View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by DSpace@MIT

APPROACHES TO INCORPORATING ROBUSTNESS INTO AIRLINE SCHEDULING

Yanina V. Ageeva and John-Paul Clarke

Report No. ICAT-2000-8
August 2000

MIT International Center for Air Transportation
Department of Aeronautics & Astronautics
Massachusetts Institute of Technology
Cambridge, MA 02139 USA

Approaches to Incorporating Robustness into Airline Scheduling

by

Yana Ageeva

Submitted to the Department of Electrical Engineering and Computer Science
on August 31, 2000, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science

Abstract

The airline scheduling process used by major airlines today aims to develop optimal schedules which maximize revenue. However, these schedules are often far from "optimal" once deployed in the real world because they do not accurately take into account possible weather, air traffic control (ATC), and other disruptions that can occur during operation. The resulting flight delays and cancellations can cause significant revenue loss, not to mention service disruptions and customer dissatisfaction. A novel approach to addressing this problem is to design schedules that are robust to schedule disruptions and can be degraded at any airport location or in any region with minimal impact on the entire schedule. This research project suggests new methods for creating more robust airline schedules which can be easily recovered in the face of irregular operations. We show how to create multiple optimal solutions to the Aircraft Routing problem and suggest how to evaluate robustness of those solutions. Other potential methods for increasing robustness of airline schedules are reviewed.

Thesis Supervisor: John-Paul Clarke
Title: Assistant Professor

Acknowledgments

I would like to thank my thesis advisor, John-Paul Clarke, without whose ideas, constant encouragement, support, understanding and patience this work would never have been possible. Thanks for inspiring me, J-P, and for providing me with an opportunity I would never have gotten anywhere else. I would also like to thank Michael Clarke and Milorad Sucur of the Research group at Sabre, who provided me with advice and suggestions. In addition, I thank Professor Cynthia Barnhart and Manoj Lohatepanont for teaching me Airline Scheduling, and also Professors Arnold Barnett and Richard Larson for inspiring my interest in Operations Research in the first place.

My love, deep respect and appreciation go to Edwin and Jeffrey. I was lucky to have my friends' support with everything – from helping me with my writing to offering suggestions, ideas, and help with technical issues, to giving amazingly strong emotional support, let alone enhancing my ramen and caffeine diet with other nutrients. :-) Thanks for everything, dear friends – I could not have possibly done this without you. Thanks for always being there for me.

And, of course, this work would have been much harder without other friends who kept me sane (well, for some definition of sane, anyway), drank coffee with me and constantly encouraged me. I would like to thank my officemate, Terran, for all his help and for getting me out of trouble more than once. :-) Thanks, Andrew, for providing me with good advice and for caring. Thanks, Scott and Steven, for your support.

Contents

1 Introduction **10**

1.1 The Problem 10

1.2 Thesis Overview 11

2 Background **13**

2.1 The Basics of Airline Operations 13

2.1.1 Overview 13

2.1.2 The Schedule Planning Process 14

2.1.3 Schedule Construction and Coordination 16

2.1.4 Fleet Assignment 19

2.1.5 Aircraft Maintenance Routing 21

2.1.6 Crew Scheduling 22

2.1.7 Yield Management 23

2.2 Irregular Airline Operations 24

2.2.1 Airline Operations Control Centers 24

2.2.2 Irregular Airline Operations 25

2.3 Previous Work on Irregular Operations 26

2.3.1 The Airline Schedule Recovery Problem 26

2.3.2 Analyzing Robustness of a Prospective Schedule 27

3 The Flight String Model for Aircraft Maintenance Routing **29**

3.1 Overview 29

3.2 Notations 30

3.2.1 Decision Variables 30
3.2.2 Parameters 31
3.2.3 Sets 31
3.3 Aircraft Routing String Model Formulation 31
3.3.1 Objective Function 31
3.3.2 Constraints 31
3.4 String-Based Model Solution 33
3.4.1 Linear Program Solution 33
3.4.2 Pricing Subproblem for the String Model 34

4 Robust Airline Scheduling **35**
4.1 The Concept of Robustness. Fault Tolerant Recovery Paths 35
4.2 Examples. Sequences. Points. Overlaps 36
4.2.1 Example 36
4.2.2 Points. Sequences. Overlaps 37
4.3 Tradeoff Between Robustness and Optimality 42
4.4 Building More Robust Schedules 42
4.4.1 Optimized Turn Times 43
4.4.2 Flexible Subroute Switching 43
4.4.3 Passenger Routing Redundancy 43
4.5 Methods for Incorporating Robustness into
Aircraft Routing. Subroute Switching 44
4.5.1 Incorporating a Measure of Robustness into the Objective Function 44
4.5.2 Computing Alternative Optimal Solutions and Selecting the Most Robust One 46

5 Chapter 5: Building Robustness into the String Model **48**
5.1 Overview of the Implementation 48
5.2 OPL Optimization Programming Language 48
5.2.1 What it is 48

5.2.2 Advantages of Using OPL 50
5.3 Implementing the Modified Base Model 50
5.3.1 Overview 50
5.3.2 Generating Initial Set of Routings 52
5.3.3 How Sequences are Stored in OPL model 56
5.3.4 Restricted Master Problem. Solving the LP 57
5.3.5 The Pricing Subproblem 59
5.3.6 Problems with OPL 61
5.4 Making the String Model Robust 61
5.4.1 Obtaining Alternative Solutions by Adding Constraints 63
5.4.2 Removing Multiple Copies (Perl) 64
5.4.3 C++ code 64
5.4.4 Algorithm for Finding Overlapping Sequences 64
5.4.5 Avoiding Complexity. Sequence Storage Model. 66
5.4.6 Data structures. Objects. Sequence Storage. 66

6 Results and Conclusions 71

6.1 Overview 71
6.2 Test Data 71
6.3 Robustness Factors 72
6.3.1 Number of Alternative Solutions 72
6.3.2 Robustness Time Tolerance 73
6.4 Results 73
6.4.1 Improvement 74
6.4.2 Network 1: 14 Flights 74
6.4.3 Network 2: 22 Flights 77
6.4.4 Network 3: 26 Flights 79
6.4.5 Network 4: 37 Flights 81
6.5 Conclusions 84
6.5.1 Problems and Future Model Improvement 84

7 Future Research **86**

7.1 Robustness Measure as Part of LP's Objective Function 86

7.2 Robustness as Part of the Pricing Subproblem 87

7.3 Through Flights 87

7.4 Hybrid Airline Scheduling and Robustness 87

7.5 Robust Crew Scheduling 88

7.6 Estimating Performance of a Schedule Based on Historical Data . . . 88

7.7 Robust Airline Schedules in Practice 88

7.7.1 Possible Geometrical Representation of a Robust Schedule 89

7.8 Social Aspects of Airline Scheduling 90

7.8.1 Customer Acceptance 90

7.8.2 Other Related Questions 90

A Mathematical Tools **91**

A.1 Mathematical Programming 91

A.2 Linear Programs (LP). Integer Programs(IP) 91

A.3 SIMPLEX 92

A.3.1 Pricing 92

A.3.2 Pivot Selection 92

A.4 CPLEX 92

List of Figures

2-1 Steps of the Scheduling Process 15

2-2 Hub and Spoke Networks 21

4-1 Example 1: Robust Schedules 36

4-2 Flight 37

4-3 Sequence 38

4-4 Sequences, Overlaps, and Robustness 40

5-1 Flight String Model for Aircraft Routing. Solution Steps 51

5-2 Building Robustness into the String Model. Solution Steps 62

5-3 Sequence Store Design 67

6-1 Robustness Distribution (14 flights, 500 solutions) 76

6-2 Robustness Distribution (22 flights, 500 solutions) 78

6-3 Robustness Distribution (26 flights, 500 solutions) 81

6-4 Robustness Distribution (37 flights, 500 solutions) 83

7-1 Representing Robust Schedules Geometrically 89

List of Tables

6.1 Test Data 72
6.2 Results: Minimum Robustness Coefficient(14 flights) 75
6.3 Results: Maximum Robustness Coefficient(14 flights) 75
6.4 Results: Average Robustness Coefficient(14 flights) 75
6.5 Results: Improvement (14 flights, 54 solutions) 75
6.6 Results: Minimum Robustness Coefficient (22 flights) 77
6.7 Results: Maximum Robustness Coefficient (22 flights) 77
6.8 Results: Average Robustness Coefficient (22 flights) 79
6.9 Results: Improvement (22 flights, 260 solutions) 79
6.10 Results: Minimum Robustness Coefficient (26 flights) 80
6.11 Results: Maximum Robustness Coefficient (26 flights) 80
6.12 Results: Average Robustness Coefficient (26 flights) 80
6.13 Results: Improvement (26 flights, 376 solutions) 80
6.14 Results: Minimum Robustness Coefficient (37 flights) 82
6.15 Results: Maximum Robustness Coefficient (37 flights) 82
6.16 Results: Average Robustness Coefficient (37 flights) 82
6.17 Results: Improvement (37 flights, solutions) 82

Chapter 1

Introduction

1.1 The Problem

The airline industry has seen strong growth in passenger traffic over the last few years, supported by a strong economy, airline deregulation, and an increasingly mobile population. This growth has come at a price, however, as current airline operations systems have not been able to keep up. The resulting flight delays and cancellations have made headlines this past year as irate travelers, frustrated airline executives, and even Congress have searched for ways to improve airline performance.

Presently airlines use scheduling systems that create optimal (i.e. revenue maximizing) schedules based on no disruptions or irregularities in operations. These schedules, however, are often far from "optimal" once deployed in the real world because they do not take into consideration many of the possible weather and air traffic control (ATC) delays that occur during operation, except through the addition of increased time buffers in both block times and turn times. In the face of weather/ATC delays airlines are unable to reschedule flights efficiently. For a lot of airlines lack of robust real-time decision making tools means that rescheduling of flights in the aftermath of irregularities has been a manual process performed by operation controllers.

Constantly increasing traffic volume and lack of efficient decision support systems for coping with operational irregularities have resulted in an increasing number of customer complaints and a significant revenue drop for airlines. The volume of com-

plaints in 1998 represents a 26% increase over 1997 [3]. For a major U.S. domestic carrier the financial impact of daily irregularities in operations can exceed $400 million per year in lost revenue, crew overtime pay, and passenger hospitality costs [7]. In fact, severe weather conditions and the associated loss of airport capacity, coupled with increased passenger traffic have caused the typical major U.S. airline to lose an estimated ten percent of its expected revenue based on the optimal schedule achieved during the strategic phase of the airline scheduling process.

Clearly, then, the so-called "optimal" schedules used today by commercial airlines are far from optimal in practice. One approach to resolving this dilemma is to develop real-time algorithms to re-optimize the schedule after irregularities occur. Another approach is to build robustness into the schedule when it is being developed so that schedule adaptations can be more easily made.

1.2 Thesis Overview

This thesis seeks to

1. develop methods for incorporating operational considerations into the schedule planning process

2. determine whether optimal (i.e. revenue maximizing) airline scheduling solutions can be made more robust

3. evaluate the trade-off between optimality and robustness

Chapter 2 describes the basics of airline operations and reviews some previous efforts in managing irregularities.

Chapter 3 describes the flight string model for aircraft maintenance routing, which serves as the base model for a more robust routing model introduced in the later chapters.

Chapter 4 defines the concept of robustness, discusses ways to build more robust airline schedules, and suggests methods for incorporating a measure of robustness into

the scheduling process. The chapter also explains the trade-off between optimality and robustness.

Chapter 5 discusses a new aircraft routing model, based on the flight string model, which attempts to create more robust routing schedules. The model, details of its implementation, and tools used to solve the model are also reviewed.

Chapter 6 outlines the results of this research project, explains how the model was tested, what we expected to find and what we ended up finding as a result of our work. We also discuss some of the limitations of the tools used to implement the model and point out how results could have been different if we had access to other, more powerful tools.

Chapter 7 discusses possible directions for future research in the field. We summarize several ideas that were considered as part of this research work, but were not implemented due to insufficient time and/or resources, as well as a number of ideas that were born in the process. We believe that developing these ideas in the future may lead to improved airline schedules and will therefore cause increase in revenue for airlines.

Appendix A defines several mathematical terms and tools used by operations researchers to design airline scheduling systems.

Chapter 2

Background

This chapter discusses the basics of airline operations in general, as well as steps that airlines currently take to manage irregular operations in particular. In both cases the focus of discussion is the scheduling aspect. After reviewing previous efforts in operations research that may prove useful in building a robust airline scheduling system, two specific approaches to account for irregularities in planned airline operations are compared. One approach, discussed in section 2.3.1, is to develop real-time algorithms to re-optimize the schedule after irregularities occur. Another, discussed in section 2.3.2, is to build robustness into the schedule when it is being developed by analyzing historical data and using it to predict the likelihood of certain irregular events occurring in future operations.

2.1 The Basics of Airline Operations

2.1.1 Overview

Airline schedule planning is a complex process that takes into consideration a number of different factors: equipment maintenance, crews, facilities, marketing, as well as seasonal considerations. Alexander Wells, a researcher in the field of airline operations, describes the airline schedule planning process as an attempt to "put together a jigsaw puzzle, constructed in three dimensions, while the shape of key pieces is

constantly changing." [12]

Airlines rely on a variety of computer technologies in conjunction with operations research (OR) and artificial intelligence (AI) based decision support systems. Airlines that make first use of new technologies end up developing significant competitive advantages over the rest of the industry through increased productivity, reduced costs, and improved profitability. Recent advances in operations research, specifically mathematical programming, as well as in raw computing power, have made it possible for airlines to solve problems that were considered intractable several years ago [7].

In addition to technological advances, new opportunities for improvement have opened due to the deregulation of US domestic market. For example, one important consequence of deregulation was the development of so called *hub and spoke* networks. Hub airports are used by airlines to transfer passengers traveling from one community to another in the area surrounding the airport. They also serve as transfer points between local communities and international or other domestic destinations. Hub and spoke networks have provided airlines with an opportunity to better manage their limited resources such as aircraft and crew members. At the same time, use of hubs has made aircraft routing and crew scheduling more complex because of an increased number of possible feasible solutions.

2.1.2 The Schedule Planning Process

The airline schedule planning process can be represented by a sequence of distinct steps (see Figure 2-1), where the output of a given step is fed as an input into subsequent steps. The scheduling process begins with schedule construction, a process concerned with generating a feasible plan for which cities to fly to and at what times. After the schedule is fixed, the airline has to decide what aircraft type (767 or 737, for example) will be assigned to each flight, a process known as *fleet planning*. Fleet planning is followed by *aircraft maintenance routing* – assigning a specific aircraft (*tail number*) in the airline's fleet to each flight. This assignment basically represents aircraft routing taking into consideration that each aircraft has to be able to undergo planned periodic maintenance at certain stations and at a certain frequency. Both

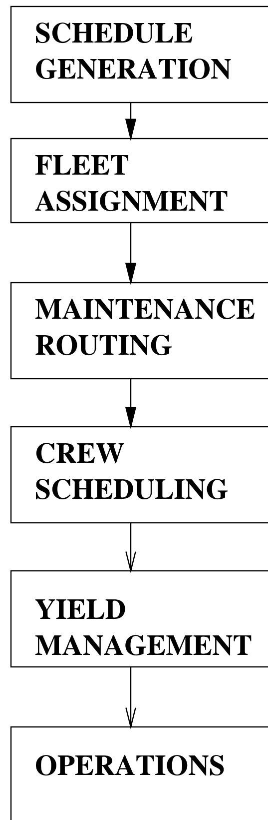


Figure 2-1: Steps of the Scheduling Process

fleet assignment and maintenance routing manage the airline's equipment. *Crew scheduling*, on the other hand, involves deciding who flies the equipment, i.e. assigning specific crew (pilots and cabin crew) to the schedule. *Revenue (Yield) Management* maximizes revenue by selectively accepting and rejecting reservation requests based on their relative value [2].

Historically each of the above steps of the scheduling process was treated as an independent problem. However, in the recent years there has been an attempt to combine some of these phases of the scheduling process, a process frequently referred to as *hybrid airline scheduling*. A joint formulation and optimization framework can result in higher revenues. However, limitations of currently available computer hardware and optimization algorithms prevent the problem from being solved without breaking it up into sequential stages and solving each of them separately. As technology and

computational power improve, hybrid airline scheduling will become more and more common.

With all of this effort given to planning an efficient schedule, one should not forget that implementing that schedule is an equally important task. This phase is known as *Operations* phase. The factors that make airline operations difficult include bad weather conditions, ATC delays, crew unavailability, and mechanical problems. *Irregular operations* are operations affected by any of these factors, and it is the responsibility of the airline's operations controllers to run operations as close to plan as possible, even in light of irregularities. Flights are interlinked very tightly in a schedule, and a small change in the arrival or departure times of a flight can have a large effect on later flights throughout the system. If a particular aircraft is experiencing a mechanical breakdown, it might be better to cancel some other flight and switch the aircraft to minimize revenue losses. If a flight is delayed, it may be necessary to purposely delay connecting flights to make sure that the delayed passengers can get to their destination. Unfortunately, the process shown in Figure 2-1 provides no feedback mechanism. Once a schedule has been designed and is in operation, changes are usually made locally and often manually. Such changes may improve a small part of the schedule, but often result in even larger disruptions in the rest of the schedule.

There is an increasing need to provide a feedback mechanism between the schedule design process and operations. The schedule design process should take into consideration possible disruptions in operations to ensure that changes to the schedule during operations are minimal, and that opportunities for efficient changes in fact exist.

Irregular airline operations are discussed in more detail in section 2.2. Detailed explanations of each of the steps of the airline scheduling process are presented below.

2.1.3 Schedule Construction and Coordination

A scheduling system must take into consideration a variety of factors. The goal is to come up with a schedule that meets the desirable objectives while satisfying all the limiting constraints. For example, to guarantee maximum profit it is desirable to

keep aircraft in the air as much as possible. However, at the same time, enough time must be provided on the ground for maintenance and servicing of aircraft.

Barnhart [2] defines a schedule as a "list comprising four things: origin city, destination city, time of departure (which also roughly determines the time of arrival) and flight frequency." Flight frequency is defined as the days of the week when this particular flight is offered – i.e., Monday through Friday, or week-ends only.

No airline schedule is static, of course. For example, schedules are affected by seasonal changes. There are other aspects that make schedules highly unpredictable over the long term. The schedule affects every operational decision and has a big impact on the profitability of an airline. It defines the airline's market and therefore defines the airline's strengths as well as future plans. While every airline wants to defend its traditional profit-making markets where it has a large market share, it may also want to improve its market share somewhere else by increasing the frequency to a competitor's hub. However, in the short run – over a few months at least – schedules can be kept relatively stable.

A schedule is usually generated several months before deployment date. A number of factors are taken into consideration, including but not limited to the following [12]:

1. *Size and Composition of the Airline's Fleet.* The frequency of service and the number of markets that the airline can maintain are restricted by the types of aircraft in the airline's fleet. Not all types of aircraft can fly between all pairs of cities. A 737, for example, can not fly between Boston and Honolulu nonstop.

2. *Slot-constrained Airports.* Some airports, including Chicago O'Hare and JFK, are slot-constrained. It can be very expensive for an airline to acquire additional slots at such airports.

3. *Bilateral agreements.* International flights are guided by bilateral agreements which specify which airlines can fly where.

4. *Traffic Flow.* Traffic flow varies from city to city, depending on geography, route structure, and alternative service. Some cities, because of favorable geography,

get high traffic flow. However, even for a particular city traffic flow might vary from time to time – if the city gets a lot of its traffic because it is a connecting point between two major destinations, an increase in the number of nonstop flights between those destinations will lead to a decrease in traffic flow for the city in between.

5. *Schedule Salability.* In the airline industry schedule salability is highly sensitive to even minor changes in departure/arrival times. With strong competition imposed by other airlines, a 15 minute difference in departure time can lead to millions of dollars in lost revenue. Salability also varies by route and direction of flight on the same route, as well as airport of departure/destination. Factors like accessibility and favorable location can make an airport more popular than other airports that serve the same city.

6. *Time Zones.* The time zone effect has to do with the fact that one gains three hours on westbound flights and loses three hours going eastbound. Passengers normally do not like arriving at their destination after 11pm which means that someone traveling from San Francisco to Boston will want to leave no later than 3pm. Many people will prefer a "red-eye" flight to one that gets them to their destination after midnight.

7. *Noise Considerations.* Departures and arrivals are rarely scheduled between 11pm and 7am due to high opposition from airport communities.

8. *Station Personnel.* The peaking of personnel and ground equipment need to be minimized. If two flights can be scheduled in such a way that only one jet ground crew is needed to service the planes, then the airline can save the work of 10-12 people that would be required if another ground crew were necessary.

9. *Equipment Turnaround Time.* At the end of every trip certain operations must be performed – refueling, cabin cleaning, catering services. There are established standards for turn time required of aircraft depending on the type of aircraft

and flight length. Again, extra time should be allowed to accommodate late arrivals/departures.

10. *Load Factor Leverage.* Costs of operating a schedule vary only slightly as load factor changes. However, even a slight change in load factor will result in direct proportional change in revenue. Thus, it is even more important to consider possible load factor changes when making even small adjustments to an existing schedule.

Unfortunately, at the present time no airline uses a model that captures all of the above factors. This is partially the case because formulating many of these factors mathematically can be difficult. For example, it is difficult to design and solve a model that captures competitors' moves and strategies.

2.1.4 Fleet Assignment

The fleet assignment process seeks to assign an aircraft type to each flight segment in a given flight schedule. This selection is based on factors such as revenue and operating cost, as well as operational constraints.

Some of the obvious operational constraints in fleet assignment are airport runway lengths and aircraft fuel capacity – the router must ensure that the types of aircraft scheduled to use a certain airport can use the runways as well as have enough fuel capacity to get to the next destination. Also, weather patterns can be an important consideration. Cold weather patterns in a northern city can make it inadvisable to overnight aircraft in that city – early morning departures might become impractical due to the need to remove the snow and ice from the aircraft. Other factors include ground operations and facility limitations. Ground service can not be arranged in a random fashion. Limitations exist on gate positions, ground equipment, passenger service facilities, and personnel. The goal of ground service is to accommodate as many flights as possible as efficiently as possible, given physical limitations and aiming for maximum utilization of personnel and equipment. For example, the schedule planner must make sure that there are enough gate positions for the number of planes

on the ground simultaneously, including possible early and late departures. Sufficient time has to be provided for passenger and baggage transfer. Also, enough ticket-counter space must be provided for efficient passenger check-in.

Presently there are several distinct ways that airlines schedule flights. Among the most popular ones are *skip-stop* scheduling, *local service*, *nonstops*, and *cross-connections (hub and spoke)*. A brief description of each of them is presented below [12]:

1. *Skip-stop Schedules* Skip-stop schedules are schedules that provide service to a number of stations A_1, A_2, $\ldots$, A_n, by scheduling flights $A_1->A_3->A_5->A_n$ and $A_2->A_4->A_6->A_{n-1}$ (as a hypothetical example). In this type of schedule one or more of intermediate stops are "skipped" and service to those stations is provided by another flight. This way fast service is provided between intermediate stations (A_1 and A_3, for example). On the other hand, no service is provided between consecutive cities (A_1 and A_2, for example).

2. *Local Service* This type of service is performed by short-range aircraft providing service between all consecutive points on a segment. Connections to long-range aircraft are provided at some of the intermediate stations.

3. *Nonstops* Nonstops provide fast service between end points. However, they do not service intermediate cities.

4. *Cross-connections (Hub and Spoke Networks)* One important consequence of deregulation in airline industry was the creation of hub and spoke networks, which has become perhaps the most popular type of airline scheduling. In this type of scheduling several points of departure are fed into a single hub airport, from which connecting flights carry passengers to other destinations. The main advantage of cross-connections is the enormous "multiplier" effect as to the number of city pairs an airline can serve (see figure 2-2). Later on we will see that hubs can also be very useful in designing a robust airline schedule. On the other hand, hub networks introduce congestion of traffic and the need to transfer planes.

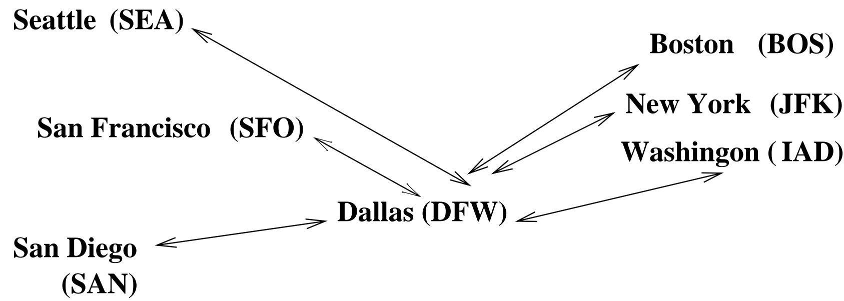


Figure 2-2: Hub and Spoke Networks

Typically the fleet assignment problem is solved as an integer programming model that assigns multiple aircraft types to a flight schedule. A number of other methods, based on the integer programming model along with other advanced techniques, have been introduced in the last few years. Currently researchers are attempting to combine the business processes of schedule construction and fleet assignment – problems that have traditionally been considered independent. Also, a hot research subject in the fleet assignment area has been schedule recovery in the event of irregular operations.

2.1.5 Aircraft Maintenance Routing

The most essential prerequisite for any airline operations is, of course, safety. Safety should never be sacrificed to meet any other goals. It is essential therefore to provide each type of aircraft in the fleet with a separate maintenance-routing plan. All routing plans must be coordinated to provide the best overall service. Certain stations are equipped with facilities and personnel to provide periodic mechanical checks which range all the way from simple and frequent maintenance such as en route service (to be performed at each stop) to complex and rare maintenance like airplane overhaul (to be performed every 6,000 flight hours, for example). Other types of maintenance checks include preflight check (at trip termination) and engine removal and installation. Different kinds of stations are responsible for different types of maintenance checks, and the aircraft router must ensure that the airplane gets to its scheduled station

before its time expires. Unfortunately keeping up the maintenance schedule is not possible without making constant adjustments. For example, if the aircraft happens to have a mechanical breakdown in Denver en route to San Francisco where it is scheduled for a maintenance check, the router then has to substitute another aircraft for the airplane in question. A tied-up airplane is now off the planned schedule designed to allow the carrier to accomplish all required inspections. Furthermore, now there is a danger of a maintenance station overload – once the tied-up airplane finally gets to its maintenance station, it will have to compete with other airplanes for a spot. It is therefore apparent that maintenance people have to work in close contact with the scheduling planners – a slight change in the way one of these groups does things will likely affect the other.

The aircraft routing problem is usually solved after the fleet assignment has been completed. Candidate flight segments are linked to specific aircraft tail numbers within a given sub-fleet of the airline. Traditionally aircraft maintenance routing has been done manually, but in the recent years researchers have made attempts to automate the process. Another area of research has to do with daily demand fluctuations. The latter frequently require changes in existing schedule, and those changes, in turn, result in modification of maintenance routing.

2.1.6 Crew Scheduling

After fuel costs, crew costs are the second largest component of direct operating costs for an airline. The crew scheduling process consists of several important elements such as crew pairing generation, crew rostering, and crew recovery. The *crew pairing* problem determines how to construct a set of pairings to cover all given flight segments at a minimum cost. An operational constraint in crew pairing is that flight crews are governed by working limitations found both in the Federal Aviation Regulations (FAR) and employment agreements. For example, flight-crew members must have had at least 16 hours of rest since the completion of their last assigned flight. Also, flight crews may not exceed a maximum of 40 flight hours during any seven consecutive days. Release from all duty for 24 hours must be granted to each flight-crew member during

any seven-consecutive-day period. The *crew rostering* problem considers activities like vacations and training to construct monthly work schedules for each employee. The goal of the *crew recovery* problem is to rebuild broken crew pairings using the existing pairings as well as reserve crews. Most methods for solving the crew scheduling problem involve integer programming solution techniques (branch and bound, for example), frequently combined with column generation. A lot of research has been done in crew scheduling. However, not a lot of work has been done on schedule recovery in the event of irregular airline operations. Current research in the area is focused on developing robust methods to create crew schedules that are less affected by irregularities.

2.1.7 Yield Management

The goal of yield management is to maximize revenue by selectively accepting and rejecting reservation requests based on their relative value. This is usually done in several different ways, including overbooking, fare mix, group control, and traffic flow control. *Overbooking* is a term used to describe the process of accepting more reservation requests than the number of seats on the plane. Overbooking compensates for the effects of events like cancellations and no-shows. Discount allocation or *fare mix* is a technique used to protect seats for late booking higher valued passengers by limiting the number of seats allocated to lower valued passengers booking early.

In the early stages of yield management utilization revenue maximization was accomplished through a leg-based inventory control. That basic process later evolved into a segment-based scheme, and currently carriers use an origin-destination based approach. Littlewood [11] developed a method to predict demand for each fare class by flight leg by looking at historical booking data. Williamson [13] describes the use of mathematical programming and network flow theory in the origin-destination seat inventory control problem.

2.2 Irregular Airline Operations

Clarke points out that for a typical airline, approximately ten percent of its scheduled revenue is lost due to irregularities in airline operations, with a large percentage being caused by severe weather conditions and associated loss of airport capacity [7]. An airline's ability to reschedule flights in a fast and efficient manner can be crucial when trying to withstand the competition from other airlines. However, not a lot of research has been done in the area of schedule recovery. Until a few years ago schedule recovery was being done manually and in many cases it still is. However, advances in mathematical programming and computer processing speed have enabled researchers to look for ways to perform schedule recovery automatically, at least in part.

When irregularities occur in an airline's operations, the primary goal of the airline is to get back on the original schedule as soon as possible, while minimizing flight cancellations and passenger travel delays. The Airline Operations Control Center plays a crucial role in this process.

2.2.1 Airline Operations Control Centers

The Airline Operations Control Center is responsible for the tactical stage of the carrier's scheduling – executing the schedule developed during the strategic stage, updating the schedule to accommodate minor operational deviations, and rerouting for irregular operations. An AOCC normally operates 24 hours a day, and its size varies depending on the size of the airline. Each AOCC works in close contact with the Maintenance Operations Control Center (MOCC) and various Stations Operations Control Centers (SOCC) which in turn are responsible for airline maintenance activities and station resources, respectively. Within the AOCC there are three functional groups, each having a separate role in the schedule execution process. These groups are: 1) the Airline Operation Controllers, 2) On-line Support, and 3) Off-line Support. The operation controllers are the only group within the AOCC that has the power to resolve any problems that come up during daily airline operations.

They are also the group that maintains the "Current Operational Schedules" (COS), the most up-to-date version of the airline system resource schedule that includes delays and cancellations, irregular routings for aircraft and crews, as well as additional flights. The on-line support group is responsible for functions like flight and crew dispatch. During regular operations dispatchers are responsible for the successful release of a flight given maintenance and airport restrictions, as well as availability of required operational support (fuel, gates, airport facilities) at airports of departure and destination. Finally, the off-line services provide supporting resources for all AOCC personnel by maintaining the navigational database, meteorology, and flight technical services.

In the event of irregular operations, the dispatcher informs the operations controller of the problem, and the controller's role is to devise operational schedules as quickly as possible. The result of this work is a new COS, a plan to be followed to return to the schedule developed during the strategic stage of planning, called the Nominal Schedule of Services (NSS) [6].

2.2.2 Irregular Airline Operations

Any airline's NSS is prone to unexpected daily changes due to factors such as severe weather conditions and ATC delays. Clarke [6] summarizes major causes of flight delays at major hub airports:

1. *Weather* Wind, fog, thunderstorm, low cloud ceiling (these conditions account for 93% of all flight delays at major hub airports)

2. *Equipment* Air traffic radar/computer outage, aircraft failure

3. *Runway* Unavailability because of construction, surface repair, disabled aircraft

4. *Volume* Aircraft movement rate exceeds capacity of the airport at a given time

Any aircraft routing is bound to be affected by delays, as most routings are optimally determined during the strategic phase of planning, without any consideration

for unexpected irregularities. This “optimality” provides very little slack time in the flight sequence, which means that any delay/cancellation early in the day is likely to affect the schedule for the rest of the day unless the airline can take effective steps to correct the problem and get back to its NSS. Most carriers have developed procedures to follow in the event of unexpected disruptions in operations. However, most of these procedures are implemented manually, with little or no reliance on automated decision support systems. Given how much data the operations controllers have to deal with, it only follows that they are generally forced to take a localized approach in dealing with irregularities. This in turn frequently results in less efficient decisions – flight cancellations for example, that later turn out to be unnecessary.

2.3 Previous Work on Irregular Operations

The following is a review of some research that has been done in the attempt to deal with irregular airline operations. Some of the most interesting studies done in the area are the Airline Schedule Recovery Problem and a study done at Southwest airlines.

2.3.1 The Airline Schedule Recovery Problem

Clarke [4, 5] discusses the Airline Schedule Recovery Problem (ASRP) which addresses how airlines can efficiently reassign operational aircraft to scheduled flights in the aftermath of irregularities. The model incorporates various aspects of the airline's tactical process. For example, aircraft routing/rotation and fleet scheduling are done simultaneously. ASRP is a model for solving the aircraft rescheduling problem that is best described as a hybrid of the traditionally defined fleet assignment problem and the aircraft routing/rotation problem. Furthermore, the mathematical formulation of ASRP allows possible flight delays and cancellations to be considered simultaneously. The author uses concepts from Linear Programming and Network Flow Theory to develop a number of algorithms (both heuristic and optimization-based) for solving the ASRP problem. The solution methodologies were validated by performing a case

study based on a set of operational data provided by two major carriers. The results of using each of the algorithms on the given data were compared to each other and also to the actual operations associated with the historical data (normal operations). Then irregularities, such as constraints on aircraft departures and aircraft movement, were simulated and used to compare performance of three different algorithms when applied to the original data combined with the simulated constraints. The results demonstrate that it is possible to develop efficient procedures for flight rescheduling.

2.3.2 Analyzing Robustness of a Prospective Schedule

Another interesting study related to schedule robustness was conducted at Southwest Airlines [9]. The idea was to develop a tool to analyze robustness of a prospective schedule. Such a tool can potentially be very useful to schedulers as it would allow them to analyze the on-time performance of a prospective schedule based on historical data for block time and turn times as well as explicit business rules (in cases where historical data does not apply). The data used consisted of actual departure and arrival times, cancellations, delay codes, and passenger data (e.g. baggage load, enplanements). To account for seasonal effects of block times and turn times, all of the historical data used in a simulation is based on the same season of the year as the proposed schedule. The historical data is processed by sorting it into appropriate buckets and then feeding it into a statistical package to create distribution functions for all possible scenarios. The distributions are then used in the simulation. The simulation tool is still in the process of being developed and its power is limited by the fact that Southwest is a point-to-point carrier with an average turn-time of 20 minutes. The simulation focus for a point-to-point airline will differ from that for a typical hub and spoke carrier. However, the approach taken by Southwest presents an interesting direction for airline scheduling research. If it is possible to develop tools for evaluating robustness of a prospective schedule, airline schedulers can use those tools to incorporate methods for handling irregularities into the strategic stage of planning.

The Southwest study is unique in that it developed a tool to "measure" the ro-

bustness of a schedule believed to be optimal under normal operational conditions. However, it did not create a framework or method for building robustness into the schedule a-priori.

Chapter 3

The Flight String Model for Aircraft Maintenance Routing

This chapter discusses the Flight String Model for Aircraft Routing [1]. The strength of this particular approach is in its ability to capture complicated constraints such as maintenance requirements and aircraft utilization restrictions.

3.1 Overview

The output of the fleet assignment problem is used as input into the aircraft routing problem. Given an assignment of flights to fleets, the airline must determine a sequence of flights, or routes, to be flown by individual aircraft such that assigned flights are included in exactly one route, each aircraft visits maintenance stations at regular intervals (usually about once in three days), and there is always an aircraft available for a flight's departure. The goal of the aircraft routing problem is to find a minimum-cost set of such aircraft routings. The costs to be minimized include the maintenance costs as well as the negative through revenues associated with through flights.

A *string* is a sequence of connected flights that begins and ends at a maintenance station (not necessarily the same one), satisfies flow balance, and is maintenance feasible, i.e. does not exceed maximum flight time before maintenance (while airlines

usually require a maintenance check every 40-45 hours of flying, the maximum time between checks is restricted to three to four calendar days). In addition, strings must satisfy the minimum turn time requirement – enough time has to be allocated between any two flights to provide service of aircraft. An *augmented string* is a string with a minimum maintenance time attached at the end of the string.

The model includes two types of variables, augmented string variables x_s and ground variables y.

3.2 Notations

3.2.1 Decision Variables

x_s is an augmented string variable: it equals 1 if $s \in S$ is selected as part of the solution set, 0 otherwise.

$y_{(e_1,e_2)}$ are ground variables, used to count the number of aircraft on the ground at a maintenance station between every two adjacent events (described below) at that station.

To ensure flow balance at maintenance stations, each station is associated with a set of *events* – the set of flights arriving at and departing from that airport. For any arriving flight $i \in F$, where F is the set of flights, its event time is its arrival time plus minimum maintenance time. For any departing flight, its event time is its departure time. At each maintenance station, all events are sorted and numbered in increasing order of time. In case a tie between an arriving and a departing flights occurs, the arriving flight is given priority. Ties involving only departing or only arriving flights are broken arbitrarily. Let $e_{i,a}(e_{i,d})$ be the event number corresponding to the arrival (departure) of flight i. Also, let $e^+_{i,a}(e^+_{i,d})$ be the next event at a station after the arrival (departure) of i. Similarly, $e^-_{i,a}(e^-_{i,d})$ is the event at a station preceding the arrival (departure) of i.

The number of ground variables has an upper bound defined by the number of flights terminating or starting at a maintenance station.

3.2.2 Parameters

c_s maintenance cost of string s

a_{is} equals 1 if flight $i \in F$ is in augmented string s and equals 0 otherwise.

N number of available aircraft

t_n the "count time"

r_s number of times (possibly greater than 1) augmented string s crosses the count line.

p_j number of times (0 or 1) ground arc $j \in G$ crosses the count line.

3.2.3 Sets

F the set of all flight legs i

S the set of all maintenance feasible strings s. The size of this set is exponential in the number of flights.

S_i^- the set of augmented strings ending with flight i and maintenance

S_i^+ the set of augmented strings beginning with flight i

G the set of all ground arcs y

3.3 Aircraft Routing String Model Formulation

The model can be described in the following way:

3.3.1 Objective Function

$$Min \sum_{s \in S} c_s x_s \tag{3.1}$$

The objective is to minimize the total cost of the selected strings.

3.3.2 Constraints

Flight Coverage:

Each flight must be assigned to exactly one routing.

$$\sum_{s \in S} a_{is} x_s = 1 \qquad \forall i \in F \tag{3.2}$$

Flow Balance:

The number of aircraft arriving at a station must equal the number of aircraft departing. For every departing flight there is an aircraft available.

$$\sum_{s \in S_i^+} x_s - y_{(e_{i,d}^-, e_{i,d})} + y_{(e_{i,d}, e_{i,d}^+)} = 0 \qquad \forall i \in F \tag{3.3}$$

$$-\sum_{s \in S_i^-} x_s - y_{(e_{i,a}^-, e_{i,a})} + y_{(e_{i,a}, e_{i,a}^+)} = 0 \qquad \forall i \in F \tag{3.4}$$

Aircraft Count:

The number of aircraft used cannot exceed the number available. This constraint ensures that the total number of aircraft in the air and on the ground does not exceed the size of fleet. It is enough to ensure that the count constraint is satisfied at one particular point of time – the flow balance constraints ensure that if the count constraint is satisfied at some point of time, it is also satisfied at any other point of time.

$$\sum_{s \in S} r_s x_s + \sum_{j \in G} p_j y_j \leq N \tag{3.5}$$

Other Constraints:

The number or aircraft on the ground at any time has to be non-negative.

$$y_j \geq 0 \qquad \forall j \in G \tag{3.6}$$

The number of aircraft assigned to a string has to be 0 or 1.

$$x_s \in \{0, 1\}, \qquad \forall s \in S \tag{3.7}$$

Notice that the solution to the aircraft routing problem does not explicitly specify connections between selected strings. However, the flow balance constraints guarantee

that such connections exist.

3.4 String-Based Model Solution

Due to the fact that the number of potential strings is exponential in the number of flights, the String Model uses column generation to reduce the number of columns (strings) used in the solution process.

3.4.1 Linear Program Solution

Column generation is a technique used to solve large linear program (LP) problems. When the number of variables in LP is too large to enumerate the constraint matrix explicitly, the column generation algorithm starts by selecting an initial set of variables for which the LP is then solved and dual costs associated with each constraint are determined. Those dual costs are then used to compute reduced costs associated with other, nonbasic variables. In a minimization problem, the variables with negative reduced costs are the variables which correspond to strings that may improve the solution. The columns (strings) with negative reduced costs are therefore added to the original set of strings used in the restricted problem. Adding variables to the constraint matrix is what is referred to as the *column generation process*. The linear program with a restricted set of columns is called the *restricted master problem.*

The restricted master problem is solved repeatedly, with a new set of columns added at every iteration. This process is repeated until no more variables have negative reduced cost, at which point optimality of the solution is achieved. When all reduced costs are nonnegative, it is guaranteed that no new column can improve the current optimal solution. Thus, column generation is a way to find an optimal LP solution without examining all problem variables. When the number of potential variables is very large, column generation can generate an optimal solution by solving LP with a very limited number of columns, leaving millions of other, non-optimal columns out of the LP. In the case of aircraft routing, as we have already pointed out, the number of string variables is exponential in the number of flights. Column

generation, therefore, comes in as a very handy technique in this case.

The column generation algorithm can be described as a two-step process:

Step 1. Solving the Restricted Master Problem:

Find an optimal solution to the current restricted master problem, with only a subset of string variables included.

Step 2. Solving the Pricing Subproblem and Updating the Restricted Master Problem:

Generate columns with negative reduced cost. If no more columns are generated, an optimal solution has been found, and the LP has been solved. Otherwise, update the Restricted Master Problem by adding new columns and go back to step 1.

The LP is solved by using optimization software packages such as CPLEX, which in turn use linear programming algorithms such as SIMPLEX.

3.4.2 Pricing Subproblem for the String Model

The pricing subproblem for the string model can be described in the following way:

$$RC_s = c_s - \sum_{i \in F} a_{is}\pi_i - \lambda_m^b + \lambda_n^e - r_s\sigma, \tag{3.8}$$

where RC_s is the reduced cost associated with string s, π_i is the dual variable corresponding to the cover constraint for flight i, λ_m^b and λ_n^e are the dual variables associated with the flow balance constraints for strings beginning with flight m and ending with flight n, and σ is the dual variable corresponding to the count constraint.

Chapter 4

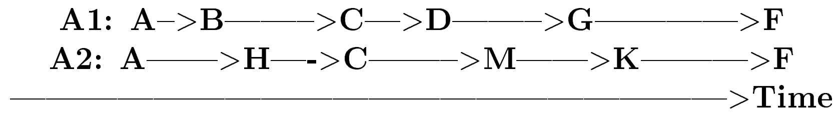
Robust Airline Scheduling

4.1 The Concept of Robustness. Fault Tolerant Recovery Paths

An airline schedule is said to be *robust* if it provides enough flexibility that parts of the schedule can be recovered in the event of irregularities in operations. In the event of severe weather conditions, for example, this flexibility allows an airline to easily recover its schedule by switching aircraft around and moving passengers to alternative itineraries. Similarly, if an aircraft originally assigned to a routing which covers high-demand flights happens to break down, a highly robust schedule may provide an option to reassign another aircraft to this routing and to get it back on its original routing before the next maintenance check. In general, a more recoverable schedule could help avoid significant revenue losses which would otherwise result from flight delays and cancellations.

In this research project we attempted to incorporate a measure of robustness into airline scheduling design. More specifically, we concentrated on the aircraft maintenance routing stage of the scheduling process. Instead of developing schedule reoptimization tools that can be used in the event of irregular airline operations, we focused on building fault tolerant recovery paths into the original schedule developed during the strategic phase of planning.

Schedule 1:



Schedule 2:

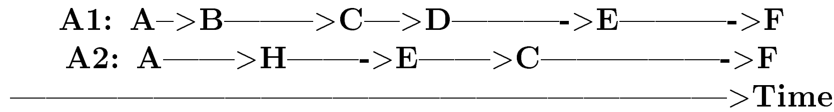


Figure 4-1: Example 1: Robust Schedules

4.2 Examples. Sequences. Points. Overlaps

Aircraft routing schedules can be thought of in terms of flights, sequences, points and overlaps. This section defines these terms and describes how they can be used to help measure robustness.

4.2.1 Example

Let us consider two hypothetical airline schedule examples in which aircraft 1 and 2 (A_1 and A_2) are assigned to two different routes (see Figure 4-1).

It can be observed that the first schedule is more robust than the second one as it allows for more flexibility in the face of irregular operations or demand fluctuations. For example, if A_1 experiences a mechanical problem at point C, it could be replaced with A_2. Or if the demand on segments (C, M), (M, K) and (K, F) were to suddenly show a temporary increase, while the demand on segments (C, D), (D, G) and (G, F) decreases, there is an option of switching A_1 and A_2 to bring higher profit (provided that CAP_{A_1} is greater than CAP_{A_2}, where CAP_A is capacity of aircraft A).

Notice that the two routes in Schedule 1 also meet at point F at about the same time. In the case of a mechanical failure, the functioning aircraft, which was reassigned to the other route, can still get back on its original route before the next maintenance check. The same is true for both aircraft if switching took place because

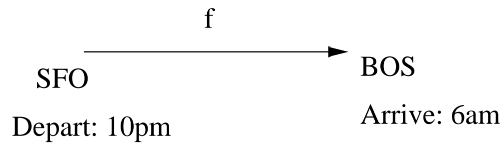


Figure 4-2: Flight

of fluctuations in the demand for the two routes.

4.2.2 Points. Sequences. Overlaps

Flights

A *flight* (see Figure 4-2) is defined by its origin, destination, departure and arrival times (given in minutes of the day starting from midnight). Thus, the flight in figure 4-2 would have the following parameters:

$Org(f) = SFO$,

$Dest(f) = BOS$,

$Dep(f) = 1320$,

$Arr(f) = 360$.

Sequences

A *sequence* is defined by a sequence of flights f (see Figure 4-3). In a sequence, any flight's destination is the same as the next flight's origin. For example, $Dest(f_0) = SFO = Org(f_1)$. For any sequence s and flight $f \in s$, Index(f) is defined to be the index of flight f in sequence s. Also, for any index i, $Flight(i)$ is the flight at index i in the sequence. Thus, $Index(f_0) = 0$ and $Flight(0) = f_0$. In addition, we say that sequence s has length n ($length(s) = n$) if there are n flights in sequence s. Sequence s in Figure 4-3 has length 5, and $Flight(2) = f_2$.

Points

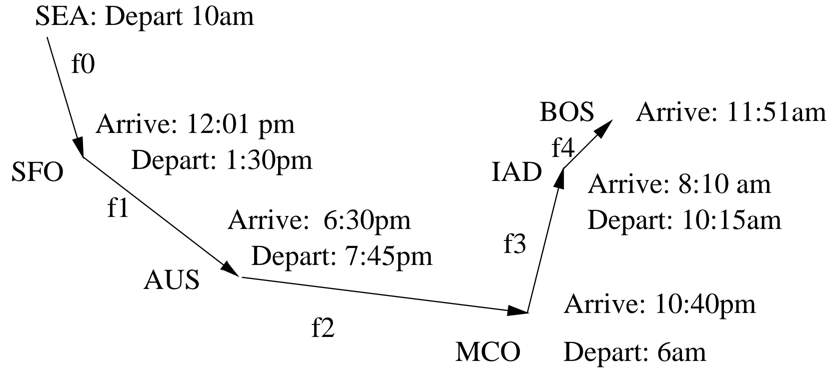


Figure 4-3: Sequence

A *point* $P_{s,k}$ is the interval of time that an aircraft assigned to routing s spends at an airport between the arrival of flight k and departure of flight $k+1$. Thus, points are defined by time and airport. More precisely, for any sequence s let us define:

Airport($P_{s,k}$)	the airport of point $P_{s,k}$
Arr($P_{s,k}$)	the arrival time at point $P_{s,k}$
Dep($P_{s,k}$)	the departure time from point $P_{s,k}$

$P_{s,0}$ corresponds to the point immediately preceding the first flight in a sequence. The last point $P_{s,n}$, where $n = length(s)$, corresponds to the point immediately following the last flight in s.

Let $n = length(s)$. Then

$$Arr(P_{s,k}) = \begin{cases} Arr(Flight(k-1)) & \forall k \in [1, n] \\ Dep(Flight(0)) & k = 0 \end{cases}$$

Similarly,

$$Dep(P_{s,k}) = \begin{cases} Dep(Flight(k)) & \forall k \in [0, n-1] \\ Arr(Flight(k-1)) & k = n \end{cases}$$

And lastly,

$$Airport(P_k) = \begin{cases} Org(Flight(k)) & k < n \\ Dest(Flight(n)) & k = n \end{cases}$$

In our example, then,

$$\begin{array}{lll} Arr(P_0) = 600 & Dep(P_0) = 600 & Airport(P_0) = SEA \\ Arr(P_2) = 1110 & Dep(P_2) = 1185 & Airport(P_2) = AUS \\ Arr(P_5) = 711 & Dep(P_5) = 711 & Airport(P_5) = BOS \end{array}$$

To define an order for points in a sequence, let $P_{s,i} > P_{s,j}$ if $i > j$. Thus, $P_{s,2} > P_{s,1}$.

Robustness Related Concepts

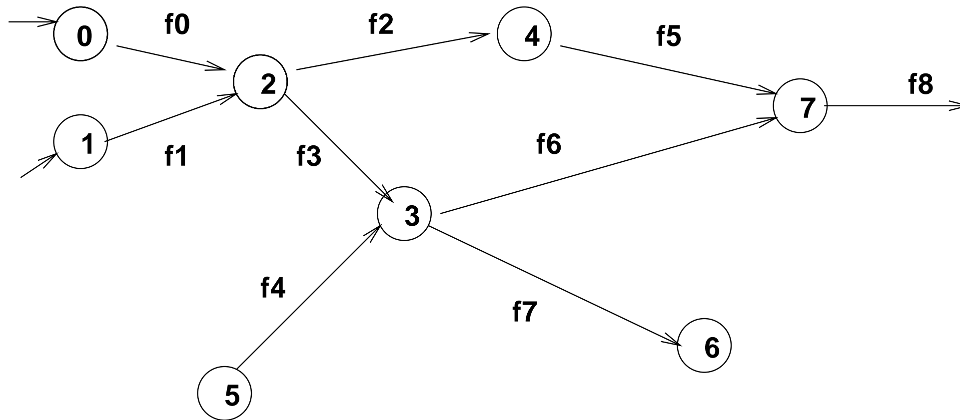
Definition 1: Sequences s_1 and s_2 *meet* at points $P_{s_1,i}$ and $P_{s_2,j}$ *within T=deltaT on departure* (or arrival) if $Airport(P_{s_1,i}) = Airport(P_{s_2,j})$ and $abs(Dep(P_{s_1,i}) - Dep(P_{s_2,j})) \leq deltaT$ $(abs(Arr(P_{s_1,i}) - Arr(P_{s_2,j})) \leq deltaT$ for arrival).

Figure 4-4 shows a network of eight flights, flight information, and potential flight sequences. One can observe that sequences s_0 and s_1 meet at points $s_{0,1}$ and $s_{1,1}$ within T=15 on departure. They also meet at points $s_{0,3}$ and $s_{1,3}$ within T=10 on arrival.

Definition 2: *Overlaps* are another very important sequence related concept, which will serve as the base for defining a measure of robustness. An *overlap within T=deltaT* occurs at point $P_{s_1,i}$ if there exist points $P_{s_1,i'} : i' > i, P_{s_2,j}, P_{s_2,j'} : j' > j$, such that s_1 and s_2 meet at $P_{s_1,i}$ and $P_{s_2,j}$ within a time interval *deltaT* on departure, and also s_1 and s_2 meet at $P_{s_1,i'}$ and $P_{s_2,j'}$ within *deltaT* on arrival.

Again, in Figure 4-4 we can see that sequence s_0 has an overlap within T=15 at point $P_{s_0,1}$, because s_0 and s_1 meet at points $P_{s_0,1}$ and $P_{s_1,3}$. Similarly, sequence s_1

Flight Network



Flight Information

Flight Number	Departure Time	Arrival Time	Origin	Destination
0	180	270	0	2
1	150	285	1	2
2	350	480	2	4
3	340	450	2	3
4	300	460	5	3
5	540	771	4	7
6	531	763	3	7
7	548	714	3	6

Possible Sequences

s_0 : $f0- > f2- > f5- > f8$
s_1 : $f1- > f3- > f6- > f8$
s_2 : $f4- > f7$
s_3 : $f1- > f3- > f7$
s_4 : $f4- > f6- > f8$

Figure 4-4: Sequences, Overlaps, and Robustness

has an overlap within T=15 at point $P_{s_1,1}$.

Definition 3: A sequence s is considered *robust within* $T = deltaT$ *at point* $P_{s,i}$ if there exists an overlap within deltaT at point $P_{s,i}$. Hence, sequence s_0 (Figure 4-4) is robust at point $P_{s_0,1}$.

Moreover, a sequence is considered *absolutely robust within* $deltaT$ if for every point on the sequence there exists an overlap within deltaT. (It is important to point out, however, that no sequence has an overlap at its ending point, because an overlap on a sequence is defined by two points, and the ending point is the last point on a sequence).

The number of *potential* overlaps for any sequence s can be computed as $\sum_{s \in S} length(s)$, where S is the set of all sequences in the solution. In general, one of the ways to increase robustness of an airline schedule is to provide ways for more aircraft routes to intersect at different points, so that aircraft can be easily switched if needed (subject to operational and maintenance constraints). More specifically, the goal is to minimize $P - A$, where P is the number of potential overlaps and A is the number of actual overlaps.

Definition 4: A more precise way to measure robustness is to compute the percentage of points in the system that have overlaps, i.e. $(A/P) * 100\%$, which we will refer to as the *coefficient of robustness* $coeff_r$. The higher $coeff_r$, the more robust a schedule is.

Returning to figure 4-4, suppose a routing solution consists of sequences s_0, s_1, and s_2. Then the number of potential overlaps is 8. The actual number of overlaps is 2, since sequence s_0 has an overlap at point 1, sequence s_1 has an overlap at point $P_{s_1,1}$, and sequence s_2 has no overlaps (even though it meets sequence s_1 at point $P_{s_2,1}$). Therefore, the coefficient of robustness $coeff_r = 100\frac{2}{8}(\%) = 25(\%)$.

Notice that if our routing solution consisted of sequences s_0, s_3, and s_4 instead, then there would be no overlaps in the system, and, hence, the coefficient of robustness would be 0%!

4.3 Tradeoff Between Robustness and Optimality

It is important to note that there is a trade-off between robustness and optimality in a schedule. One should expect that a highly robust airline schedule will not correspond to the maximum of the objective function in the LP model (which is how optimality is defined in this case). However, that in itself does not mean that robust schedules will result in lower profit than optimal schedules. An optimal schedule that does not take into consideration the airline's performance in the aftermath of irregular operations might not be as profitable in the end as a less optimal but a more robust schedule. Cancelled flights translate into revenue loss, and delayed flights result in loss of passenger goodwill which in turn also translates into revenue loss.

It is also true that there is frequently more than one optimal solution to an airline scheduling problem. For example, there may be several unique solutions to the aircraft maintenance routing problem, which all have the same cost. It is possible that some of these solutions are more robust than others. In this case, selecting a more robust solution may result in more efficient and profitable airline operations, and will therefore help ensure that the cost of operations is as close to the "optimal" as possible.

Our goal was to incorporate robustness into the airline scheduling process at the strategic stage of planning, to explore the above described trade-off between robustness and optimality, and to learn whether there is any correlation between the two. The result of this work is a new model that incorporates operational measures that provide robustness.

4.4 Building More Robust Schedules

As already mentioned above, the key to making airline schedules more robust lies in providing fault tolerant recovery paths. Fault tolerant recovery paths can be built into the system by incorporating the following factors into the aircraft routing model during the strategic phase of planning:

4.4.1 Optimized Turn Times

A common approach to scheduling flights at hub airports is first-in first-out (FIFO). In many instances, however, delays from incoming aircraft can disrupt hub operations either through missed passenger connections or missed equipment/crew connections. Historical data provides a means of determining which incoming flights are delayed because of issues at origin airports. This data can be used to develop probability density functions for arrival times and thus determine the expected value for actual correction times. For example, if we know that a certain airport (e.g. ORD) is likely to be affected by snow storms the first couple weeks of January, we might allow for longer connection times during that period of time.

4.4.2 Flexible Subroute Switching

If two routings meet at more than one node within a certain time window, an aircraft can be switched from one routing to the other and then returned to its original routing at a subsequent meeting node. Thus, if a flight is severely delayed or cancelled and the relative demand for the routing is favorable, route switching can provide robustness by allowing a flight with high demand to be flown when it otherwise would not be flown. The idea is similar to the one used in demand driven dispatch (used by Continental Airlines, for example, on certain markets).

4.4.3 Passenger Routing Redundancy

A schedule can be made more robust by ensuring alternative routing for passengers affected by potential flight delays and cancellations. Providing alternative routing can become a very complex task if one has to look at all possible Origin/Destination pairs used by a major airline. A good place to start is to implement passenger redundancy at least for the airline's most important markets (hubs, for example). In fact, if overlaps are what defines how robust a schedule is, we may want to assign various degrees of robustness to a sequence point depending on whether it is a hub or not. Also, even if two points are both hubs/not hubs, different degrees of robustness can be

associated with those points depending on how important it is to have fault tolerant recovery paths based at those specific points.

4.5 Methods for Incorporating Robustness into Aircraft Routing. Subroute Switching

In this work we concentrated mostly on subroute switching for aircraft maintenance routing. This work, however, can be expanded to be used in other areas of airline scheduling such as crew scheduling. Also, most of the model developed is applicable to passenger routing redundancy.

4.5.1 Incorporating a Measure of Robustness into the Objective Function

One way to account for robustness while generating a routing schedule is by incorporating a measure of robustness into the objective function used in the LP/IP. To review, the following is the objective function used in the linear program for the string model, described in the previous chapter:

$$Min \sum_{s \in S} c_s x_s$$

The above minimization problem can, of course, be presented as a maximization problem as well:

$$Max \sum_{s \in S} p_s x_s \tag{4.1}$$

where p_s is revenue obtained from including string s in the solution.

If we can find a way to measure robustness of a string rs_s, then the problem can be turned into a linear program which maximizes revenue and robustness at the same time:

$$Max \sum_{s \in S} (p_s + rs_s)x_s \tag{4.2}$$

Unfortunately, in the course of our research we discovered a number of problems which made implementing this approach impossible. Robustness is defined by the overlaps in the system. Therefore, robustness of a string rs_s (see equation 4.2) depends not only on the string s itself, but also on other strings in the solution. Since we do not know which strings will end up in the solution generated by the LP, rs_s is actually not a constant in the LP.

In our robustness model (described in detail in chapter 5), we tried to incorporate robustness into the LP/IP objective function (using OPL, described in chapter 5) the following way:

```
minimize
// cost of the solution:
// pair[i] is a variable deciding if sequence i is in the solution
// pairCst[i] is the cost of sequence i
  sum (i in Columns) pairCst[i]*pair[i] +
  //potential overlaps
  // routeFlt[i].up+1 is the number of points in sequence i
  sum (i in Columns) pair[i]*(routeFlt[i].up+1) -
  // actual overlaps
  sum (i in Columns: pair[i] > 0)
   (sum (j in [overlaps[i].low..overlaps[i].up])
    (max (k in [overlaps[i,j].low..overlaps[i,j].up])
                              pair[overlaps[i,j,k]]))
```

overlaps is an array which stores all potential overlaps in the system and is updated every time a new sequence is added. The first dimension of *overlaps* corresponds to the column number(sequence id), the second dimension corresponds to indices of points on the sequence, and the third dimension stores ids (numbers) of other sequences that

overlap with a given sequence at a given point. For example, overlaps[i,j] refers to an array containing ids of all sequences that overlap with i at point j.

The objective function consists of two parts:

1. **Cost of the sequences in the solution** The cost of the selected solution

   $\sum_{i \in Columns} pairCst[i] * pair[i]$

2. **"Missing" robustness cost** As was mentioned earlier in this chapter, robustness can be measured as $P - A$, which is the difference between the number of potential overlaps and the number of actual overlaps. Since a measure or robustness is being incorporated into a minimization problem, we want to minimize the number of points that do not have overlaps, which can be described as $A - P$. Notice that

   ```
   (max (k in [overlaps[i,j].low..overlaps[i,j].up])
                                   pair[overlaps[i,j,k]]))
   ```

   determines whether any of the potential overlaps for sequence i at point j are actually included in the solution. If none of them are, the expression evaluates to 0.

The problem with the above described approach is that the objective function becomes nonlinear, and is thus not suited for solution using a linear program.

4.5.2 Computing Alternative Optimal Solutions and Selecting the Most Robust One

Frequently there are a number of unique solutions to the routing/scheduling problem that have the same cost. If there are alternative optimal solutions, we can compare them based on robustness and select the most robust one. The strength of this method lies in the fact that we are *guaranteed* that the final solution is an improvement over the original "optimal" solution, because the new solution provides as much robustness

as possible while preserving the same "optimal" cost. This way, we end up with a solution which is still "optimal", but at the same time additional flexibility built into the schedule helps avoid disruptions in the face of irregular operations.

The focus of our research lies in this particular approach, and the details of the model and its implementation are discussed in the next chapter.

Chapter 5

Chapter 5: Building Robustness into the String Model

5.1 Overview of the Implementation

Our model is based on the string model for aircraft routing, which was discussed in detail in chapter 3. The modified base model was implemented in Optimization Programming Language (OPL) [10]. Some of the robustness work was implemented in OPL as well, but most of it was done in Perl and C++, which operated on the data obtained from OPL.

5.2 OPL Optimization Programming Language

5.2.1 What it is

OPL is a modeling language created by ILOG for combinatorial optimization, linear and integer programming. Many mathematical programming and optimization problems are not only very challenging from the computational and algorithmic standpoints, but also require substantial development effort since modeling such problems can be nontrivial.

OPL was inspired by other modeling languages like AMPL, and its goal is to

provide support for modeling mathematical programming problems, as well as giving access to many optimization algorithms. While the language has the capability to solve non-optimization problems, its ability to solve such problems is very limited, and it is certainly designed specifically for optimization.

The idea of a modeling language like OPL is to provide a language whose syntax is similar to the syntax used in textbooks and papers. OPL provides data structures for mathematical objects like sets as well as computer-language equivalents to algebraic and logical notations such as unions and intersections. For example,

$\sum_{i=1}^{n} a_i x_i$

can be written as

```
sum {i in [1..n]} a[i]*x[i]
```

Also, a minimization problem can be easily described in OPL the following way:

```
minimize
  sum(i in Items) value[i] * amount[i]
subject to
  forall (r in Resources)
    sum (i in Items) use[r,i] * amount[i] <= capacity[r];
```

Notice that the syntax above looks very similar to what one would use in a textbook to describe an integer programming problem. If one had to actually model and implement a problem like this in a standard programming language, it would take significantly more time and effort, because one would have to implement low-level concepts instead of focusing on modeling the problem.

In addition, OPL provides a way to solve sequences of related models, to make modifications to those models and to solve the modified models, a feature which came in very handy in our implementation and will be discussed later in this chapter.

While there exist other modeling languages, AMPL for example, OPL is the only one that performs constraint programming. Constraint programming in OPL allows optimization problems to be solved by specifying the *constraint* part and the *search*

part. The constraint part consists of a set of constraints to be satisfied, and the search part describes how to search for solutions.

5.2.2 Advantages of Using OPL

OPL was chosen to be the tool for implementing our base model for several reasons. First, the high-level abstraction that OPL provides allows one to focus on developing the model and its applications as opposed to the low-level details of implementing integer or linear programming. Second, OPL's ability to perform constraint programming can be exploited to generate sequences during the column generation process. While the popular tool CPLEX has proved to be extremely useful in solving mathematical programming problems, it does not have the ability to do constraint programming. OPL in fact uses CPLEX libraries to solve LP/IP problems, but at the same time it offers much more functionality.

As mentioned in the previous section, OPL provides a way for models to interact with each other, to be modified, and to be solved with several instances of data/constraints. *OPLScript* is a script language for OPL which supports all of these functionalities. OPLScript treats models as first-class objects, which means they can be developed and updated independently from the scripts that use them. In our problem, OPLScript allows us to generate new sequences during the pricing subproblem of column generation, add them to the restricted master problem, and then run the LP on new input, the updated set of sequences.

The modified base model was developed using ILOG's OPL Studio 3.020 – a graphical user interface to OPL.

5.3 Implementing the Modified Base Model

5.3.1 Overview

The steps of the solution process for the basic string model are shown in figure 5-1.

The overall algorithm is controlled by an OPL script called routing.osc. The

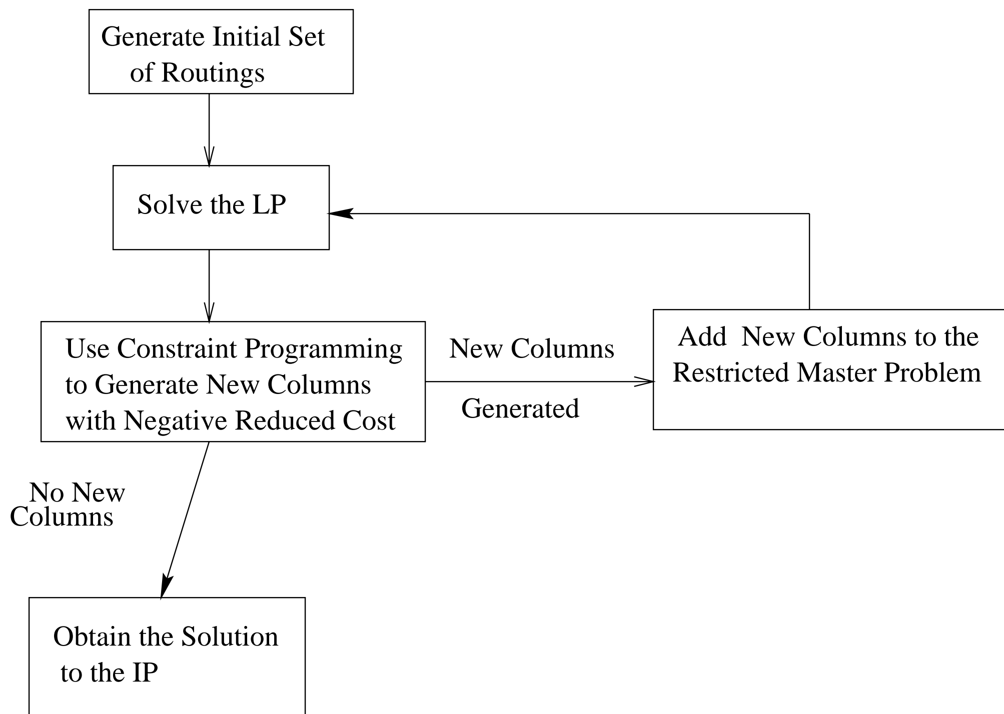


Figure 5-1: Flight String Model for Aircraft Routing. Solution Steps

script starts by generating an initial set of routings. Then the problem alternates between solving the LP relaxation of the set partitioning problem and solving a routing generation problem that produces new columns for the master problem. The solution strategy uses constraint programming as a pricing subproblem algorithm for linear programming column generation. Each column represents a "routing", a potential sequence of flights to be flown by an aircraft. Thus, the master problem must find a set of routings that covers every flight at a minimum cost, does not use more airplanes than are available, and maintains aircraft flow balance. This is a particular kind of set partitioning problem. Once there are no more columns with negative reduced cost, the columns are fixed and the set partitioning problem is solved to find an integer optimal solution. Finally, the OPL script prints the routings used by aircraft.

5.3.2 Generating Initial Set of Routings

The main script solves a constraint programming model (cvrroute.mod) sequentially for every given flight. The idea is to generate a set of routings that cover each flight in the network a number of times. The model exploits OPL's constraint programming functionality to search for the longest sequences, i.e. routings which maximize utilization.

However, to make sure a subset of these sequences actually form a solution when the LP is solved, a small number of "shortest" routings (with minimum utilization) are also included. The model cvrroute2.mod is responsible for generating this second, smaller set of sequences which cover each flight.

Models cvrroute.mod and cvrroute2.mod are almost identical. The main difference is in how the search procedure is defined. cvrroute.mod uses

```
search {
    tryall (i in fltRng : isInDomain(fltSeq[i], coverFlt)
            ordered by decreasing <i, dsize(fltSeq[i])>)
        fltSeq[i] = coverFlt;
    generateSize(citySeq);
    generateSize(fltSeq);
};
```

ordering sequences by length in decreasing order, whereas cvrroute2.mod ranks them in increasing order instead:

```
search {
    tryall (i in fltRng : isInDomain(fltSeq[i], coverFlt)
            ordered by increasing <i, dsize(fltSeq[i])>)
        fltSeq[i] = coverFlt;
    generateSize(citySeq);
    generateSize(fltSeq);
};
```

The following is an overview of how the constraint part of the above models works. The constraint part is also used later, during the subpricing problem, to generate columns with negative reduced cost.

Since the model is solved repeatedly for each flight, the flight being covered has to be imported into the model:

```
// Run-time data
Flight  coverFlt = ...;
```

A number of variables are declared in the model:

```
// Variables
var City    citySeq[cityRng];               // sequence of cities
var Flight  fltSeq[fltRng];                 // sequence of flights
var int+    startTime   in 0..Horizon;      // start time of the string
var int+    endTime     in 0..Horizon;      // end time of the string
var int+    endIndex    in 1..nSeq;         // index of last non-dummy
                                            // flight in the sequence
var Flight  startFlight;                    // first flight in sequence
var Flight  endFlight;                      // last flight in sequence
var int+    r           in 0..nDays;        // number of times the
                                            // routing crosses the count
                                            // line
var int+    duty    in 0..maxDuty;          // elapsed time
var int+    flight  in 0..maxDuty;          // time spent flying
var int+    util in 0..100;                 // overall utilization
var int+    cost in 0..maintenanceCost;     // cost of flying the string
```

To generate sequences of variable length, the set of flights includes a "dummy" flight for each airport, a flight whose origin and destination are the same. Each dummy flight departs at 1439 (last minute of the day) and arrives at 0 (first minute

of the next day). These flights can be used to “fill up” the end of `fltSeq` if a sequence is shorter than `nSeq`, maximum length allowed.

As potential sequences of flights/cities are generated in the search part of the model, the constraint part computes values for the above variables:

```
startFlight = fltSeq[1];
endTime     = (Arr[fltSeq[endIndex]] + minDuty) mod Horizon;
endIndex    = max (i in fltRng) (citySeq[i-1] <> citySeq[i])*i;
r = sum (i in fltRng)
          ((i <= endIndex)*(Dep[fltSeq[i]] > Arr[fltSeq[i]])) +
    sum (i in fltRng: i>= 2)
          ((i <= endIndex)*(Arr[fltSeq[i-1]] > Dep[fltSeq[i]])) +
    (Arr[fltSeq[endIndex]] + minDuty > Horizon);
duty        = r*Horizon + endTime - startTime;
flight      = sum (i in fltRng)
            ((citySeq[i-1] <> citySeq[i])*
             (Arr[fltSeq[i]]-Dep[fltSeq[i]]));
util        = 100*flight/(maxDuty-minDuty);
```

A special logical predicate, an equivalent of a simple function, is defined to help ensure that any flight’s destination is the next flight’s origin.

```
// Predicate for cities and flights
predicate p(Flight f, City o, City d) return o = Org[f] & d = Dst[f];
```

The constraint part ensures that a number of constraints described below are satisfied:

Link flights and cities using predicate:

```
forall (i in fltRng)
   p(fltSeq[i], citySeq[i-1], citySeq[i]);
```

Minimum Turn Time:

```
// Arr[fltSeq[i-1]] and Dep[fltSeq[i]] are on the same day
forall (i in fltRng : i>= 2)
   (i <= endIndex)*(Arr[fltSeq[i-1]] < Dep[fltSeq[i]]) =>
   (Arr[fltSeq[i-1]] + minStop +
   (hubStop-minStop) * (citySeq[i-1] in Hub)
                                 <= Dep[fltSeq[i]]);

// Dep[fltSeq[i]] is the day after Arr[fltSeq[i-1]
forall (i in fltRng : i>= 2)
   (i <= endIndex)*(Arr[fltSeq[i-1]] >= Dep[fltSeq[i]]) =>
   (Horizon - Arr[fltSeq[i-1]] + Dep[fltSeq[i]] >=
            minStop + (hubStop-minStop) * (citySeq[i-1] in Hub));
```

Constrain Dummy Flights to End of Sequence:

```
citySeq[0] <> citySeq[1];
forall (i,j in fltRng : 1 < i < j)
   (citySeq[i-1] = citySeq[i]) => (citySeq[j-1] = citySeq[j]);
```

Limit Elapsed Time:

```
duty <= maxDuty;
```

No Flight Can Be Repeated More Than Once:

```
forall (f in Flight)
  (Org[f] <> Dst[f]) => sum(i in fltRng) (fltSeq[i] = f) <= 1;
```

Start and End Sequence at a Maintenance Station:

```
// Sequence must start at a maintenance station and end at a
// maintenance station (not necessarily the same one)
citySeq[0] in Maintenance;
citySeq[nSeq] in Maintenance;
```

Cover the Specified Flight:

```
sum (i in fltRng) (fltSeq[i] = coverFlt) = 1;
```

As sequences are generated, they are stored in the main script, and then later imported by all internal models which are called from inside the main script.

5.3.3 How Sequences are Stored in OPL model

Two arrays are responsible for storing sequences:

```
Open Flight routeFlt[int+, int+]
Open int pairIdx[Flight,int+]
```

As new columns (sequences) are generated, they are added to the `routeFlt` array, which is an expandable array whose first dimension represents sequence numbers and second dimension represents flights in each sequence. Thus, the kth flight in sequence i is stored in `routeFlt[i,k]`.

For efficiency and complexity reasons, flight sequences are also stored in `pairIdx`, an array which stores sequences referenced by flights, i.e. each flight is associated with sequence numbers which contain that flight.

Notice that if sequences were only stored in `routeFlt`, then the flight cover constraint in the LP/IP would have to be implemented as follows:

```
// flight cover constraint
   forall (f in Flight : Org[f] <> Dst[f])
      cvr[f]:
         sum (i in Columns)
             (sum (j in [routeFlt[i].low..routeFlt[i].up])
                   pair[i]*(routeFlt[i,j] = f))
                          = 1;
```

Storing sequences in pairIdx helps avoid the above double summation complexity by transforming the flight constraint into the following much simpler form:

```
// flight cover constraint
    forall (f in Flight : Org[f] <> Dst[f])
         cvr[f]:
            sum (i in [pairIdx[f].low..pairIdx[f].up])
                pair[pairIdx[f,i]]
                       = 1;
```

It is important to point out that avoiding complexity in the LP/IP of this model is essential. Not only do we gain efficiency, but making IP/LP simple in OPL actually helps avoid some of the memory problems introduced that are by OPL Studio otherwise (see Section 5.3.6).

5.3.4 Restricted Master Problem. Solving the LP

After the initial set of routings has been obtained, OPLScript calls the LP model (linroute.mod) which is responsible for solving the restricted master problem.

The following variables and constraints are declared in the model:

```
var float+  pair[Columns] in 0..1;       // Amount for each routing
var float+  ground[Maintenance] in
            0..numAircraft;              // Number of planes on the
                                         // ground overnight
constraint  cvr[Flight];                 // Cover each flight
constraint  flow_balance[Flight];        // Flow Balance
constraint  maintenance_flow[City];      // Number arriving =
                                         // Number departing
constraint  aircraft_count;              // Use no more aircraft than
                                         // are available
```

The variables declared above are equivalent to x_s and y used in the string model (see section 3.1). The objective function looks similar to the one described in 3.3.1:

```
minimize
```

```
sum (i in Columns) pairCst[i]*pair[i]
```

Also, the flight cover and aircraft count constraints are defined in the model similarly to the equivalent constraints described in 3.3.2:

```
// flight cover constraint (for each non-dummy flight):
forall (f in Flight : Org[f] <> Dst[f])
     cvr[f]:
        sum (i in [pairIdx[f].low..pairIdx[f].up])
            pair[pairIdx[f,i]] = 1;

// aircraft count
aircraft_count:
   sum (city in Maintenance) ground[city] +
   sum(i in Columns) pair[i]*crossCount[i] <= numAircraft;
```

Notice, however, that the flow balance constraints differ from the ones in the original string model. Instead of defining a ground variable y for every ground arc, we only define a ground variable for the overnight arc at each station. Then, the flow balance constraints transform into the following:

```
// Number of sequences ending at a maintenance station has to
// be equal to the number of sequences starting:
forall (c in City: c in Maintenance)
   maintenance_flow[c]:
      sum(i in Columns) pair[i]*((Org[startFlight[i]] = c) -
                                 (Dst[endFlight[i]] = c))  = 0;

// For every sequence starting at a maintenance station, there is
// an airplane available at departure:
forall (f in Flight: Org[f] in Maintenance)
   flow_balance[f]:
```

```
ground[Org[f]] +
sum(i in Columns) pair[i]*(Dst[endFlight[i]] = Org[f])*
    (((Arr[endFlight[i]] + minDuty) mod Horizon) < Dep[f]) -
sum(i in Columns) pair[i]*(Org[startFlight[i]] = Org[f])*
    (Dep[startFlight[i]] < Dep[f]) -
sum(i in Columns) pair[i]*(startFlight[i]  = f) >= 0;
```

In the last constraint, the number of aircraft available at a maintenance station at a flight's departure from that station is determined by the number of aircraft present at the airport in the beginning of the day plus the number of strings that arrive before the flight's departure, minus the number of strings that depart the airport prior to the flight's departure.

5.3.5 The Pricing Subproblem

In the column generation phase, the constraint program is used twice. First, model optroute.mod is used to determine the cost of an optimal routing with respect to the current set of dual values. In other words, we look for a routing which has the most negative reduced cost. Then, entroute.mod is used to search for all routings that have reduced cost of at most 2/3 of the optimal routing (minCost). This gives us a large set of entering columns and eliminates one of the major weaknesses of column generation: a large number of iterations needed to improve the objective value in the master problem.

After the RMP is solved by the LP, the main script updates the two models used for the pricing subproblem with new dual costs:

```
forall (f in Flight : cp.Org[f] <> cp.Dst[f]) {
  op.fltCst[f] := ftoi(nearest(lp.cvr[f].dual));
  ep.fltCst[f] := ftoi(nearest(lp.cvr[f].dual));
}
forall (c in cp.Maintenance) {
  ep.maintCst[c] := ftoi(nearest(lp.maintenance_flow[c].dual));
```

```
      op.maintCst[c] := ftoi(nearest(lp.maintenance_flow[c].dual));
    }
    forall (f in Flight: cp.Org[f] in cp.Maintenance) {
      op.flowCst[f] := ftoi(nearest(lp.flow_balance[f].dual));
      ep.flowCst[f] := ftoi(nearest(lp.flow_balance[f].dual));
    }
    op.countCst := ftoi(nearest(lp.aircraft_count.dual));
    ep.countCst := ftoi(nearest(lp.aircraft_count.dual));
```

Both models are based on the constraint programming model used to generate the initial set of columns. However, optroute.mod solves a maximization problem, and entroute sets a lower bound on the objective function.

The objective function computes the reduced cost of a column (routing) using the above dual costs:

```
// Define objective function
obj = sum (i in fltRng) fltCst[fltSeq[i]] +
      maintCst[Org[startFlight]] -
      maintCst[Dst[endFlight]] +
      sum(f in Flight) (Dst[endFlight] = Org[f])*
                       (endTime < Dep[f])*flowCst[f] -
      sum(f in Flight) (Org[startFlight] = Org[f])*
                     (startTime < Dep[f])*flowCst[f] -
      sum(f in Flight) (startFlight = f)*flowCst[f] +
      countCst - cost;
```

It should be pointed out that the above equation actually computes a positive value, so it should be maximized, not minimized.

The goal of optroute.mod is to maximize the objective function:

```
maximize
    obj
```

entroute.mod, on the other hand, uses the solve functionality of constraint programming to find all routings which have a reduced cost of at least minCost:

```
solve {
  obj >= minCost;
}
```

5.3.6 Problems with OPL

As it has already been mentioned above, OPL is a language designed specifically for optimization problems. While it also provides some functionality similar to that offered by more standard programming languages, this functionality is very limited.

These limitations force a particular storage model for sequences. Unlike a more complete language like C++, we can not create objects that represent sequences nor can we easily perform any non-optimization related work on these sequences, since OPL does not allow functions to be defined.

Moreover, OPL Studio does not allow explicit memory management. Either by mistake or by design memory frequently does not get released after a model within an OPLScript is solved.

Since robustness is directly related to sequence overlaps, determining how robust a given solution is requires searching through a large number of sequences. Given the limitations in OPL functionality, however, it is impossible to perform this kind of computation and memory intensive work inside OPL. Unfortunately, OPL also does not allow external programs to be accessed from within OPL itself. Most of the robustness related work, therefore, has to be done outside of OPL, completely separately from the base model implemented in OPL.

5.4 Making the String Model Robust

The solution process for the Robust String Model is presented in figure 5-2.

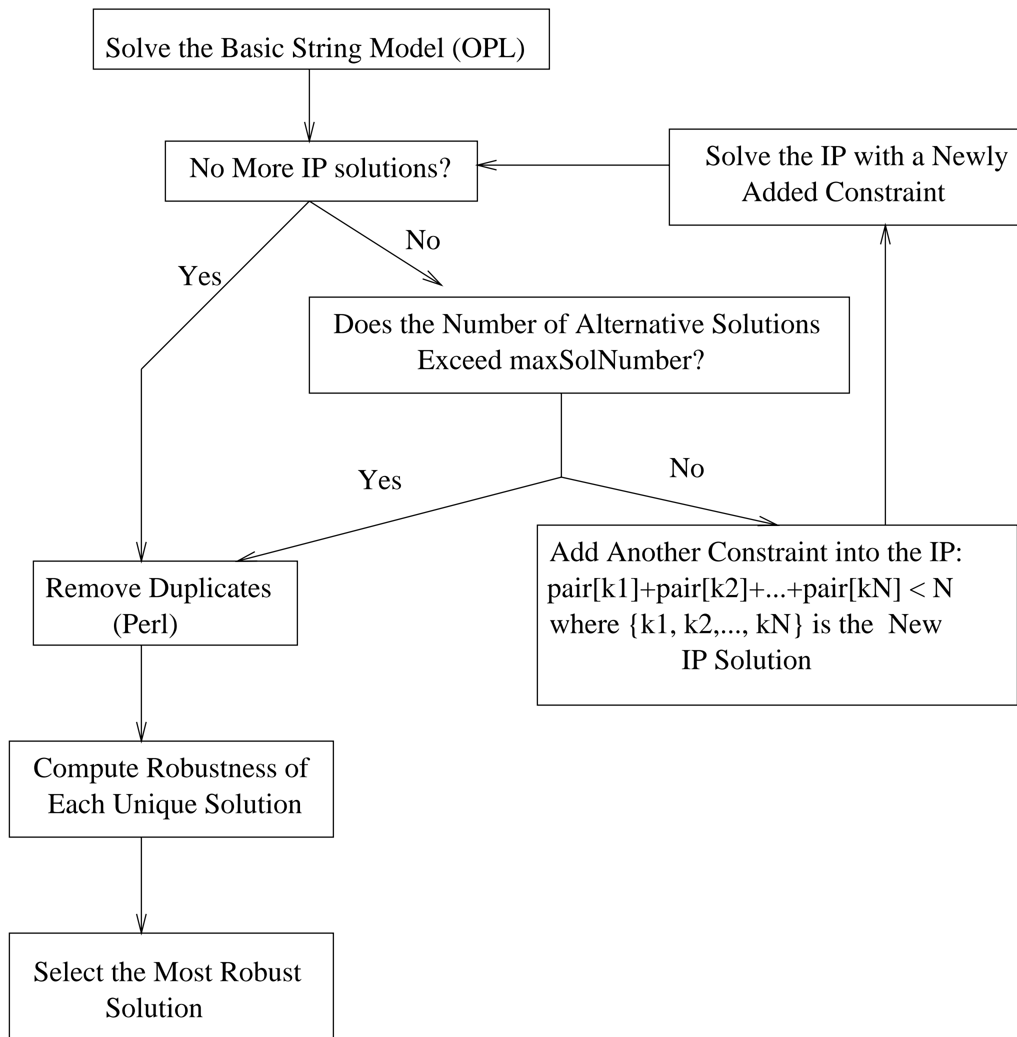


Figure 5-2: Building Robustness into the String Model. Solution Steps

5.4.1 Obtaining Alternative Solutions by Adding Constraints

After an optimal IP solution is obtained, the OPLScript which is in charge of the overall program control iterates through the IP model and obtains a number of alternative optimal solutions which have the same cost as the original optimal solution. This process is accomplished by incorporating a new constraint into the IP at every iteration. It can be described as the following:

$$\sum_{i \in prev_sol} pair[s_i] < \sum_{i \in prev_sol} 1 \tag{5.1}$$

where s_i is column i, $prev_sol$ is the set of columns (i's), and

$$pair[s_k] = \begin{cases} 1 & s_k \in IP \\ 0 & otherwise \end{cases}$$

Basically, the above described constraints ensure that none of the already generated solutions are generated again by requiring that no set of sequences representing an existing solution is a subset of a new solution.

The main script routing.osc keeps track of all of the IP solutions that are generated at each solution generation by storing them in an expandable array:

```
Open int solColIdx[int+, int+];
```

`solColIdx` is an array whose first dimension represents solution number and the second dimension stores indices of columns included in the corresponding solution.

The IP in turn imports (at every iteration) the array of existing solutions from the main script, declares an additional constraint `new_solution` for every existing solution, and defines each of the constraints in the following way:

```
import Open solColIdx;
constraint  new_solution[[solColIdx.low..solColIdx.up]];
```

```
// generate only new solutions:
   forall (solIndex in [solColIdx.low..solColIdx.up])
     new_solution[solIndex]:
       sum (i in [solColIdx[solIndex].low..solColIdx[solIndex].up])
         pair[solColIdx[solIndex,i]] <= solColIdx[solIndex].up;
```

5.4.2 Removing Multiple Copies (Perl)

During the constraint programming stage it is possible for multiple copies of the same string to be generated. Having multiple copies of the same sequence will frequently lead to multiple copies of the same solution obtained from the IP. A set of unique solutions, therefore, must be selected.

Once a specified number of alternative solutions have been generated and output to a file, a script written in Perl looks through these solutions, sorts them, and keeps only one copy of each solution.

Any two solutions that include *exactly the same* set of sequences (order does not matter), are considered the same.

5.4.3 C++ code

Once a set of unique IP solutions has been obtained from OPL and output to a file, a program written in the C++ programming language processes each solution and assigns a measure of robustness to each of them. Results are then output to a file.

5.4.4 Algorithm for Finding Overlapping Sequences

For each unique alternative solution obtained, all sequences in the solution are stored and then processed according to the algorithm described below.

For every sequence s_1 in the solution:

```
1) foreach point p1 on sequence s1
2)   foreach sequence s2 which is present at airport(p1) at
```

```
     point p2 on departure (not including s1 at p1) within
     a time interval deltaT {
3)     foreach point p1'in s1: p1'> p1
4)       foreach point p2' in s2: p2' > p2
5)         if (p2 and p2' form an overlap) {
           // Make sure the number of days in between p1 and p1'
           // is the same as the number of days in between
           // p2 and p2'.  Otherwise it's not an actual overlap
6)           if (abs((arr(p1')-dep(p1)) -
                     (arr(p2')-dep(p2))) <= 2*deltaT) {
             // An overlap is found. Update the overlap array
             // and go back to step 2) to look for other sequences
             // that might overlap with s1 at p1
                  declare there is an overlap at point p1 for
                      sequence s1
                  go to next p1 on sequence s1
             } // end if 6)
           } // end if 5)
         } // end foreach 4)
       } // end foreach 3)
     } // end foreach 2)
   } // end foreach 1)
```

Since robustness of a string is defined by the percentage of points in the system that have overlaps, the goal of the above algorithm is to look at every point in the system and find out if its sequence has an overlap there.

Notice that even when sequences s_1 and s_2 meet at two sets of points (p_{1_1}, p_{2_1}) and (p_{1_2}, p_{2_2}) within a specified time interval deltaT, an overlap is not guaranteed. The subsequences of s_1 and s_2 defined by the specified points can still have different length. Since departure and arrival times are defined in minutes of the day, even

if the subsequences start and end at about the same time, it is still unknown how many times flights and points in those subsequences cross the count line. Hence, the number of times the subsequence of s_1 crosses the count line can be different from the number of times s_2 cross the count line, which in turn means the two subsequences can have different length.

5.4.5 Avoiding Complexity. Sequence Storage Model.

When the flight network is large and includes thousands of flights, the number of sequences in the solution can be quite high. Step 2 of the algorithm described in the previous section can become quite complex if we have to look at every point on every sequence.

The complexity of finding pairs (s_2, p_2) can be avoided by storing sequences by airport and departure time at every point. Each airport is associated with a number of time intervals that cover 24 hours (1440 minutes). For each *airport-time(A-T)* combination, there is a linked list of sequences which are present at A during the time interval T. Thus, every sequence is referenced in the linked lists as many times as there are points on that sequence (not including the last one, since no overlaps are possible at that point). Therefore, in step 2, pairs (s_2, p_2) can be found by looking at the linked list representing *(airport(p1), TimeInterval(p1))* and the linked lists which correspond to the neighboring time intervals for the same airport.

5.4.6 Data structures. Objects. Sequence Storage.

As was pointed out earlier, one of the problems with OPL is that there is no good way to store sequences such that they can be efficiently processed later to evaluate the robustness of a solution. The object-oriented functionality of C++, on the other hand, provides the data storage flexibility that can be exploited to process solutions efficiently.

Figure 5-3 shows how sequences are stored in our C++ model. Here we give a short description of data structures used.

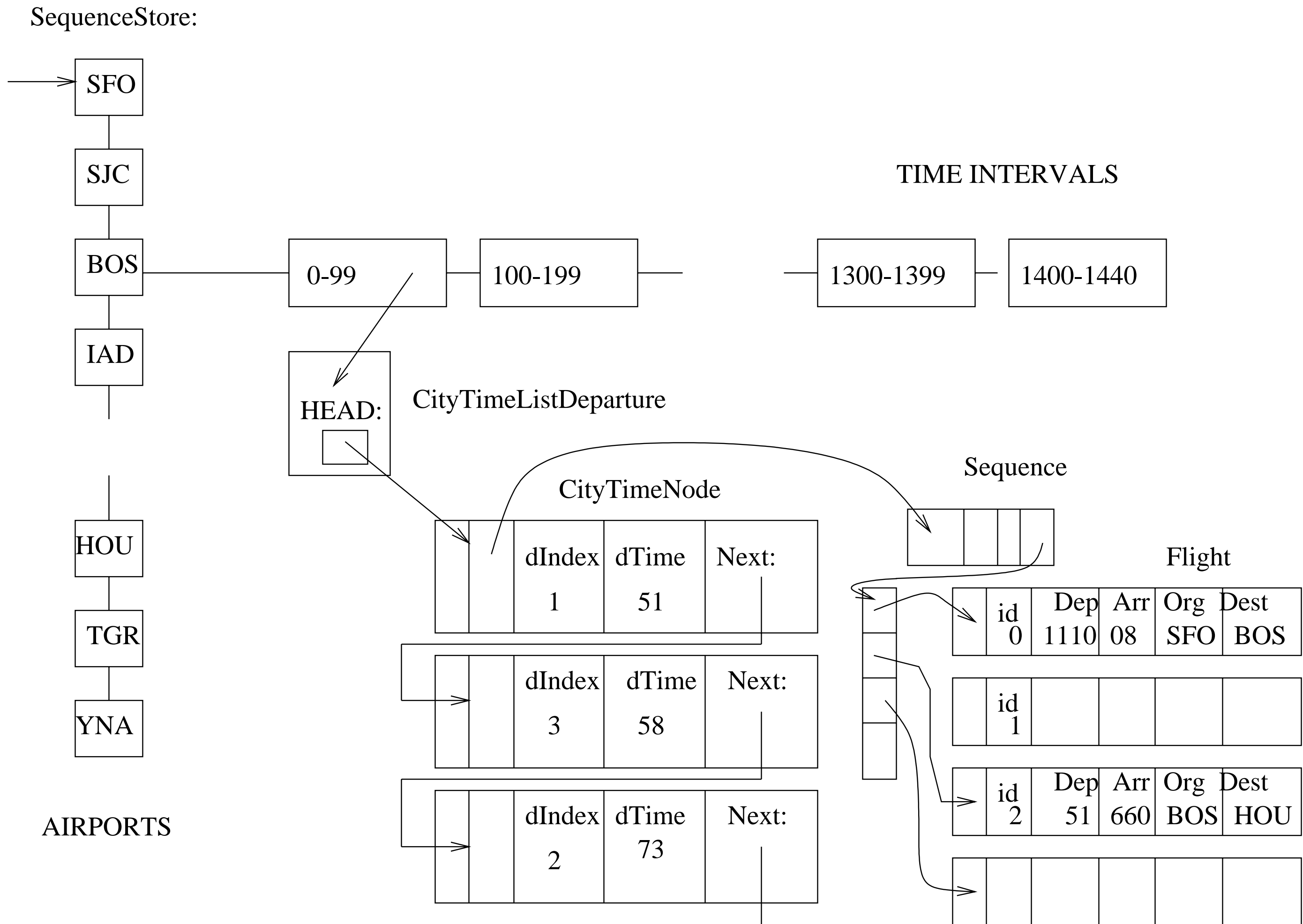


Figure 5-3: Sequence Store Design

Flights

Flight objects contain flight id, departure and arrival times, and origin and destination cities. Times are defined in minutes of the day [0..1440]. Cities are represented with integer ids.

Sequences

Sequences are defined by the following components:

```
// number of times this sequence crosses the count line
int nDays;
// maximum number of flights in a sequence
int maxSize;
// current number of flights in a sequence
int size;
// sequence arrival time determined by end of maintenance
// which follows the arrival of the last flight in a sequence
int endTime;
// array of flights representing a sequence
Flight** fl;
```

Sequence objects provide methods for accessing departure, arrival, and airport information at every point on a sequence, as well as duration time of any subsequence specified by indices of start and end points of the subsequence.

SequenceNode

A SequenceNode is an object that encapsulates a sequence and provides a pointer to the next SequenceNode. SequenceNodes are used to create linked lists of sequences (SequenceList).

SequenceList

A SequenceList object represents a linked list of Sequences. A SequenceList's head points to the first element in the list or NULL if the list is empty. The class

provides methods for adding and removing elements as well as accessing the head of the list.

CityTimeNode

CityTimeNode objects represent departures associated with sequence points. They are used to create CityTimeListDeparture objects which represent lists of sequence points whose departures fall within a particular Time interval and are associated with a particular airport (City). A CityTimeNode contains a pointer to a sequence, an index of a point on the sequence, departure time for that point, and a pointer to a CityTimeNode representing next departure at the airport of interest.

CityTimeListDeparture

CityTimeListDeparture objects represent sequence points whose departures fall within a particular time interval of the day at a particular airport. Within this time interval (inside the list) points are sorted by departure time in increasing order. The head element points to the first CityTimeNode on the list (the one that has the earliest departure).

The class provides methods for inserting elements (sequence points), removing elements and determining if a sequence point is contained in the list, as well as extracting the head of the list.

Notice that keeping points sorted by departure time within a list helps access an event's neighbors at an airport, a functionality useful when we are trying to determine if two sequences meet at a pair of points.

SequenceStore

This is the most important class, responsible for storing alternative solutions and determining their robustness.

SequenceStore objects are responsible for sorting and storing sequences by time and airport. SequenceStore maintains a two dimensional array of CityTimeListDeparture* elements, the first dimension representing airports, and the second dimension

representing consecutive time intervals of the day. A list of sequences SL is also maintained to keep track of all sequences entered into the SequenceStore.

Using the algorithm described in sections 5.4.4 and 5.4.5, SequenceStore was used to determine robustness of a set a sequences, defined by the percentage of total points in the system that have overlaps with at least one other point in the system. The results are described in chapter 6.

Chapter 6

Results and Conclusions

6.1 Overview

The robustness model described in Chapter 5 was tested on subsets of an actual airline maintenance routing schedule. For each subset a number of equally optimal alternative solutions were compared based on robustness. It was found that in some cases the model provided an increase in robustness of up to 35% as compared to the original string model. At the same time, the optimal cost of the final solution was preserved.

6.2 Test Data

As shown in table 6.1, several subsets of an airline schedule used by a major airline were used to test the robustness model. Test networks consisted of up to 6 airports, one of which was a hub, and up to three of which were maintenance stations. The networks contained between 14 and 37 randomly picked flights, with the smaller networks being subsets of the larger networks. Minimum turn time for hubs was set to 40 minutes; 30 minutes were provided for other, non-hub stations. Maximum length of a string was defined to be 4320 minutes (three days), a common length of time allowed by airlines between maintenance checks. Minimum maintenance time was set to three hours.

Table 6.1: Test Data

Network	Flights	Airports	Hubs	Maintenance
1	14	5	1	2
2	22	6	1	2
3	26	6	1	2
4	37	6	1	3

6.3 Robustness Factors

It would seem obvious that varying the number of alternative solutions and robustness time tolerance would alter the degree to which robustness of the final solution could be improved. In fact, increasing the number of alternative solutions generated in the model corresponds directly to improving robustness since the final result is the most robust solution in the set. The impact of varying the time tolerance on robustness, on the other hand, is significantly less clear as there is no direct correlation in this case.

6.3.1 Number of Alternative Solutions

For each test network the OPL part of the model generated a set of 500 alternative solutions. Then, for each of those sets a Perl script generated five subsets by extracting 100, 200, 300, 400, and 500 top solutions from the original set. Another Perl script was responsible for removing multiple copies of solutions from each of those subsets and for sorting the remaining unique solutions by flight number. Thus, for every test network, five sets of unique solutions were generated. We will call these sets *final* sets.

It was our expectation that using larger final sets would result in higher robustness of the final solution.

6.3.2 Robustness Time Tolerance

To determine the effect that varying robustness time tolerance would have on the robustness of the final solution, every final set in each of the four networks was tested with $deltaT$ of 20, 30, 40, 50, 60, 70, 80, and 90 minutes.

As mentioned above, we expected there would be no direct correlation between $deltaT$ and improvement in robustness of the final solution. It seems reasonable to assume that robustness of any solution increases as $deltaT$ goes up, since there is more opportunity for overlaps among the sequences in the solution. Thus, while maximum and average robustness of any final set go up as $deltaT$ increases, minimum robustness goes up as well. Therefore, the improvement, or the difference between maximum and minimum/average robustness, is not guaranteed to change.

6.4 Results

For each file containing robustness of a final set for a given $deltaT$, a Perl script generated an information file which describes the number of unique solutions, coefficients of robustness for the least and most robust solutions, and average robustness of the set. The latter was computed as the sum of robustness coefficients of all solutions in the set, divided by the total number of solutions in the set.

A solution with minimum robustness corresponds to the worst case (least robust) solution obtained from the original string model. Conversely, a solution with maximum robustness corresponds to the best case (most robust) solution obtained from the original string model and is the solution selected by the improved string model. The average robustness of the set indicates the likely robustness of a solution obtained from the original model.

In the following sections, robustness coefficients are rounded to the nearest whole number.

6.4.1 Improvement

From these three measurements, we can determine the improvement in robustness computed by the improved string model as compared to the original model.

For each network and each $deltaT$, *maximum* improvement was defined to be the difference between the highest and lowest achieved robustness among the solutions in the largest final set, and *average* improvement was defined to be the difference between the highest and average achieved robustness. Since the standard (non-robust) maintenance routing problem selects any of the alternative optimal solutions, the average improvement most closely corresponds to the expected improvement over the original string model. However, it is important to note that in the worst case - that is, the case in which the standard maintenance routing problem selects the least robust solution - the improved string model can offer the maximum improvement over the original model.

The maximum and average improvement in robustness are given for each network in the following sections.

6.4.2 Network 1: 14 Flights

When run against the 14 flight network, a minimum of 27 and maximum of 54 unique solutions were generated for subsets of 100 and 500 alternative solutions, respectively. Tables 6.2 - 6.4 show minimum, maximum, and average robustness computed for the 14 flight network. As shown, for values of $deltaT < 40$, all solutions computed had robustness of 0. However, for $deltaT = 40$, the model was able to obtain a significant improvement in robustness - 28% (see Table 6.5.) Of particular interest was the improvement for $deltaT = 80, 90$ - a difference of 35%. Figure 6-1 shows robustness distribution of the final set of 500 solutions for each specified value of $deltaT$. Notice that for larger values of $deltaT$ there exists a larger number of robust solutions, whereas most of the solutions for small values of $deltaT$ are nonrobust. Furthermore, robustness of the solutions is more equally distributed for larger values of $deltaT$.

Table 6.2: Results: Minimum Robustness Coefficient(14 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	27	0	0	0	0	0	0	7	7
200	38	0	0	0	0	0	0	7	7
300	46	0	0	0	0	0	0	7	7
400	48	0	0	0	0	0	0	7	7
500	54	0	0	0	0	0	0	7	7

Table 6.3: Results: Maximum Robustness Coefficient(14 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	27	0	0	14	14	14	14	28	28
200	38	0	0	14	14	14	14	28	28
300	46	0	0	28	28	28	28	42	42
400	48	0	0	28	28	28	28	42	42
500	54	0	0	28	28	28	28	42	42

Table 6.4: Results: Average Robustness Coefficient(14 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	27	0	0	8.3	8.3	8.3	8.3	17.9	17.9
200	38	0	0	7.4	7.4	7.4	7.4	16.9	16.9
300	46	0	0	7.6	7.6	7.6	7.6	17.7	17.7
400	48	0	0	7.3	7.3	7.3	7.3	17.5	17.5
500	54	0	0	7	7	7	7	17.5	17.5

Table 6.5: Results: Improvement (14 flights, 54 solutions)

Improvement	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
Maximum	0	0	28	28	28	28	35	35
Average	0	0	21	21	21	21	24.5	24.5

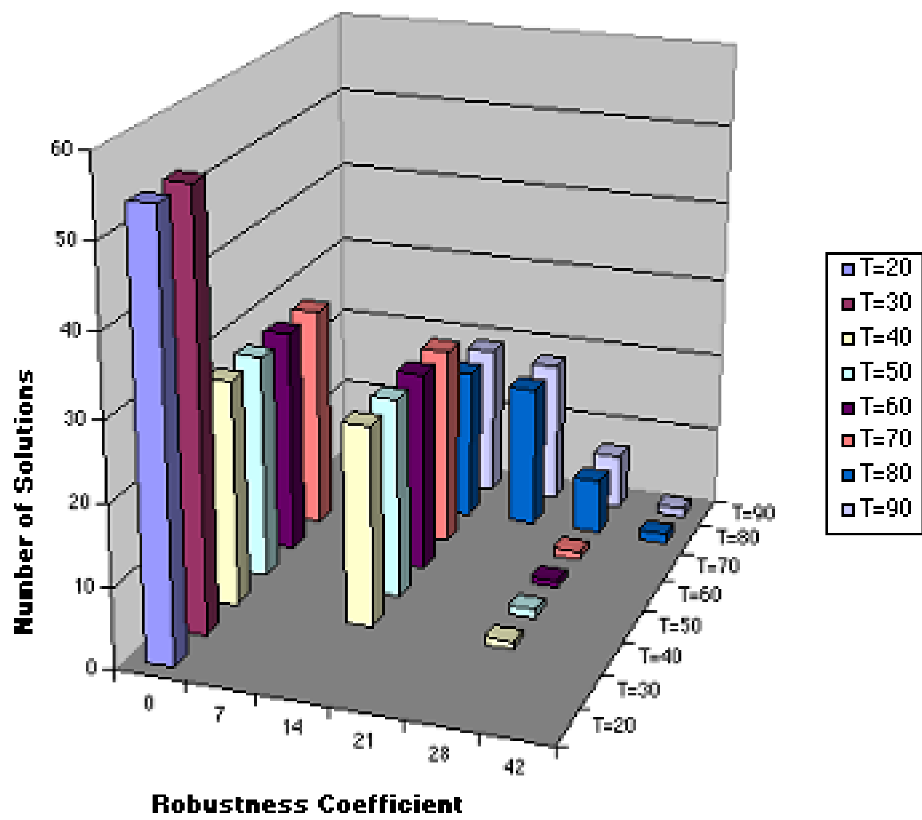


Figure 6-1: Robustness Distribution (14 flights, 500 solutions)

6.4.3 Network 2: 22 Flights

For this network, the model computed a significantly larger number of unique solutions as compared to Network 1 – between 59 and 260 solutions were generated. As with the 14 flight network, all of the computed solutions for $deltaT < 40$ in the 22 flight network had a robustness coefficient of 0. Within the flight network, improvements in maximum robustness could be found at $deltaT = 40$ (18%) and $deltaT = 80$ (23%). Results are shown in tables 6.6 - 6.9. Figure 6-2 shows robustness distribution of the final set of 500 solutions for each specified value of $deltaT$. Notice that for $deltaT = 40$ and $deltaT = 50$ there exists a small number of robust solutions, while the majority of the solutions are nonrobust; for larger values of $deltaT$, robustness of the solutions is more equally distributed.

Table 6.6: Results: Minimum Robustness Coefficient (22 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	59	0	0	0	0	0	0	4	4
200	115	0	0	0	0	0	0	4	4
300	172	0	0	0	0	0	0	4	4
400	225	0	0	0	0	0	0	4	4
500	260	0	0	0	0	0	0	4	4

Table 6.7: Results: Maximum Robustness Coefficient (22 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	59	0	0	18	18	18	18	27	27
200	115	0	0	18	18	18	18	27	27
300	172	0	0	18	18	18	18	27	27
400	225	0	0	18	18	18	18	27	27
500	260	0	0	18	18	18	18	27	27

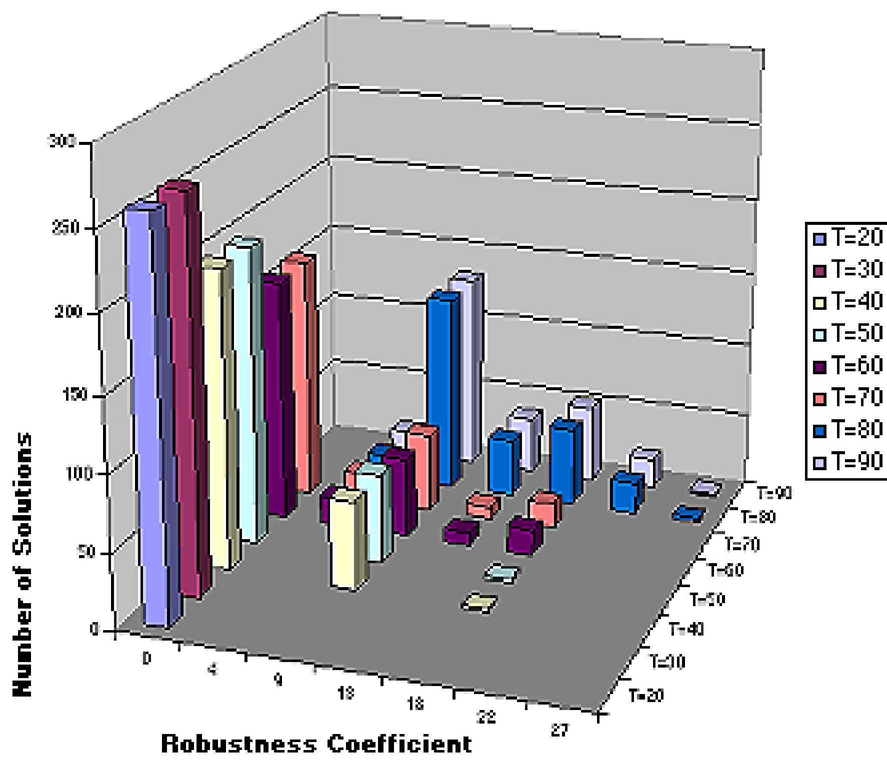


Figure 6-2: Robustness Distribution (22 flights, 500 solutions)

Table 6.8: Results: Average Robustness Coefficient (22 flights)

Number of Solutions	Unique Solutions	△T =20	△T =30	△T =40	△T =50	△T =60	△T =70	△T =80	△T =90
100	59	0	0	1.4	1.4	3.1	3.1	11.3	11.3
200	115	0	0	2.7	2.7	5.1	5.1	13	13
300	172	0	0	2.6	2.6	4.6	4.6	12.7	12.7
400	225	0	0	2.4	2.4	4.1	4.1	12.7	12.7
500	260	0	0	2.2	2.2	3.8	3.8	12.2	12.2

Table 6.9: Results: Improvement (22 flights, 260 solutions)

Improvement	△T =20	△T =30	△T =40	△T =50	△T =60	△T =70	△T =80	△T =90
Maximum	0	0	18	18	18	18	23	23
Average	0	0	15.8	15.8	14.2	14.2	14.8	14.8

6.4.4 Network 3: 26 Flights

Testing the 26 flight network, like the 22 flight network, resulted in a fairly large number of unique solutions (376 for the final set of size 500). Unlike the smaller networks, a number of solutions for $deltaT = 20, 30$ had some level of robustness built into them. In fact, the two largest improvements in robustness appeared to occur at $deltaT = 20$ (12%) and $deltaT = 40$ (27%). Results for this network can be found in tables 6.10 - 6.13. Figure 6-3 shows robustness distribution of the final set of 500 solutions for each specified value of $deltaT$. Similarly to networks 1 and 2, robustness of the solutions is higher for larger values of $deltaT$. In fact, for larger values of $deltaT$, distribution of robustness resembles normal distribution.

Table 6.10: Results: Minimum Robustness Coefficient (26 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	70	3	3	11	11	11	11	19	19
200	145	3	3	7	7	7	7	11	11
300	221	3	3	7	7	7	7	11	11
400	296	3	3	7	7	7	7	11	11
500	376	3	3	7	7	7	7	11	11

Table 6.11: Results: Maximum Robustness Coefficient (26 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	70	7	7	23	23	30	30	34	34
200	145	7	7	26	26	30	30	34	34
300	221	15	15	30	30	34	34	38	38
400	296	15	15	34	34	34	34	38	38
500	376	15	15	34	34	34	34	38	38

Table 6.12: Results: Average Robustness Coefficient (26 flights)

Number of Solutions	Unique Solutions	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
100	70	5.7	5.7	15.3	15.3	17.3	17.3	25	25
200	145	6.1	6.1	16.9	16.9	18.6	18.6	25.6	25.6
300	221	6.5	6.5	17.8	17.8	19	19	26.3	26.3
400	296	6.7	6.7	17.4	17.4	18.4	18.4	26.2	26.2
500	376	6.8	6.8	17.4	17.4	18.4	18.4	26.2	26.2

Table 6.13: Results: Improvement (26 flights, 376 solutions)

Improvement	$\triangle$T =20	$\triangle$T =30	$\triangle$T =40	$\triangle$T =50	$\triangle$T =60	$\triangle$T =70	$\triangle$T =80	$\triangle$T =90
Maximum	12	12	27	27	27	27	27	27
Average	8.2	8.2	16.6	16.6	15.6	15.6	11.8	11.8

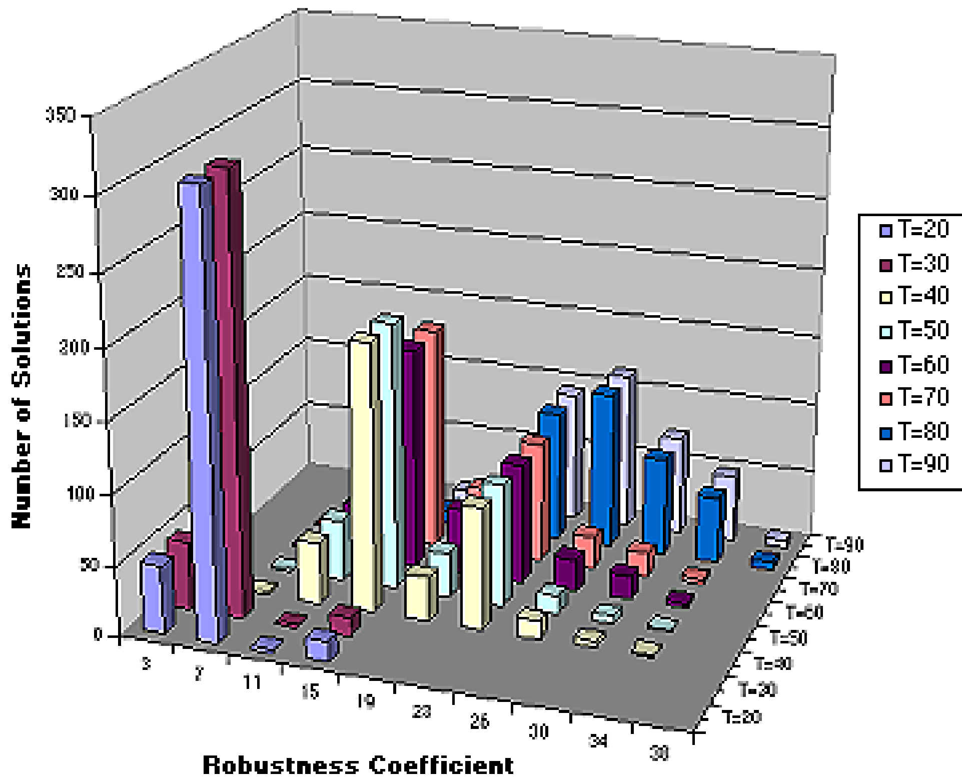


Figure 6-3: Robustness Distribution (26 flights, 500 solutions)

6.4.5 Network 4: 37 Flights

This network was the largest test network. As was the case with network 1, the number of unique solutions in the 37 flight network appeared to be fairly small – between 22 and 91 unique solutions were generated for the network's final sets. For all *deltaT* in each final set, all solutions had some level of robustness built in, with a minimum of 2% for *deltaT* = 20 in the final set of size 500 and maximum of 51% for *deltaT* = 90. Maximum improvements were found at *deltaT* = 20 (8%), *deltaT* = 30 (13%), and *deltaT* = 90 (27%). Results are summarized in tables 6.14 - 6.17. Figure 6-4 shows robustness distribution of the final set of 500 solutions for each specified value of *deltaT*. Notice that, similarly to all smaller test networks, robustness of the solutions is distributed more equally for larger values of *deltaT*.

Table 6.14: Results: Minimum Robustness Coefficient (37 flights)

Number of Solutions	Unique Solutions	△T =20	△T =30	△T =40	△T =50	△T =60	△T =70	△T =80	△T =90
100	22	5	10	16	16	16	18	24	32
200	37	5	8	16	16	16	16	21	29
300	42	5	8	16	16	16	16	21	29
400	67	2	8	16	16	16	16	21	24
500	91	2	8	16	16	16	16	21	24

Table 6.15: Results: Maximum Robustness Coefficient (37 flights)

Number of Solutions	Unique Solutions	△T =20	△T =30	△T =40	△T =50	△T =60	△T =70	△T =80	△T =90
100	22	10	21	35	35	37	37	43	51
200	37	10	21	35	35	37	37	43	51
300	42	10	21	35	35	37	37	43	51
400	67	10	21	35	35	37	37	43	51
500	91	10	21	35	35	37	37	43	51

Table 6.16: Results: Average Robustness Coefficient (37 flights)

Number of Solutions	Unique Solutions	△T =20	△T =30	△T =40	△T =50	△T =60	△T =70	△T =80	△T =90
100	22	5.2	11.8	23.7	23.7	24.1	24.9	30.7	38.2
200	37	5.3	11.2	22.6	22.6	22.9	23.5	28.8	36.1
300	42	5.2	11	22.1	22.1	22.4	22.9	28.3	35.7
400	67	5.4	10.7	21.2	21.2	22.3	22.9	28	35.1
500	91	5.4	10.6	21.2	21.2	22.6	23	28.3	35.6

Table 6.17: Results: Improvement (37 flights, solutions)

Improvement	△T =20	△T =30	△T =40	△T =50	△T =60	△T =70	△T =80	△T =90
Maximum	8	13	19	19	21	21	22	27
Average	4.6	10.4	13.8	13.8	14.4	14	14.7	15.4

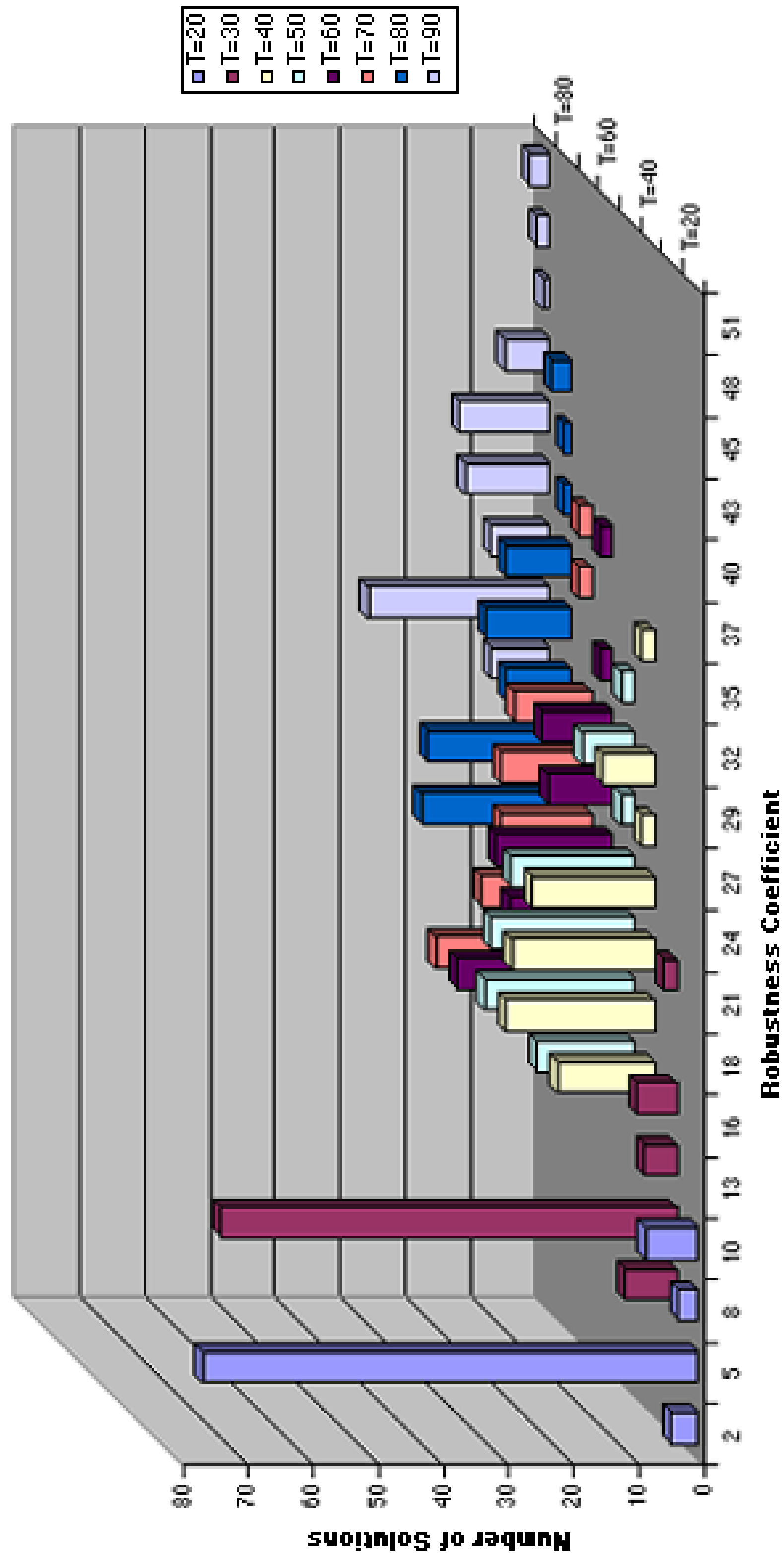


Figure 6-4: Robustness Distribution (37 flights, 500 solutions)

6.5 Conclusions

The results obtained suggest that the alternative solutions approach to the aircraft routing problem can significantly improve robustness of the solution while preserving its optimal cost. An improvement in robustness of up to 35% over the original model has been shown in some cases. As was expected, for each network robustness of the final solution was directly related to the number of alternative solutions in the final set – the larger the final set, the greater an improvement over the original model.

In addition, we also discovered that there was in fact a correlation between $deltaT$ and robustness. Contrary to our original assumption (see Section 6.3.2), we found that larger values of $deltaT$ resulted in larger improvement. Also, we discovered that in most cases the largest improvement occurred between $deltaT = 30$ and $deltaT = 40$. This last fact is especially interesting as 40 minutes is a reasonable value for time tolerance to allow for two aircraft to be switched at an airport. Put another way, if two aircraft depart within 40 minutes of each other and one of them is substituted for the other, there will be a maximum of 40 minute delay (20 minutes on average). Finally, the graphs showing robustness distribution for each of the test cases suggest that larger values of $deltaT$ result in a larger number of solutions incorporating some level of robustness. Furthermore, robustness of the solutions in each network is distributed more equally for larger values of $deltaT$.

6.5.1 Problems and Future Model Improvement

Unfortunately, due to insufficient time and limitations introduced by OPL, we were not able to test the model on sufficiently large data sets. Also, our implementation did not make full use of the search functionality of OPL's constraint programming, and, therefore, some inefficiency was introduced in the pricing subproblem of column generation.

One of the goals of our research was to explore the trade-off between robustness and optimality. We have shown that, in general, the solution to the maintenance routing problem can be improved significantly by increasing its robustness. Moreover,

since in our model the robust solution is selected from the set of optimal solutions to the original string model, no optimality is traded for robustness. However, we did not try to determine how much extra robustness can be gained by selecting a slightly less optimal solution. This issue still remains to be explored in the future.

Chapter 7

Future Research

Over the course of this research work on robust airline scheduling several related areas of interest were identified. This chapter discusses some suggestions for future work.

7.1 Robustness Measure as Part of LP's Objective Function

In section 4.5.1 a few suggestions were offered on how to incorporate a measure of robustness into the objective function. While we were unable to come up with a way to do so and keep the objective function linear at the same time, we hope this approach will become possible in the future.

The strength of this approach (if implementable) is in that we are guaranteed the final solution is optimal, since robustness consideration is a part of the column generation process. While the alternative solutions approach guarantees improvement of the original optimal solution, it might not result in the most robust solution since once an optimal solution is generated, the column generation process stops. Some of the columns making up the most robust optimal solution might in fact never be considered.

7.2 Robustness as Part of the Pricing Subproblem

Neither of the two approaches discussed in sections 4.5.1 and 4.5.2 has the ability to generate columns that are robust from the start. The first method does not take robustness into consideration until after all of the new columns at a given iteration of the column generation process are added to the restricted master problem and the LP is being solved. The second, alternative solutions, approach does not take robustness into consideration until a set of optimal solutions has been found.

If, in the pricing subproblem of the column generation process we can generate columns that are robust from the start, we can make the solution process significantly more efficient. Fewer columns will have to be generated, which may significantly reduce complexity of the LP/IP as well as the solution time in general. Also, some or all of the work described in section 5.4 can be eliminated because the solutions found will have robustness built in from the start.

7.3 Through Flights

For simplicity purposes, our robustness model assigned the same cost to every string and did not take into consideration potential revenue obtained from through flights in the network. It would be interesting to explore the trade-off between revenue obtained from through flights and robustness in the schedule.

7.4 Hybrid Airline Scheduling and Robustness

The techniques presented in this research project to build robustness into aircraft routing can also be applied to hybrid airline scheduling solutions. One of the examples, immediately related to aircraft routing, is the combined fleet assignment/maintenance routing string model [8]. A joint formulation and optimization framework used in hybrid airline scheduling can result in higher revenues. For the same reason, applying robustness to hybrid airline scheduling can potentially result in higher robustness of the final schedule.

7.5 Robust Crew Scheduling

Robust airline scheduling, and our model in particular, can be applied to other parts of the airline scheduling design. Crew scheduling in many ways is similar to aircraft routing. Building robustness into crew scheduling will allow a crew, delayed in an event of irregular operations for example, to be switched from one pairing to another and then returned to its original pairing before the end of duty.

7.6 Estimating Performance of a Schedule Based on Historical Data

A set of historical data from an actual airline could be used to compare the airline's actual performance in the past to its hypothetical performance given a series of more robust schedules.

7.7 Robust Airline Schedules in Practice

While the method developed as a result of this research project has been shown to increase robustness of aircraft routing schedules in theory, it still remains to be seen how robust schedules perform during actual operations. A robust airline schedule which has subroute switching opportunities built into it at the strategic stage of planning will still have to be maintained and updated during the tactical stage. Whether robustness built into the schedule will actually be used during actual operations depends on how efficiently schedule maintenance and updates are performed during the tactical stage.

In fact, we hope that future research will include developing efficient ways to maintain airline schedules, such that robustness built into a schedule is actually used when necessary.

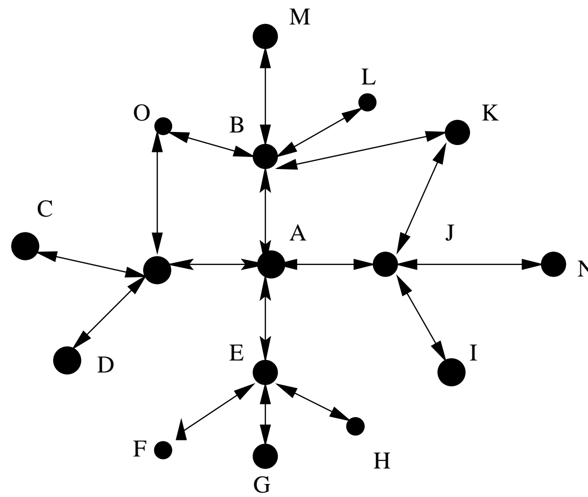


Figure 7-1: Representing Robust Schedules Geometrically

7.7.1 Possible Geometrical Representation of a Robust Schedule

As we have already seen, an aircraft routing schedule is defined by a set of sequences, whose robustness in turn is defined by the overlaps in that set of sequences. One way to describe a robust schedule is by using a fractal-like structure (see Figure 7-1).

The above structure represents a highly robust airline schedule in which there is a large amount of flexibility to switch aircraft in the event of irregular operations or demand fluctuations. In general, one might guess that hub-and-spoke networks provide higher robustness than point-to-point networks. By definition, hubs are used to transfer passengers from one route to another. The goal, then, is to have a number of routes intersect at the same airport at about the same time, which frequently will result in an overlap at that airport.

In the future it may be possible to determine robustness of an airline schedule directly from its geometrical representation.

7.8 Social Aspects of Airline Scheduling

7.8.1 Customer Acceptance

Another, more social-related, side of the issue that would be interesting to address is the issue of customer acceptance of robust versus optimal schedules. That is, given an airline with a robust schedule which takes into account potential delays but has a slightly less convenient scheduled arrival/departure time or higher fares than a competing airline with a more optimal but less robust schedule, would customers choose the more robust airline?

7.8.2 Other Related Questions

Making Information More Available to the Public

It would be interesting to explore how availability of flight delay information to the public affects customer satisfaction. For example, if a passenger learns that not only his flight was delayed but many other, will he be more accepting and patient?

Government Policy and Research

Another area of interest are government imposed airline fees based on on-time performance. Little research has been done to determine how the government policy has historically affected research in airline operations.

Appendix A

Mathematical Tools

A.1 Mathematical Programming

A *mathematical program* is an optimization problem of the (standard) form:

$$Maximize\ \ f(x): \qquad x \in X,\ \ g(x) \leq 0,\ \ h(x) = 0, \tag{A.1}$$

where X is a subset of R^n and is in the domain of the real-valued functions, f, g and h. The relations, $g(x) \leq 0$ and $h(x) = 0$ are called *constraints*, and f is called the *objective function*.

A point x is *feasible* if it is in X and satisfies the constraints: $g(x) \leq 0$ and $h(x) = 0$. A point $x*$ is *optimal* if it is feasible and if the value of the objective function is not less than that of any other feasible solution: $f(x*) \geq f(x)$ for all feasible x. The sense of optimization is presented here as *maximization*, but it could just as well be *minimization*, with the appropriate change in the meaning of optimal solution: $f(x*) \leq f(x)$ for all feasible x.

A.2 Linear Programs (LP). Integer Programs(IP)

A linear program is an instance of a mathematical program that can be described in the following form:

$$Opt\{cx : Ax = b, \;\; x >= 0\} \tag{A.2}$$

Other forms of the constraints are possible, such as $Ax \leq b$.

Integer programs are linear programs in which the variables are required to be integer-valued.

A.3 SIMPLEX

An algorithm invented to solve a linear program by progressing from one extreme point of the feasible polyhedron to an adjacent one. The method is an *algorithm strategy*, where some of the tactics include *pricing* and *pivot selection*.

A.3.1 Pricing

This is a tactic in the simplex method, by which each variable is evaluated for its potential to improve the value of the objective function.

A.3.2 Pivot Selection

In the simplex method, this is a tactic to select a basis exchange. The incoming column is based on its effect on the objective function, and the outgoing column is based on its effect on feasibility.

A.4 CPLEX

A collection of mathematical programming software solvers.

Bibliography

[1] Cynthia Barnhart. Flight string models for aircraft fleeting and routing. Working Paper, MIT Center for Transportation Studies., 1997.

[2] Cynthia Barnhart and Kalyan Talluri. *Airline Operations Research*, chapter 10. 1996.

[3] Headley Bowen. The airline quality rating. *http://www.unomaha.edu/ unoai/research/aqr99.html*, 1999.

[4] Michael Clarke. The airline schedule recovery problem. Working paper, MIT International Center for Air Transportation., 1997.

[5] Michael Clarke. *Development of Heuristic Procedures for Flight Rescheduling in the Aftermath of Irregular Airline Operations*. PhD dissertation, Massachusetts Institute of Technology, International Center for Air Transportation, 1997.

[6] Michael Clarke. Irregular airline operations: A review of the state-of-the practice in airline operations control centers. Working paper, MIT International Center for Air Transportation., 1997.

[7] Michael Clarke and Barry Smith. The impact of operations research on the evolution of the airline industry: A review of the airline planning process. jul 1999.

[8] Cynthia Barnhart et al. Flight string models for aircraft fleeting and routing. *Transportation Science*, 32(3), August 1998.

[9] Alex Heinold. On-time performance simulation model at southwest airlines. AGIFORS Schedule and Strategic Planning Study Group Meeting., 1999.

[10] Pascal Van Hentenryck. *ILOG OPL Optimization Programming Language*. ILOG, January 200.

[11] Kenneth Littlewood. Forecasting and control of passenger bookings. In *12th Annual Symposium Proceedings*, pages 95–117, Nathanya (Israel), October 1972. AGIFORS.

[12] Alexander Wells. *Air Transportation: A Management Perspective*. Wadsworth Books, 1995.

[13] E.L Williamson. *Airline Network Seat Inventory Control: Methodologies and Revenue Impacts.* PhD dissertation, Massachusetts Institute of Technology, Flight Transportation Laboratory, 1992.