval in Algorithm 9.8 is ap-

Algorithm 9.6 Basic algorithms calculating the distances to closest reset possibilities.

data

ResetPossible(f^1, f^2, m): Is it possible to reset constraint m on connection (f^1, f^2), including activity f^2?

ConsumptionForward(f^1, f^2, m): Consumption of resource type m on connection (f^1, f^2), including activity f^2. If consumption is reset on the connection or on activity f^2, the consumption *before* the reset is returned.

ConsumptionBackward(f^1, f^2, m): Consumption of resource type m on connection (f^1, f^2), including activity f^2. If consumption is reset on the connection or on activity f^2, the consumption *after* the reset is returned.

```
procedure CalculateForwardDistance($f, f', m$)
    if ResetPossible($f, f', m$) then
        return ConsumptionForward($f, f', m$)
    else
        return forward_dist$_{f',m}$+ ConsumptionForward($f, f', m$)
    end if
end procedure

procedure CalculateBackwardDistance($f', f, m$)
    if ResetPossible($f', f, m$) then
        return ConsumptionBackward($f', f, m$)
    else
        return backward_dist$_{f',m}$+ ConsumptionBackward($f', f, m$)
    end if
end procedure
```

plied. SuccessorRemoval simply checks if the removed successor was part of the forward shortest path tree, and re-computes the forward shortest path tree of the activity if all shortest path arcs in the forward direction have now been removed. If the re-computation changes the shortest path distance, re-computation is applied recursively to the predecessors. Also, when the shortest path distance is increased, the backward distances of all predecessors are checked by CheckPredecessors in Algorithm 9.9. Equivalent algorithms for predecessor removals are also necessary, of course, as well as algorithms handling backtracking and `vehicle` domain changes.

Clearly, handling the cumulative constraint this way requires more complicated propagation algorithms than the `pricing` constraint. One major drawback is that the propagation algorithms make the constraint handling

Algorithm 9.7 Algorithm initializing the distances to closest reset possibilities.

```
data
    CalculateForwardDistance: Algorithm 9.6
    CalculateBackwardDistance: Algorithm 9.6

procedure Initialize
    for all $m \in M$ do
        for all $f \in F$, in increasing departure time order do
            $P^{sp}_{f,m} \leftarrow \{\}$
            if $f$ is carry-in for aircraft $t$ then
                $\texttt{backward\_dist}_{f,m} \leftarrow r^t_m$
            else
                $\texttt{backward\_dist}_{f,m} \leftarrow \infty$
                for all $j \in D(\texttt{predecessor}_f)$ do
                    $dist \leftarrow$ CalculateBackwardDistance$(j, f, m)$
                    if $dist < \texttt{backward\_dist}_{f,m}$ then
                        $\texttt{backward\_dist}_{f,m} \leftarrow dist$
                        $P^{sp}_{f,m} \leftarrow \{j\}$
                    else if $dist = \texttt{backward\_dist}_{f,m}$ then
                        $P^{sp}_{f,m} \leftarrow P^{sp}_{f,m} \cup j$
                    end if
                end for
            end if
        end for
        for all $f \in F$, in decreasing departure time order do
            $\texttt{forward\_dist}_{f,m} \leftarrow \infty$
            $S^{sp}_{f,m} \leftarrow \{\}$
            if $|D(\texttt{successor}_f)| = 0$ then
                $\texttt{forward\_dist}_{f,m} \leftarrow 0$
            else
                for all $i \in D(\texttt{successor}_f)$ do
                    $dist \leftarrow$ CalculateForwardDistance$(f, i, m)$
                    if $dist < \texttt{forward\_dist}_{f,m}$ then
                        $\texttt{forward\_dist}_{f,m} \leftarrow dist$
                        $S^{sp}_{f,m} \leftarrow \{j\}$
                    else if $dist = \texttt{forward\_dist}_{f,m}$ then
                        $S^{sp}_{f,m} \leftarrow S^{sp}_{f,m} \cup i$
                    end if
                end for
            end if
        end for
    end for
end procedure
```

Algorithm 9.8 Successor removal propagation for the alternative cumulative constraint handling propagation.

```
1: procedure SuccessorRemoval($f, f'$)
2:     for all constraint types $m \in M$ do
3:         if $S^{sp}_{f,m} \cap f' \neq \emptyset$ then
4:             $S^{sp}_{f,m} \leftarrow S^{sp}_{f,m} \setminus f'$
5:             if $S^{sp}_{f,m} = \emptyset$ then
6:                 ReScanBackward($f, m$)                    ▷ Algorithm 9.10
7:             end if
8:         end if
9:     end for
10: end procedure
```

Algorithm 9.9 Algorithm checking distance for predecessors.

```
data
    AllVehiclesCareAbout($V, m$): Does constraint $m$ apply to all vehicles in set $V$?

1: procedure CheckPredecessor($f, m$)
2:     for all $i \in D(\texttt{predecessor}_f)$ do
3:         $dist \leftarrow$ CalculateBackwardDistance($i, f, m$)
4:         $V \leftarrow D(\texttt{vehicle}_f) \cap D(\texttt{vehicle}_i)$
5:         if $dist + \texttt{forward\_dist}_{f,m} > l_m$ and
               AllVehiclesCareAbout($V, m$) then
6:             $D(\texttt{predecessor}_f) \leftarrow D(\texttt{predecessor}_f) \setminus i$
7:         end if
8:     end for
9: end procedure
```

less flexible. For example, using the `pricing` constraint, it is fairly trivial to allow aircraft dependent cumulative constraint limits, since labeling is performed for one aircraft at a time.[3] In the propagation algorithms described here, this is very difficult to handle, since all aircraft are considered simultaneously. It could be handled e.g. by using the largest limit in the algorithms, i.e. using $l_m = max_{t \in T}(l^t_m)$. But if the limits differ a lot, this would give very poor propagation.

The maintenance handling proposed above has been tested on a number of instances where the cumulative constraints have been made very tight, causing the `pricing` constraint approach to fail. Unfortunately, the performance

[3] Algorithm 5.1 shows how aircraft dependent limits are treated in the pricing algorithm.

Algorithm 9.10 Algorithm recursively scanning distances of predecessors.

data
See Algorithm 9.9

```
procedure ReScanBackward($f, m$)
    $q \leftarrow$ empty queue
    $q$.push($f$)
    while $q$ not empty do
        $f' \leftarrow q$.pop()
        $dist_{old} \leftarrow \texttt{forward\_dist}_{f',m}$
        $\texttt{forward\_dist}_{f',m} \leftarrow \infty$
        for all $i \in D(\texttt{successor}_{f'})$ do
            $dist_i \leftarrow$ CalculateForwardDistance($f', i, m$)
            $V \leftarrow D(\texttt{vehicle}_{f'}) \cap D(\texttt{vehicle}_i)$
            if $dist_i + \texttt{backward\_dist}_{f',m} > l_m$ and
                AllVehiclesCareAbout($V, m$) then
                $D(\texttt{successor}_{f'}) \leftarrow D(\texttt{successor}_{f'}) \setminus i$
            end if
            if $dist_i < \texttt{forward\_dist}_{f',m}$ then
                $\texttt{forward\_dist}_{f',m} \leftarrow dist_i$
            end if
        end for
        for all $i \in D(\texttt{successor}_{f'})$ do
            if $dist_i = \texttt{forward\_dist}_{f',m}$ then
                $S^{sp}_{f',m} \leftarrow S^{sp}_{f',m} \cup i$
            else
                $S^{sp}_{f',m} \leftarrow S^{sp}_{f',m} \setminus i$
            end if
        end for
        if $\texttt{forward\_dist}_{f',m} > dist_{old}$ then
            CheckPredecessors($f', m$)
        end if
        if $\texttt{forward\_dist}_{f',m} \neq dist_{old}$ then
            for all $j \in D(\texttt{predecessor}_{f'})$ do
                if $S^{sp}_{j,m} \cap f' \neq \emptyset$ then
                    $q$.push($j$)
                end if
            end for
        end if
    end while
end procedure
```

of the resulting model was not good. It was difficult to find good variable and value ordering heuristics, causing excessive backtracking even for instances for which the cumulative constraints are not tight. However, some of the smaller test instances, which could not be solved with the `pricing` constraint approach, could be solved with the shortest path approach. The conclusion is thus that handling the cumulative constraints with the shortest path propagation is promising, but better variable and value ordering heuristics must be invented. Of course, the shortest path propagation can also be used together with the `pricing` constraint, to provide additional propagation. However, our tests indicate that this does not make much difference in terms of search performance. Because of these problems, we have not used the shortest path propagation in our constraint model.

9.8 Computational Results

In this section we will present computational results using model Csp-Tas for a set of real-world test instances. The instances come from different planning months at the same medium sized airline. The aircraft included are a mix of M82, M83 and M88 aircraft. Observe that these instances differ from the ones used in the second part of this thesis. The reason is simply that these results are newer, and the instances used here better represent the current production problems solved. The instances also contain more aircraft and activities than the previously presented instances.

Table 9.1 presents the instances, and shows the running times of model Csp-Tas and of an aggressive fixing heuristic method based on column generation (see Chapter 11), which was the best known method for quickly producing solutions prior to model Csp-Tas. The pure column generation fixing heuristics (as described in Chapter 7) are not included in the comparison, as they have the property that they can leave activities unassigned. Comparing running times of heuristics that can leave activity unassigned with heuristics that cannot do this does not make sense, and hence this comparison was skipped.

For all instances, a constraint specifying that the aircraft should return to base with regular intervals is present. For some instances, so-called ‘A-check’ and lubrication maintenance constraints are also present. From the last two columns of the table, it is clear that Csp-Tas produces solutions significantly faster than the fixing heuristic method. For these instances, the constraint model is about twice as fast as the fixing heuristic.

Table 9.2 shows the benefit of the reachability algorithm. The table shows the behavior of the basic model Csp-Tas^{relax} compared to Csp-Tas^{relax} combined with the reachability algorithm. When the reachability algorithm is included, the variable ordering is changed to first instantiating the earliest uninstantiated activity of the most restricted aircraft, using the labels in the reachability algorithm. We thus do not generate and fix entire routes, but

Instance	#Flight legs	#Aircraft	#Maintenance Constraints	Fix. Heur. Time	CSP-TAS Time
Instance 1	4696	33	1	140	70
Instance 2	5571	31	1	247	98
Instance 3	5127	31	1	184	82
Instance 4	4423	31	2	232	99
Instance 5	5304	31	1	175	78
Instance 6	5816	31	2	209	99
Instance 7	4932	31	1	156	68
Instance 8	5068	31	1	1468	106
Instance 9	4987	31	1	218	66
Instance 10	3906	30	3	321	112
Instance 11	4048	30	3	235	112
Instance 12	4249	30	3	249	134
Instance 13	3865	30	3	184	100

Table 9.1 Test instances and running times for model CSP-TAS, compared to a column generation-based fixing technique.

still use the reachability labels to help guide the search. For entries marked with a star, no solution was found within the predefined limits, which were set to 1800 seconds running time or as many backtracks as there are activities after preprocessing in the problem. The running time limit is never reached in Table 9.2, and the measured times for the cases where no solution is found are thus the times to reach the backtracking limit. The reachability propagation clearly helps, as without it only two instances can be solved within the limits, while when it is added, all but two instances are solved.

Table 9.3 shows the importance of the ordering heuristics. The first columns show the full model using the old ordering heuristics, i.e. first-fail and short-connections-first, while the last columns show the full model with the route-fixing ordering. When using the old heuristics, we have applied the `pricing` constraint propagation every $|F|/|T|$ search nodes, to approximately apply it once for every fixed aircraft route. This was to get a fair comparison with the route-fixing ordering, which applies the propagation exactly once per route. It is clear from the table that the old ordering heuristics do not work well at all, even in the presence of reachability propagation and the `pricing` constraint. No solution is found for any instance, and the excessive thrashing leads to very long running times.

	Csp-Tas^{relax}		Csp-Tas^{relax} with reachability	
Instance	#Backtracks	Time	#Backtracks	Time
Instance 1	*2388	73	2	91
Instance 2	*2965	122	18	63
Instance 3	0	57	1	74
Instance 4	*2379	74	*2379	450
Instance 5	*2839	74	44	75
Instance 6	*2652	76	29	71
Instance 7	*2474	786	5	58
Instance 8	*2858	467	34	67
Instance 9	*2604	72	*2604	1283
Instance 10	*2040	63	3	54
Instance 11	*2232	102	17	68
Instance 12	26	70	2	68
Instance 13	*2163	69	591	67

Table 9.2 The performance impact of the reachability algorithm for model Csp-Tas^{relax} – results when using Csp-Tas^{relax} with and without the reachability algorithm. '*' markings means that no solution was found within the set search limit, which was either 1800 seconds or as many backtracks as there are activities in the problem.

9.9 Summary

In this chapter, we have presented a full constraint model, Csp-Tas shown in Algorithm 9.2, for the tail assignment problem. The model includes a reachability algorithm that provides efficient propagation required to handle the flight restriction constraints. It also uses the column generation pricing algorithm to model cumulative constraints, which would otherwise require substantial effort to re-implement. Soft cumulative constraints, while handled in the pricing algorithm, are ignored by the constraint model.

One potential drawback of the maintenance constraint handling is that the propagation is not that strong. However, since these constraints are seldom extremely tight in practice, the limited propagation is enough to achieve good performance. The fact that the handling of these constraints has not been re-implemented in the constraint model makes it easier to add new cumulative constraints. To achieve stronger propagation, an alternative propagation algorithm for cumulative constraints has been investigated. This algorithm maintains shortest path trees for distances to reset possibilities. Unfortunately, it does not work well enough in practice, because no good enough

	Csp-Tas with Csp-Tas^{relax} ordering		Csp-Tas with Csp-Tas ordering	
Instance	#Backtracks	Time	#Backtracks	Time
Instance 1	*2388	1182	177	70
Instance 2	1391	*1878	0	98
Instance 3	*2757	1238	0	82
Instance 4	*2379	1353	146	99
Instance 5	*2839	1191	116	78
Instance 6	*2652	1755	0	99
Instance 7	*2474	1458	0	68
Instance 8	1268	*1867	0	106
Instance 9	*2604	1227	0	66
Instance 10	*2040	1071	0	112
Instance 11	*2232	1009	0	112
Instance 12	*2226	1172	0	134
Instance 13	*2163	752	0	100

Table 9.3 The performance impact of the ordering heuristics for model Csp-Tas – Results when using the old and new variable ordering heuristics for Csp-Tas. '*' markings means that no solution was found within the set search limit, which was 1800 seconds or as many backtracks as there are activities in the problem.

variable and value ordering heuristics have been found. It is possible that the reachability algorithm, as well as the idea of re-using a column generation pricing problem, could be of use also for other types of applications, similar to tail assignment.

By re-using a column generation pricing problem to check feasibility and improve ordering heuristics, we have shown that mathematical programming can be integrated with constraint programming to model difficult constraints. Further integration between the approaches will likely benefit a lot from the results presented here. We also conjecture that adding future constraints to our tail assignment model will often be easier for the constraint model than for the full column generation model, making it an excellent tool for experimentation.

Ten

Preprocessing with Constraint Programming

In this chapter we will show how constraint programming can be used to preprocess the tail assignment problem, which in turn improves the performance of the column generation algorithm, both in terms of running time and solution quality.

As discussed in the preceding chapters, we have chosen to model the flights and connections with a connection network. The use of a connection network makes it possible to consider individual connections. The price we have to pay for using a connection network representation is that the network size grows quadratically with the number of activities in the problem. To reduce the negative effects on performance caused by the size of the network, we have developed preprocessing methods that filter away connections from the network. Not only does this reduce the memory consumption, but it also reduces the complexity for both the RMP and the pricing problem. Removing connections directly impacts the performance of the pricing problem, by decreasing the number of arcs. Furthermore, when the connection removal leads to a flight having a unique predecessor and/or successor, one can also aggregate activities. Removing connections might thus also reduce the RMP complexity, by decreasing the number of constraints. It should be observed that this kind of preprocessing does not change the existence or quality of solutions to the problem.

In Sections 10.1 and 10.2, two preprocessing techniques, one rather simple and one more general based on constraint propagation, are presented. Section 10.3 shows how cost information can be used to obtain powerful heuristic preprocessing, and in Section 10.4 we summarize this chapter.

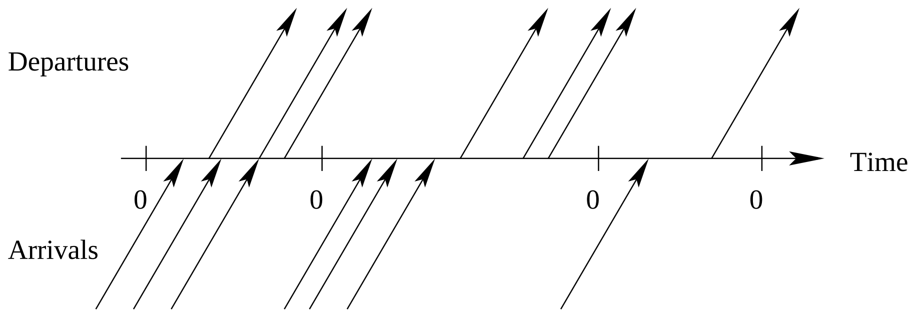


Figure 10.1 The aircraft count filter finds points with no aircraft on ground. The number of aircraft initially available on ground at each airport is given by the carry-in activities of the aircraft.

10.1 Simple Aircraft Balance Preprocessing

A lot of connections can be removed by studying the balance of arriving and departing activities at a particular airport. The number of aircraft initially available on ground at each airport is given by the carry-in activities of the aircraft. Counting the number of aircraft that are on ground at a particular airport at each point in time, a set $\mathcal{T}$ of time points when no aircraft are on ground can be detected. Now, assuming that all activities will be flown, all connections from activities arriving before $\tau \in \mathcal{T}$ to activities departing after τ can be removed, since it is clear that no aircraft can be on ground at time τ. The connections removed in this way might very well be legal according to all constraints, but they can never be part of a solution with all activities assigned. With the same reasoning, it is also possible to deduce that none of the activities arriving before τ can be the last activity in a route. However, if all activities cannot be covered, the balance does not necessarily hold, and the procedure is not valid. We call this simple preprocessing procedure the *aircraft count filter*. Figure 10.1 shows an example where there are four points in time with no aircraft on ground, provided there are no aircraft on ground initially. In the figure, arriving and departing activities at an airport are shown as incoming and outgoing arcs, respectively.

The method is very similar to the procedure of 'making islands', which identifies islands of grouped arriving and departing flights, and often is used in North American hub-and-spoke networks to heuristically reduce problem sizes for aircraft planning problems [112]. But since we are using a dated connection network, and thus do not have to make any assumptions about at which times there are no aircraft on ground, the method is exact in our case. The aircraft count filter is very fast, and can remove a lot of connections, specially for instances where connections are generated without any maximum connection time constraint.

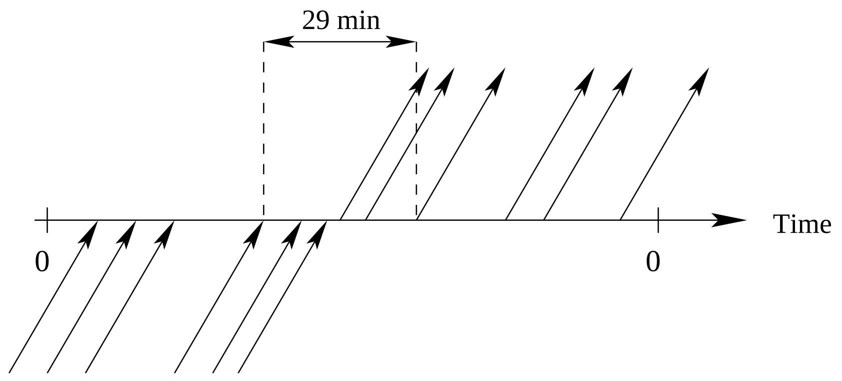


Figure 10.2 Example where aircraft counting cannot make any reductions, assuming a minimum connection time of 30 minutes.

However, it is not difficult to see that there are situations in which the aircraft count filter cannot make any reductions, even though reductions are possible. Figure 10.2 shows such an example, assuming a minimum connection time of 30 minutes. Since there are no aircraft on ground initially, there is no point in time with zero aircraft on ground, except after all six departures. Thus, the aircraft count filter cannot make any reduction between these times. But it is clear that since the connection time is too short for the last three arrivals to connect to the first three departures, these departures must be connected to the first three arrivals. It is therefore possible to remove the connections from the first three arrivals to the last three departures - *but the aircraft count filter will not detect this.* Next, we will look at a more general preprocessing method, that can handle this case as well as even more complicated cases.

10.2 Consistency-Based Preprocessing

To get a robust and general algorithm that can detect several impossible 'patterns' of flights, we propose to use consistency techniques. In the constraint models described in Chapter 9, removing connections that cannot be used in a solution is equivalent to establishing consistency. To achieve arc consistency with respect to the `all_different`, the algorithm of Régin [181] can be used. Filtering away impossible connections is thus quite straightforward – set up the constraint model, reduce domains as much as possible without any variable instantiation, and filter away the connections that are determined as being inconsistent. Since the main ingredient in the model is the `all_different` constraint, most of the domain reductions will come from its consistency algorithm. The arc consistency algorithm for `all_different` has polynomial time complexity, and runs very quickly in practice. However, for instances which are constrained by e.g. flight restrictions, reductions will also be obtained by propagation for these constraints. For this *propagation filter*, we use model CSP-TAS^{relax} rather than the full model CSP-TAS, because the extra constraints in CSP-TAS are not very useful when no variable instantiations are performed, but they still take some time to propagate.

Since the propagation obtains arc consistency for `all_different`, *all* impossible connections with respect to the `all_different` constraint over the `successor` variables will be detected, including the ones caught by the aircraft count filter and the ones in Figure 10.2. But it can also catch other, more subtle, patterns without having to explicitly identify them beforehand. For example, it will detect patterns not only at single airports, as the aircraft count filter, but patterns involving several airports. Unfortunately, it is difficult to graphically exemplify these kinds of patterns, as they are often quite complicated. Since a relaxation of the problem is used, where maintenance and other individual constraints are not included, and the propagation does not achieve global consistency with respect to all constraints, there might still exists connections left in the connection network after the filtering that are not possible to use in a solution with all activities assigned.

Table 10.2 shows the effect of the aircraft count and propagation filters on a set of test instances. The details of the test instances are presented in Tables 6.1 and 10.1. For some instances, as many as 97% of the connections can be removed with the aircraft count filter alone, while for others as few as 4.5% of the connections can be removed. The amount of reduction when using the aircraft count filter depends a lot on the network structure. For instances where many connections are allowed at remote airports, the filter will work better than on problems where most connections are at major hubs. The table further shows that as many as 86% more connections are removed when using the propagation filter, compared to the aircraft count filter alone. Overall, the propagation filter reduces about twice as much as the aircraft count filter. The number of activities is reduced by up to 60% by node aggregation.

Since the aircraft count filter removes a subset of the connections removed by the propagation filter, it suffices in theory to only apply the propagation filter. However, in practice it is a good idea to apply them both. The reason is that the aircraft count filter can be applied even before connections are generated, since it is only based on activity balance, and it can thus help us to avoid ever generating many useless connections. Since checking all the connection constraints on a large number of connections can be quite time consuming, generating fewer connections can often lead to substantially shorter running times. Unlike the aircraft count filter, the propagation filter requires the entire connection network to be generated before any filtering can be made.

Name	Period	Fleet	#Aircraft	#Flight legs	Maint. constr.	Flight restr.
M81_1w	1 week	MD81	66	2667	No	No
M82/3_1m	1 month	MD82/83	34	4689	Yes	Yes
A320_2_1m	1 month	A320	22	3220	No	No
MDDC9_1w	1 week	MD82/83/88 DC9	48	3533	Yes	Yes
ALL_1w	1 week	MD82/83/88 DC9 B757/767	69	4568	Yes	Yes
MDS_1m	1 month	MD82/83/88	33	6659	Yes	Yes
MDS_1m2	1 month	MD82/83/88	31	5946	Yes	Yes

Table 10.1 Test instances for Chapter 10, in addition to those presented in Table 6.1.

	No preprocessing		Aircraft count filter		Propagation filter	
Instance	Activities	Conn.	Activities	Conn.	Activities	Conn.
A320_1m	5661	2319958	3290	1126153	3188	40451
B737_1m	6013	4715507	3065	4518890	3049	218582
B757_1m	4872	3193135	2619	2982643	2580	105051
M81_1w	2581	269744	1894	257436	1889	156731
M82/3_1m	2431	457654	1377	17189	1345	10757
A320_2_1m	3942	234905	2694	182176	2653	149843
DC9_1w	544	8317	221	4287	216	3365
DC9_1w2	568	36529	259	5102	251	3177
DC9_10d	735	62491	334	6704	324	3898
DC9_2w	1140	151001	504	9165	488	5420
DC9_1m	2378	664300	1034	18059	996	10105
MD82_2w	802	46606	354	3401	342	2783
MD82_1m	1760	231450	785	12888	769	11697
MDDC9_1w	1425	193650	851	175468	842	42486
ALL_1w	1932	83457	1221	76804	1205	69005
MDS_1m	4931	2286232	2666	160415	2474	24151
MDS_1m2	5816	2488467	2864	167347	2796	29943

Table 10.2 Reductions in the number of activities and connections by the aircraft count and propagation filters. The table shows the number of activities and connections left after the filters have been applied.

Effect on Column Generation

Table 10.3 shows the column generation performance without preprocessing, and when using the aircraft count and propagation filters. In the table we show the running times, including preprocessing time, and objectives when using 100 column generation iterations on the root LP. The reason why we only show results for the LP relaxation is that the main contribution of the preprocessing is the improved convergence, which is best illustrated on the relaxed problem. There are two reasons why we only show a fixed number of iterations, rather than solving to LP optimality. Firstly, this is what is done in practice, since it is often difficult to judge how long it will take to reach LP optimality, and it is not necessary to find the optimal LP solution to find good integer solutions. Secondly, for the unfiltered instances it would take far too long to solve to LP optimality, as only the first 100 iterations take a very long time. Instances for which LP optimality is not reached within 100 iterations are marked with '*'.

For all instances, the effect of using preprocessing is significant. In fact, for the larger ones it is very hard to even find an LP solution with all activities assigned without preprocessing. This is the reason for the substantial objective value differences for some of the tests – without preprocessing a large portion of the objective is made up of slack costs. The running times for all instances are substantially decreased already when using only the aircraft count filter. But when using the propagation filter, both the running times and the objective can be further improved, in many cases dramatically. Out of the 13 test cases, LP optimality is reached for seven instances within 100 iterations when using the propagation filter, compared to two in the unfiltered case. For the instances for which no optimal solution was found within 100 iterations, most LP objective are within 0.40% from the lower bound, and do not use any slack columns. The exceptions are the three big instances in the bottom, for which the lower bound is of poor quality. For the cases where LP optimality is reached using only the aircraft count filter, the running times are lower when using the propagation filter. The time required to perform the filtering is in all cases negligible compared to the total running time.

One of the advantages with propagation-based preprocessing is that one does not have to resort to heuristic, or very specific, methods to keep the problem compact. One example of a heuristic preprocessing procedure could be not to allow long connections, e.g. longer than 24 hours, at small airports, but allow them at the hub. However, with such specific heuristics, it is often difficult to set the parameters correctly - what is a hub, why should 24 and not 48 hours be used, and what is a 'small' airport, for example? It is not uncommon that some remote airports in a flight network are slightly bigger than other remote airports, but smaller than the main hub. At such airports it is sometimes possible to have long connections, and sometimes not. Also, removing long connections to reduce the network might have the effect of

	No preprocessing		A/C count filter		Prop. filter	
Instance	Time	LP obj.	Time	LP obj.	Time	LP obj.
A320_1w	8267	*1300120	2951	*715596	1803	628145
B737_1w	4813	*4978248	2215	4973650	495	4973650
B757_1w	4564	*4507347	2766	4388950	821	4388950
DC9_1w	467	12101800	37	12101800	30	12101800
DC9_1w2	505	16662800	53	16662800	41	16662800
DC9_10d	1358	*21380724	210	21314300	147	21314300
DC9_2w	3615	*40780568	456	*33236381	438	*33088102
DC9_1m	20827	*135714910	2011	*85470479	1921	*71206556
MD82_2w	1932	*30682915	189	23628800	145	23628800
MD82_1m	12322	*89358995	1109	*58233165	1041	*54891828
MDDC9_1w	57509	*109951592	19178	*82975360	10531	*82884451
MDS_1m	52177	*305487048	16780	*222635955	15189	*99030227
MDS_1m2	119090	*2992461711	20872	*1542280574	18754	*402956666

Table 10.3 Effect of the propagation filter on column generation running time and solution quality. Here, only the linear programming relaxation of Path-Tas has been solved. The running times include the preprocessing time, which in all cases is negligible compared to the total running time. Objective values marked with '*' are not optimal.

removing opportunities for using aircraft as standby. Since we often want to use many stand-by opportunities, this is not good.

Our propagation filter does not use any heuristic assumptions, and thus avoids many of the problems with heuristic reduction methods. But there is no problem combining the propagation filter with a heuristic filter - the propagation will then tighten the heuristically restricted network even further, if possible.

10.3 Cost-Based Preprocessing

The preprocessing methods presented so far only deal with the feasibility aspect of the tail assignment problem. However, even better than removing connections that cannot be part of a solution, would be to remove connections that cannot be part of an *optimal* solution. As it is normally not known what is an optimal solution before solving the problem, this is difficult to achieve. Instead, we must settle for removing connections that cannot be part of a good enough solution, according to some measure.

A *Generalized Cardinality Constraint* (`GCC`) is a generalized version of the `all_different` constraint that allows values to be assigned to more than one variable. For a `GCC`, each value has a lower and an upper bound specifying how many times it may be assigned. The `all_different` constraint is a special case of `GCC`, where all values have bound $(0, 1)$. `costGCC` is a further extension of `GCC`, where each variable-value pair is associated with a cost, and there is a built-in sum constraint that limits the total cost to be below a specified bound H. Establishing arc consistency for `costGCC` thus means removing all values that will always lead to a solution with cost strictly greater than H. In [182], Régin discusses an efficient technique for obtaining arc consistency for `GCC`, and in [183] `costGCC` is discussed. Régin's consistency algorithm for `costGCC` uses reduced cost information to achieve arc consistency, and in that aspect it has some similarities with the work by Focacci et al. [90] on the TSP.

Since connection costs are available as input data, it is trivial to extend models Csp-Tas and Csp-Tas^{relax} to use `costGCC` instead of `all_different`. If a solution, and thus an upper bound, is known in advance, `costGCC` consistency can be used as an exact method to remove connections that can never improve this solution. The usefulness of the propagation will then depend on the quality of the solution – a poor quality solution will likely not give rise to much propagation, while a solution close to the lower bound will. However, one can also use `costGCC` as a heuristic preprocessing technique, without having a known upper bound. By specifying `costGCC`'s upper bound H to be a few percent above the network flow lower bound, as calculated by the `costGCC` propagation algorithm[1] [183], connections can be removed that can never be used in a solution with cost less than or equal to H. This requires some knowledge about the specific problem, however. If there no solution exists to the real problem below the heuristically set upper bound, the optimal solution might be filtered away.

Table 10.4 shows results when using the `costGCC` propagation filter as described above. A gap of $x\%$ means that only connections that are used by some solution to the `costGCC` constraint within $x\%$ of the lower bound are left after filtering. It is important to note that filtering with gap $x\%$ does *not* provide any guarantee that a solution within $x\%$ of the lower bound is found – it rather means that all connections that can *only* lead to solutions not within the gap are removed. So clearly, if no solution within the gap exists, there is a risk that connections necessary to obtain a solution are removed, and the reduced problem becomes infeasible. Instances for which the propagation made the problem infeasible are marked with '-', and non-optimal solutions are marked with '*'. The slack cost is 0 in all cases.

The table starts at gap 1.00% since most instances are not affected by gaps much higher than that. Observe that Table 10.4 measures the entire column generation solution process, including the integer fixing phase. The results are thus not directly comparable to those in Table 10.3. The reason for including

[1]This is the same bound as calculated by Tas^{relax}.

		Gap 1.00%		Gap 0.25%	
Instance	LB	Time	IP obj.	Time	IP obj.
A320_1w	628145	268	628145	276	628145
B737_1w	4969800	1507	*5093200	1565	*5014550
B757_1w	4385060	1439	*4413300	1202	*4389650
DC9_10d	21314200	206	*21479000	162	*21363900
DC9_2w	33062400	620	*34242200	418	*33787300
DC9_1m	69513000	2444	*72150000	2494	*71802000
MD82_2w	23628800	191	*23758200	84	*23628900
MD82_1m	52775500	1369	*55266500	1024	*55031700
		Gap 0.05%		Gap 0.00%	
Instance	LB	Time	IP obj.	Time	IP obj.
A320_1w	628145	283	628145	240	628145
B737_1w	4969800	1145	*5000750	-	-
B757_1w	4385060	217	4388950	-	-
DC9_10d	21314200	86	*21314400	-	-
DC9_2w	33062400	430	*33108400	-	-
DC9_1m	69513000	2121	*69939200	-	-
MD82_2w	23628800	64	23628800	13	23628800
MD82_1m	52775500	771	*53635600	82	52775500

Table 10.4 Effect of the `costGCC` propagation filter on column generation running time and solution quality. Here, the complete model PATH-TAS has been solved, and not only its linear programming relaxation. The running times include the preprocessing time, which is in all cases negligible compared to the total running time. Objective values marked with '*' are not optimal, see e.g. Table 7.2.

the integer phase here is that the filtering will affect the existence of solutions, unlike the aircraft count and standard propagation filters. For most instances it is possible to see a trend of reduced running times with reduced gaps. For many instances, no solution could be found when the gap was decreased below a certain point. In all these cases we can prove that no solution within the gap exists, since no possible routes exist for some aircraft in the reduced problem.

Clearly, the `costGCC` filter can dramatically reduce running times and increase solution quality, for problems for which there is a solution better than the set bound. For example, the running time for the MD82 1month instance decreases from 1369 to 82 seconds, while the objective decreases from 55266500 to 52775500 (-4.5%), when using gap 0.00% instead of 1.00%. The drawback is that without knowing how to set the gap, it can easily be set too low, resulting in no solution being found. Also, for instances where the network flow lower bound is poor, the filtering gaps must be set very high in order not to remove optimal solutions, which often results in very little filtering. Hence the quality of the lower bound is crucial for this filtering to work really well.

One slight shortcoming of all preprocessing methods described so far, is that they are based on the assumption that no activities are left unassigned in a solution. We have seen that model Path-Tas uses slack variables to allow unassigned activities, if this is necessary. This can be the case e.g. when working with incomplete or experimental data. Allowing unassigned activities in Csp-Tas and Csp-Tas^{relax} can be done by adding an activity as its own successor ($D(\texttt{successor}_f) \cap f \neq \emptyset$). However, this has the drawback that the propagation becomes very weak, since a solution can then always be obtained by leaving all activities unassigned. With `costGCC` propagation, unassigned activities can be allowed in a better way. By setting a high penalty (slack) cost on the self assignments, the same procedure as before can be used, specifying H so as not to allow more than 5 activities to be unassigned, for example. Using different penalty costs, it is also possible to model that some activities are more important than others to assign.

Another possible use of the `costGCC` constraint could be to use it to find optimal solutions with our constraint models. By replacing `all_different` by `costGCC` in the models, tightening the upper bound each time a solution is found, and continue the search, the optimal solution would in theory be found eventually. However, as discussed in Section 8.6, this is not a very powerful optimization method, as it essentially only uses the objective function as a constraint, and not as a guide for the search. Since there is little hope that this method would work well in practice, we have not tested it.

10.4 Summary

In this chapter, we have shown how the use of preprocessing increases the performance of column generation for tail assignment. Using constraint programming alone does not provide strong enough optimization, and using column generation alone gives poor performance. We have shown how constraint propagation on model CSP-TAS^{relax} can be used to achieve powerful preprocessing, clearly outperforming standard balance-based preprocessing techniques in terms of removed connections. The preprocessing also improves column generation performance substantially, both in terms of running time and convergence.

We have also shown how to extend model CSP-TAS^{relax} with connection costs, to obtain a heuristic preprocessing technique. This technique can be very effective if one knows an upper bound on the optimal value in advance. By filtering away connections that can never be used in an optimal solution, the column generator produces a solution 15 times faster than with only the normal preprocessing methods for one of our test instances. Unfortunately, it can also lead to no solution being found, in case the upper bound is set too tight. A possible use of this preprocessing method could be to use it in a 'bootstrapping' fashion, gradually tightening the problem to find better and better solutions.

It might be possible to use integration approaches similar to what we propose for fleet assignment, or other vehicle planning problems. For example, in [125], Jarrah et al. describe a feasibility checker for airline re-fleeting that solves a matching problem at each airport. This could potentially be changed into a filtering technique, using the consistency algorithm for `all_different`.

Eleven

Constraint Programming for the Integer Heuristics

In Chapter 7 we presented the integer fixing heuristics used to obtain integer solutions after an LP relaxed solution has been found by column generation. We showed that there is a trade-off between solution quality and running time when using the fixing heuristics. Using the variable fixing heuristic it is possible to obtain solutions of poor quality relatively quickly, while using the connection fixing heuristic leads to very good solutions, but can potentially take a lot of time. We also showed that the hybrid variable and connection fixing heuristic was one way of getting the best of both worlds – relatively high quality solutions with relatively short running times. Unfortunately, the hybrid heuristic can lead to poor quality solutions in some cases.

In this chapter we will discuss how the constraint programming models described in Chapter 9 can improve the performance of the fixing heuristics. We will also present an aggressive version of the hybrid fixing heuristic, which combines column generation and constraint programming to quickly obtain solutions.

11.1 Look-Ahead in the Fixing Heuristics

To avoid solutions with unassigned activities, a heuristic backtracking method for the fixing heuristics was presented in Section 7.4. The backtracking method is heuristic, and does not guarantee that a solution without unassigned activities is found. Also, no method for detecting conflicts early is used, so backtracking can only be performed once an integer solution is found. In Chapter 7 we therefore focused on finding solutions without backtracking, and just used the backtracking ability as a last option. But no matter how careful a fixing strategy is used, it can always happen that a set of variables or connections is fixed that will make it impossible to find a solution without unassigned activities, unless in each fixing iteration it is checked whether a solution to the resulting reduced problem exists. But unfortunately, checking for existence of a solution is as difficult as finding a solution to the original problem.

In constraint programming, constraint propagation and consistency checking is used to avoid excessive backtracking. As explained in Section 8.5, propagation performed during search is called look-ahead. To insert look-ahead in the fixing heuristics, we propose a generalization of the consistency-based preprocessing method described in Chapter 10.

Throughout the fixing process, an instance of Csp-Tas^{relax} is maintained. Since the `successor` variables correspond directly to the arcs in the Path-Tas pricing network, enforcing Path-Tas fixing decisions in Csp-Tas^{relax} is as trivial as posting $\texttt{successor}_f \neq \texttt{successor}_{f'}$ in Csp-Tas^{relax} for all removed connections (f, f'). When subsequent propagation in Csp-Tas^{relax} leads to domain reductions, these are transfered back to the Path-Tas pricing network. This can lead to more connections being removed, and the sets of restricted activities, R_t, being enlarged. The latter happens if domains of `vehicle` variables are reduced. The result of this integration with Csp-Tas^{relax} is a very strong look-ahead mechanism for the fixing heuristic. This connection between the constraint and column generation models is similar to the connection between CP and local improvement heuristics proposed by De Backer et al. [21].

Algorithm 11.1 shows the adjusted fixing heuristic using look-ahead. Here, the callback function RestrictAndEnforce which is used in Chapter 7 cannot be used, since Csp-Tas^{relax} must be propagated before the fixing decision is enforced. Algorithm 11.1 therefore generalizes the fixing step, only specifying that a set of connections is fixed. Clearly, this applies to both the variable and connection fixing heuristics. In steps 6–8 the fixing decisions are transferred to Csp-Tas^{relax}, which is propagated. If some domain in Csp-Tas^{relax} becomes empty, no solution without unassigned activities exists. Csp-Tas^{relax} is then restored to the state it was in before step 6, and backtracking is performed. If no conflict is detected, the domain reductions in Csp-Tas^{relax} are transferred back to Path-Tas, by removing further connections and adjusting the R_t sets. From a constraint programming perspective, the interaction between the Path-Tas and Csp-Tas^{relax} models can be seen as adding a tunneling constraint between them. Step 18 transfers the reduced `vehicle` domains to Path-Tas.

There is one small caveat with the look-ahead mechanism proposed in Algorithm 11.1. Since Path-Tas allows unassigned activities at a very high cost, from a strict feasibility point of view there always exists a trivial solution leaving all activities unassigned. But Csp-Tas^{relax} never allows unassigned

Algorithm 11.1 The general integer fixing heuristic with look-ahead.

data

SOLVE(m): A function that solves model m, returning objective value

PROPAGATE: A function that propagates the constraint model. Returns a set of removed connections, and new restriction sets.

COLUMNGENERATION(i): Algorithm 6.1

INTEGRAL(x): Returns **true** if x is integral, **false** otherwise

σ: Number of column generation iterations to perform between fixing iterations

ϵ: Relative optimality tolerance

z_u: Known upper bound on integer objective, or ∞

δ: Number of restrictions to undo when backtracking

Recall that F_r is the set of rows covered by column r.

```
1:  procedure LOOK-AHEADINTEGERHEURISTIC
2:      $\underline{z} \leftarrow$ SOLVE(TAS$^{relax}$)
3:      $(\overline{x}, \underline{z}, \overline{z}) \leftarrow$ COLUMNGENERATION$(100, \underline{z})$      ▷ 100 iter., as an example
4:      loop
5:          Restrict the solution space by removing connections $C$
6:          for all connections $(f, f') \in C$ do
7:              POST successor$_f \neq$ successor$_{f'}$ in CSP-TAS$^{relax}$
8:          end for
9:          $(C', R') \leftarrow$ PROPAGATE(CSP-TAS$^{relax}$)      ▷ $R' = \bigcup_{t \in T} R'_t$
10:         $conflict \leftarrow$ false
11:         if some domain in CSP-TAS$^{relax}$ empty then
12:             Restore CSP-TAS$^{relax}$
13:             $conflict \leftarrow$ true
14:         end if
15:         if $\neg conflict$ then
16:             Remove from RMP all columns $k$ s.t.
                    $|(F_k \cap f) \cup (F_k \cap f')| = 2$ for some $(f, f') \in (C \cup C')$
17:             Remove from pricing network all connections in $C \cup C'$
18:             $R_t \leftarrow R'_t \quad \forall t \in T$
19:             $\underline{z}' \leftarrow$ SOLVE(TAS$^{relax}$)
20:             $(\overline{x}, \underline{z}', \overline{z}) \leftarrow$ COLUMNGENERATION$(\sigma, \underline{z}')$
21:         end if
22:         if $\neg conflict$ and INTEGRAL$(\overline{x})$ and $\overline{z} < z_u$
                and $100 \times \frac{\overline{z} - \underline{z}}{\underline{z}} < \epsilon$ then
23:             return $(\overline{x}, \underline{z}, \overline{z})$
24:         else if $conflict$ or INTEGRAL$(\overline{x})$ or $\overline{z}' \geq z_u$ then
25:             Undo the last $\delta$ solution space restrictions
26:         end if
27:     end loop
28: end procedure
```

		Without look-ahead			With look-ahead		
Instance	Tas^{relax}	Time	IP obj.	Slack	Time	IP obj.	Slack
A320_1w	628145	3248	1858794	0	3437	985362	0
B737_1w	4969800	1404	7060700	2000000	1444	5093200	0
B757_1w	4385060	1448	4489650	0	1266	4623270	0
DC9_10d	21314200	202	21831300	0	211	21422000	0
DC9_2w	33062400	575	36210700	2000000	511	34242200	0
DC9_1m	69513000	2881	216785100	142000000	2370	72150000	0
MD82_2w	23628800	181	25703900	2000000	162	23712200	0
MD82_1m	52775500	1182	65713900	8000000	1295	55266500	0

Table 11.1 Performance of the hybrid fixing heuristic with and without look-ahead, doing 100 column generation iterations before the fixing heuristic is started.

activities,[1] which means that the look-ahead will not work for instances where it is impossible to assign all activities. While such instances are often ignored in research work, they do appear in practice, e.g. due to bad or experimental data, and the heuristics must be able to work even for such instances.[2] To overcome this problem, a full search with Csp-Tas^{relax} is performed initially, to decide if a solution exists. If no solution exists, or can be found within some time limit, is is assumed that some activities must be left unassigned, and look-ahead is not used in the fixing heuristic.

Table 11.1 shows the improvement in performance when using look-ahead for the hybrid fixing heuristic, without backtracking, with an aggressive fixing strategy – 60% of the variables are fixed initially, before starting connection fixing, and $\sigma = 5$. Columns 2-5 are taken from Table 7.3 in Section 7.3, and show that this aggressive fixing strategy does not work well without look-ahead. In five out of eight cases, not all activities are assigned, and slack columns are used. The last three columns show that when look-ahead is used, the slack cost is zero for all instances, leading to dramatically better solutions being found for most instances. Remember that in Section 7.3 we discussed the fact that the objective function for the A320_1w instance is unstable, leading to objective values far above optimum. This is still the case when using look-ahead, but the objective value is still vastly improved. For one of the instances the solution quality is slightly worse with look-ahead, but with decreased running time. It might seem strange that worse solutions can be

[1]It is possible to extend all our constraint models to allow unassigned activities, but this leads to very poor `all_different` propagation, and can obviously result in poor solutions.

[2]Models Csp-Tas and Csp-Tas^{relax} alone *cannot* be used for such instances. However, they can be extended to handle unassigned activities, as Kilborn [133] demonstrates.

obtained when using look-ahead, but since it potentially reduces the pricing network and RMP more than before, the search can take a different path in the search tree, potentially even leading to worse solutions.

11.2 An Aggressive Fixing Heuristic

The column generation approach is not able to quickly find solutions, as the results in Chapter 7 show. On the other hand, Chapter 9 shows that using constraint programming it is possible to quickly obtain solutions. The drawback is that the constraint programming solution quality is much worse, since model Csp-Tas does not consider costs at all. We will now describe a heuristic strategy that improves the fixing heuristic in Algorithm 11.1 further, making it possible to quickly obtain solutions even to large instances, while maintaining some control of solution quality.

The aggressive heuristic is shown in Algorithm 11.2. Unlike the previously described heuristics, backtracking is crucial for this algorithm to work well. Only in very few cases it will be possible to find solutions without using backtracking when using this heuristic. The heuristic is based on the normal look-ahead hybrid heuristic shown in Algorithm 11.1, but has a few additional features:

- before starting the heuristic, only very few column generation iterations are performed. During the heuristic, column generation is stopped as soon as the LP solution does not contain any slack columns, or after at most σ iterations. The idea is that our goal should be feasibility rather than optimality. Since the cost function is considered both by the RMP and pricing problem, the heuristic will still strive toward a high quality solution;

- initially, a round of variable fixations without any column re-generation is performed. This means that a fairly large portion of the problem gets fixed right away. Here it is really important to use look-ahead, to make sure that no conflicting variables are fixed. For the initial fixing, which is not shown explicitly in Algorithm 11.2, step 22 is not performed;

- when the initial fixing is completed, the hybrid heuristic is continued, fixing variables and connections with at most σ column generation iterations in each fixing iteration;

- finally, and most importantly, the aggressive heuristic does more careful feasibility checking than Algorithm 11.1. Instead of using Csp-Tas^{relax} which does not handle cumulative constraints, for look-ahead, the Csp-Tas^{hybrid} model is used. Csp-Tas^{hybrid}, which is shown in Model 11.1, is a variation of Csp-Tas^{relax} which also checks the cumulative constraints and runs the reachability algorithm at the root node, and ignores them

Algorithm 11.2 The aggressive fixing heuristic.

```
data
    See Algorithm 11.1
    HASSOLUTION: A function which solves the constraint model , returning
        true if a solution exists, and false if not
    FEASCOLUMNGENERATION: Algorithm 6.1, stopping as soon as $\overline{x}_j <$
        $10^{-6}$ for all slack columns $j$

1: procedure AGGRESSIVEINTEGERHEURISTIC
2:     $\underline{z} \leftarrow$ SOLVE(TAS$^{relax}$)
3:     $(\overline{x}, \underline{z}, \overline{z}) \leftarrow$ COLUMNGENERATION$(5, \underline{z})$            ▷ 5 iter., as an example
4:     $feasible \leftarrow$ HASSOLUTION(CSP-TAS$^{hybrid}$)
5:     loop
6:         Restrict the solution space by removing connections $C$
7:         if $feasible$ then
8:             for all connections $(f, f') \in C$ do
9:                 POST successor$_f \neq$ successor$_{f'}$ in CSP-TAS$^{hybrid}$
10:            end for
11:            $(C', R') \leftarrow$ PROPAGATE(CSP-TAS$^{hybrid}$)
12:        end if
13:        $conflict \leftarrow$ false
14:        if some domain in CSP-TAS$^{hybrid}$ empty
               or ¬HASSOLUTION(CSP-TAS$^{hybrid}$) then
15:            Restore CSP-TAS$^{hybrid}$
16:            $conflict \leftarrow$ true
17:        end if
18:        if $\neg conflict$ then
19:            Remove from RMP all columns $k$ s.t.
                   $|(F_k \cap f) \cup (F_k \cap f')| = 2$ for some $(f, f') \in (C \cup C')$
20:            Remove from pricing network all connections in $C \cup C'$
21:            $R_t \leftarrow R'_t \quad \forall t \in T$
22:            $\underline{z}' \leftarrow$ SOLVE(TAS$^{relax}$)
23:            $(\overline{x}, \underline{z}', \overline{z}) \leftarrow$ FEASCOLUMNGENERATION$(\sigma, \underline{z}')$
24:        end if
25:        if $\neg conflict$ and INTEGRAL$(\overline{x})$ and $\overline{z} < z_u$
               and $100 \times \frac{\overline{z}-\underline{z}}{\underline{z}} < \epsilon$ then
26:            return $(\overline{x}, \underline{z}, \overline{z})$
27:        else if $conflict$ or INTEGRAL$(\overline{x})$ or $\overline{z}' \geq z_u$ then
28:            Undo the last $\delta$ solution space restrictions
29:        end if
30:    end loop
31: end procedure
```

Model 11.1 CSP-TAS^{hybrid}. CSP-TAS^{relax} combined with the `reachability` propagation algorithm and the `pricing` constraint at the root node.

$$\forall f \in F : \quad D(\texttt{successor}_f) = \bigcup_{f' \in F} (f' \text{ possible follow-on of } f)$$

$$\forall f \in F : \quad D(\texttt{predecessor}_f) = \bigcup_{f' \in F} (f \text{ possible follow-on of } f')$$

$$\forall f \in F : \quad D(\texttt{vehicle}_f) = \bigcup_{t \in T} (t \text{ can be assigned to } f)$$

$$\texttt{inverse}(\texttt{successor, predecessor}) \tag{11.1}$$

$$\wedge$$

$$\texttt{all\_different}(\texttt{successor}_1, \ldots, \texttt{successor}_{|F|}) \tag{11.2}$$

$$\wedge$$

$$\texttt{all\_different}(\texttt{vehicle}_{G(f)}) \quad \forall f \text{ s.t. } |T_f| \neq |\mathtt{T}| \tag{11.3}$$

$$G(f) : \text{Activities overlapping the start time of activity } f$$

$$\wedge$$

$$\texttt{tunneling}(\texttt{successor, predecessor, vehicle}) \tag{11.4}$$

$$\wedge$$

$$\text{at root node}$$

$$\texttt{reachability}(\texttt{successor, predecessor, vehicle}) \tag{11.5}$$

$$\texttt{pricing}(\texttt{successor, predecessor, vehicle}) \tag{11.6}$$

during search. Compared to Algorithm 11.1, the constraint model is here *solved*, rather than only propagated, before each fixing decision. While solving CSP-TAS rather than CSP-TAS^{hybrid} would give even stronger propagation, it would also lead to significantly longer running times. CSP-TAS^{hybrid} at least guarantees that a solution to CSP-TAS^{relax} exists, that all activities can be reached, and that at least one route exists for each aircraft.

Table 11.2 shows results when using the aggressive heuristic with two different strategies. In the first test, no variables are fixed initially, and variables corresponding to 40% of the aircraft are fixed before connection fixing is started. In the second test, variables corresponding to 15% of the aircraft are fixed initially, and then variable fixing is used. Here, connection fixing is thus not used at all. The convergence criterion (step 25 in Algorithm 11.2) is simply that no slack columns are used, i.e. the search stops as soon as a

		Initially fixed variables: 0% Variable fixing: 40% Connection fixing: 60%		
Instance	TAS^{relax}	Time	IP obj.	Slack cost
A320_1w	628145	1850	3069872	0
B737_1w	4969800	1161	5525100	0
B757_1w	4385060	621	5097630	0
DC9_1w	12101800	21	12548400	0
DC9_1w2	16662800	22	17295600	0
DC9_10d	21314200	67	22252600	0
DC9_2w	33062400	259	34135500	0
DC9_1m	69513000	1913	71419700	0
MD82_1w	11984100	5	12769100	0
MD82_2w	23628800	42	25004000	0
MD82_1m	52775500	579	56499400	0
		Initially fixed variables: 15% Variable fixing: 85% Connection fixing: 0%		
Instance	TAS^{relax}	Time	IP obj.	Slack cost
A320_1w	628145	711	4213497	0
B737_1w	4969800	557	5518050	0
B757_1w	4385060	311	5362830	0
DC9_1w	12101800	16	12863700	0
DC9_1w2	16662800	15	17205400	0
DC9_10d	21314200	28	22255900	0
DC9_2w	33062400	45	34365200	0
DC9_1m	69513000	161	73038600	0
MD82_1w	11984100	3	12666900	0
MD82_2w	23628800	20	25080000	0
MD82_1m	52775500	89	56864200	0

Table 11.2 Performance of the aggressive fixing heuristic. The percentages show how many percent of the aircraft are fixed initially without re-generation of columns, with variable fixing, and with connection fixing.

		Initially fixed variables: 10%		
		Variable fixing: 70%		
		Connection fixing: 20%		
Instance	TAS^{relax}	Time	IP obj.	Slack cost
A320_2w	928459	845	4830921	0
B737_2w	9858450	1007	10846700	0
B757_2w	7664360	775	9068990	0
M81_1w	8404600	8624	10573700	0
M82/3_1m	100510	5219	586646	0
A320_2_1m	1061	112	1154	0
MDDC9_1w	82080900	1219	107904600	0
ALL_1w	110000900	6918	142440700	0
MDS_1m	53068150	748	80586100	0
MDS_1m2	83043500	812	148256500	0

Table 11.3 Performance of the aggressive fixing heuristic for large test instances. The percentages show how many of the variables, in terms a percentage of the number of aircraft, are fixed initially without re-generation of columns, and with variable and connection fixing.

solution without unassigned activities has been found. Comparing the results to the previous fixing heuristic results in Tables 7.1, 7.2 and 7.3, it is clear that the results in Table 11.2 are extremely good in terms of running time. We are in fact able to find integer solutions, covering all activities, in less time than it normally takes to find an optimal relaxed solution. When it comes to solution quality, the results are not so good. However, the deterioration is of course expected, as the primary purpose of this aggressive heuristic is to quickly find solutions.

Table 11.3 shows how the aggressive heuristic performs on a set of larger test instances. Solutions are obtained rather quickly, except in some of the largest cases, where it still takes more than 2 hours. For many of the instances, the results are quite far from the lower bounds. For most of these instances we have not been able to find any solutions very close to the lower bound, and we thus suspect that the lower bounds are quite poor.

In Section 9.8 we compared the aggressive fixing heuristic to CSP-TAS used standalone. Table 9.1 shows that CSP-TAS used standalone is substantially faster, but the table does not show the solution quality, simply because the purpose of CSP-TAS is to quickly obtain a solution, rather than to find a solution of high quality.

11.3 Summary

In this chapter we have demonstrated how constraint programming can be used to improve the column generation based fixing heuristics described in Chapter 7. Integrating look-ahead using model Csp-Tas^{relax} in the heuristics, it is possible to find solutions without unassigned activities even when using very aggressive hybrid fixing strategies. We believe that this kind of integration of constraint programming and column generation has great potential. Since the column generation process only looks at a restricted part of the problem at any given time, the use of a constraint model with a global feasibility-focused view can be very useful indeed. We have clearly shown that it is very useful for the tail assignment problem, but we also conjecture that it could be useful for other, similar, problems. Since crew planning problems are similar to tail assignment in structure, the type of integration we have presented might be possible also for such problems.

To quickly obtain solutions of better quality than those obtained by Csp-Tas, an aggressive fixing heuristic was developed. It combines the look-ahead version of the fixing heuristics with extra feasibility checks and few column generation iterations, and makes it possible to obtain solutions to large instances within minutes. There is of course a trade-off between solution quality and running time, and by tuning the algorithm it is possible to choose whether to emphasize solution quality or running time.

Part IV

The Complete Approach and Final Results

Twelve

The Complete System

In the previous parts of this thesis, we have described how to solve the tail assignment problem using column generation or constraint programming, and also how the approaches can be combined in various way.

In practice, it is often too time consuming to find proven optimal solutions to large tail assignment instances. For many instances, just completing the relaxed column generation phase can take far too long. Instead, we use local search [1] heuristics to gradually improve an existing solution. Our local search heuristics give a scalable improvement approach, that also has the benefit that it can produce several solutions of gradually improving quality, which is very good from a usability point of view. The local search heuristics apply the column generation or constraint programming models iteratively to subsets of the entire problem.

The obvious drawback of using a local search heuristic is that it cannot guarantee optimality. In practice, this is not a huge problem, and many large-scale optimization problems are solved in a similar fashion. And as we have already seen in previous chapters, optimality has been sacrificed for computational efficiency at numerous places in our column generation approach.

However, before the local search heuristics can be started, there are several other steps. Figure 12.1 gives an informal view of the complete solution approach. The first step is always to perform a few rudimentary feasibility checks on the flight schedule and the constraints. In an environment with thousands of flight legs and many constraints, it is not uncommon that data handling errors are made that render the problem infeasible. Some feasibility checks can be performed even before the connection network is generated, while other checks must be performed using the full connection network. The feasibility checks are covered more thoroughly in Section 12.1.

For large instances, it is sometimes necessary to reduce the flight network heuristically before proceeding with the optimization, simply to get a computationally feasible problem. Section 12.4 presents a few possible reduction

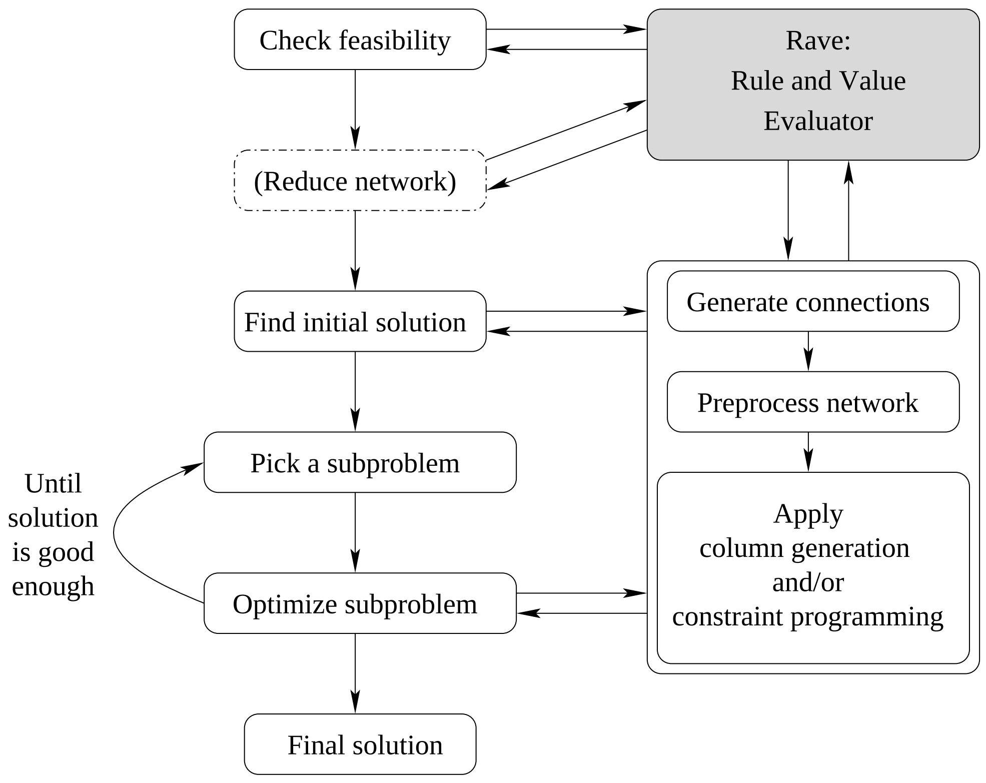


Figure 12.1 The complete solution approach, also shown more formally in Algorithm 12.1.

techniques.

The actual solution process starts by finding an initial solution, which covers the entire solution period and satisfies all constraints. The main purpose of the initial solution, except to provide an initial starting point for the local search, is to show that the flight schedule is balanced, and that all constraints are possible to satisfy. In that sense, the initial solution search can be seen as a deeper, complete, feasibility check. Section 12.2 describes the methods we use to find initial solutions.

Throughout the solution process, Carmen Systems' Rave system is used for evaluation of constraints and optimizer parameters. While Rave, which is presented in more detail in Section 12.5, is not in itself part of this thesis, the way it represents some of our constraints is tail assignment specific, and thus important here.

12.1 Checking problem feasibility

The first thing that happens once the activities have been loaded is to check the flight restrictions. If there is some activity that cannot be assigned to any aircraft, the problem is clearly impossible to solve without unassigned activities. Secondly, activity balance is checked. This is done using a time-line network, as described in Section 3.1. The time-line network is a more compact representation than the connection network, so it can be set up quickly. However, since the time-line network does not model individual connections between activities, it is a relaxation of the connection network (since it potentially contains connection possibilities not present in the connection network). On the time-line network, a network flow problem, similar to model Tas^{relax}, is solved. In the network flow problem, each activity has a loop-back arc connecting to its own start node, to allow activities to be left unassigned by the network flow solution. The costs are 0 on all normal arcs, and 1 on the loop-back arcs. If this problem has an optimal objective value larger than 0, no solution covering all activities can exist for the complete tail assignment problem.

Further, the optimal flow solution can suggest a number of activities which when removed will make the problem balanced. Even though the column generation solution approach can handle unassigned activities, making the flight network balanced often improves the performance. It will make preprocessing possible, thus reducing the number of connection and RMP rows, but it will also directly influence the convergence, since no high cost slack columns will 'pollute' the duals. The constraint models cannot handle unassigned activities well, so for those it is crucial that unbalanced activities are removed.

However, since the time-line network is a relaxation of the connection network, and the network flow solution does not consider maintenance or flight restriction constraints, there is still no guarantee that a solution to the complete tail assignment problem exists. After the connection network has been created, we check whether all activities can be reached in the column generation pricing network, and whether at least one legal route exists for each aircraft. This is the same type of feasibility check as that used in our constraint programming models.

12.2 Finding an Initial Solution

We have already discussed two ways to quickly obtain solutions. In Chapter 9 our fastest approach, the constraint based model Csp-Tas, is presented. In Chapter 11, an aggressive fixing heuristic combining models Csp-Tas^{hybrid} and Path-Tas is presented. The aggressive fixing heuristic is not as fast as Csp-Tas on most instances, but can often produce better solutions, since Csp-Tas does not consider solution quality at all. It is also better than Csp-Tas at handling instances for which cumulative constraints are very constraining.

While finding the initial solution, a global lower bound is calculated, using TAS^{relax}. If LP optimality is reached in the column generation phase, which is unlikely since only very few iterations are performed, the global lower bound is adjusted to the optimal LP objective value, which as discussed in Chapter 5 is always at least as good as the TAS^{relax} bound. Thus, once an initial solution is found, both a lower and an upper bound on the optimal objective exist, which can be used to monitor convergence.

Observe that it might be tempting to relax some of the complicating constraints, e.g. maintenance constraints, to find an initial 'solution' faster. Methods such as this are sometimes used to find initial solutions for column generation RMPs. By appropriately penalizing the broken constraints, one can try to make sure that a final solution will be found which does not violate the constraint, if such a solution exits. We do not do this, because we *only* want to produce solutions satisfying all constraints, so as not to fool a user of the system into believing that all constraints can be satisfied, when they cannot.

12.3 Overall Local Search

Local search is a broad name for heuristics which gradually improve solutions by small local changes. Sometimes, local search is categorized as a type of *neighborhood search* [9]. In neighborhood search, a specific type of local change defines a *neighborhood* containing the set of solutions that can be reached from the current solution by applying the change. Standard local search, which is sometimes called *hill climbing*, iteratively moves to the best solution in the neighborhood of the current solution, until no more improvement is possible. Observe that this normally requires exhaustively searching the neighborhood of the current solution.

The most standard local changes for routing problems are the 2-opt and 3-opt swap methods for TSP and VRP. These methods attempt to disconnect 2 or 3 arcs of the routes and re-connect them in a better way. In our case, this corresponds to swapping partial routes between aircraft. The benefit of using such a simple local change is that the resulting method would be very fast. Unfortunately, this kind of swap is difficult to do for us, because of the additional constraints on routes. Considering preassigned activities, flight restriction and cumulative constraints, it is difficult to apply fast simple changes, and still guarantee that the resulting solution will satisfy all constraints. By relaxing the difficult constraints and adding penalty costs, standard local search techniques can still be applied. But since we want to make sure that all solutions we produce are feasible, this is not possible for us to do. Ahuja et al. [9] present *very large-scale neighborhood search*, where the goal is to be able to define large neighborhoods and still be able to quickly find the best solution in the neighborhood. They propose to use *cyclic exchanges* and present a network algorithm to efficiently search the neighborhood. Unfortunately, their approach has the same problem as traditional local search in that it largely ignores additional complicating constraints.

Due to the feasibility problems with local swap improvements, we instead use a local search approach based on partially 'locking' the current solution, and re-optimizing the unlocked part with the column generation solution approach. This way it can be guaranteed that intermediate solutions satisfy all constraints, since the column generator will either find a better solution, satisfying all constraints, or re-use the current solution. This can be seen as a kind of very large-scale neighborhood search, since the neighborhoods defined by all solutions that can be obtained by partially re-optimizing the problem can be very large indeed. Since column generation is used to find the best neighbor, it is also not necessary to explicitly enumerate all neighbors.

Time Window Local Search

The most natural way to partially lock the current solution is to pick a *time window* $[t_1, t_2]$, fixing the current solution outside this interval, and re-optimizing the subproblem within $[t_1, t_2]$. Normally, the time window is chosen to a be few days long, and is slid over the planning period, as illustrated in Figure 12.2, to gradually improve the current solution. It is a good idea to let the time windows overlap each other, since a connection at the end of a time window might look very good in the current window, but prevent improvements in the following window. Similar techniques are often used for large scale planning problems. For example, Kharraziha et al. [132] uses sliding time windows, as well as a random locking approach, to solve large scale airline crew rostering problems.

The appropriate length of the time window depends on the size of the instance. For instances containing a lot of aircraft, a shorter time window is more efficient than a long one, while for smaller instances longer time windows can be used. Generally, the time window length should not be too short compared to the cumulative constraints used. Observe that for small instances it might be possible to define the entire planning period as the time window, which gives a global optimization approach.

Aircraft Subset Local Search

For some highly constrained instances, the sliding time window improvement method might work poorly because of its limited time horizon. While the connection costs are local in character, there can still be situations where a more global perspective is necessary to avoid suboptimality. For example, in order to reduce the penalty cost of a certain soft cumulative constraint, it might be necessary to have a time horizon of at least 15 days, but the problem size makes it impractical to use such large time windows. In such

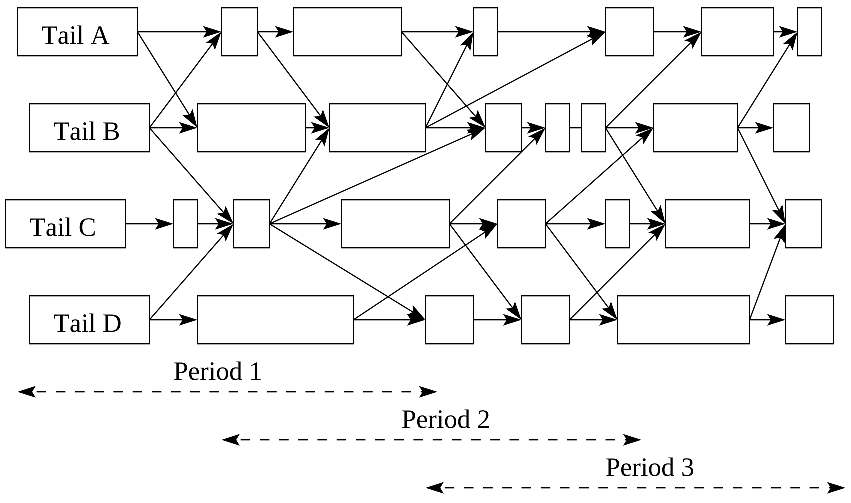


Figure 12.2 The sliding time window local search heuristic.

cases, a better method is to lock a number of routes, over the entire planning period, and re-optimize over the entire planning period for a subset of the aircraft. Since it is normally difficult to determine which aircraft should be included in the subset, an iterative sliding subset approach is used. That is, in each iteration a number of aircraft from the current subset are selected to be replaced. In Figure 12.2, this can be seen as a vertical variant of the sliding time window method.

Parallel Local Search

One obvious improvement when using a problem division strategy, such as our time window and aircraft subset local search heuristics, is to parallelize the process and solve each subproblem on a dedicated processor. For the time window heuristic, parallelization is difficult. Since each subproblem must have fixed carry-in and carry-out activities at the start and end of the time window, to make sure it can connect with the existing route, one cannot optimize adjacent time windows in parallel. One solution to this could be to make sure that at least one activity is fixed between the time windows. A more serious problem is the feasibility with respect to cumulative constraints. To make sure that a route is feasible, the exact resource consumption must be known, which might not be the case if another processor is simultaneously changing the route. This could potentially be solved by forcing the necessary fixed activities between the time windows to reset the resource consumption. But as a further complication, the cost might also depend on previous time windows, making the optimization in each time window potentially suboptimal.

For the aircraft subset heuristic, parallelization is much more straightforward. Here, the subproblem are completely independent, and since the entire planning period is considered by each subproblem, the cumulative constraints are not affected by parallelization. This indicates that this process could potentially be almost perfectly parallelized. We have not implemented any parallelization, however, as that requires knowledge which is outside of the scope of this thesis.

Other Improvement Methods

For large instances, combining the time window and aircraft subset methods creates a local search heuristic that scales well with increasing number of aircraft, and longer planning periods.

Of course, many other problem division strategies exist. For example, one could imagine randomly selecting connections in the current solution to unlock, or simultaneously selecting several disjoint time windows or aircraft subsets to unlock. However, these approaches have not been tested.

The most fundamental problem with hill climbing algorithms, such as our improvement algorithm, is that they get stuck in local optima, since they never move away from solutions for which no improving neighbor exists. A number of algorithms which address this problem have been proposed, e.g. simulated annealing [134] and tabu search [102]. We have not found the problem with local optima serious enough to justify implementation of either of these algorithms.

Discussion

The complete solution approach, using an iterative sliding time window local search heuristic, is shown in Algorithm 12.1. This approach iteratively increases or decreases the time window length and the amount by which to slide the time window each time a full sweep over the planning period has been completed. It combines column generation, local search and constraint programming on many levels. First an initial solution is found either with a constraint model using the column generator pricing solver, or using an aggressive fixing heuristic where feasibility checking using a constraint model is a key component. Then the resulting solution is improved with column generation accelerated by constraint programming preprocessing and feasibility checking, wrapped into a local search framework. The process can of course be further customized e.g. by looping backward over the planning period, or by selecting just a few time windows to improve.

There are several benefits of this approach, apart from the fact that it creates several solutions. For example, it scales well to large instances. Of course, subdividing large problems into too small chunks will affect the possibilities to reach optimality, so there is still a trade-off between running time and solution quality. Observe that the local search heuristics can handle unassigned

Algorithm 12.1 The complete approach: Constraint programming accelerated column generation, wrapped into local search.

```
data
    Csp-Tas: Algorithm solving Csp-Tas
    AggressiveIntegerHeuristic: Algorithm 11.2
    Look-AheadIntegerHeuristic: Algorithm 11.1, here applied to a
        sub-period of an existing solution
    $l_{tw}$: time window length
    $\kappa_{tw}$: amount by which to move window
    $\Delta_l$: change to apply to $l_{tw}$ when a full sweep is completed
    $\Delta_\kappa$: change to apply to $\kappa_{tw}$ when a full sweep is completed
    $\epsilon$: optimality tolerance
    $[t_s, t_e]$: planning period

1: procedure CompleteApproach
2:     Check feasibility using time-line network relaxation
3:     Find initial solution $\hat{x}$ with objective value $\hat{z}$:
           $(\hat{x}, \underline{z}, \hat{z})$ = Csp-Tas() or
           $(\hat{x}, \underline{z}, \hat{z})$ = AggressiveIntegerHeuristic()
4:     $[t_s^{curr}, t_e^{curr}] \leftarrow [t_s, t_s + l_{tw}]$
5:     repeat
6:         $(\hat{x}, \underline{z}', \hat{z}) \leftarrow$ Look-AheadIntegerHeuristic($\hat{x}, [t_s^{curr}, t_e^{curr}]$)
7:         $[t_s^{curr}, t_e^{curr}] \leftarrow [t_s^{curr} + \kappa, t_e^{curr} + \kappa]$
8:         if $t_s^{curr} > t_e$ then
9:             $l_{tw} \leftarrow l_{tw} + \Delta_l$
10:            $\kappa_{tw} \leftarrow \kappa_{tw} + \Delta_\kappa$
11:            $[t_s^{curr}, t_e^{curr}] \leftarrow [t_s, t_s + l_{tw}]$
12:        end if
13:    until $100 \times \frac{\hat{z} - \underline{z}}{\underline{z}} < \epsilon$ or some search limit is reached
14:    return $(\hat{x}, \underline{z}, \hat{z})$
15: end procedure
```

activities, which means that it can also be used for finding initial solutions to large instances, by starting with a solution with all activities unassigned, and gradually building routes. However, this is not the intended use of the sliding time window method, and might lead to problems e.g. when used for highly constrained instances.

In the time window optimization, the connection fixing heuristic is typically used unless the subproblems are too large, in which case a careful version of the hybrid heuristic is used. Up to 100 initial column generation iterations are performed, which means that the PATH-TAS^{relax} lower bound for the subproblem is often found. In fact, an important difference of our local search heuristic compared to many others is that we have a global lower bound, computed during the search for an initial solution, against which objective values can be compared. This means that we have a good idea of how good the current solution is, and can stop the improvement phase whenever a solution is found which is within a certain tolerance of the lower bound, or tune the algorithm depending on the gap to the lower bound.

Another useful feature of this solution approach is that it makes it easy to try different cost scenarios without complete re-optimization. Once an initial solution has been found, and possibly manually inspected, the cost function can be redefined, and the improvement method can be re-started to obtain improving solutions with respect to the new cost function. This facilitates re-fleeting close to day of operation, for example. Changing constraints without complete re-optimization is possible, but not recommended, since it can potentially cause the existing solution to violate the constraints, and lead to problems when improvement is applied.

12.4 Connection Network Reduction Techniques

When solving very large tail assignment instances, the first bottleneck is the sheer size of the complete connection network. For some instances, just setting up the network can take several minutes. The huge number of connections will make the pricing problem more complicated to solve, and will also degrade the column generation convergence, because so many possible routes exist. In Chapter 10 we have already seen how the connection network can be preprocessed to reduce its size. While the aircraft balance filter can be applied before the connection network is created, the propagation filter can only be applied once the connection network has been created. For huge problems this might be problematic, since just creating the connection network takes a long time.

A very common technique to accelerate column generation is to reduce the size of the pricing network [73]. The motivation behind using a reduced network is that when the pricing network is less dense, the running time of the pricing phase can be decreased. Also, in the beginning of the column generation process, before the duals have taken on stable values, the pricing is quite inexact, in the sense that a lot of columns that might not be useful later on are produced. Not considering the complete pricing network in this initial phase normally does not not have a huge impact on the column generation convergence, but can improve the running time performance substantially. An example of using a reduced network is given by Hjorring and Hansen [117], who present a column generation approach which uses an approximate pricing network combined with a network refinement technique. Using a reduced connection network will also make the constraint models easier to solve, since a tighter network means smaller domains and more constraint propagation.

We will now present a few connection network reduction techniques for tail assignment.

k-FIFO Filtering

When matching arriving aircraft to departing flights at a particular airport, a very natural strategy is to assign the first arriving aircraft to the first departing flight (First In First Out). With FIFO assignment, the ith arriving aircraft is thus assigned to the ith departing flight. Because of the operational constraints, this strategy is unlikely to produce a feasible tail assignment solution if it is carried out globally.

If the ith arriving aircraft can instead choose between the departing flights in the interval $[i-k, i+k]$, the value of k decides whether it is possible to create a feasible solution. Obviously, $k = 0$ corresponds to pure FIFO assignment, while $k = \infty$ corresponds to choosing any aircraft for a departure. If k is chosen relatively small, but still large enough to make it possible to satisfy all operational constraints, many connections can be removed from the network, only leaving at most $2k+1$ connections from each activity. Figure 12.3 shows an example with six arrivals and departures, and the remaining connections left from the first and fifth arrivals when 1-FIFO filtering has been performed. The other connections have been left out so as not to clutter the figure.

Also, it should be noted that this kind of filtering is reasonable, since flight networks are often designed to be operated in a FIFO fashion. In fact, pure FIFO or LIFO (Last In First Out) assignment is often used to create approximate aircraft routes e.g. as input to crew pairing optimizers.

The success of this approach will clearly depend on the characteristics of the flight network. The method works well for networks where most of the activities go from a main hub to an outlying airport and back. In such a network it is fairly simple to route the aircraft from anywhere back to the main hub for maintenance, and it is probably possible to use relatively small values of k. In networks that do not have this structure it might be more difficult to find feasible solutions, as several activities might then be forced to be flown in sequence without possibility of returning to the hub, restricting the possibilities to get maintenance.

One major benefit of the k-FIFO filter is that it can be applied before

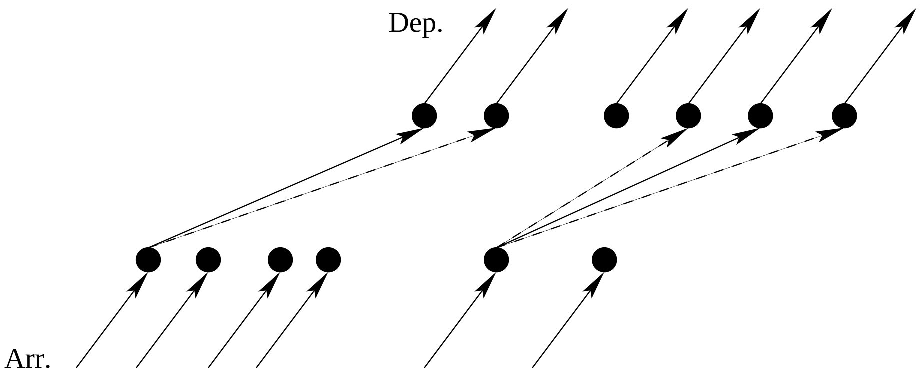


Figure 12.3 An example of 1-FIFO filtering. Only the connections from the first and fifth arriving activities are shown in the figure, for clarity.

the connection network is generated, thus prohibiting many connections from ever being considered. This is of course also the drawback of the method – that connections that can be very good from a cost point of view are removed without being considered. For example, k-FIFO filtering is not suitable for instances where the cost function maximizes standby time, since the filtering will attempt to minimize the aircraft connection time.

Propagation-Based Filtering

In Chapter 10, a heuristic preprocessing method based on the `costGCC` constraint is presented. This filter is able to remove connections that can never be part of a solution better than a selected objective. Tests in Chapter 10 indicate that while this method is not perfectly suited for exact preprocessing, it can be used as a heuristic filter. Unfortunately, it can remove too many connections if used aggressively, causing the remaining problem to become infeasible. Unlike the k-FIFO filter, the `costGCC` filter has the drawback that it requires all connections to be generated initially. In fact, it even requires the cost of each connection to be calculated, which can be time consuming, especially for multi-fleet instances. On the other hand, it will not remove connections that can be used in good solutions.

LOF Filtering

Our final network reduction technique fixes sequences similar to lines of flights (LOFs) [104], by breaking up an existing solution into shorter sequences. The sequences are not always full day LOFs, but can sometimes be shorter than a full day. Also, the starting solution does not necessarily have to satisfy all constraints, but can be a solution to e.g. CSP-TAS^{relax}. It is of course

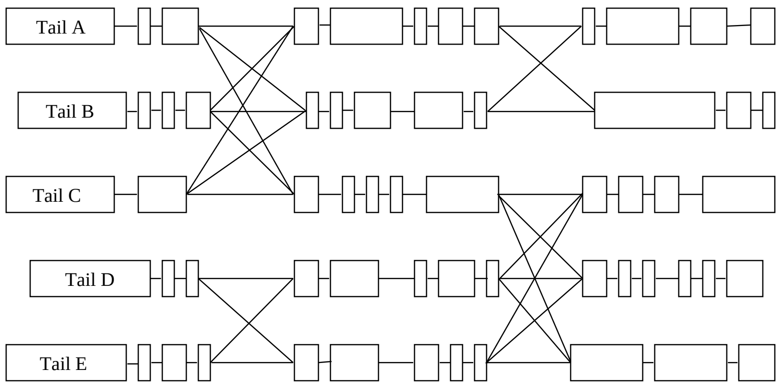


Figure 12.4 Creation of activity sequences, LOFs.

necessary to use a fast relaxed model, such as Csp-Tas^{relax}, to obtain the starting solution, or there is no point of filtering.

Algorithm 12.2 shows the process for creating LOFs. Given a starting solution, all connections in the solution shorter than LOF_{break} are fixed, by removing all connections in the connection network conflicting with them. Initial LOFs are then created by sequencing single-connection activities. Finally, if the duration of a LOF is longer than LOF_{max}, it is split where the number of connection possibilities is maximized. The purpose of this final split is to avoid long, maintenance infeasible, LOFs.

Tail assignment instances often have a structure where many aircraft are on ground at the main hub during the night. The many connection possibilities over nights make multi-day instances *much* more complex than single-day instances. Since maintenance is typically performed at the main hub at night, there is a good chance that feasible routes can be constructed even if the sequence of activities during the day are considered fixed, and only night connections are optimized.

The LOF filter was used to create the initial partitioning for the constraint aggregation method in Section 6.9.

Discussion

We have now presented three heuristic connection network filtering methods. These filters can be used either to a priori filter away connections, never to re-insert them again. But it can also be used to temporarily remove connections and re-insert them again, to increase the chances of finding good solutions. One alternative is to run the column generator for a number of iterations with a reduced connection network, and then re-insert connections when optimality is approached, or when no more negative reduced cost columns can be generated. Another possibility is to use a heuristic filter when finding the initial solution,

		Without filtering		
Instance	$\textsc{Tas}^{relax}$	Time	IP obj.	Slack cost
A320_1w	628145	5136	628145	0
B737_1w	4969800	1875	5076050	0
B757_1w	4385060	1609	4389000	0
DC9_1w	12101800	63	12101800	0
DC9_1w2	16662800	75	16664200	0
DC9_10d	21314200	235	21495300	0
DC9_2w	33062400	601	34400500	0
DC9_1m	69513000	2435	219724700	127000000
MD82_1w	11984100	12	11984100	0
MD82_2w	23628800	205	23651200	0
MD82_1m	52775500	1301	56418700	0
		LOF_{break}: 8h, LOF_{max}: 48h		
Instance	$\textsc{Tas}^{relax}$	Time	IP obj.	Slack cost
A320_1w	628145	2603	1828700	0
B737_1w	4969800	1517	5109150	0
B757_1w	4385060	1210	4623270	0
DC9_1w	12101800	58	12118600	0
DC9_1w2	16662800	73	16663600	0
DC9_10d	21314200	141	21896800	0
DC9_2w	33062400	330	33980900	0
DC9_1m	69513000	1624	71897300	0
M82_1w	11984100	13	11984100	0
M82_2w	23628800	131	24998900	0
M82_1m	52775500	952	55924400	0

Table 12.1 Performance of the column generator with and without LOF filtering.

Algorithm 12.2 The LOF filter.

```
data
    $LOF_{break}$: Connections longer than this are considered to break LOFs
    $LOF_{max}$: The maximum duration of a LOF

1: procedure LOFFilter
2:     $R_{init} \leftarrow$ initial routes
3:     for all $r \in R_{init}$ do
4:         for all connections $c$ in $r$ do
5:             if duration of $c < LOF_{break}$ then
6:                 Fix connection $c$ in connection network
7:             end if
8:         end for
9:     end for
10:    Create LOFs by sequencing single-connection activities
11:    for all resulting LOFs $l$ do
12:        if duration of $l > LOF_{max}$ then
13:            Split $l$ where the number of connection possibilities is maxi-
               mized
14:        end if
15:    end for
16: end procedure
```

and then use all connections in the improvement phase.

Table 12.1 shows results of using the filter during the column generation phase. Connections are re-inserted when

- the sum of used slack columns is below 10^{-6}, or;
- no more columns can be generated, or;
- the integer fixing heuristic is started.

When either of the criteria is fulfilled, *all* the removed columns are re-inserted. One could imagine inserting only some, based e.g. on their reduced cost. This strategy has been tested, but was not very successful, as it turned out that almost all connections were re-inserted right away anyway. The results in Table 12.1 were obtained with the hybrid fixing heuristic.

The solution quality is slightly worse for some instances, and better for others, when using filtering. For most instances the running times decrease when using the filter. For the B737_1w instance the running time is slightly worse. What happens is that the process converges really fast to a solution with 0 slack cost, but around 29% above the lower bound. When columns are re-inserted, the objective does not converge fast enough to the lower bound, and the unfiltered case actually reaches the end of the relaxed phase faster. We do not use filtering by default in our optimizer, since it is not predictable enough when looking for close to optimal solutions. Instead, filtering is available as an option to use on huge instances.

12.5 Rave – Modeling Legality and Costs

To specify constraints and costs, we use a modeling language called *Rave* (*R*ule *A*nd *V*alue *E*valuator). Rave is a modeling language designed by Carmen Systems, specifically suited to airline planning problems. In [13], Andersson et al. give a general overview of Rave, and outline its usefulness for various planning applications. In [142], Kohl and Karisch describe more in detail how Rave is used in Carmen Systems' crew rostering system, and in [19] Augustsson discusses partial evaluation in Rave.

Using Rave, it is easy to specify connection, cumulative and flight restriction constraints, as well as costs. The fact that the Rave rules can be changed and dynamically re-linked to the optimizer software makes it easy to try out different constraints and cost scenarios. It might sound as if Rave is just a replacement for simple parameterized minimum connection times, but Rave is in fact much more than that.

An example Rave rule specifying the minimum connection time is shown in Algorithm 12.3. The rule specifies that if the connection is a flight-to-flight connection within the same flight (different leg numbers of the same multi-leg flight), the variable `%min_connection_time_same_flight%` should be used to give the minimum connection time. If it is a flight-to-flight connection but not within the same flight, `%min_connection_time_thru%` should be used, and otherwise `%min_connection_time_default%` should be used. The `%min_%` variables are all defined elsewhere in the rules, and could e.g. access a table to get the actual times to use. Observe that nowhere are the two flights about which the rule reasons mentioned – this is built into the language.

To handle cumulative constraints, we have extended Rave with a special syntax, specifying the consumption, reset and limit characteristic of each cumulative constraint individually. Algorithm 12.4 shows a cumulative constraint limiting the time away from base. The rule states that the number of days away from base at the start of the planning period are given by the Rave variable `%initial_days%`, the consumption is updated when a flight leg or connection passes midnight, and reset when the base is visited for long enough.

Soft cumulative constraints are defined by specifying a penalty function above *or* below the resource limit. The penalty functions are stated as mathematical expressions with one unknown variable, representing the amount by which the constraint is violated. The operators shown in Table 12.2 are sup-

Algorithm 12.3 An example Rave minimum connection time rule.

```
rule (on) connection_time_rule =
  %connection_time% >= %min_connection_time%;
  remark  "Minimum connection time rule";

%connection_time% = next(leg(chain), departure) - arrival;

%min_connection_time% =
  if %is_flight_to_flight_connection%
     AND %is_same_flight_connection% then
    %min_connection_time_same_flight%
  else if %is_flight_to_flight_connection% then
    %min_connection_time_thru%
  else
    %min_connection_time_default%
```

Algorithm 12.4 An example Rave cumulative constraint definition.

```
 %cumulative_base_visit%(String arg) =
  if arg = "remark" then                 "Base visit rule"
  else if arg = "limit" then             "3"
  else if arg = "initial" then           "initial_days"
  else if arg = "reset_leg" then         "leg_is_long_base_visit"
  else if arg = "reset_conn" then        "is_long_base_visit"
  else if arg = "consumption_leg" then   "leg_passes_midnight"
  else if arg = "consumption_conn" then "passes_midnight"
  else if arg = "reset_value_leg" then   "1"
  else if arg = "reset_value_conn" then "1"
  else "";
```

ported in mathematical expressions. To demonstrate the versatility of the mathematical expressions, Table 12.3 shows a few examples of penalty functions and corresponding expressions, and Figure 12.5 depicts the resulting penalty functions. Algorithm 12.5 shows a soft version of the constraint in Algorithm 12.4. The limit has been decreased to 2, and a quadratic penalty above the limit has been added. Observe that the name of the unknown variable in the expressions is irrelevant. The column generator is only able to handle non-decreasing penalty functions, but more general functions can be expressed.

$>$	$\geq$	$<$	$\leq$	$!=$	$==$
$+$	$-$	$*$	$/$	$()$	$?:$

Table 12.2 Possible operators in soft cumulative constraint penalty functions.

Soft constraint	Example penalty	Expression
Constant penalty 100 to all violating routes	100	`100`
Linear penalty for maintenance violation	$10 + 5z$	`10 + 5 * z`
Quadratic penalty for uneven aircraft utilization	$(y + 1)^2$	`(y + 1)^2`
'Strange' penalty (Mainly to show the expressiveness of the syntax)	x if $x < 5$ x^2 if $5 \leq x < 9$ 90 if $x \geq 9$	`x < 5 ? x : (x < 9 ? x^2 : 90)`

Table 12.3 Examples of soft cumulative constraint penalty functions.

Algorithm 12.5 An example Rave soft cumulative constraint definition.

```
%cumulative_base_visit%(String arg) =
 if arg = "remark" then                "Base visit rule"
 else if arg = "limit" then            "2"
 else if arg = "initial" then          "initial_days"
 else if arg = "reset_leg" then        "leg_is_long_base_visit"
 else if arg = "reset_conn" then       "is_long_base_visit"
 else if arg = "consumption_leg" then  "leg_passes_midnight"
 else if arg = "consumption_conn" then "passes_midnight"
 else if arg = "reset_value_leg" then  "1"
 else if arg = "reset_value_conn" then "1"
 else if arg = "penalty_above" then    "10*z^2"
 else "";
```

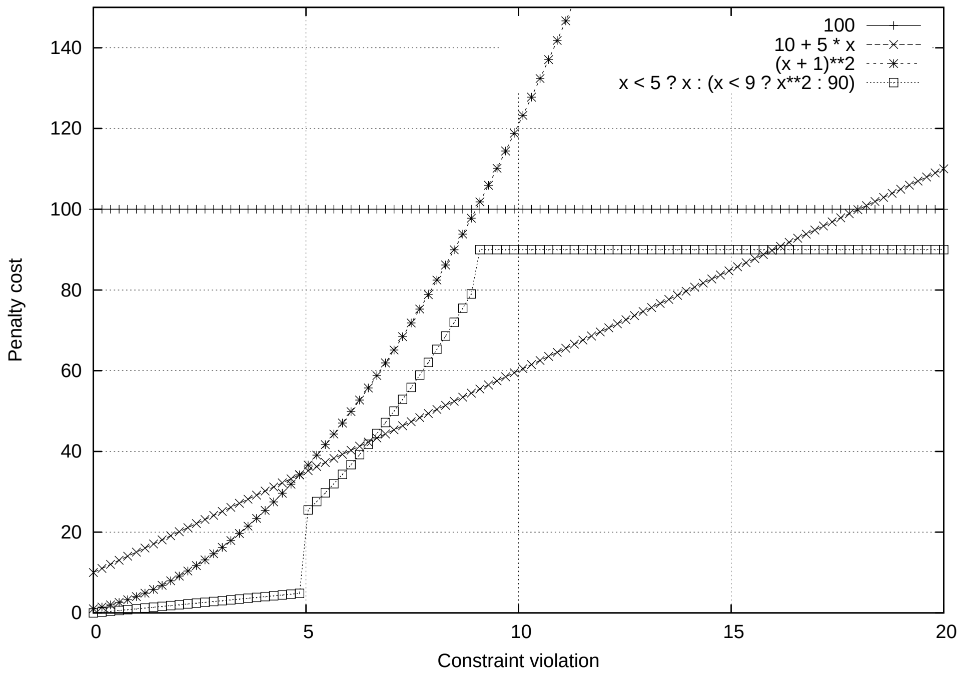
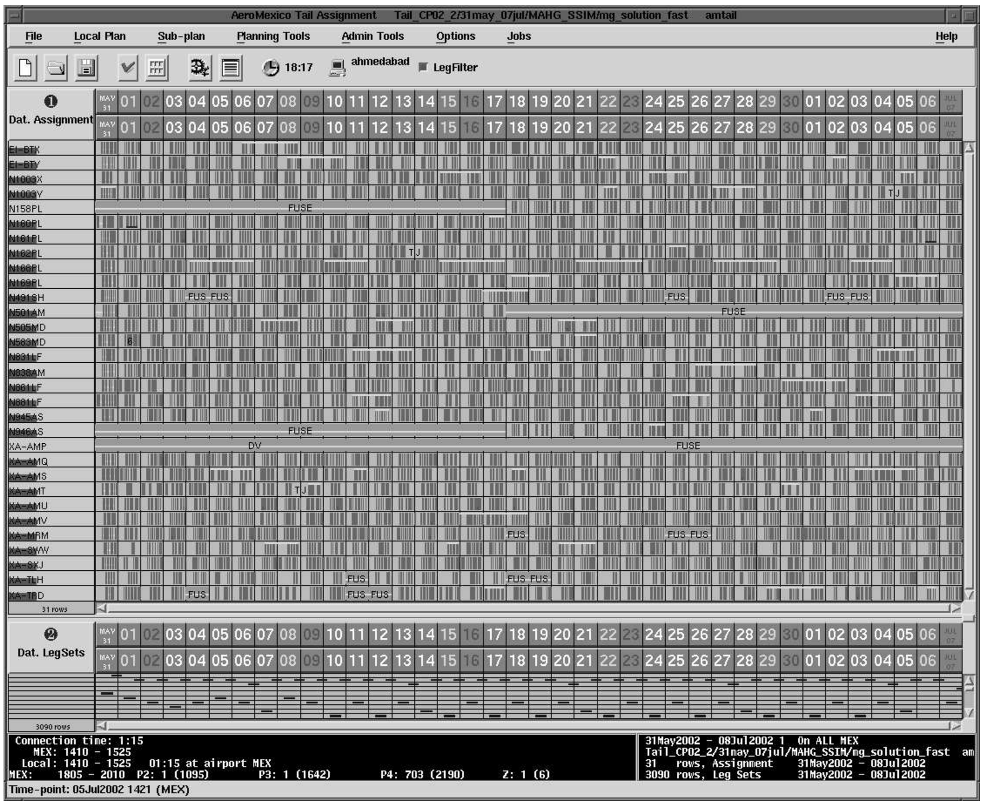


Figure 12.5 Constraint violation versus penalty cost for example penalties in Table 12.3.

12.6 The User Interface

The graphical user interface (GUI) of the tail assignment system, shown in Figure 12.6, is the most visible component to users of the system. The GUI, which is of course not part of this thesis, is where the user of a system handles preassigned activities, resolves flight imbalance problems, and sees the result of optimization runs. Since it is integrated with Rave, the GUI is also used to control constraints and costs, as well as to set optimization parameters, e.g. the maximum number of column generation iterations to perform. It can also produce customizable graphical reports of just about any aspect of the problem, using a report generator extension to Rave.

Which optimization methods to use for a particular instance is determined by *scripts* written in the *Python* script programming language [217]. When the optimizer is started, it starts a Python interpreter, which runs the Python script, and calls optimizer callback functions via predefined method calls. An example script, performing a method similar to that described in Algorithm 12.1, is shown in Algorithm 12.6. The optimizer interface is completely customizable, making it easy to try out different solution strategies by writing a simple script.



```
Connection time: 1:15
   MEX: 1410 - 1525
 Local: 1410 - 1525   01:15 at airport MEX
MEX:     1805 - 2010  P2: 1 (1095)        P3: 1 (1642)        P4: 703 (2190)       Z: 1 (6)
```

```
31May2002 - 08Jul2002 1   On ALL MEX
Tail_CP02_2/31may_07jul/MAHG_SSIM/mg_solution_fast  am
31    rows, Assignment     31May2002 - 08Jul2002
3090 rows, Leg Sets        31May2002 - 08Jul2002
```

Time-point: 05Jul2002 1421 (MEX)

Figure 12.6 The graphical user interface.

Algorithm 12.6 An example Python script.

```
initial_method("cp")

# Use a window size that increases by 6 hours every
# iteration, and a step that is half the window.
# Stop if window is 720 hours (30 days).
solveHours = 18
while solveHours <= 720:
    solveHours = solveHours + 6
    stepHours = solveHours / 2
    set_parameter("improve_sweeps", "1")
    set_parameter("improve_length", solveHours)
    set_parameter("improve_step", stepHours)
    improve_method()
```

12.7 Summary

In this chapter we have shown how the mathematical models can be combined with the constraint models, and wrapped into local search. The result is a system that tightly integrates constraint programming and mathematical programming on several levels. It first uses mixed constraint programming and column generation to find an initial solution covering the entire planning period and satisfying all constraints covering, and then uses local search combined with column generation, accelerated by constraint propagation, to create multiple solutions in an improvement phase. We also discussed how the local search improvement heuristics could potentially be parallelized.

We presented three heuristic connection network reduction methods. The purpose of the methods is to reduce the connection network, either initially to speed up column generation, or during the complete solution process. The k-FIFO filtering method is a generalization of FIFO flight matching, and is very simple and intuitive. It can be applied even before the connection network has been created, making it useful for huge instances where creating the network is very time consuming. The heuristic `costGCC` filter described in Section 10.3 can also be used as an a priori filter, but requires the connection network to be created before it can be applied. The LOF filter creates LOF-like activity sequences, leaving only long connections left to be optimized. Using this reduction method, column generation running times can be reduced by up to about 75% on some instances, with only slightly worse solution quality, as Table 12.1 shows.

Finally, we presented some non-optimization components of the tail assignment system, such as the Rave rule language and the graphical user interface.

Thirteen

Computational Results and Modeling Examples

Throughout this thesis, we have presented computational results to demonstrate the effect of various models and algorithms. However, so far we have not really analyzed the test instances or the solutions obtained, but rather only compared the obtained running times and objective values.

In this chapter we will therefore present computational results for our complete solution approach for a number of well described tail assignment planning scenarios. We will also discuss how our tail assignment approach can be used to model various types of constraints and costs, and analyze the solutions obtained from our complete solution approach. We have chosen to focus on the ability of our approach to model a wide variety of objective functions, since that is something that is lacking in many other approaches.

We will present five planning scenarios, ranging from commercial planning to various operational stability optimizations. The first two scenarios are real production scenarios, solved by the airlines which currently use our system. The last three scenarios are experimental benchmarking scenarios which show how our approach can handle constraints and costs presented in the literature, or which have been proposed by airlines.

The tests have been performed by applying both the Csp-Tas and the aggressive fixing initial methods, independently of each other. The purpose of applying both methods is to compare the running time required to find an initial solution, and the quality of the initial solution obtained. Remember that Csp-Tas does not consider the objective function at all, but only focuses on finding a solution quickly. The initial solution obtained by Csp-Tas has then been further improved by the local search algorithm presented in Figure 12.1, and more specifically the time window heuristic described in Section 12.3. The improvement strategy is the following: start with a period length of 24 hours and a step length of 12 hours, and perform a sweep over the entire planning period. Repeat at most 20 such sweeps, increasing the period by six hours in every sweep. The step is always half the period length, and the process is stopped prematurely only if optimality is reached.

We will not present lower bounds for the instances used here. Since we will present results not in terms of objective function values, but in terms of interesting properties of the obtained solutions, the lower bounds are not useful for comparison. They bound the objective function values, but since they do not represent primal feasible solutions, one cannot measure properties of them which are defined only for feasible solutions, such as the number of short connections. However, as a general remark we can mention that the lower bounds for the first two scenarios are good, while the bounds for the remaining scenarios are less good, since for these scenarios the objective functions vary more between individual aircraft.

Running times will be presented for the first three scenarios, showing that our combined solution method does indeed meet the criteria of both producing solutions quickly, and producing solutions of high quality, if longer running times are permitted. For the final two scenarios, no running times are presented, simply because these scenarios are of an experimental nature. All running times are reported as *minutes:seconds*.

13.1 Minimizing Medium-Length Connections

In this section we will show how operational robustness can be increased by our tail assignment model. Operational robustness is, loosely speaking, a measure of how much a planned tail assignment solution must be changed when disturbances occur close to the day of operations. Of course, robustness is difficult to measure exactly short of actually observing the real-world behavior. In fact, more and more research has recently been aimed toward using *simulation* to evaluate various disruption scenarios. For example, the SimAir [189] simulation tool has been used to evaluate robustness of fleet assignment solutions [188] and crew planning solutions [197]. A similar simulation tool is the MIT Extensible Air Network Simulator, MEANS [51].

Here, we will not use simulation to verify robustness. Instead, we will identify criteria that are considered to improve or deteriorate robustness, from experience, and optimize using these criteria. Short connections, slightly above the minimum connection time, are desirable since they give high utilization of the aircraft. Connections longer than some limit, e.g. three hours, are also good, because they allow the aircraft to be used as a standby aircraft, to be used in case of operational disruptions. Medium-length connections, on the other hand, are considered bad for robustness. These connections sometimes force aircraft to occupy gates during a long periods, and make it impossible to use aircraft for standby duty.

To prevent medium-length connections, a cost function such as the one

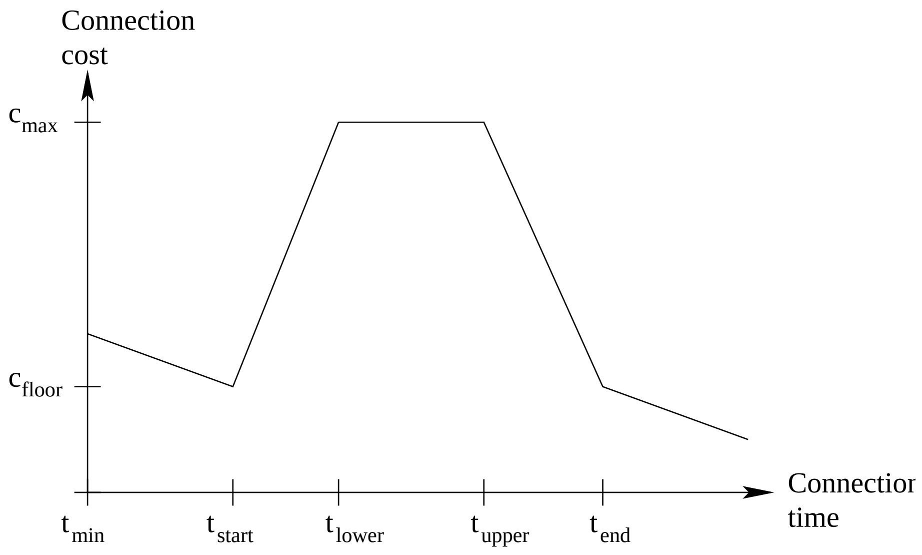


Figure 13.1 Cost function penalizing medium-length connections.

depicted in Figure 13.1 is used. The connection cost on the horizontal axis is the c_{ijt} parameter in TAS, Model 4.1, which gives the cost of each connection for a specific aircraft. For these tests, all aircraft have the same costs. The minimum legal connection time is denoted by t_{min}, so there are in fact no connections with connection time less than this. Connections close to the minimum connection time are penalized, up to an optimal connection time denoted t_{start}. Connections between t_{lower} and t_{upper} in duration are penalized with a constant penalty, and connections longer than t_{end} are penalized with a linearly decreasing penalty.

The values of the threshold times in Figure 13.1 typically depend on the airport, time of day, type of flights, and other factors. We have here chosen to simplify the cost function to be the same for all airports and times. The exception is t_{min}, which is individual for each connection.

For our tests, we want to avoid connections which are between two and six hours long, and in particular connections which are between three and five hours. To model this, we have used $t_{start} = 2$ hours, $t_{lower} = 3$ hours, $t_{upper} = 5$ hours and $t_{end} = 6$ hours. c_{floor} is 100, c_{max} 5000, and the penalties decrease by one unit per minute in the intervals $[t_{min}, t_{start}]$ and $[t_{end}, \infty)$. Table 13.1 shows the running times and the number of 2-6 and 3-5 hour connections, when using the CSP-TAS and aggressive fixing initial methods, and when using the time window improvement method.

The CSP-TAS initial method finds initial solutions within 2 minutes. The aggressive initial method only gives slightly better solution quality than CSP-TAS, and takes longer time. Within around 30 minutes we obtain improved

	Initial method						Improvement method		
	Csp-Tas			Aggr. fixing			Improved Csp-Tas		
Instance	#2-6h conn's	#3-5h conn's	Time	#2-6h conn's	#3-5h conn's	Time	#2-6h conn's	#3-5h conn's	Time
1A	151	30	00:05	144	30	00:11	111	16	02:50
1B	60	17	00:12	50	11	00:24	0	0	01:55
1C	319	107	01:01	298	98	02:47	169	61	23:13
1D	347	100	01:51	309	80	05:49	141	38	28:25
1E	443	103	01:29	412	98	10:03	214	56	31:15

Table 13.1 Running times and number of 2-6 and 3-5 hour connections used in the initial and improved solutions.

Instance	#Activities	#Aircraft	#Maintenance constraints
1A	1338	9	2
1B	2378	17	1
1C	4932	33	1
1D	5816	31	1
1E	5571	31	1

Table 13.2 The test instances for Section 13.1. Number of activities, aircraft and cumulative constraints.

solutions with substantially fewer medium-length connections than for the initial solutions. For one of the instances we can even get rid of *all* medium-length connections. While not shown in the table, the Tas^{relax} lower bound shows that the final results are optimal for all instances.

Information about the test instances is shown in Table 13.2. Instance 1A includes an 'A check' maintenance constraints, forcing all aircraft to be maintained every 450 flying hours. For all instances, there is also one generic cumulative constraint, forcing the aircraft to return to their home base every six days. The are also a number of production-type flight restrictions present.

13.2 Avoiding Tight Crew Connections

In crew pairing, many constraints and costs depend on the aircraft routes. Since aircraft routes are most often constructed *after* the crew pairing step, approximate routes are often used when solving the crew pairing problem.

These routes can be constructed by an approximate aircraft routing solver, or they can be extracted directly from the fleet assignment solution. In either of these cases, the constructed routes are likely to be impossible to operate in practice, due to all the operational constraints which are ignored. By instead using the tail assignment model, exact aircraft routes can be used already in the crew pairing step, which is of course an improvement. Further, by iterating between the crew pairing and tail assignment problems, aircraft routes more suited to the crew can be generated. We will here show how the tail assignment model can be used to minimize the number of *tight crew connections*.

Tight crew connections are connections where the crew has short time to connect from one activity to another, and thus have little time to swap aircraft. Since connections which do not require an aircraft change can be performed much faster than connections involving an aircraft change, it is clearly desirable from a robustness point of view to let the aircraft follow the tight crew connections. This has been observed e.g. by Cordeau et al. [57], and subsequently by Mercier et al. [170], who solve an integrated aircraft routing and crew pairing problem using Bender's decomposition, with the specific purpose of minimizing the number of tight connections used. Cohn and Barnhart [55] propose an integrated maintenance routing and crew pairing model, in which a crew pairing model is extended with variables representing complete maintenance routing solutions.

We use a procedure which is perhaps less elegant from a theoretical point of view than the approaches mentioned above, but perhaps more useful in practice. We iteratively solve the tail assignment and the crew pairing problem:

1. define some arbitrary cost function for the tail assignment problem;
2. solve the tail assignment problem;
3. solve the crew pairing problem using the aircraft routes from the tail assignment solution;
4. if the crew pairing quality is satisfactory, stop;
5. re-define the tail assignment cost function to penalize connections which break the tight crew connections in the current crew pairing solution. Go to step 2.

This process has been shown to generate crew pairing containing fewer tight crew connections with aircraft changes [116]. It might seem a bit strange to let the crew pairings dictate the aircraft routes, and not the other way around, since we have previously discussed how aircraft routes are given as input to crew pairing. The reason is that the crew pairing costs are normally considered more important than the costs modeled by the tail assignment cost function.

Instance	#Activities	#Aircraft	#Maintenance constraints
2A	4987	31	1
2B	5068	31	1
2C	5304	31	1
2D	5304	31	1
2E	5304	31	1

Table 13.3 The test instances for Section 13.2. Number of activities, aircraft and cumulative constraints.

	Initial method				Improvement method	
	Csp-Tas		Aggr. fixing		Improved Csp-Tas	
Instance	#Tight conn's	Time	#Tight conn's	Time	#Tight conn's	Time
2A	415	01:21	351	04:55	67	113:39
2B	733	01:53	721	30:13	134	193:42
2C	488	01:19	472	03:33	23	134:25
2D	928	01:23	872	03:47	82	128:24
2E	915	01:18	881	03:48	63	123:26

Table 13.4 Running times and number of tight crew connections used in the initial and improved solutions.

Here, we will only look at how the tail assignment problem is solved in single iterations of the iterative process, to find solutions violating as few tight crew connections as possible. We have decided to isolate the tail assignment part of the process simply because that is the topic of this thesis.

The cost function is simple: if the connection violates a tight crew connection, i.e. connects from the arrival of the tight connection to another departure, or vice versa, it is penalized by a cost of 5000. The test instances are shown in Table 13.3. They all contain 31 aircraft and around 5000 flight legs, as well as a number of flight restrictions. Instances 2C-2E are taken from the same combined crew pairing/tail assignment instance, representing iterations 2, 5 and 7 of the process described above.

Table 13.4 shows the results of running the Csp-Tas and aggressive initial methods, and running the improvement method, starting with the Csp-Tas solution. Again, the aggressive fixing heuristic finds slightly better solutions than Csp-Tas, but takes longer time. The last column shows that we are able to find solutions using very few of the tight connections with the improvement method. It is interesting to note that the number of violated tight crew connections *increases* between instances 2C and 2D. As mentioned above, these instances represent iterations 2 and 5 in the same iterative crew pairing/tail assignment run, so clearly the number of tight connections does not decrease monotonously in the iterative process. This is not surprising, since the crew pairing cost function strives to reduce the total crew cost, and not only the tight crew connection penalty cost.

The fact that the results in Table 13.4 are part of an iterative crew pairing and tail assignment process must be considered when analyzing them. What the results show is that the tail assignment optimizer, given a list of undesirable connections e.g. obtained in a crew planning step, can create solutions containing only very few of these connections. This can be useful not only in an iterative approach, but also as a cost function for a stand-alone tail assignment or aircraft routing solver, to increase operational robustness.

13.3 Commercial Planning: Minimizing the Utilization of Leased Aircraft

In the previous two sections, we have optimized operational robustness, by minimizing the number of medium-length connections and the number of tight crew connections with aircraft changes. We will now demonstrate how the tail assignment optimizer can be used also for commercial planning.

Airlines often lease parts of their fleet, to make their capacity more flexible. They will then get away from the high costs attached to owning the aircraft, but on the other hand have to pay high fees when actually using the aircraft. The fee of using a leased aircraft of course varies depending on the type of aircraft, and the leasing agreement, but €820 (about $1000) per flying hour is not an unrealistic estimate.

Since using leased aircraft is so expensive, in a fleet consisting of mixed leased and owned aircraft it is desirable to utilize the leased aircraft as little as possible, but still cover all activities. To show how our tail assignment optimizer can minimize the utilization of leased aircraft, we have re-used the instances from Section 13.1, and simulated that a few randomly selected aircraft in each instance are leased. The cost function is the same as that in Section 13.1, except that the leased aircraft are given a penalty of 50 cost units per flying minute. Observe that adding costs which depend on the individual aircraft is not a problem in models Tas and Path-Tas. The maintenance and flight restriction constraints are also the same as in Section 13.1. Observe that the cost function does not model the true lease cost, but is a mix between two optimization criteria.

Table 13.5 shows the total flying time for the leased aircraft, and the optimization running time for the five instances. In the improved solutions the leased aircraft are used substantially less than in the initial solutions. On av-

		Initial method				Improvement method	
		Csp-Tas		Aggr. fixing		Improved Csp-Tas	
Instance	#Leased Aircraft	Lease time	Time	Lease time	Time	Lease time	Time
3A	1	458:00	00:03	488:10	00:25	275:45	04:54
3B	2	534:50	00:12	541:55	00:21	348:50	12:40
3C	4	1676:12	00:57	1689:57	03:06	1234:30	93:46
3D	4	1556:13	01:43	1625:52	03:53	1029:55	228:39
3E	4	1777:06	01:29	1709:33	04:21	1144:35	255:02

Table 13.5 Running times and total flying time for the leased aircraft in the initial and improved solutions.

erage for the five instances, the flying time per leased aircraft in the improved solutions is decreased by 131 hours and 15 minutes, compared to the initial solutions. With an estimated hourly lease cost of €820, this means monthly savings of about €820 $\times$131.25 = €107625 per leased aircraft. Observe that since we only compare initial solutions to improved solutions, and the leased aircraft are randomly selected, it might seem difficult to draw conclusions about actual savings from these tests. However, what we demonstrate is that our improvement optimizer is able to produce much better solutions than solution approaches which do not care about these costs. And such approaches are not uncommon, many airlines for example do not plan their tail assignments until very late in the planning process, at which time it might be difficult to consider lease costs.

Since we here used the same cost function as in Section 13.1, with the leasing cost penalties added, it is interesting to see how the solutions compare with respect to the number of medium-length connections used. That is, are we able minimize the utilization of leased aircraft, and still obtain solutions using few medium-length connections, or do the criteria conflict with each other? It seems reasonable that the criteria do conflict somewhat, as in order to assign more activities to fewer aircraft, fewer long connections, and hence more medium-length connections, must be used. On the other hand, the reduced utilization of the leased aircraft would lead to these using more long connections. Table 13.6 shows that the new solutions use slightly more medium-length connections than the optimized solutions in Table 13.1, but still much fewer than the initial solutions in the same table.

Instance	#2-6 h. conn's	#3-5 h. conn's
3A	120	18
3B	10	0
3C	192	62
3D	161	38
3E	240	56

Table 13.6 Number of medium-length connections used for the improved solutions in Table 13.5.

13.4 Equal Aircraft Utilization

Often, airlines want the aircraft in a particular fleet to be equally utilized. This makes sure that all aircraft age at the same rate in terms of flying hours, which is important to reduce the cost of ownership. For aircraft of different age, it might be more correct to enforce non-uniform utilization, rather than equal utilization, but we will here limit ourselves to the case of equal utilization.

When the aircraft routing problem is solved with a cyclic model, as e.g. in [53], equal utilization can be handled automatically by forcing the solution to be a Hamiltonian cycle through the flight network, and let each aircraft fly the same route, offset by the length of the planning period. This has the benefit of not only enforcing equal aircraft utilization, but also to subject all aircraft to approximately the same conditions in terms of wind, temperature, and so on. Barnhart et al. [26] show how to adjust their string-based aircraft routing model to force all aircraft to fly the same route, and thus enforce equal utilization. They dynamically add *connectivity constraints*, similar to TSP subtour elimination constraints, to ensure that the route forms a Hamiltonian cycle.

In our more general case, equal aircraft utilization is more difficult to enforce. In fact, it will in most cases not be possible to use all aircraft exactly equally in a fixed period. This is the case also for the Hamiltonian cycle approach; it only obtains equal utilization in the limit, and not for an arbitrary fixed period. Instead, to obtain an as equal utilization as possible, we will use soft cumulative constraints. By penalizing aircraft that are used more than a specified amount of hours, the optimizer will strive toward using equal utilization. However, it will not subject all aircraft to the same weather conditions, as the Hamiltonian cycle approach attempts to do.

We start by determining the *target* flying time per aircraft in the planning period. In the simplest case, this is obtained by dividing the sum of the flight hours of all activities in the period by the number of aircraft. If some aircraft have a lot of preassigned maintenance scheduled in the period, the calculation can be adjusted to account for this, setting a lower target flying time for

Instance	#Activities	#Aircraft	#Maintenance constraints
4A	1803	64	3
4B	3906	30	3
4C	4161	29	1
4D	4048	31	1
4E	4244	31	3

Table 13.7 The test instances for Section 13.4. Number of activities, aircraft and cumulative constraints.

these aircraft. Similarly, if the desired situation is not equal utilization, but some aircraft should be used more than others, this can also be accounted for by adjusting the target time. Here, our target is equal utilization, which allows us to report results in terms of the standard deviation from the target. Now, given the target number of flying hours for each aircraft, a cumulative constraint is added to the problem. The limit for the constraint, above which a penalty is added, is the target time, and the penalty function is quadratic in the number of flying hours above this limit. A quadratic penalty is used, rather than a linear one, to ensure that using two aircraft α hours more than the target incurs a smaller penalty than using one aircraft 2α hours more than the target.

The test instances are shown in Table 13.7. They contain 29-64 aircraft, and 1-3 cumulative constraints. The cumulative constraints are return-to-base constraints, A checks, and lubrication checks. The instances are multi-fleet instances, and contain preassigned activities, so it is not possible to obtain completely equal utilization. Figure 13.2 shows the number of flying hours for each aircraft for instance 4E, for the initial and optimized solutions. The figure clearly shows that the utilization is a lot more equal after optimization. Some aircraft, the ones with a lot of preassigned activities, are still not used much, but most other aircraft are used around 360-370 hours.

Table 13.8 shows the detailed results for these instances. The table shows the optimal number of flying hours, calculated as described above, the minimum and maximum number of flying hours, and the standard deviation relative to the optimal flying time over the aircraft. Clearly, for some instances the improved solutions are quite a bit better in terms of equal utilization than the initial solutions, while for a few instances only minor improvements can be observed. Since some aircraft have preassigned activities, and thus cannot be used as much as other aircraft, there is clearly a lower bound on the standard deviation which we can possibly obtain. In light of this, the standard deviation improvements in Table 13.8 really are a bit better than they appear, as Figure 13.2 also shows. To account for the preassigned activities,

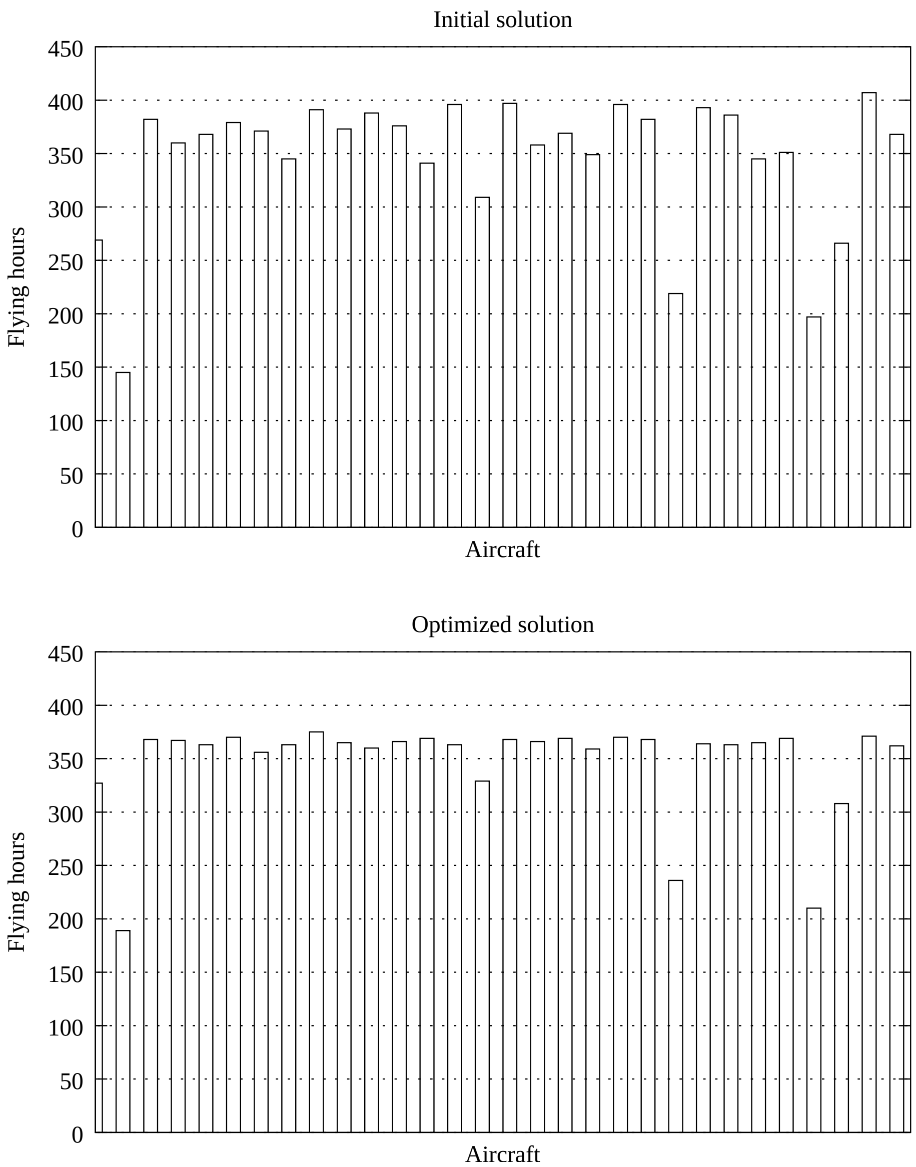


Figure 13.2 The flying time, in hours, for the individual aircraft in instance 4E, before and after optimization.

		Initial Csp-Tas			Improved Csp-Tas		
Instance	Optimal	Min	Max	Std. dev.	Min	Max	Std. dev.
4A	80:49	48:30	126:55	14:22	46:40	126:55	13:21
4B	314:27	0:00	389:15	88:42	0:00	344:20	85:42
4C	358:09	142:40	412:55	66:38	169:00	384:35	58:17
4D	329:52	0:00	384:10	66:53	0:00	346:45	61:59
4E	346:21	145:15	407:25	63:01	189:45	375:00	47:17

Table 13.8 The utilization of aircraft in the initial and improved solutions. The optimal flying time, minimum and maximum flying time, and the standard deviation from the optimal flying time over the aircraft.

we could instead have showed the percentage-wise utilization of each aircraft. This has not been done simply because it is a less exact indicator of what we are actually optimizing.

13.5 Avoiding Multiple Visits to Critical Airports

In [79], Elf et al. present an aircraft routing model for a major European airline, where one of the main objectives is to limit the number of landings per day and aircraft at so-called *air traffic critical* airports. These are airports which have been identified as particularly prone to delays, e.g. due to traffic congestion. By distributing the visits to these airports over all aircraft, Elf et al. hope to avoid excessive delay propagation for some aircraft.[1] They identify several types of delay-prone violation patterns. A *direct ATC violation* is an aircraft returning directly to an air traffic critical airport. A more complicated violation consists in several visits per day by an individual aircraft to an air traffic critical airport, and the most general case counts the delay probability propagation during each day, and puts an upper limit on this.

We have not met this type of objective for real instances, but will in this section demonstrate how our model can handle similar constraints. The complicated delay probability propagation violations could probably be modeled using cumulative constraints, but since we cannot provide realistic data for how the delay probability should be calculated, we will not consider it directly. Instead, our goal will be to limit the number of short connections per day at air traffic critical airports. We will use a soft cumulative constraint to model this, which counts the number of short connections at the air traffic

[1]One could possibly also argue that isolating the visits to air traffic critical airports to a few aircraft would better control the delay propagation, but since the suggestion of Elf et al. comes from real world requirements, we will show how it can be captured by our model.

Instance	#Activities	#Aircraft	#Maintenance constraints
5A	2509	10	1
5B	2056	15	1
5C	1726	14	1
5D	3655	25	1
5E	817	9	1

Table 13.9 The test instances for Section 13.5. Number of activities, aircraft and cumulative constraints.

critical airports, defined here as connections with connection time less than 90 minutes, and which is reset each night.

For each test instance, we have selected the three most visited airports as critical airports. This means that the main hub of the airline under consideration is defined as air traffic critical. So, while maintenance constraints force each aircraft to return to the main hub for maintenance with regular intervals, our soft constraint penalizes routes which visit the main hub too frequently. The number of short connections allowed at critical airports per day has been determined from case to case, and the penalty function is quadratic in the number of additional visits per day.

Table 13.9 shows the test instances used in this section. They are slightly smaller than some of the previous instances, but on the other hand contain a more limiting maintenance constraint. Flight restriction constraints are also present.

Table 13.10 shows the average number of days in the planned month that the aircraft perform more than the allowed limit of short connections at the critical airports. In the optimized solutions the average number of violations is substantially less than in the original solutions. Figure 13.3 shows the total number of violations per day over the planning period, before and after optimization. It is interesting to note that in the optimized case, a clear pattern emerges in the violations, except in the initial few days, which form a transition period from the fixed carry-in activities.

13.6 Summary

In this chapter, we have shown that our tail assignment model and solution method can indeed produce solutions quickly for real-world instances. As discussed in Section 1.3, whether or not the running time of an algorithm is acceptable depends on expectations and practical considerations. When we state that our solution method finds solutions quickly enough, we mean that

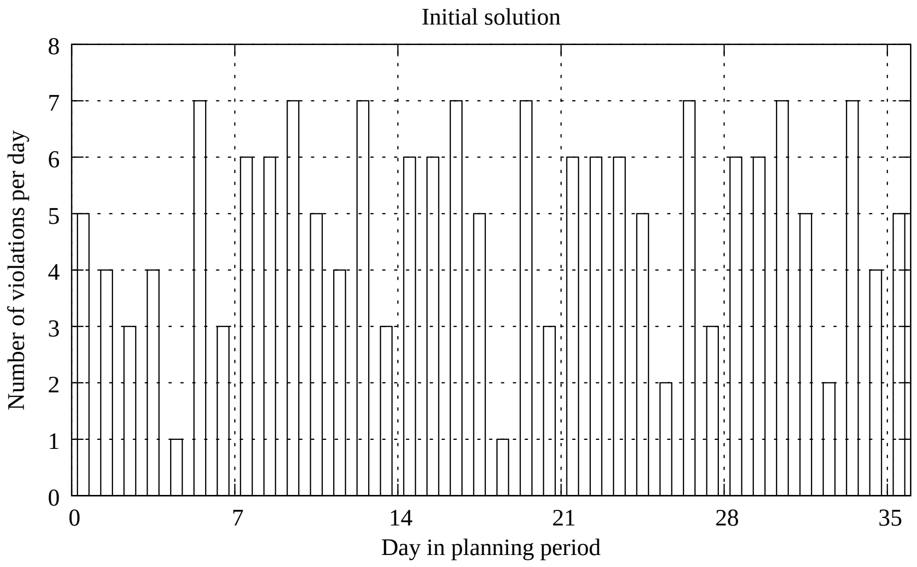
		Initial Csp-Tas	Improved Csp-Tas
Instance	Allowed visits	Average #violations	Average #violations
5A	3	16.90	4.80
5B	3	4.80	0.84
5C	1	12.33	8.33
5D	2	4.21	1.71
5E	1	7.55	2.00

Table 13.10 Results of minimizing the number of short visits per day and aircraft to air traffic critical airports. The table shows the allowed number of short visits per day, and the average number of days in the planned month for which the aircraft violate the limit, initially and after optimization.

the running times are acceptable to users of our system. We have also shown that given more time, high quality solutions can be obtained.

We have further demonstrated how our tail assignment model can be used for many types of planning scenarios. We have shown examples of how to increase operational robustness by increasing aircraft standby time, integrating with crew planning and reducing the number of landings per day to air traffic critical airports. We have also shown how tail assignment can be used to optimize commercial and financial aspects by minimizing lease costs and enforcing target aircraft utilization.

In all cases but one, we have only used one optimization criteria at a time. In the case of the lease cost minimization, we used the lease cost as the primary objective function, while the secondary goal was to minimize medium-length connections. We demonstrated that while the primary goal was met, the secondary goal was also fairly well optimized. However, this type of multi-criteria optimization is inherently difficult, and our approach does not allow any special kind of treatment for multiple optimization criteria. Instead, the task of weighing various criteria together in one objective function is left to the user of the system.



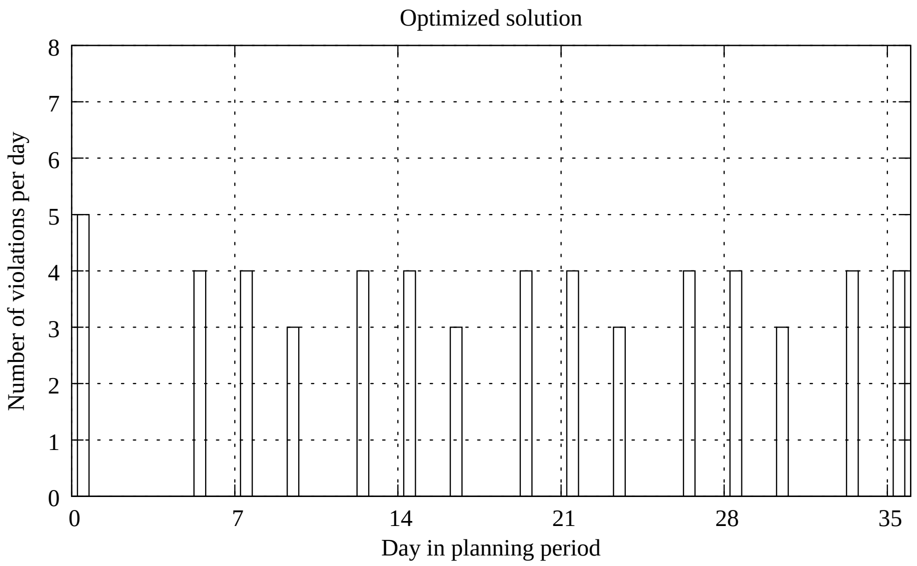


Figure 13.3 The total number of short visit violations per day over the planning period, before and after optimization.

Fourteen

Summary and Conclusions

In this thesis, we have presented the tail assignment problem, which is the approach we propose for the general aircraft assignment problem. We have described the real-world problem and the planning process of which it is a part, as well as analytical models and solution methods. In this chapter we will summarize and discuss the models, solution methods and computational results.

14.1 The Tail Assignment Problem

As discussed in the first part of this thesis, many different approaches have been proposed to solve the general aircraft assignment problem. Some approaches focus on just finding maintenance feasible routes for the aircraft, while other create a single, cyclic, route which all aircraft fly, and try to maximize the possibilities for passengers to use the same aircraft on multi-leg itineraries. In the tail assignment problem, all operational constraints, such as connection constraints, flight and destination restrictions, and maintenance constraints, are modeled. We can also model any objective function which can be decomposed into aircraft dependent connection components. The tail assignment problem is solved for a fixed time period, which means that specific information about pre-planned long-term maintenance stops and other temporary restrictions are handled for each individual aircraft.

We also allow general cumulative constraints, i.e. constraints restricting the accumulation of some value along an aircraft route. The maintenance constraints are special cases of cumulative constraints, but cumulative constraints can also be used e.g. to limit the number of landings per day for each aircraft at a specific airport. Cumulative constraints can even be soft, i.e. allow the accumulated value to exceed the limit, but at a cost. This makes it possible to model constraints which should be satisfied 'as much as possible', as well as more complicated aspects, such as equal aircraft utilization.

Already the solution to the fleet assignment problem, which provides the input for aircraft assignment, partitions the aircraft fleets and creates fleet-specific aircraft assignment problems. However, there can be several reasons why it is a good idea to take a step back and solve the aircraft assignment problem for several fleets together. One trivial reason, which we have found to occur in practice, is that no feasible aircraft assignment solution exists to the fleet assignment solution combined with the operational constraints, making it necessary to re-plan the fleet assignments. Another reason to change fleet assignments late in the planning process is that the demand estimates might have changed substantially since the fleet assignments were originally planned, making it profitable to re-plan closer to flight departure. The tail assignment problem is more flexible than most other approaches to aircraft assignment by allowing multiple fleets to be planned simultaneously.

In our tail assignment approach, all constraints and costs are directly controlled by the user of the system, via Carmen Systems' Rave rule language. This makes it easy to customize constraints and costs for various planning scenarios, as well as add and remove constraints and cost components with minimal effort.

All in all, the tail assignment problem can be used to model most of the constraints and objective functions which have been proposed for similar problems in the literature. It captures operational constraints which are often ignored by other approaches, and makes it possible to add customized constraints with minimal effort. It can also be used to solve multiple fleets simultaneously, making it useful for a wide range of planning scenarios.

A Comparison with Other Airline Planning Problems

As explained in Chapter 2, the tail assignment problem addresses one of a number of large-scale optimization problems solved by airlines regularly. Further, the tail assignment problem is our specific formulation of the general aircraft assignment problem. Rather than solving the more well studied aircraft routing or maintenance routing problems, we have chosen to identify drawbacks of these problems and formulate our own problem to address these drawbacks. This results in an approach that probably cannot be used directly by all airlines, but which in our opinion is more complete and flexible than most of the other presented approaches to aircraft assignment. By allowing aircraft of different fleet types to be solved simultaneously, the tail assignment problem can be used together with a fleet assignment optimizer which creates tail-infeasible fleetings, and by solving for a fixed time period while considering all operational constraints, it can be integrated with crew planning problems.

The tail assignment problem has many similarities to some of the other airline planning problems, both in terms of modeling and computational difficulties, but it also differs in many ways. The fact that both the fleet assignment and the tail assignment problems deal with aircraft might seem like a very important similarity, but the computational similarities between these two problems are in fact small. The fleet assignment problem only deals with anonymous aircraft types and has few side constraints, making network flow based solution approaches suitable. In tail assignment the individual aircraft are crucial, and must be given an important role in the modeling of the problem. In fact, the attention to individual aircraft makes tail assignment computationally more similar to the crew planning problems, and especially to crew rostering.

There are three main differences between the tail assignment and the crew rostering problem, apart from the obvious fact that they optimize different types of resources. Firstly, the crew rostering problem must deal with a huge number of extremely complicated route and global constraints, modeling work-time regulations and union agreements, while tail assignment has a more simple set of constraints to deal with. The complicated constraints in crew rostering makes a model and solution approach using complete rosters very suitable, as such an approach separates the roster-specific constraints from the constraints involving several rosters. As we have discussed, this separation is one of our motivations for using column generation to solve the tail assignment problem. However, the complicated roster constraints in crew rostering often make the column generation pricing problem very difficult to model, and it is therefore difficult to use an exact column generation algorithm for crew rostering without simplifying the real-world problem.

Secondly, the aircraft are utilized *much* more than the crew. While each crew member cannot work more than, say, 12 hours without a rest period, an aircraft can fly for several days before it needs maintenance. While this might sound like a small difference, it has a huge impact on the performance of the column generation algorithm. In the restricted master problem constraint matrix, it leads to very high density columns. Column densities of around five percent are not uncommon, compared to just a fraction of a percent in typical crew planning instances. High density matrices in turn leads to poor performance of the factoring routines used by commercial linear programming solvers, resulting in poor linear programming performance. We have tested different linear programming solution techniques to reduce this problem.

Finally, the relatively simple connection constraints in tail assignment, combined with many aircraft coming back to the main airports often, leads to an explosion in the number of possible connections. This combinatorial explosion of connection possibilities is encountered regardless of the solution method used, but the implications vary. For column generation, it means that many overlapping columns will be generated, which leads to poor integrality properties of the linear programming relaxation. This affects the performance of the fixing heuristics, and is one of the main reasons why some of them are so computationally inefficient. It also leads to many columns which can never be used in a solution being generated, which in turn leads to poor column generation convergence. The effect on the constraint programming approach is small, due to its efficient propagation. We have presented various preprocessing and connection reduction techniques which reduce this combinatorial explosion.

The tail assignment problem does not have that many similarities to the crew pairing problem. While crew pairing is typically solved using column generation, it differs from tail assignment in that it considers anonymous individuals, and in that it only creates partial working patterns. Also, the crew pairing problem is typically not solved for a fixed time period, but for a cyclic period of a few days.

14.2 Solution Methods

As we discussed in the first part of this thesis, the tail assignment problem can be seen either as an optimization problem or as a feasibility problem, depending on the exact planning scenario in which it is used. Due to this dual nature of the problem, we have presented two main solution techniques, based on mathematical and constraint programming, respectively. We have further shown how these two techniques can be combined in various ways to obtain hybrid solution methods which can be used both to quickly find solutions, and to find close to optimal solutions.

The Mathematical Programming Approach

Our first attempt to solve the tail assignment problem was to use mathematical programming techniques, as described in Part II of the thesis.

The Mathematical Model and Column Generation

We started by presenting a mathematical programming model which formulated the tail assignment problem as an integer multi-commodity network flow problem with side constraints. Due to the complicating side constraints, which are the cumulative constraints, we proposed column generation as a suitable method to solve the multi-commodity flow model. Column generation is an established solution technique for similar problems, and gives a nice separation between selection and generation of aircraft routes which makes it straightforward to handle the cumulative constraints as well as the other constraints. To use column generation, our multi-commodity flow problem is re-formulated as a path flow problem, where each route through the networks corresponds to a column in the restricted master problem (RMP). This way, the RMP becomes a standard set partitioning problem, and the pricing problem becomes a resource constrained shortest path problem with resource reset possibilities. To solve the pricing problem, we use a slightly enhanced version of a standard labeling algorithm, which handles resource resets and soft constraints, and uses specialized label handling.

Column Generation Acceleration Techniques

We then proceeded to test a number of techniques to accelerate the convergence and decrease the running time of the basic column generation algorithm. Such techniques are absolutely vital to get acceptable performance in practice. We presented various column management strategies, and tested the impact of using different algorithms to solve the RMP. We also presented more advanced techniques, such as a dual re-evaluation strategy to avoid generating similar columns for different aircraft, and dual stabilization and constraint aggregation techniques. Unfortunately, the constraint aggregation technique, which is perhaps the most interesting acceleration technique we present, turned out not to be practically possible to use. While it does improve the initial convergence dramatically, it is very difficult to combine with our fixing heuristics.

We showed how each acceleration technique improves the performance of the basic column generation algorithm, and combining the techniques was shown to have a very positive effect on both convergence and the running time required to reach linear programming optimality. While no test instance could be solved to linear programming optimality in 1000 iterations with the basic column generation algorithm, LP optimality was reached for all test instances in at most 567 iterations when using all acceleration techniques.

Integer Heuristics

Finally, we presented three fixing heuristics to find integer solutions. The heuristics fix variables of the RMP, connections in the flight network, or both, in a greedy fashion. It then re-generate columns, and repeats this process until an integer solution is found, or backtracking is necessary due to poor solution quality. The fixing and backtracking is done in a heuristic fashion, which means that the methods cannot guarantee that an optimal solution is found. Instead, the goal has been to find as good solutions as possible in as short time as possible.

We have shown that the pure variable fixing heuristic does not produce very good solutions, but is very fast, while the pure connection fixing heuristic can produce very good solutions, but requires a lot longer running time. As a consequence, we propose a hybrid fixing heuristic, which initially fixes variables, and then switches to connection fixing. This gives a good trade-off between running time and solution quality, but even this heuristic is not fast enough to be useful when solutions must be produced very quickly.

Algorithmic Difficulties with the Mathematical Programming Approach

Two main computational challenges were encountered for the mathematical programming solution approach. Firstly, the column generation convergence was initially very poor as the objective value approached optimality. This 'heavy tail' behavior of column generation is fairly well documented for various types of applications, and several techniques have been proposed to solve it. We investigated a number of well known techniques, as well as a few less well known ones, and found that the convergence could be improved dramatically. However, since we are interested in integer solutions, one can question the value of putting much effort on reaching LP optimality. In our case, the LP solution provides very little information except for the possibly improved lower bound. In fact, we seldom try to reach LP optimality in practice, but instead stop the column generation algorithm early, and start our integer heuristics instead.

This leads to the second computational challenge encountered, namely the difficulty of avoiding conflicting fixations in the integer fixing heuristics. Because of the poor integrality properties of the linear programming solutions, the heuristics are very sensitive to bad fixations. Since most variables in the linear programming solution initially take values far from integer, the heuristics are initially forced to perform fixing based on very fractional solutions, making it easy to fix connections or variables which cannot lead to any integer solution without unassigned activities. Our computational tests indicate, however, that our heuristics can still find close to optimal solutions, at the cost of longer running times. We have also shown how constraint programming can be used to avoid conflicting fixations.

The Constraint Programming Approach

As an alternative solution method, especially focused on quickly finding feasible solutions, we have used constraint programming, as described in Part III of the thesis.

We have presented a full constraint model for the tail assignment problem, which mixes traditional and specialized constraints and propagation algorithms. A large portion of the tail assignment problem can be formulated by using `successor` and `vehicle` variables, which represent the possible successors and aircraft which can be assigned to each activity, together with straightforward constraints, including the well-known `all_different` constraint. However, to make sure that cumulative constraints are satisfied, and to improve the propagation of flight restriction constraints to avoid excessive backtracking, specialized constraints, propagation algorithms and search heuristics must be used.

To model the cumulative constraints, the column generation pricing algorithm was re-used. The constraint model asks the pricing algorithm at various points during the search whether it is still possible to generate at least one route per aircraft, and whether all activities can be reached by at least one route. These checks can be done by simple extensions of the pricing labeling algorithm. In the process of checking feasibility, the pricing algorithm will also produce routes for the aircraft, which the constraint solver uses for its variable and value ordering heuristics. Re-using the pricing algorithm is a very important feature in practice, since it allows a single implementation of the complicated cumulative constraints.

While the restriction constraints are modeled by the `vehicle` variables, in cases where a lot of such constraints are present, the propagation with respect to these variables is not enough, resulting in thrashing and very long running times. To improve the propagation, we presented an algorithm which during the search keeps track of which activities are reachable by each aircraft. This algorithm can be used to improve the propagation of the restriction constraints, and substantially reduces trashing for instances containing many such constraints.

All in all, the constraint model can produce solutions to our largest tail assignment instances in 1–2 minutes of running time, which is clearly competitive for instances of this size.

Algorithmic Difficulties with the Constraint Programming Approach

As is often the case for constraint programming, the main contribution to reduce running times is to find the right type and amount of propagation, combined with variable and value ordering heuristics to avoid excessive backtracking. We have presented propagation algorithms which are very efficient in reducing thrashing, and we have also used ordering heuristics based on information from the column generation pricing algorithm.

Maintenance constraints which are very difficult to satisfy can cause performance problems for our constraint programming approach. It can cause excessive backtracking and subsequently very long running times. We have presented an alternative way to handle maintenance constraints which works better for these kinds of instances.

In Chapter 10, we discussed how the `all_different` constraint could be exchanged for a `costGCC` constraint to add connection costs to the constraint model. We could of course have done this for the full model, instead of just using it for preprocessing purposes. However, without a good upper bound, only very limited extra propagation would have been obtained, at the cost of longer running times. The more important reason to use the `costGCC` constraint would instead have been to find optimal solutions within the constraint programming search, by tightening the upper bound as better and better solutions are found. We decided not to try this, partly because of the extra implementation effort necessary to implement the incremental `costGCC` propagation algorithm to be used during search, but more importantly because we do not believe that the search would be very efficient compared to the column generation approach. The performance would very much rely on good solutions being found quickly, as well as clever ordering heuristics. At the moment, constraint programming alone is simply not suitable for optimization.

Combining the Approaches

The mathematical and constraint programming approaches can each be used to solve the tail assignment problem, to either obtain high-quality solutions, or to obtain solutions quickly. However, they can also be combined in various ways, to complement each other and create a hybrid solution approach.

As discussed above, the column generator pricing algorithm is re-used in the constraint programming model to handle the cumulative constraints. We have also shown how the constraint model can be used as a powerful preprocessing algorithm for tail assignment instances. By using only the propagation logic of the constraint model, and not instantiate any variables, it is possible to remove a lot of connections from the connection network that can never be used in a solution covering all activities. Similarly, it is possible to exclude some aircraft/activity combinations which can be shown not to be part of any solution covering all activities. While propagation of this kind is fundamental in constraint programming, we also show how it has a huge positive impact on column generation running time and convergence. The actual preprocessing takes very little time, but has still been shown to be much more powerful than traditional preprocessing based on simple activity balance reasoning, for our test instances.

Taking the preprocessing idea one step further, we have shown how constraint propagation can be used also in the fixing heuristics, to avoid conflicting fixing decisions, making it possible to use aggressive fixing. By maintaining an instance of the constraint model parallel to the mathematical model, and test feasibility of potential fixing decisions before doing them, even fairly aggressive fixing strategies can often be used with only small amounts of backtracking necessary. Once a fixing decision has been implemented, propagation is done in the constraint model, and the impossible connections and aircraft/activity combinations are removed in the mathematical model. We have presented a fixing heuristic which combines aggressive fixing with this kind of feasibility checking and propagation, and which produces solutions almost as quickly as the pure constraint model. Compared to the constraint model, which does not care about solution quality at all, this heuristic often produces better solutions.

Finally, we have shown how our mathematical and constraint programming approaches can be combined with local search improvement techniques to obtain a complete hybrid approach to the tail assignment problem. This approaches uses the constraint model, or the aggressive fixing heuristic combined with constraint propagation, to find an initial solution for the entire problem. It then proceeds to improve this solution gradually, by applying the column generator to subproblems of it. The subproblems can either be selected as covering the entire period but only include a few aircraft, to include all aircraft for a limited time period, or to include a few aircraft for a limited time period. The improvement is typically applied iteratively, by sweeping a 'time window' over the complete planning period, and stopped when the solution quality is sufficiently good, or when a time limit is reached.

The resulting solution method typically obtains an initial solution quickly, and then provides solutions of increasingly high quality. The process is completely controlled by the user of the system via Rave parameters and script functionality, making it possible to customize completely, for example in such a way that it solves the problem using column generation over the entire planning period. The existence of a network flow or linear programming lower bound for the entire problem, as well as for each subproblem, means that improvement heuristic can measure the solution quality, unlike many local search methods. The fact that our hybrid solution approach normally cannot guarantee optimality is not a problem in practice.

14.3 Computational Results

Throughout the thesis, we have presented computational results to validate and test the various acceleration strategies and algorithms we have proposed. In the last part of the thesis, we also presented a set of modeling examples, to show how various planning scenarios can be modeled and solved with our tail assignment approach, and especially to highlight the benefit of using a flexible optimization approach. The scenarios were a mix of actual planning scenarios and more experimental ones. The experimental scenarios all model aspects of aircraft assignment which have either been proposed by airlines, or in the literature.

First, we presented a real-world planning scenario, where the objective was to use as few medium-length connections as possible. Medium-length connections are bad from a utilization point of view, as they often require an aircraft to remain at the gate for a rather long time. During connections longer than e.g. 5 hours, the aircraft can be used for standby duty, to be used in case disruptions occur. We showed how our optimizer can produce solutions using very few medium-length connections.

The second scenario is also a real-world one. Here, the objective was to improve the crew planning by avoiding aircraft changes for crews during short connections. Normally, this is done in an iterative process using a crew pairing optimizer and our tail assignment optimizer, but for the purposes of this thesis we only showed how to solve a single iteration. We showed how solutions with substantially less aircraft changes than originally proposed could be produced by our optimizer.

The third example scenario differed from the previous ones in that it was a commercial rather than an operational planning problem. The objective was to use leased aircraft as little as possible, since using leased aircraft incurs a substantial hourly cost. We simulated a scenario where a number of aircraft in a fleet were leased, and showed how the utilization of these aircraft could be substantially reduced. For the one-month problems, the utilization of the aircraft was decreased by 131 hours and 15 minutes in average over the five instances, corresponding to many thousand Euros in leasing costs. We also showed that the decreased utilization of the leased aircraft did not substantially decrease the solution quality in terms of the number of medium-length connections used, demonstrating that our model can be used for multi-criteria objective functions.

The fourth scenario used soft cumulative constraints to obtain target aircraft utilization. Since aircraft age in terms of flying hours is an important measure for the cost of an aircraft, and thus needs to be controlled, airlines often want the aircraft to be utilized to a certain degree each month. In case all aircraft are of approximately the same age, they should normally be used equally, but if they are of different age, each aircraft might have a target utilization which it should follow. For simplicity, we have showed how our optimizer can be used in case equal aircraft utilization is desired. By using a soft cumulative constraint with a quadratic penalty, we obtained utilization much more equal than a solution which does not care about utilization targets, which can e.g. represent a solution created close to flight departure, when ensuring equal aircraft utilization is not possible.

Finally, we showed how our tail assignment approach can be used to increase operational stability by avoiding multiple visits per day by the same aircraft to delay-critical airports. This stability criterion has been proposed in the literature [79] for a European airline. We use customized cumulative constraints to simulate that three of the airports in our instances are delay critical, and could show how the average number of days for which the aircraft visit the delay-critical airports too often can be substantially reduced. In one of the optimized solutions we even found a regular pattern, indicating that an underlying structure of the flight network made it necessary to break the constraint for a number of days.

Altogether, our computational results show that our tail assignment model and solution approach can model and solve real-world planning scenarios in reasonable time, and obtain solutions of high quality.

While we have discussed the ability of the tail assignment problem to model re-fleeting, i.e. solving several fleets together close to the day of operation, using real up to date revenue data, we have not presented any computational results for such cases. Some of our test instances contain several fleets which can be mixed, but none of them do actual revenue focused re-fleeting. The main reason for this is that the type of data required for this is very valuable to the airlines, and we have simply not had access to it. Obviously, we could have used randomly generated revenue data to simulate re-fleeting, but we decided that this would not be meaningful.

Real World Experience

The tail assignment optimizer described in this thesis is presently in use at two medium-sized airlines. The complete planning solution contains a graphical user interface, manual planning features and the Rave rule system, along with the optimizer. The main usage of the system is to provide robust aircraft routes within a few weeks of operation. At one of the airlines, the optimizer also uses revenue data to enable re-fleeting. Tests have been carried out that show great potential for integration with a crew pairing optimizer, to increase robustness of crew schedules as well.

Before using the tail assignment optimizer, both airlines constructed the tail routes by hand. Since the exact cost function and constraints used in the optimizer were formulated as our model was developed, and are not exactly the ones used before, it is difficult to directly compare the quality of solutions. Instead, the main benefit of the system has been that the construction of tail routes can now be done much faster, and with greater control. Many scenarios, using well-defined costs and rules, can be tested in the same amount time it previously took to construct a single solution. High quality solutions have also contributed to an increase in aircraft availability and increased revenues.

14.4 Final Remarks

The tail assignment approach presented in this thesis is currently maintained as a commercial product by Carmen Systems. It is continually improved, both in terms of modeling and computational efficiency, and in terms of software quality. One of the ideas for future improvements is improved integration with crew planning products, as we have shown that integrated tail and crew planning can be very beneficiary.

Also, it would be interesting to develop methods to allow our solution approach to be used on *really* large instances, involving many hundreds of aircraft, as this could make it possible to use our approach as a full fleet assignment solver. While fleet assignment is a well documented problem, one might question whether it is actually necessary to create a fleeting solution many months before the day of operation, only to find that it is infeasible when operational constraints are considered. Instead, one could explore the possibilities of using our tail assignment approach to create a solution which considers all operational constraints, just long enough in advance to be able to create crew rosters in time. This potentially makes integration with crew planning even more fruitful, and also makes it possible to use accurate revenue data based on the current booking situation. The impact on booking and gate allocation systems, which might need information about fleet types early, would then of course have to be investigated further.

One of the really strong points of our tail assignment solution approach is the integration of column generation and constraint programming. This type of integration could potentially be interesting in other areas as well, and how similar integration could be used for crew planning is definitely something that should be investigated further.

Finally, we believe that using our tail assignment approach can be a real benefit to many airlines. Switching from current practice to using tail assignment might be a big step for some airlines, and even involve organizational changes. However, we are convinced that the benefits of an approach which can potentially be used throughout the planning process, from fleet assignment and all the way to the day of operation, are worth the change. In our opinion, the future of resource optimization is to make decisions when detailed operational and commercial information is available, rather than to solve abstract long-term planning problems which must be adjusted to reality later.

Bibliography

[1] E. Aarts and J. K. Lenstra, editors. *Local search in Combinatorial Optimization*. Princeton University Press, 2003. Cited on pages: 58, 195

[2] J. Abara. Applying Integer Linear Programming to the Fleet Assignment Problem. *Interfaces*, 19:20–28, 1989. Cited on pages: 19

[3] A. Abdelghany, G. Ekollu, R. Narasimhan, and K. Abdelghany. A Proactive Crew Recovery Decision Support Tool for Commercial Airlines During Irregular Operations. *Annals of Operations Research*, 127(1–4):309–331, 2004. Cited on pages: 25

[4] A. Aggoun and N. Beldiceanu. Extending CHIP in order to solve complex scheduling and placement problems. *Mathematical Computational Modelling*, 17(7):57–73, 1992. Cited on pages: 36, 133

[5] R. K. Ahuja, J. Goodstein, A. Mukherjee, J. B. Orlin, and D. Sharma. A Very Large-Scale Neighborhood Search Algorithm for the Combined Through and Fleet Assignment Model. Working Paper 4388-01, MIT Sloan School of Management, Cambridge, MA, USA, 2001. Cited on pages: 21, 24, 26, 27, 40

[6] R. K. Ahuja, J. Liu, J. Goodstein, A. Mukherjee, J. B. Orlin, and D. Sharma. Solving Multi-Criteria Combined Through Fleet Assignment Models. In T. A. Ciriani, G. Fasano, S. Gliozzi, and R. Tadei, editors, *Operations Research in Space and Air*, pages 233–256. Kluwer Academic Publishers, 2003. Cited on pages: 21, 27, 40

[7] R. K. Ahuja, J. Liu, J. B. Orlin, J. Goodstein, and A. Mukherjee. A Neighborhood Search Algorithm for the Combined Through and Fleet Assignment Model with Time Windows. Submitted to Networks, MIT Sloan School of Management, Cambridge, MA, USA, 2003. Cited on pages: 40

[8] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993. Cited on pages: 49, 50, 52, 55, 56, 62, 68, 95

[9] R. K. Ahuja, J. B. Orlin, and D. Sharma. Very large-scale neighborhood search. *International Transactions in Operational Research*, 7:301–317, 2000. Cited on pages: 198

[10] Applying MSG-3 to Out of Production Aircraft. *Aircraft Technology Engineering & Maintenance*, 50, February/March 2001. Cited on pages: 33, 34

[11] A. I. Ali, R. V. Helgason, J. L. Kennington, and H. Lall. Computational Comparison among Three Multicommodity Network Flow Algorithms. *Operations Research*, 28(4):995–1000, 1980. Cited on pages: 51

[12] R. Anbil, J. J. Forrest, and W. R. Pulleyblank. Column Generation and the Airline Crew Pairing Problem. *Documenta Mathematica – Journal der Deutschen MathematikerVereinigung, number III in extra volume: proceedings of the ICM*, pages 677–686, 1998. Cited on pages: 66, 67

[13] E. Andersson, A. Forsman, S. E. Karisch, N. Kohl, and A. Sørensson. Problem Solving in Airline Operations. Carmen Research and Technology Report CRTR-0404, Carmen Systems AB, Gothenburg, Sweden, June 2004. Cited on pages: 157, 209

[14] E. Andersson, E. Housos, N. Kohl, and D. Wedelin. Crew Pairing Optimization. In G. Yu, editor, *OR in the Airline Industry.* Kluwer Academic Publishers, 1998. Cited on pages: 22, 66

[15] L. H. Appelgren. A column generation algorithm for a ship scheduling problem. *Transportation Science*, 3:53–68, 1969. Cited on pages: 107

[16] K. R. Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003. Cited on pages: 127, 137, 138

[17] A. A. Assad. Multicommodity Network Flows – A Survey. *Networks*, 8:37–91, 1978. Cited on pages: 50

[18] Air Transport Association of America, Inc. `http://www.airlines.org/`. Cited on pages: 33

[19] L. Augustsson. Partial Evaluation in Aircraft Crew Planning. In *Partial Evaluation and Semantics-Based Program Manipulation, Amsterdam, The Netherlands, June 1997*, pages 127–136. New York: ACM, 1997. Cited on pages: 209

[20] B. Awerbuch and T. Leighton. Improved Approximation Algorithms for the Multi-Commodity Flow Problem and Local Competitive Routing in Dynamic Networks. In *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, pages 487–496, May 1994. Cited on pages: 56

[21] B. D. Backer, V. Furnon, P. Kilby, P. Prosser, and P. Shaw. Solving Vehicle Routing Problems using Constraint Programming and Metaheuristics. *Journal of Heuristics*, 1(16), 1997. Cited on pages: 148, 184

[22] F. Barahona and R. Anbil. The Volume Algorithm: producing primal solutions with a subgradient method. Technical report, IBM Research Division, T.J. Watson Research Center, Yorktown Heights, New York, 1998. Cited on pages: 67

[23] J. F. Bard and I. G. Cunningham. Improving Through-Flight Schedules. *IIE Transactions*, 19(3):242–252, November 1987. Cited on pages: 40

[24] C. Barnhart. Dual-Ascent Methods for Large-Scale Multicommodity Flow Problems. *Naval Research Logistics*, 40:305–324, 1993. Cited on pages: 56

[25] C. Barnhart, P. Belobaba, and A. R. Odoni. Applications of Operations Research in the Air Transport Industry. *Transportation Science*, 37(4):368–391, November 2003. Cited on pages: 16

[26] C. Barnhart, N. L. Boland, L. W. Clarke, E. L. Johnson, G. L. Nemhauser, and R. G. Shenoi. Flight String Models for Aircraft Fleeting and Routing. *Transportation Science*, 32(3):208–220, August 1998. Cited on pages: 21, 24, 27, 28, 31, 37, 38, 42, 114, 223

[27] C. Barnhart, C. A. Hane, and P. H. Vance. Using Branch-and-Prich-and-Cut to Solve Origin-Destination Integer Multicommodity Flow Problems. *Operations Research*, 48(2):318–326, 2000. Cited on pages: 57

[28] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-Price: Column Generation for Solving Huge Integer Programs. *Operations Research*, 46(3):316–329, May-June 1998. Cited on pages: 57, 59, 66, 76, 100, 107, 111, 114, 115, 116

[29] C. Barnhart, T. S. Kniker, and M. Lohatepanont. Itinerary-Based Airline Fleet Assignment. *Transportation Science*, 36(2):199–217, May 2002. Cited on pages: 20, 39

[30] M. C. Bartholomew-Biggs, S. C. Parkhurst, and S. P. Wilson. Global Optimization Approaches to an Aircraft Routing Problem. *European Journal of Operational Research*, 146(2):417–431, 2003. Cited on pages: 23

[31] M. S. Bazaraa, H. D. Sherali, and C. M. Shetty. *Nonlinear Programming: Theory and Algorithms*. John Wiley & Sons, Inc., 1993. Cited on pages: 51

[32] N. Bélanger, G. Desaulniers, F. Soumis, and J. Desrosiers. Periodic Airline Fleet Assignment with Time Windows, Spacing Constraints, and Time Dependent Revenues. Les Cahiers du GERAD G-2003-41, GERAD, June 2003. Cited on pages: 20

[33] N. Bélanger, G. Desaulniers, F. Soumis, J. Desrosiers, and J. Lavigne. Weekly Airline Fleet Assignment with Homogeneity. Les Cahiers du GERAD G-2002-70, GERAD, December 2002. Cited on pages: 19, 20

[34] N. Beldiceanu and E. Contejean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, December 1994. Cited on pages: 36, 133

[35] H. Ben Amor, J. Desrosiers, and J. M. Valério de Carvalhò. Dual-optimal Inequalities for Stabilized Column Generation. Les Cahiers du GERAD G-2003-20, GERAD, 2003. Cited on pages: 94

[36] M. E. Berge and C. A. Hopperstad. Demand Driven Dispatch: A Method for Dynamic Aircraft Capacity Assignment, Models and Algorithms. *Operations Research*, 41(1):153–168, January-February 1993. Cited on pages: 21, 38

[37] C. Bessière. Arc-consistency and arc-consistency again. *Artificial Intelligence*, 65:179–190, 1994. Cited on pages: 131

[38] C. Bessière, E. C. Freuder, and J.-C. Régin. Using Constraint Metaknowledge to Reduce Arc Consistency Computations. *Artificial Intelligence*, 107:125–148, 1999. Cited on pages: 131

[39] C. Bessière and J.-C. Régin. Refining the basic constraint propagation algorithm. In *Proceedings of IJCAI-01*, pages 309–315, 2001. Cited on pages: 131

[40] R. E. Bixby, M. Fenelon, Z. Gu, E. Rothberg, and R. Wunderling. MIP: Theory and Practice - Closing the Gap. In *System Modelling and Optimization*, pages 19–50, 1999. Cited on pages: 107

[41] A. Bockmayr and T. Kasper. Branch and Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming. *INFORMS Journal on Computing*, 10(3):287–300, 1998. Cited on pages: 7, 138

[42] R. Borndörfer, M. Grötschel, and A. Löbel. Scheduling Duties by Adaptive Column Generation. ZIB-Report 01-02, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, Germany, January 2001. Cited on pages: 66, 76, 114

[43] J.-M. Cao and A. Kanafani. The Value of Runway Time Slots for Airlines. *European Journal of Operational Research*, 126:491–500, 2000. Cited on pages: 25

[44] A. Caprara, F. Focacci, E. Lamma, P. Mello, M. Milano, P. Toth, and D. Vigo. Integrating Constraint Logic Programming and Operations Research Techniques for the Crew Rostering Problem. *Software – Practice and Experience*, 28(1):49–76, January 1998. Cited on pages: 7, 139, 147

[45] J. Castro and A. Frangioni. A Parallel Implementation of an Interior-Point Algorithm for Multicommodity Network Flows. In J. M. L. M. Palma, J. J. Dongarra, and V. Hernández, editors, *Vector and Parallel Processing - VECPAR 2000: 4th International Conference, vol. 1981 of Lecture Notes in Computer Science*, pages 301–315. Springer-Verlag, June 2000. Cited on pages: 56

[46] A. M. Cheadle, W. Harvey, A. J. Sadler, J. Schimpf, K. Shen, and M. G. Wallace. ECL^{i}PS^e: An Introduction. Technical Report IC-Parc-03-1, IC-Parc, Imperial College London, London, UK, 2003. Cited on pages: 138

[47] V. C. P. Chen, D. Günther, and E. L. Johnson. Solving for an optimal airline yield management policy via statistical learning. *Journal of the Royal Statistical Society*, 52(1):19–30, 2003. Cited on pages: 18

[48] Y. Chen. Arc Consistency Revisited. *Information Processing Letters*, 70:175–184, 1999. Cited on pages: 131

[49] G. Christodoulou and P. Stamatopoulos. Crew Assignment by Constraint Logic Programming. In *Proceedings of the 2nd Hellenic Conference on Artificial Intelligence SETN-2002 (Companion Volume)*, pages 117–127, Thessaloniki, Greece, 2002. Cited on pages: 146

[50] V. Chvátal. A greedy heuristic for the set covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979. Cited on pages: 42

[51] J.-P. Clarke. MEANS – MIT Extensible Air Network Simulation. Presentation at AGIFORS Airline Operations Conference, 2003. Cited on pages: 216

[52] L. W. Clarke, C. A. Hane, E. L. Johnson, and G. L. Nemhauser. Maintenance and Crew Considerations in Fleet Assignment. *Transportation Science*, 30(3):249–260, August 1996. Cited on pages: 21, 27, 31, 38

[53] L. W. Clarke, E. L. Johnson, G. L. Nemhauser, and Z. Zhu. The Aircraft Rotation Problem. *Annals of Operations Research*, 69:33–46, 1997. Cited on pages: 24, 30, 31, 37, 42, 223

[54] M. D. D. Clarke and G. Laporte. The Airline Schedule Recovery Problem. Technical report, International Center for Air Transportation 33-

212, MIT, Cambridge, MA, USA and Centre de recherche sur les transports, Université de Montréal, Canada, 1997. Cited on pages: 26

[55] A. M. Cohn and C. Barnhart. Improving Crew Scheduling by Incorporating Key Maintenance Routing Decisions. *Operations Research*, 51(3), May–June 2003. Cited on pages: 23, 27, 219

[56] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium on the Theory of Computing*, pages 151–158, 1971. Cited on pages: 128

[57] J.-F. Cordeau, G. Stojković, F. Soumis, and J. Desrosiers. Benders Decomposition for Simultaneous Aircraft Routing and Crew Scheduling. *Transportation Science*, 35(4), November 2001. Cited on pages: 24, 26, 42, 219

[58] J.-F. Cordeau, P. Toth, and D. Vigo. A Survey of Optimization Models for Train Routing and Scheduling. *Transportation Science*, 32(4):380–404, November 1998. Cited on pages: 44

[59] S. D. Curtis, B. M. Smith, and A. Wren. Forming Bus Driver Schedules using Constraint Programming. Technical Report Report 99.05, School of Computer Studies, University of Leeds, March 1999. Cited on pages: 146

[60] G. B. Dantzig. Programming of Interdependent Activities 1. Mathematical Model. *Econometrica*, 17(3 & 4):200–211, 1949. Cited on pages: 65

[61] G. B. Dantzig and P. Wolfe. Decomposition Principle for Linear Programs. *Operations Research*, 8:101–111, 1960. Cited on pages: 7, 52, 63, 64, 65

[62] G. B. Dantzig and P. Wolfe. The Decomposition Algorithm for Linear Programs. *Econometrica*, 29(4):767–778, 1961. Cited on pages: 64

[63] K. Darby-Dowman and J. Little. Properties of Some Combinatorial Optimization Problems and Their Effect on the Performance of Integer Programming and Constraint Logic Programming. *INFORMS Journal on Computing*, 10(3):276–286, 1998. Cited on pages: 138

[64] Dash Optimization Ltd. *Xpress-Optimizer Reference Manual, release 14*, 2002. Cited on pages: 11, 62, 67, 91, 93, 107

[65] M. S. Daskin and N. D. Panayotopoulos. A Lagrangian Relaxation Approach to Assigning Aircraft to Routes in Hub and Spoke Networks. *Transportation Science*, 23:91–99, 1989. Cited on pages: 24, 41

[66] P. R. Day and D. M. Ryan. Flight Attendant Rostering for Short-Haul Airline Operations. *Operations Research*, 45(5):649–661, September-October 1997. Cited on pages: 22

[67] A. de Silva. Bus Driver Scheduling by Combining Constraint Programming and Linear Programming. *Annals of Operations Research*, 108(1–4):277–291, November 2001. Cited on pages: 139, 147

[68] R. Dechter. *Constraint Processing.* Morgan Kaufmann, 2003. Cited on pages: 127, 128, 131, 137, 138

[69] G. Desaulniers, J. Desrosiers, Y. Dumas, S. Marc, B. Rioux, M. M. Solomon, and F. Soumis. Crew Pairing at Air France. *European Journal of Operational Research*, 97:245–259, 1997. Cited on pages: 22

[70] G. Desaulniers, J. Desrosiers, Y. Dumas, M. M. Solomon, and F. Soumis. Daily Aircraft Routing and Scheduling. *Management Science*, 43(6):841–855, July 1997. Cited on pages: 21, 24, 26, 31, 42

[71] G. Desaulniers, J. Desrosiers, M. Gamache, and F. Soumis. Crew Scheduling in Air Transportation. Les Cahiers du GERAD G-97-26, GERAD, November 1997. Cited on pages: 22

[72] G. Desaulniers, J. Desrosiers, and M. M. Solomon. Accelerating Strategies in Column Generation Methods for Vehicle Routing and Crew Scheduling Problems. Les Cahiers du GERAD G-99-36, GERAD, August 1999. Cited on pages: 66

[73] G. Desaulniers, J. Desrosiers, and M. M. Solomon. Accelerating Strategies for Column Generation Methods in Vehicle Routing and Crew Scheduling Problems. In C. C. Ribeiro and P. Hansen, editors, *Essays and Surveys in Metaheuristics*, pages 309–324. Kluwer Academic Publishers, 2001. Cited on pages: 59, 66, 77, 91, 203

[74] G. Desaulniers, J. Desrosiers, and M. M. Solomon. *Column Generation.* Kluwer Academic Publishers, 2005. Cited on pages: 66

[75] M. Desrochers and F. Soumis. A generalized permanent labelling algorithm for the shortest path problem with time windows. *INFOR*, 26(3):191–212, 1988. Cited on pages: 71, 157

[76] J. Desrosiers and M. E. Lübbecke. A Primer in Column Generation. Les Cahiers du GERAD G-2004-02, Technische Universität Berlin and GERAD, 2004. Cited on pages: 66, 77

[77] O. du Merle, D. Villeneuve, J. Desrosiers, and P. Hansen. Stabilized Column Generation. *Discrete Mathematics*, 194:229–237, 1999. Cited on pages: 66, 94

[78] H. El Sakkout. Modelling Fleet Assignment in a Flexible Environment. In *Proceedings of the Second International Conference on the Practical Application of Constraint Technology (PACT 96)*, pages 27–39, 1996. Cited on pages: 19, 139, 147

[79] M. Elf, M. Jünger, and V. Kaibel. Rotation Planning for the Continental Service of a European Airline. In W. Jager and H.-J. Krebs, editors, *Mathematics – Key Technologies for the Future. Joint Projects between Universities and Industry*, pages 675–689. Springer Verlag, 2003. Cited on pages: 24, 36, 40, 43, 226, 240

[80] I. Elhallaoui, D. Villeneuve, F. Soumis, and G. Desaulniers. Dynamic Constraint Aggregation of Set Partitioning Constraints in Column Generation. Les Cahiers du GERAD G-2003-45, HEC Montréal, 2003. Submitted to *Operations Research*. Cited on pages: 66, 97, 98, 99, 100, 104

[81] S. Elhedhli and J.-L. Goffin. The Integration of an Interior-Point Cutting Plane Method within a Branch-and-Price Algorithm. Les Cahiers du GERAD G-2001-19, GERAD, 2001. Cited on pages: 67, 91

[82] N. Eliasson. Nonlinear pricing and surrogate column generation in linear programming. Master's thesis, Linköping University, April 1999. Cited on pages: 66

[83] G. Erling and D. Rosin. Tail Assignment with Maintenance Restrictions - A Constraint Programming Approach. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2002. Cited on pages: 152

[84] Europa – The European Union On-Line. `http://europa.eu.int/`. Cited on pages: 31

[85] S. Even, A. Itai, and A. Shamir. On the complexity of timetable and multicommodity flow problems. *SIAM Journal on Computing*, 5(4):691–703, December 1976. Cited on pages: 50

[86] T. Fahle, U. Junker, S. E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben. Constraint Programming Based Column Generation for Crew Assignment. *Journal of Heuristics*, 8(1):59–81, 2002. Cited on pages: 7, 66, 90, 147, 148

[87] J. C. Falkner and D. M. Ryan. A Bus Crew Scheduling System using a Set Partitioning Model. *Asia-Pacific Journal of Operational Research*, 4:39–56, 1987. Cited on pages: 114

[88] A. Farkas. *The Influence of Network Effects and Yield Management on Airline Fleet Assignment Decisions*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1995. Cited on pages: 20

[89] T. A. Feo and J. F. Bard. Flight Scheduling and Maintenance Base Planning. *Management Science*, 35(12), December 1989. Cited on pages: 25, 41

[90] F. Focacci, A. Lodi, M. Milano, and D. Vigo. Solving TSP through the Integration of OR and CP Techniques. In M. G. Wallace, Y. Caseau, E. Jacquet-Lagreze, H. Simonis, and G. Pesant, editors, *Electronic Notes in Discrete Mathematics*, volume 1. Elsevier, 1999. Cited on pages: 138, 139, 178

[91] L. R. Ford, Jr. and D. R. Fulkerson. A Suggested Computation for Maximal Multi-Commodity Network Flows. *Management Science*, 5(1):97–101, November 1958. Cited on pages: 52, 54

[92] J. J. Forrest. Mathematical Programming with a Library of Optimization Subroutines, October 1989. presented at the ORSA/TIMS Joint National Meeting, New York. Cited on pages: 67

[93] A. Frangioni and G. Gallo. A Bundle Type Dual-Ascent Approach to Linear Multicommodity Min-Cost Flow Problems. *INFORMS Journal on Computing*, 11(4):370–393, 1996. Cited on pages: 56

[94] D. Frost and R. Dechter. Looking at full look-ahead. In E. C. Freuder, editor, *Proceedings of the Second International Conference on the Principles and Practice of Constraint Programming - CP'1996, vol. 1118 of Lecture Notes in Computer Science*, pages 539–540. Springer-Verlag, 1996. Cited on pages: 135

[95] M. Gamache and F. Soumis. A Method for Optimally Solving the Rostering Problem. In G. Yu, editor, *OR in the Airline Industry*. Kluwer Academic Publishers, 1998. Cited on pages: 88, 114

[96] M. Gamache, F. Soumis, G. Marquis, and J. Desrosiers. A Column Generation Approach for Large-Scale Aircrew Rostering Problems. *Operations Research*, 47(2):247–263, April-March 1999. Cited on pages: 22, 59, 66

[97] M. Gamache, F. Soumis, D. Villeneuve, J. Desrosiers, and E. Gélinas. The Preferential Bidding System at Air Canada. *Transportation Science*, 32(3):246–255, August 1998. Cited on pages: 22

[98] E. R. Gansner and S. C. North. An open graph visualization system and its applications to software engineering. *Software – Practice and Experience*, 30(11):1203–1233, 2000. Cited on pages: 142

[99] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness*. Freeman, San Fransisco, 1979. Cited on pages: 50, 60, 70

[100] J. Gaschnig. Performance measurement and analysis of search algorithms. Technical report cmu-cs-79-124, Carnegie Mellon University, Pittsburgh, PA, USA, 1979. Cited on pages: 137

[101] F. Glover. Future paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*, 13(5):533–549, 1986. Cited on pages: 58

[102] F. Glover and M. Laguna, editors. *Tabu Search*. Kluwer Academic Publishers, 1997. Cited on pages: 58, 148, 201

[103] R. Gopalan and K. T. Talluri. Mathematical Models in Airline Schedule Planning: A Survey. *Annals of Operations Research*, 76(1):155–185, 1998. Cited on pages: 16, 19, 34, 39, 40, 100

[104] R. Gopalan and K. T. Talluri. The Aircraft Maintenance Routing Problem. *Operations Research*, 46(2):260–271, March–April 1998. Cited on pages: 24, 34, 40, 100, 205

[105] S. Götz, S. Grothklags, G. Kliewer, and S. Tschöke. Solving the Weekly Fleet Assignment Problem for Large Airlines. In *Proceedings of the III Metaheuristic International Conference*, 1999. Cited on pages: 19

[106] M. Grönkvist. Structure in Airline Crew Optimization Problems. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 1998. Cited on pages: 22, 91, 115, 116

[107] M. Grönkvist. Tail Assignment – A Combined Column Generation and Constraint Programming Approach. Lic. Thesis, Chalmers University of Technology, Gothenburg, Sweden, 2003. Cited on pages: 112, 117

[108] Z. Gu, E. L. Johnson, G. L. Nemhauser, and Y. Wang. Some Properties of the Fleet Assignment Problem. Technical report, School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA, USA, 1993. Cited on pages: 19

[109] T. Gustafsson. A heuristic approach to column generation for airline crew scheduling. Lic. Thesis, Chalmers University of Technology, Gothenburg, Sweden, 1999. Cited on pages: 22, 59, 66, 67

[110] M. T. Hajian, H. El Sakkout, M. G. Wallace, B. Richards, and J. M. Lever. Towards a Closer Integration of Finite Domain Propagation and Simplex-Based Algorithms. *Annals of Operations Research*, 81:421–431, 1998. Cited on pages: 7, 138, 139

[111] C. Halatsis, P. Stamatopoulos, I. Karali, T. Bitsikas, G. Fessakis, A. Schizas, S. G. Sfakianakis, C. Fouskakis, T. Koukoumpetsos, and D. Papageorgiou. Crew Scheduling Based on Constraint Programming:

The PARACHUTE Experience. In *Proceedings of the 3rd Hellenic-European Conference on Mathematics and Informatics HERMIS '96*, pages 424–431, 1996. Cited on pages: 146

[112] C. A. Hane, C. Barnhart, E. L. Johnson, R. E. Marsten, G. L. Nemhauser, and G. Sigismondi. The fleet assignment problem: solving a large-scale integer program. *Mathematical Programming*, 70:211–232, 1995. Cited on pages: 19, 30, 31, 32, 39, 172

[113] R. M. Haralick and G. L. Elliot. Increasing tree-search efficiency for constraint satisfaction problems. *Artificial Intelligence*, 14:263–313, 1980. Cited on pages: 135

[114] W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In C. S. Mellish, editor, *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), Vol. 1*, pages 607–615, Montréal, Québec, Canada, August 1995. Morgan Kaufmann. Cited on pages: 90

[115] J. L. Higle and A. E. C. Johnson. Flight Schedule Planning with Maintenance Considerations. Submitted to OMEGA, The International Journal of Management Science, Systems and Industrial Engineering, The University of Arizona, Tucson, AZ, USA, 2004. Cited on pages: 28

[116] C. A. Hjorring. Integrating fleet and crew planning. Presentation at 41st AGIFORS Annual Symposium, Sydney, Australia, 2001. Cited on pages: 28, 219

[117] C. A. Hjorring and J. Hansen. Column generation with a rule modelling language for airline crew pairing. In *Proceedings of the 34th Annual Conference of the Operational Research Society of New Zealand*, pages 133–142, Hamilton, New Zealand, December 1999. Cited on pages: 22, 59, 64, 66, 69, 204

[118] C. A. Hjorring, S. E. Karisch, and N. Kohl. Carmen Systems' Recent Advances in Crew Scheduling. In *Proceedings of the 39th Annual AGIFORS Symposium*, pages 404–420, New Orleans, LA, USA, October 1999. Cited on pages: 22

[119] K. L. Hoffman and M. Padberg. Solving Airline Crew Scheduling Problems by Branch-and-Cut. *Management Science*, 39(6):657–682, June 1993. Cited on pages: 22, 108

[120] K. Holmberg and D. Yuan. A Multicommodity Network Flow Problem with Side Constraints on Paths Solved by Column Generation. *INFORMS Journal on Computing*, 15(1):42–57, 2003. Cited on pages: 57

[121] ILOG Inc. *ILOG CPLEX 7.5 Reference Manual*, 2001. Cited on pages: 11, 40, 57, 62, 67, 91, 93, 107, 114, 138

[122] ILOG Inc. *ILOG Solver 5.2 Reference Manual*, 2001. Cited on pages: 11, 139, 148

[123] I. Ioachim, J. Desrosiers, F. Soumis, and N. Bélanger. Fleet assignment and routing with schedule synchronization constraints. *European Journal of Operational Research*, 119:75–90, 1999. Cited on pages: 20, 26

[124] V. Jain and I. E. Grossmann. Algorithms for Hybrid MILP/CP Models for a Class of Optimization Problems. *INFORMS Journal of Computing*, 13(4), 2001. Cited on pages: 138

[125] A. I. Jarrah, J. Goodstein, and R. Narasimhan. An Efficient Airline Re-Fleeting Model for the Incremental Modification of Planned Fleet Assignments. *Transportation Science*, 34(4):349–363, November 2000. Cited on pages: 21, 38, 181

[126] A. I. Jarrah and J. C. Strehler. An optimization model for assigning through flights. *IIE Transactions*, 32(3):237–244, March 2000. Cited on pages: 40

[127] A. I. Jarrah, G. Yu, N. Krishnamurty, and A. Rakshit. A Decision Support Framework for Airline Flight Cancellations and Delays. *Transportation Science*, 27:266–280, 1993. Cited on pages: 25

[128] K. L. Jones, I. J. Lustig, J. M. Farvolden, and W. B. Powell. Multicommodity network flows: The impact of formulation on decomposition. *Mathematical Programming*, 62:95–117, 1993. Cited on pages: 53, 54

[129] U. Junker, S. E. Karisch, N. Kohl, B. Vaaben, T. Fahle, and M. Sellmann. A Framework for Constraint Programming based Column Generation. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on Principles and Practices of Constraint Programming - CP'1999, vol. 1713 of Lecture Notes in Computer Science*, pages 261–274. Springer-Verlag, 1999. Cited on pages: 90, 139

[130] N. M. Kabbani and B. W. Patty. Aircraft Routing at American Airlines. In *Proceedings of the Thirty-Second Annual Symposium of AGIFORS*, 1992. Cited on pages: 24, 42

[131] J. L. Kennington and M. Shalaby. An effective subgradient procedure for minimal cost multicommodity flow problems. *Management Science*, 23(9):994–1004, May 1977. Cited on pages: 50, 55

[132] H. Kharraziha, M. Ozana, and S. Spjuth. Large Scale Crew Rostering. Carmen Research and Technology Report CRTR-0305, Carmen Systems AB, Gothenburg, Sweden, September 2003. Cited on pages: 199

[133] E. Kilborn. Aircraft Scheduling and Operation – a Constraint Programming Approach. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2000. Cited on pages: 151, 186

[134] S. Kirkpatrick, C. D. Gelatt, Jr., and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983. Cited on pages: 58, 201

[135] J. Kjerrström. A Model and Application for the Resource Constrained Shortest Path Problem with Reset Possibilities. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2003. Cited on pages: 76

[136] D. Klabjan, E. L. Johnson, G. L. Nemhauser, E. Gelman, and S. Ramaswamy. Solving Large Airline Crew Scheduling Problems: Random Pairing Generation and Strong Branching. *Computational Optimization and Applications*, 20(1), October 2001. Cited on pages: 22

[137] D. Klabjan, E. L. Johnson, G. L. Nemhauser, E. Gelman, and S. Ramaswamy. Airline Crew Scheduling with Regularity. *Transportation Science*, 35(4), November 2001. Cited on pages: 22

[138] D. Klabjan, E. L. Johnson, G. L. Nemhauser, E. Gelman, and S. Ramaswamy. Airline Crew Scheduling with Time Windows and Plane-Count Constraints. *Transportation Science*, 36(3), August 2002. Cited on pages: 22, 27

[139] G. Kliewer. Integrating Market Modeling and Fleet Assignment. Technical report, University of Paderborn, Paderborn, Germany, 1996. Cited on pages: 20, 27

[140] T. S. Kniker and C. Barnhart. Shortcomings of the Conventional Airline Fleet Assignment Model. In *Proceedings: Tristan III*, pages 17–23, Puerto Rico, June 1998. Cited on pages: 20

[141] T. S. Kniker and C. Barnhart. Passenger Mix Problem: Models, Algorithms, and Applications. Working paper, Massachusetts Institute of Technology, Cambridge, MA, USA, 1999. Cited on pages: 20

[142] N. Kohl and S. E. Karisch. Airline Crew Rostering: Problem Types, Modeling, and Optimization. Research and Technology Report CRTR-2001-1, Carmen Systems AB, Gothenburg, Sweden, September 2001. Cited on pages: 22, 209

[143] N. Kohl, M. Kremer, and L. C. Nørrevang. Carmen Fleet - Optimization of Locomotive Scheduling. Research and Technology Report CRTR-1998-2, Carmen Systems AB, Gothenburg, Sweden, June 1998. Cited on pages: 19, 52

[144] N. Kohl, A. Larsen, J. Larsen, A. Ross, and S. Tiourine. Airline Disruption Management - Perspectives, Experiences and Outlook. Research and Technology Report CRTR-0407, Carmen Systems AB, Gothenburg, Sweden, September 2004. Cited on pages: 26

[145] N. Kohl and O. B. G. Madsen. An Optimization Algorithm for the Vehicle Routing Problem with Time Windows Based on Lagrangian Relaxation. *Operations Research*, 45(3):395–406, 1997. Cited on pages: 67

[146] A. S. Krishna, C. Barnhart, E. L. Johnson, D. Mahidara, G. L. Nemhauser, R. Rebello, A. Singh, and P. H. Vance. Advanced Techniques in Crew Scheduling. Presentation at INFORMS National Meeting, 1995. Cited on pages: 108

[147] V. Kumar. Algorithms for constraint-satisfaction problems: a survey. *AI Magazine*, 13(1):32–44, 1992. Cited on pages: 127

[148] A. H. Land and A. G. Doig. An Automatic Method for Solving Discrete Programming Problems. *Econometrica*, 28:497–520, 1960. Cited on pages: 107

[149] M. Larsson and Z. Miloloza. Investigations of the Shortest Path Problem with Resource Constraints Applied to Aircraft Scheduling. Master's thesis, Chalmers University of Technology, Gothenburg, Sweden, 2002. Cited on pages: 73

[150] T. Larsson and D. Yuan. An Augmented Lagrangian Algorithm for Large Scale Multicommodity Routing. *Computational Optimization and Applications*, 27(2):187–215, 2004. Cited on pages: 56

[151] S. Lavoie, M. Minoux, and E. Odier. A new approach for crew pairing problems by column generation with an application to air transport. *European Journal of Operational Research*, 35:45–58, 1988. Cited on pages: 22, 59, 66

[152] L. Lettovsky. *Airline Operations Recovery: An Optimization Approach.* PhD thesis, School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA, USA, 1997. Cited on pages: 25

[153] L. Lettovsky, E. L. Johnson, and G. L. Nemhauser. Airline Crew Recovery. *Transportation Science*, 34(4), November 2000. Cited on pages: 25

[154] A. Levin. Scheduling and Fleet Routing Models for Transportation Systems. *Transportation Science*, 5:232–255, 1971. Cited on pages: 31, 32, 41

[155] A. Löbel. Experiments with a Dantzig-Wolfe Decomposition for Multiple-Depot Vehicle Scheduling Problems. Preprint SC 97-16, Konrad-Zuse-Zentrum für Informationstechnik, Berlin, Germany, 1997. Cited on pages: 59

[156] M. Lohatepanont and C. Barnhart. Airline Schedule Planning: Integrated Models and Algorithms for Schedule Design and Fleet Assignment. *Transportation Science*, 38(1):19–32, February 2004. Cited on pages: 16, 20

[157] M. E. Lübbecke and J. Desrosiers. Selected Topics in Column Generation. Les Cahiers du GERAD G-2002-64, Department of Mathematical Optimization, Braunschweig University of Technology, and GERAD, 2002. Submitted to *Operations Research*. Cited on pages: 66, 67, 77, 82, 91, 94, 107, 108, 114

[158] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8(1):99–118, 1977. Cited on pages: 127, 129, 131

[159] K. Marriot and P. J. Stuckey. *Programming with Constraints: an Introduction*. MIT Press, 1998. Cited on pages: 127, 138

[160] R. E. Marsten. Crew Planning at Delta Air Lines. Presentation at Mathematical Programming Symposium XV, Ann Arbor, MI, USA, 1994. Cited on pages: 108

[161] R. E. Marsten, W. W. Hogan, and J. W. Blankenship. The `Boxstep` method for large-scale optimization. *Operations Research*, 23:389–405, 1975. Cited on pages: 94

[162] R. E. Marsten, R. Subramanian, and S. Martin. RALPH: Crew Planning at Delta Air Lines. To appear in *Interfaces*. Cited on pages: 22

[163] E. Q. V. Martins and J. L. E. dos Santos. The labelling algorithm for the multiobjective shortest path problem. Technical report, Departamento de Matemática, Universidade de Coimbra, Coimbra, Portugal, November 1999. Cited on pages: 72

[164] R. D. McBride. Solving embedded generalized network problems. *European Journal of Operational Research*, 21:82–92, 1985. Cited on pages: 56

[165] R. D. McBride. Advances in Solving the Multicommodity-Flow Problem. *Interfaces*, 28(2):32–41, March–April 1998. Cited on pages: 50, 51

[166] R. D. McBride. Progress made in solving the multicommodity flow problem. *SIAM Journal on Optimization*, 8(4):947–955, November 1998. Cited on pages: 56

[167] R. D. McBride and J. W. Mamer. Solving Multicommodity Flow Problems with a Primal Embedded Network Simplex Algorithm. *INFORMS Journal on Computing*, 9:154–163, 1997. Cited on pages: 55

[168] J. I. McGill and G. J. van Ryzin. Revenue Management: Research Overview and Prospects. *Transportation Science*, 33(2):233–256, May 1999. Cited on pages: 18

[169] C. P. Medard and N. Sawhney. Airline Crew Scheduling: From Planning to Operations. Carmen Research and Technology Report CRTR-0406, Carmen Systems AB, Gothenburg, Sweden, June 2004. Cited on pages: 25

[170] A. Mercier, J.-F. Cordeau, and F. Soumis. A Computational Study of Benders Decomposition for the Integrated Aircraft Routing and Crew Scheduling Problem. Les Cahiers du GERAD G-2003-48, GERAD, October 2003. Cited on pages: 26, 219

[171] M. Milano, G. Ottosson, P. Refalo, and E. S. Thorsteinsson. The Role of Integer Programming Techniques in Constraint Programming's Global Constraints. *INFORMS Journal on Computing, Special Issue on "The Merging of Mathematical Programming and Constraint Programming"*, 14(4), April 2002. Cited on pages: 36, 133, 138

[172] A. Mingozzi, M. A. Boschetti, S. Ricciardelli, and L. Bianco. A Set Partitioning Approach to the Crew Scheduling Problem. *Operations Research*, 47(6):873–888, November-December 1999. Cited on pages: 22

[173] R. Mohr and T. C. Henderson. Arc and path consistency revised. *Artificial Intelligence*, 28:225–233, 1986. Cited on pages: 131

[174] U. Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information Science*, 7(66):95–132, 1974. Cited on pages: 127, 128

[175] K. G. Murty. *Linear Programming*. John Wiley & Sons, Inc., 1983. Cited on pages: 55, 64, 65

[176] G. Ottosson. *Integration of Constraint Programming and Integer Programming for Combinatorial Optimization*. PhD thesis, Uppsala University, Uppsala, Sweden, 2000. Cited on pages: 7, 138

[177] G. Pesant, M. Gendreau, J.-Y. Potvin, and J.-M. Rousseau. An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. *Transportation Science*, 32(1):12–29, 1998. Cited on pages: 147

[178] M. Ç. Pinar and S. A. Zenios. Parallel decomposition of multicommodity network flows using linear-quadratic penalty functions. *ORSA Journal on Computing*, 4:235–249, 1994. Cited on pages: 56

[179] S. S. Pulugurtha and S. S. Nambisan. A decision-Support Tool for Airline Yield Management Using Genetic Algorithms. *Computer-Aided Civil and Infrastructure Engineering*, 18:214–223, 2003. Cited on pages: 18

[180] A. Rakshit, N. Krishnamurty, and G. Yu. System Operations Advisor: A Real-Time Decision Support System for Managing Airline Operations at United Airlines. *Interfaces*, 26(2):50–58, March-April 1996. Cited on pages: 25

[181] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proceedings of AAAI-94*, pages 362–367, 1994. Cited on pages: 133, 139, 150, 151, 173

[182] J.-C. Régin. Generalized Arc Consistency for Global Cardinality Constraint. In *Proceedings of AAAI-96*, pages 209–215, 1996. Cited on pages: 133, 139, 178

[183] J.-C. Régin. Arc Consistency for Global Cardinality Constraints with costs. In J. Jaffar, editor, *Proceedings of the Fifth International Conference on the Principles and Practice of Constraint Programming - CP'1999, vol. 1713 of Lecture Notes in Computer Science*. Springer-Verlag, 1999. Cited on pages: 139, 178

[184] J.-C. Régin and M. Rueher. A global constraint combining a sum constraint and difference constraints. In R. Dechter, editor, *Proceedings of the Sixth International Conference on the Principles and Practice of Constraint Programming - CP'2000, vol. 1894 of Lecture Notes in Computer Science*, pages 384–395. Springer-Verlag, 2000. Cited on pages: 133

[185] B. Rexing, C. Barnhart, T. S. Kniker, A. I. Jarrah, and N. Krishnamurty. Airline Fleet Assignment with Time Windows. *Transportation Science*, 34(1):1–20, February 2000. Cited on pages: 20, 31

[186] R. Rodošek, M. G. Wallace, and M. T. Hajian. A New Approach to Integrating Mixed Integer Programming and Constraint Logic Programming. *Annals of Operations Research*, 86:63–87, 1999. Cited on pages: 7, 138

[187] J. M. Rosenberger. *Topics in Airline Operations.* PhD thesis, School of Industrial & Systems Engineering, Georgia Institute of Technology, Atlanta, GA, USA, 2002. Cited on pages: 26

[188] J. M. Rosenberger, E. L. Johnson, and G. L. Nemhauser. A Robust Fleet-Assignment Model with Hub Isolation and Short Cycles. *Transportation Science*, 38:3, August 2004. Cited on pages: 21, 216

[189] J. M. Rosenberger, A. J. Schaefer, D. Goldsman, E. L. Johnson, A. J. Kleywegt, and G. L. Nemhauser. A Stochastic Model of Airline Operations. *Transportation Science*, 36(4):357–377, November 2002. Cited on pages: 216

[190] L.-M. Rousseau. Stabilization Issues for Constraint Programming Based Column Generation. In *Proceedings of CPAIOR'04, vol. 3011 of Lecture Notes in Computer Science*, pages 402–408. Springer-Verlag, April 2004. Cited on pages: 90

[191] L.-M. Rousseau, M. Gendreau, and D. Feillet. Interior Point Stabilization for Column Generation. Technical report, submitted to *Operations Research Letters*, Centre de recherche sur les transports, Université de Montréal, Canada, 2003. Cited on pages: 66, 90, 91, 95

[192] L.-M. Rousseau, M. Gendreau, and G. Pesant. Solving small VRPTWs with Constraint Programming Based Column Generation. In *Proceedings of CPAIOR'02*, March 2002. Cited on pages: 66, 90, 139, 148

[193] R. A. Rushmeier, K. L. Hoffman, and M. Padberg. Recent Advances in Exact Optimization of Airline Scheduling Problems. Technical report, USAir Operations Research Department, July 1995. Cited on pages: 31

[194] R. A. Rushmeier and S. A. Kontogiorgis. Advances in the Optimization of Airline Fleet Assignment. *Transportation Science*, 31(2):159–169, May 1997. Cited on pages: 19, 27

[195] R. Sandhu and D. Klabjan. Integrated Airline Planning. Working paper, submitted for publication, Department of Mechanical and Industrial Engineering, University of Illinois at Urbana-Campaign, Urbana, IL, USA, 2004. Cited on pages: 27

[196] A. Sarac, R. Batta, and C. M. Rump. A Branch-and-Price Approach for Operational Aircraft Maintenance Routing. Working Paper, Department of Industrial Engineering, University of Buffalo, Buffalo, NY, USA, December 2003. Cited on pages: 24, 43

[197] A. J. Schaefer, E. L. Johnson, A. J. Kleywegt, and G. L. Nemhauser. Airline Crew Scheduling under Uncertainty. Technical report, Technical Report, Georgia Institute of Technology, Atlanta, GA, USA, May 2001. Cited on pages: 216

[198] R. R. Schneur and J. B. Orlin. A Scaling Algorithm for Multicommodity Flow Problems. *Operations Research*, 46(2):231–247, March–April 1998. Cited on pages: 56

[199] A. Schrijver, editor. *Combinatorial Optimization : Polyhedra and Efficiency*. Springer-Verlag, 2003. Cited on pages: 49, 50

[200] M. Sellmann, K. Zervoudakis, P. Stamatopoulos, and T. Fahle. Crew Assignment via Constraint Programming: Integrating Column Generation and Heuristic Tree Search. *Annals of Operations Research*, 115:207–225, 2002. Cited on pages: 66

[201] M. Sigurd and D. M. Ryan. Stabilized Column Generation for Set Partitioning Optimization. Technical report, 2004. Cited on pages: 94, 95

[202] R. Simons. Aircraft Swapping by Constraint Logic Programming. Technical report, MP in Action, Eudoxus Systems Ltd., Leighton Buzzard, UK, August 1996. Cited on pages: 21, 147

[203] B. C. Smith, J. F. Leimkuhler, and R. M. Darrow. Yield Management at American Airlines. *Interfaces*, 22(1):8–32, January-February 1992. Cited on pages: 18

[204] C. Sriram and A. Haghani. An optimization model for aircraft maintenance scheduling and re-assignment. *Transportation Research Part A: Policy and Practice*, 37(1):29–48, January 2003. Cited on pages: 41

[205] M. Stojković, F. Soumis, and J. Desrosiers. The Operational Airline Crew Scheduling Problem. *Transportation Science*, 32(3):232–245, August 1998. Cited on pages: 25

[206] J. Subramanian, S. Stidham, Jr., and C. J. Lautenbacher. Airline Yield Management with Overbooking, Cancellations, and No-Shows. *Transportation Science*, 33(2):147–167, May 1999. Cited on pages: 18

[207] R. Subramanian, R. P. Scheff, Jr., J. D. Quillinan, D. S. Wiper, and R. E. Marsten. Coldstart: Fleet Assignment at Delta Air Lines. *Interfaces*, 24(1):104–120, January-February 1994. Cited on pages: 18, 19, 27, 30, 31, 39

[208] I. Sutherland. Sketchpad: A man-machine graphical communication system. In *Proceedings of the Spring Joint Computer Conference*, pages 329–346. IFIPS, 1963. Cited on pages: 127

[209] K. T. Talluri. Swapping Applications in a Daily Fleet Assignment. *Transportation Science*, 30:237–248, 1996. Cited on pages: 21, 40

[210] K. T. Talluri. The Four-Day Aircraft Maintenance Problem. *Transportation Science*, 32:43–53, 1998. Cited on pages: 34, 41

[211] E. Tardos. A strongly polynomial minimum cost circulation algorithm. *Combinatorica*, 5(3):247–255, 1985. Cited on pages: 51, 61

[212] E. Tardos. A strongly polynomial algorithm to solve combinatorial linear programs. *Operations Research*, 34(2):250–256, 1986. Cited on pages: 50

[213] D. Teodorović and S. Guberinić. Optimal Dispatching Strategy on an Airline Network After a Schedule Perturbation. *European Journal of Operational Research*, 15:178–182, 1984. Cited on pages: 25

[214] J. A. Tomlin. Minimum-Cost Multicommodity Network Flows. *Operations Research*, 14:45–51, 1966. Cited on pages: 54

[215] J. M. Valério de Carvalhò. Using extra dual cuts to accelerate column generation. Technical report, Dept. Produção e Sistemas, Universidade do Minho, 4710-057 Braga, Portugal, 2000. Cited on pages: 66

[216] P. Van Hentenryck, Y. Deville, and C.-M. Teng. A Generic Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57(2-3):291–321, 1992. Cited on pages: 131

[217] G. Van Rossum and F. L. Drake, Jr. *The Python Language Reference Manual*. Network Theory Ltd., 2003. Cited on pages: 212

[218] F. Vanderbeck. *Decomposition and Column Generation for Integer Programs*. PhD thesis, Université Catholique do Lovain, 1994. Cited on pages: 57, 82, 83, 84, 107, 108, 114

[219] B. Verweij, K. Aardal, and G. Kant. On an Integer Multicommodity Flow Problem from the Airplane Industry. Technical Report UU-CS-1997-38, Department of Computer Science, Utrecht University, the Netherlands, 1997. Cited on pages: 57

[220] C. Voudouris. *Guided Local Search for Combinatorial Optimisation Problems*. PhD thesis, Department of Computer Science, University of Essex, UK, 1997. Cited on pages: 148

[221] D. L. Waltz. Generating semantic descriptions from drawings of scenes with shadow. In P. H. Winston, editor, *The Psychology of Computer Vision*. McGraw-Hill, 1975. Cited on pages: 127

[222] A. Warburton. Approximation of Pareto Optima in Multiple-Objective, Shortest-Path Problems. *Operations Research*, 35(1):70–76, January–February 1987. Cited on pages: 50

[223] D. Wedelin. An algorithm for large scale 0-1 integer programming with application to airline crew scheduling. *Annals of Operations Research*, 57:283–301, 1994. Cited on pages: 22, 138

[224] D. Wedelin. Cost propagation: a generalization of constraint propagation for optimization problems. In *Proceedings of CPAIOR'02*, March 2002. Cited on pages: 138

[225] T. J. Winterer. *Requested Resource Reallocation with Retiming: An Algorithm for Finding Non-Dominated Solutions with Minimal Changes.* PhD thesis, Centre for Planning and Resource Control (IC-PARC), Imperial College London, UK, 2004. Cited on pages: 21

[226] R. D. Wollmer. Multicommodity Network Flows with Resource Constraints: The Generalized Multicommodity Flow Problem. *Networks*, 1(3):245–263, 1972. Cited on pages: 57

[227] M. K. Wood and G. B. Dantzig. Programming of Interdependent Activities 2. General Discussion. *Econometrica*, 17(3 & 4):193–199, 1949. Cited on pages: 65

[228] S. Yan and C.-H. Tseng. A Passenger Demand Model for Airline Flight Scheduling and Fleet Routing. *Computers & Operations Research*, 29:1559–1581, 2002. Cited on pages: 27

[229] G. Yu, M. F. Argüello, G. Song, S. M. McCowan, and A. White. A New Era for Crew Recovery at Continental Airlines. *Interfaces*, 33(1):5–22, January-February 2003. Cited on pages: 25

Index of Cited Authors

Numbers refer to indices in the bibliography. Bold numbers refer to entries for which the author is mentioned first in the list of authors.

Index

Typeset with LaTeX 2_ε, using Peter Wilson's memoir document class with chapterstyle demo, and packages algpseudocode, amsmath, amsthm, authorindex, enumerate, eurosym, fixltx2e, float, graphicx, hyperref, multicol, multirow and url.