Arrival, departure and surface management of flights by exploiting the network simplex algorithm

by

Marius Sandvik

THESIS

for the degree of

MASTER IN APPLIED MATHEMATICS

(Master of Science)

Faculty of Mathematics and Natural Sciences
University of Oslo

May 23, 2014

Faculty of Mathematics and Natural Sciences
University of Oslo

Abstract

Given today's limited capacity at airports, and the continuous increase in air transportation, the need for new ways to handle airport logistics will soon be needed. The task of managing arriving and departing flights is complex, and is today solved by several air traffic managers at each airport. While these managers operate well within their zone, the overall performance with respect to capacity and punctuality would benefit from viewing the airport as a whole. In addition, this may increase accuracy in predicting take-off times, allowing more reliable information for the arrival of flights, and reduce the environmental impact of air traffic. This thesis provides insight into how the network simplex algorithm can be used to find a conflict free solution between flights. The focus lies on mitigating the overall delay at airports and on reducing the taxi time of each individual flight while at the same time complying with safety regulations on the runway. This research also provides a basis for future integration with other airport operations such as runway sequencing and gate allocation. A data set from Stockholm Arlanda Airport was used in order to test the algorithms.

Acknowledgements

First off, I would like to start by thanking Carlo Mannino for his guidance and advice as my main supervisor during this thesis. Our discussions provided both progress and setbacks from which I have learned a lot. I would also like to thank Geir Dahl, as my supervisor at the University of Oslo, for providing me with the opportunity to work with the people at SINTEF as part of this thesis.

Secondly I would like to thank Dag Kjenstad, Patrick Schittekat, and Morten Smedsrud, all part of the research group at SINTEF, for welcoming me with open arms, and treating me as a full member of the group. Many thanks also for the provided insight into the world of aviation, and the mysterious programming forest known as C#. In addition, I would like to thank Nina Hulleberg, who was also writing a thesis related to this research area, and with whom I had the pleasure of sharing an office, as well as everyone else in the Operational Research Group at SINTEF.

Last but not least, a big thank you to all my friends and family for their unwavering support and encouragement throughout this period, always pushing me forward in the right direction.

Contents

Chapter 1

Introduction

1.1 Motivation and background

In recent years, with the advent of the liberalisation of the airline industry, the world has seen a rapid increase in air traffic volume, and IFALPA (International Federation of Air Line Pilots' Associations) has stated [1] that: *The growth has at times threatened to overwhelm the existing system capacity and a paradigm shift is required for the system to safely keep pace with the explosion in demand.* The SESAR (Single European Sky ATM Research) project predicted in 2006 [2] that the air traffic volume in Europe would double within 2020. Hence, improvements in crucial airport operations will become more and more important in the near future. With this in mind, SESAR has said [3] that its aim, amongst other things, is to triple the capacity within the same time period and reduce the delay both on the ground and in the air. The main operations that affect this bottleneck is the handling of arriving and departing flights. That is, finding a sequence on the runway, and directing the ground movement. Due to the complexity of these operations, this is today done by dividing the problem into subproblems. In practice this means dividing the airport into zones and distributing these zones among the air traffic controllers. The main focus for each group is the detection and resolution of conflicts between the subset of flights in their zone. Each group of controllers will manage the flights within their zone as best they can, but they have no control over whether or not their decisions are in the other controllers best interest, even less if they help form a good global solution for the overall problem. Our belief is that having computers equipped with optimization technology to assist the human controllers in the decision making, will increase both the capacity at the airport, and the punctuality of the airliners, which in the end are the factors they are measured by.

1.2 Aims and scopes

This thesis aims to show how the airport ground movement problem can be formulated as a minimum cost flow problem to which the network simplex algorithm can be applied. Discussions about conflict resolution and how to incorporate this in order to find a conflict free optimal solution are also presented. As a result, we hope to contribute to airport ground movement research by giving insight into a different solution approach than the ones used in previous research.

1.3 Collaboration with SINTEF

This thesis has been written as part of a vision shared by a group of researchers at SINTEF ICT Oslo whose goal is to design a real time system which will enable individual air traffic control decisions to be based on system-wide considerations. This tool will involve modern heuristic optimization techniques and consider uncertainty, that is balancing the need for responsiveness in time-critical decision making with stability of the schedules over time. This further extends the research currently done by SESAR in the way that while some of SESAR's working packages merely suggests partial integration of existing systems, this approach develops a new concept which *truly* integrates the business trajectory from approach to departure.

1.4 Structure of the thesis

The core of this thesis is organized as follows: Chapter 2 presents an overview of existing research and contributions related to the ground movement at airports, and mentions a related research area. Chapter 3 covers existing theory of linear programming problems, duality and graph theory, as well as some network flow problems and algorithms for solving these. In chapter 4 we introduce some terminology regarding airports and flights, and give a formal formulation of the problem. The chapter also covers how this problem then can be formulated as a linear programming problem. Chapter 5 gives a detailed description of how to ensure a conflict free solution, before we in chapter 6 show how an optimal solution can be found. In chapter 7, we describe the test runs done, and present the results obtained from these runs. Finally in chapter 8 we discuss some future work to be done in this research area.

Chapter 2

Background and related research

This thesis shows how it is possible, given a sequence of arriving and departing flights, to implement the network simplex algorithm in order to solve the airport ground movement problem. As with most things, it is important to begin with an understanding of the problem at hand before introducing a potential solution method. In this chapter we describe this problem, and present some of the related research done by others in this area.

2.1 The airport ground movement problem

The task of managing flights at an airport, or the airport ground movement problem as it is often called, is in principal a routing and scheduling problem. It involves guiding flights on the ground, and getting them to their destinations in a timely manner. The goal is often to reduce the overall travel time of flights, reduce the number of cancelled flights, or increase the overall punctuality of flights, while simultaneously reducing the environmental impact of the industry. It is vital that at any given time no two flights are in conflict with one another. We will describe in more detail what we mean by conflicting flights in section 4.1.4. Various problem descriptions and constraints have been used in previous work related to this problem, depending on which airport modelled. Working with a smaller airport that has fewer active flights during peak hours, one can rely on simpler algorithms for finding the optimal routing as opposed to larger airports where more complex ones are needed.

2.1.1 Routing constraints

Various routing constraints have been investigated such as the extreme point of having all routes predetermined and fixed (Rathinam, 2008)[5], reducing the problem to finding the best possible schedule. Another, slightly more relaxed approach used by Atkin (2008) [6] divides the routes into subsets depending on their properties, and then for each flight chooses a legal route from one of these subsets. The latter seems more reasonable as different types of aircrafts have different turning restrictions and therefore different legal routes.

2.1.2 Flight separation and movement speed

It is obvious that a solution where two or more flights are conflicting is of little or no value. In order to avoid this during taxiing, take-off, and landing, separation constraints are added between the flights. No universal standard for these constraints seems to be in use, but Balakrishnan (2007) [8] used a distance separation of 200 metres between flights. The separation constraints were translated into capacity constraints invoking the separation if the traffic measured increased beyond 1. These constraints were used only for taxiing flights, not on the runway. It was also pointed out that different flights have different separation value depending on the aircrafts properties, however, this was not included in the model as it would alter the structure of the integer program substantially. Ravizza (2013)[9] gave an extensive analysis of how taxi speeds can vary and how this information can be incorporated into the solution.

2.1.3 Objectives

Different studies have had different aims when solving the ground movement problem. Some, (Rathinam, 2008)[5], have focused of minimizing the total taxi time for each flight, whilst Marín and Codina (2008)[10] used a weighted linear objective function to minimize the number of intervention of controllers to solve possible conflicts, the total delay of outgoing traffic, the total time until take-off or final parking, and the total delay of incoming traffic. A side effect of this is of course reduced fuel consumption and environmental impact.

2.2 Sequencing

Ravizza [9] stated that: *The ultimate goal to support airside operations at an airport is to integrate ground movement with other operations.* It is a straightforward observation to see that the solution of the ground movement problem affects the sequence of the departing flights and vice versa. What good is an optimal take-off sequence if it cannot be achieved by the taxiing flights? In the same way, the order of which flights are landing on the runway affects the ground movement solution. And so the ground movement problem and the arrival and departure sequencing problems should ultimately be solved as a single problem.

Whenever a flight takes off or land, a wake vortex which can affect a later flight is created behind that flight. This vortex varies depending on the type and size of the aircraft. In order to get a conflict free sequence on the runway, wake vortex separations are imposed on the arriving and departing flights. Atkin [6] divided the flights at London Heathrow Airport into 3 weight groups and produced 4 tables based on information from NATS listing the different wake vortex separations. In addition, a runway may be used in so called *mixed mode*, that is, it is used for both take-off and landings simultaneously. Bianco (2006)[11] studied the coordination of arrivals and departures in this situation by using dynamic local search heuristic algorithm for the job-shop model.

2.3 Existing models

Atkin, Burke, and Ravizza (2010)[12] gives an overview of the existing models and solution approaches to the ground movement problem. The main two forms involves the development of a MILP (Mixed Integer Linear Program) formulation to which a commercial solver is applied to find the optimal solution, and if the model is formulated so that the solver did not yield a solution within reasonable time, heuristic methods are used. The advantages and disadvantages of both methods are discussed before turning to other solution approaches. It is stated that a comparison between the methods would be desirable, but that several publications did not provide all the information needed to reproduce their results. The need for generic benchmark scenarios in order to both quantify algorithms and encourage future research by those not in contact with an airport is also addressed.

2.4 Related research areas

2.4.1 Job-shop scheduling

The job-shop scheduling problem is a problem where jobs are allocated to different resources in a sequential manner. Usually a finite set of jobs is given to be executed on a finite set of machines or resources, and each job has to undergo a chain of operations in order to be completed. A standard condition is that each operation has an execution time, and each resource can handle only one job at a time. The goal is to find out how to allocate jobs on the resources in order to minimize the overall completion time. In our setting, flights can be treated as jobs, areas on the airport can be treated as resources, and a flights taxi route as the chain of operations.

2.4.2 Complexity

To our knowledge, proof that the simultaneous ground movement problem is NP-hard has yet to be published, but Schüpbach and Zenklusen (2011)[4] showed that a simplified version of the conflict-free vehicle routing problem is NP-hard. However, since the problem description along with the objective function of the ground movement problem varies, some instances which can be solved in polynomial time may exits.

2.5 Conclusions

It is apparent that previous research has utilised objectives and constraints with significant differences. This is a natural result of the differences in airports being focused on. However, it is clear that having standardised instances would be beneficial when comparing the different approaches. Benefits can also be gained from integrating the ground movement problem with other airport operations, especially sequencing of flights. Previous research have mainly focused on MILP formulations, and so a different type of formulation may provide interesting information about both results and the integration of other airport operations.

Chapter 3

Background theory

In this chapter, we present some theory regarding linear programming, graphs, and algorithms that will be useful in later chapters. The first two sections covers linear programming problems and duality, before we in the third present some key concepts on graph theory. Finally, the last two sections presents an overview of two network flow problems and their solution algorithms which are vital to the research presented later.

3.1 Linear programming problems

In this section we look at theory regarding linear programming (LP) problems. The material presented in this section is based on the books by Bertsimas (1997)[14] and Vanderbei (2008)[13]. Proofs of the theorems and corollaries presented can all be found in [14]. We start off by describing what an LP problem is.

3.1.1 Introduction

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.1}$$

A problem on the form (3.1) is called a *linear programming problem*. The vector $\mathbf{x} = (x_1, \ldots, x_n)$ is the *optimization variable* of the problem, The vector $\mathbf{c} = (c_1, \ldots, c_n)$ is the *cost vector*, and together $\mathbf{c}'\mathbf{x}$ forms the *objective function*. The functions $\mathbf{A}\mathbf{x}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$ are the (inequality) *constraint functions*, and $\mathbf{b} = (b_1, \ldots, b_n)$ are the bounds for the constraints. Note that the objective function and the constraints $\mathbf{A}\mathbf{x} \leq \mathbf{b}$ can be written on the form

$$\sum_{i=1}^{n} c_i x_i$$
$$\sum_{i=1}^{n} a_{ji} x_i \leq b_j \qquad \text{for } j = 1, \ldots, m$$

The goal is to find a vector $\mathbf{x}^*$ among all vectors $\mathbf{x}$ that satisfies the constraints so that $\mathbf{c}'\mathbf{x}^* \leq \mathbf{c}'\mathbf{x}$. This vector is then called the *optimal solution* of (3.1).

Any minimization problem can be transformed into a maximization problem by multiplying the objective function with -1. If you want the objective value to stay the same you also have to negate the answer.

$$\text{minimize} \quad \mathbf{c}'\mathbf{x} \quad = \quad -\text{maximize} - \mathbf{c}'\mathbf{x}$$

3.1.2 Standard form and slack variables

An LP problem written on the form (3.2) is said to be on *standard form*.

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.2}$$

We would like to bring all LP problems over on this form in order to solve them. This is done by introducing so called *slack variables* $\mathbf{w}$. We require that these slack variables are nonnegative, and thus (3.1) can be written as

$$\begin{aligned} \text{minimize} \quad & \mathbf{c}'\mathbf{x} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} + \mathbf{w} = \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, \mathbf{w} \geq \mathbf{0} \end{aligned} \tag{3.3}$$

To which we can apply a solution method. Note that the inequality signs can be switched around by multiplying both sides by -1.

3.1.3 Solution set and the optimal solution

We now turn our attention to the solution sets of LP problems, and where in these sets the optimal solution is located (if it exist). The following definitions are useful:

A solution that satisfies all the constraints of an LP problem is called a *feasible* solution. If no such solution exist, the problem is said to be *infeasible*. A set that can be described on the form $\{\mathbf{x} \in \mathbb{R}^n | \mathbf{A}\mathbf{x} \leq \mathbf{b}\}$, where $\mathbf{A} \in \mathbb{R}^{m \times n}$ and $\mathbf{b} \in \mathbb{R}^m$ is called a *polyhedron*. From the discussion in the previous subsection it is easy to see that any feasible solution to an LP problem must be a polyhedron.

We say that a set $S \subset \mathbb{R}^n$ is *bounded* if there exist a constant K so that the absolute value of every component of every element of S is less than or equal to K. If we let P be a polyhedron, a vector $\mathbf{x} \in P$ is called an *extreme point* of P if we cannot find two vectors $\mathbf{y}, \mathbf{z} \in P$, both different from $\mathbf{x}$, and a scalar $\lambda \in [0, 1]$, such that $\mathbf{x} = \lambda\mathbf{y} + (1 - \lambda)\mathbf{z}$.

Using these concepts, the following theorem tells us the location of the optimal solution of an LP problem.

Theorem 3.1.1. *Given an LP problem of minimizing $\mathbf{c}'\mathbf{x}$ over a nonempty polyhedron P. Then, either the optimal cost is equal to $-\infty$ or there exist an optimal solution which is an extreme point.*

In other words, when solving an LP problem, one of the following possibilities will occur:

1. There is an optimal solution.
2. The problem is unbounded and the optimal cost is $-\infty$ for minimization problems, or $+\infty$ for maximization problems.
3. The problem is infeasible, i.e. it has no solution.

3.2 Duality

In this section, we start with an LP problem which we call the *primal problem*, and introduce another LP problem, called the *dual*. Duality theory describes the relation between these two problems and is a powerful theoretical tool in linear programming. The theory in this section is based on [14].

3.2.1 The dual problem

Associated with every LP problem is another called its dual. For instance, the dual of (3.1) is

$$\begin{array}{ll} \text{maximize} & \mathbf{y}'\mathbf{b} \\ \text{subject to} & \mathbf{y}'\mathbf{A} \leq \mathbf{c}' \\ & \mathbf{y} \leq \mathbf{0} \end{array} \tag{3.4}$$

PRIMAL	minimize	maximize	DUAL
	$\geq b_i$	≥ 0	
constraints	$\leq b_i$	≤ 0	**variables**
	$= b_i$	free	
	≥ 0	$\leq c_j$	
variables	≤ 0	$\geq c_j$	**constraints**
	free	$= c_j$	

Table 3.1: Relations between primal and dual variables and constraints.

Here, $\mathbf{y}$ is known as the *dual variables*. Table 3.1 shows the relations between the primal and dual variables and constraints. If the primal had been a maximization problem, it can always be converted to an equivalent minimization problem, and then the dual can be formed according to the rules in the table.

Taking the dual of the dual problem returns the primal problem. To see this, we convert (3.4) to a minimization problem and change direction of the inequality:

$$\begin{array}{ll} -\text{ minimize} & -\mathbf{y}'\mathbf{b} \\ \text{subject to} & -\mathbf{y}'\mathbf{A} \geq -\mathbf{c}' \\ & \mathbf{y} \leq \mathbf{0} \end{array}$$

Following the rules, taking the dual of this returns

$$\begin{array}{ll} -\text{ maximize} & -\mathbf{c}'\mathbf{x} \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{b} \\ & \mathbf{x} \geq \mathbf{0} \end{array}$$

Which converted into a minimization problem identical to (3.1).

3.2.2 Weak duality

The feasible solutions of the dual problem provides lower bounds on the primal objective function value. This result is true in general and is known as the *weak duality theorem*.

Theorem 3.2.1. *(Weak duality theorem) If* $\mathbf{x}$ *is a feasible solution to the primal problem and* $\mathbf{y}$ *is a feasible solution to the dual problem, then*

$$\mathbf{y}'\mathbf{b} \leq \mathbf{c}'\mathbf{x}$$

In the same way, the feasible solutions of the primal problem provides upper bounds on the dual problem. Improving the solutions then tightens the gap between the optimal objective function value of the primal and dual problem. It is worth noting that if the primal had been a maximization problem, the feasible solutions to the dual problem would provide *upper* bounds on the primal objective function value and vice versa. We see that if one problem is unbounded then the other is *infeasible*, that is, no feasible solution to the problem exist. An important corollary can be derived from the weak duality theorem.

Corollary 3.2.2. *Let* $\mathbf{x}$ *and* $\mathbf{y}$ *be feasible solutions to the primal and the dual, respectively. Suppose that* $\mathbf{y}'\mathbf{b} = \mathbf{c}'\mathbf{x}$*. Then* $\mathbf{x}$ *and* $\mathbf{y}$ *are optimal solutions to the primal and dual, respectively.*

3.2.3 Strong duality

The next result is essential for linear programming, and the central result of LP duality. It states that if there exist a solution to an LP problem, then there exist a solution to the dual of this problem, and the respective optimal costs are equal.

Theorem 3.2.3. *(Strong duality theorem) Given a linear programming problem, if the primal problem has an optimal solution* $\mathbf{x}$*, then the dual problem has an optimal solution* $\mathbf{y}$ *such that*

$$\mathbf{y}'\mathbf{b} = \mathbf{c}'\mathbf{x}$$

A final thing to mention about duality is the relation between the primal and dual optimal solutions called the *complementary slackness* conditions. Which can be useful for instance if you want to find an optimal dual solution when only an optimal primal solution is known.

Theorem 3.2.4. *(Complementary slackness) Suppose* $\mathbf{x} \in \mathbb{R}^n$ *and* $\mathbf{y} \in \mathbb{R}^m$ *be feasible solutions to the primal and the dual problem, respectively. Let* $\mathbf{w} \in \mathbb{R}^m$ *be the primal slack variables, and let* $\mathbf{z} \in \mathbb{R}^n$ *be the dual slack variables. Then* $\mathbf{x}$ *and* $\mathbf{y}$ *are optimal solutions for their respective problems if and only if*

$$\begin{aligned} x_j z_j &= 0 \qquad \text{for } j = 1, \ldots, n \\ w_i y_i &= 0 \qquad \text{for } i = 1, \ldots, m \end{aligned}$$

3.3 Graph theory

3.3.1 Introduction to graphs

Graphs are mathematical structures consisting of points and lines connecting two and two of these points together. In this thesis we will, unless otherwise stated, define the points as nodes and the lines as edges. Many situations that arise every day can be modelled as graphs, a few examples being road networks, scheduling of tasks distributed among a workforce, and telephone networks. In 2012 Amazon.com bought Kiva Systems, a developer of mobile robotic fulfillment systems for $775 million (Boston Globe, 2013)[15]. Within a year they had mapped out an entire warehouse as a graph, and deployed over 1300 robotic workers. These workers travel along the edges making turns on the nodes while carrying merchandise ordered by customers to their designated spots. Another use of graphs, recently implemented by various social media sites, is to map each individual users social network. This is done by representing every user as a node, and having an edge connect two nodes if the two users are for example friends or following each other. These sorts of graphs are then used for marketing and advertising purposes.

3.3.2 Terminology

This subsection is based on the book by Ahuja, Magnanti, and Orlin (1993)[7]. Here we look at some of the terms and phrases often used when discussing graphs and networks which will be used in later chapters.

A *graph* $G = (N, E)$ is a representation of a finite set of *nodes* N, and a finite set of *edges* E whose elements are unordered pairs of distinct nodes (i, j). In some cases the edges have a specified direction, in which case they are referred to as *directed edges*. A directed edge e_{ij} is an edge where the nodes i and j are ordered, and we say that i is the start node while j is the end node of the edge. We also say that the edge is *outgoing* from i and *incoming* to j. It is worth noting that while we in this thesis use the term directed edges, they are in the literature sometimes referred to as *arcs*. When the graph consists of directed edges, it is called a *directed graph* or a *digraph*, see figure 3.1a. In this thesis we will be working exclusively with directed graphs. Given another graph $G' = (N', E')$ where $N' \subseteq N$ and $E' \subseteq E$, that is, all the nodes and edges in G' are also contained in G, we say that G' is a *subgraph* of G.

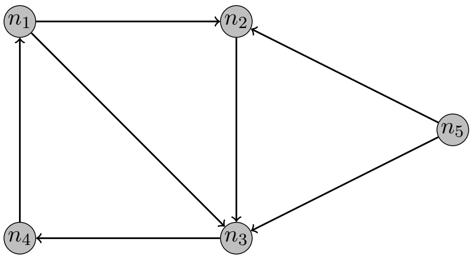
Two nodes are *adjacent* to each other if they are the endpoints of the same edge, and the edge and nodes are then said to be *incident*. By ordering a list of such nodes $(n_1, n_2, \ldots, n_k)$ we get a *path* in the graph, provided that each adjacent pair of nodes is connected by an edge. If we can create an alternating sequence of nodes and edges
$(n_0, e_1, n_1, e_2, \ldots, e_k, n_k)$ where $e_i = (n_{i-1}, n_i) \in E$, or $e_i = (n_i, n_{i-1}) \in E$ for $i = 1, \ldots, r$, or in other words a walk without any repetition of nodes, we get what is known as a *walk*. See figure 3.1b. An oriented version of a walk, in the sense that for any two consecutive nodes n_i and n_{i+1}, $e_i = (n_i, n_{i+1}) \in E$, is called a *directed walk*. In other words, a directed walk has no backward edges, see figure 3.1c. The connection between paths and walks hold for the directed versions as well, that is a *directed path* is a directed walk without any repetition of nodes.

Two nodes i and j are *connected* if there exist at least one path between them, and if every node in a graph is connected we say that the graph is connected or that we have a *connected graph*, otherwise the graph is said do be *disconnected* or *disjoint*. In addition, if this path is directed, the graph is *strongly connected*, see figure 3.1d.

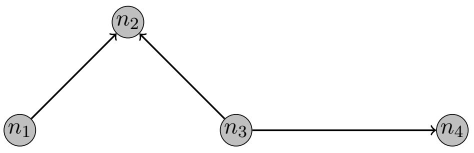
A path $(n_1, n_2, \ldots, n_k)$ together with the edge (n_k, n_1) or (n_1, n_k), that is, a path where the first and last node coincides, generates a *cycle*. See figure 3.1e. If in addition, the sum of the weights on the edges in the cycle is negative, we have what is called a *negative cycle*. If a graph does not contain any cycles, it is said to be *acyclic*.

A graph that is both connected and acyclic, is know as a *tree*. By removing one of the edges from the tree, the graph is no longer connected and no longer a tree, however, in doing so we are left with two *subtrees*, and so a subtree is a connected subgraph of a tree. Trees are an important theoretical concept that arises frequently in many algorithms for solving network flow problems, and they are an essential part of the algorithm we will look at in subsection 3.5.1 and chapter 6. We will see later that every tree contains at least two nodes that only have one incident edge. These nodes are often called *leaf nodes*. We say that these nodes have *degree* 1, while every other node in the tree has degree at least 2. Another node that has a special name, is the node that is chosen as a starting node of an algorithm, this node is referred to as the graphs *root node*.

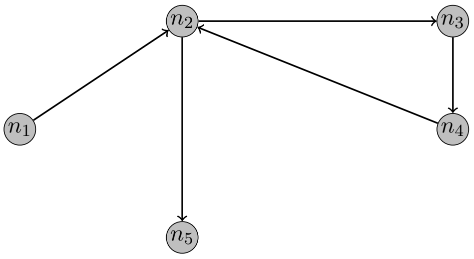
A tree $\mathcal{T}$ containing all the nodes and $n-1$ edges where n is the number of nodes of a graph G is called a *spanning tree*. See figure 3.1f. The edges belonging to $\mathcal{T}$ is called *tree edges* while those not belonging to $\mathcal{T}$ are called *nontree edges*. Solutions of some LP problems, such as the minimum cost flow problem described in the next section are spanning trees.



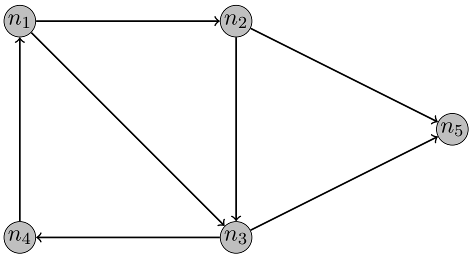
(a) Directed graph



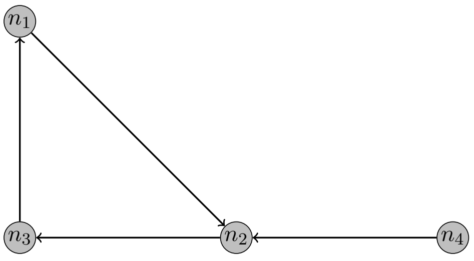
(b) Walk



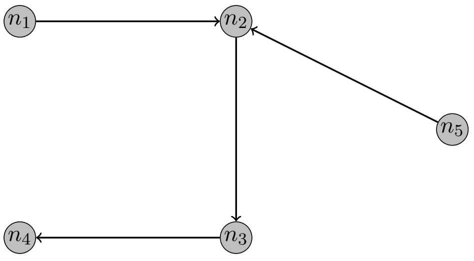
(c) Directed walk



(d) Strongly connected graph



(e) Cycle



(f) Spanning tree

Figure 3.1

3.4 Network flow problems

In this section we will look at some problems called *network flow problems*. While these types of problems arises in many instances such as the assignment problem where you pair off two disjoint sets one to one, the maximum s-t flow problem where you find a maximum flow from a node s to a node t, or generalized flow problems where flow might be generated or consumed along the edges, we will limit our discussion to the, perhaps, simplest instance namely the *shortest path* problem, which will play an important role in algorithms used in later chapters, and the *minimum cost flow* problem which is essentially the problem we want to solve when looking at an airport. For a detailed description of the other instances and more, we suggest the book by Ahuja, Magnanti, and Orlin (1993)[7]. The theory in this section is based on Vanderbei (2008)[13].

3.4.1 The minimum cost flow problem

The minimum cost flow problem can be described like this: Find the most efficient way of transporting a commodity through a network while satisfying the demands at nodes in the graph with the supplies from other nodes in the graph. The commodity can be anything from information to physical goods, and so for simplicity, we will just call it flow. The problem can easily be related to real life tasks, such as how to transport merchandise from a warehouse to retailers, how to direct cars through a road network, and as we shall see, how to guide flights along taxi routes on an airport.

Assumption

Let G be a directed graph defined by a set N of n nodes and a set E of m directed edges. Associated with each node i $\in N$ is a number b_i representing that nodes *supply* or *demand* for the flow. If $b_i > 0$ we say that i is a *supply node*, if $b_i < 0$ it is a *demand node,* and if $b_i = 0$ we say that it is a *transshipment node*. Instead of having to deal with both the supply and the demand term, we will in the proceeding stick with demand, noting that when the demand is negative, this is equivalent to supply. An important assumption is that no flow can enter or leave the graph in any way, and so we require that the total sum of demand within the graph must be zero.

$$\sum_{i \in N} b_i = 0$$

Constraints

The variables we want to decide, are the flows along each edge $e_{ij} \in E$ denoted x_{ij}. So for each node $i \in N$ (3.5) represents the total flow going out of i, and (3.6) represents the total flow coming into i.

$$\sum_{j:e_{ij} \in E} x_{ij} \tag{3.5}$$

$$\sum_{j:e_{ij} \in E} x_{ji} \tag{3.6}$$

In order to comply with the demand at each node, the total flow out of the node minus the total flow in to the node must be equal to the demand of the node. This is ensured by (3.7) which is known as the *flow balance constraints*:

$$\sum_{j:e_{ij} \in E} x_{ij} - \sum_{j:e_{ij} \in E} x_{ji} = b_i \qquad \text{for all } \ i \in N \tag{3.7}$$

In some instances the minimum cost flow problem requires capacity constraints on the edges saying that the amount of flow needs to lie between some lower and upper bound, $l_{ij} \leq x_{ij} \leq u_{ij}$. However, for the problem discussed in this thesis we will just require the flows to be nonnegative

$$x_{ij} \geq 0 \qquad \text{for all } \ e_{ij} \in E \tag{3.8}$$

Objective function

Associated with each edge $e_{ij} \in E$ is a *cost*, denoted c_{ij}, of transporting one unit of flow from node i to node j along this edge. As previously mentioned, the goal is to find the most efficient way, that is, the cheapest way, of sending the flow through the network. This translates to the following objective function:

$$\text{minimize} \sum_{e_{ij} \in E} c_{ij} x_{ij}$$

Problem formulation

Combining the objective function with the constraints (3.7)-(3.8), the problem can now be formulated as:

$$\begin{aligned} &\text{minimize} && \sum_{e_{ij} \in E} c_{ij} x_{ij} && \\ &\text{subject to} && \sum_{j:e_{ij} \in E} x_{ij} - \sum_{j:e_{ij} \in E} x_{ji} = b_i && \text{for all } i \in N \\ & && x_{ij} \geq 0 && \text{for all } e_{ij} \in E \end{aligned}$$

Written in matrix notation we get:

$$\begin{aligned} &\text{minimize} && \mathbf{c}'\mathbf{x} \\ &\text{subject to} && \mathbf{A}\mathbf{x} = \mathbf{b} \\ & && \mathbf{x} \geq \mathbf{0} \end{aligned} \tag{3.9}$$

We see immediately that (3.9) is on the LP *standard form*

Figure 3.2 shows an example of the minimum cost flow problem with 5 nodes formulated as a graph. The numbers next to the nodes are the demands of each node while the numbers next to the edges are the costs of sending one unit of flow along that edge. The solution here is to send one unit of flow along e_{n_1,n_2}, four units of flow along e_{n_1,n_3}, two units along e_{n_3,n_5}, and three units along e_{n_3,n_4}, with a total cost of 18.

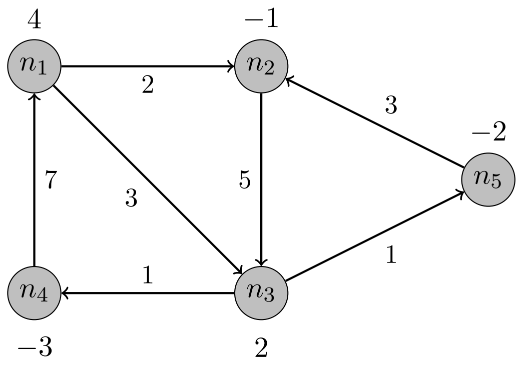


Figure 3.2: The minimum cost flow problem

Lastly, we state the dual of the minimum cost flow problem:

$$\begin{aligned} &\text{maximize} && \sum_{i\in n} b_i y_i && \\ &\text{subject to} && y_j - y_i + z_{ij} = c_{ij} && \text{for all } e_{ij}\in E \\ &&& z_{ij} \geq 0 && \text{for all } e_{ij}\in E \end{aligned} \tag{3.10}$$

3.4.2 Shortest path

As we mentioned earlier, the shortest path problem is perhaps the simplest instance of all network flow problems. It involves finding the shortest path from one point to another. The length of a path can be seen as the distance from the starting point to the end point, the time it takes to travel, the fuel consumption or the reliability. One can easily see how this model can be applied to finding the shortest taxi route from the gate to the runway on an airport. Solutions of the shortest path problem are used for instance by GPS systems to give users recommended driving routes, and are also a central part of more complex problems, such as *the traveling salesman* problem. Again, see [7] for a detailed description. More formally, the problem can be formulated as: Given a directed graph G and two distinct nodes $s, t \in N$. Assuming that each edge has some length c_{ij}, find the shortest path from the source node s to the sink node t.

In order to relate this problem to the minimum cost flow problem, we imagine that we want to send one unit of flow from s to t. This means that s has a demand of -1, t has a demand of 1, and all other nodes are transshipment nodes. In other words, $b_s = 1$, $b_t = -1$, and $b_i = 0$ for all $i \in N \setminus \{s,t\}$. We now see that the shortest path problem is just an instance of the minimum cost flow problem, and can then be formulated as:

$$\begin{aligned} &\text{minimize} && \sum_{e_{ij}\in E} c_{ij} x_{ij} && \\ &\text{subject to} && \sum_{j:e_{ij}\in E} x_{ij} - \sum_{j:e_{ij}\in E} x_{ji} = 1 && \text{for } i = s \\ &&& \sum_{j:e_{ij}\in E} x_{ij} - \sum_{j:e_{ij}\in E} x_{ji} = 0 && \text{for } i \in N\backslash\{s,t\} \\ &&& \sum_{j:e_{ij}\in E} x_{ij} - \sum_{j:e_{ij}\in E} x_{ji} = -1 && \text{for } i = t \\ &&& x_{ij} \geq 0 && \text{for all } e_{ij}\in E \end{aligned}$$

.

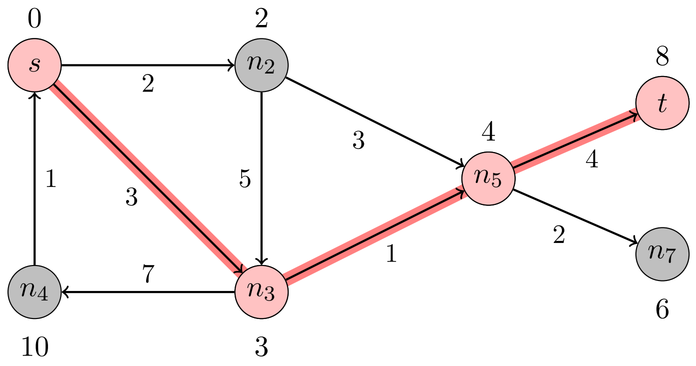


Figure 3.3: Shortest path

In figure 3.3 we see an example of the shortest path problem with the highlighted solution of the shortest path from s to t which has length 8. The distances to the other nodes are also shown.

3.5 Network flow algorithms

Having described some of the linear programming problems, it is time to look at how to solve them. Today, there exist several algorithms which solves the shortest path problem and the minimum cost flow problem. In this section we look at two of the most common ones, the *Bellman-Ford algorithm* and the *Network simplex method*. The theory in this section is based on the books by Ahuja, Magnanti, and Orlin (1993) [7] and Vanderbei (2008)[13].

3.5.1 The Bellman-Ford algorithm

The Bellman-Ford algorithm, named after two of its developers, Richard Bellman and Lester Ford, Jr. was published for the first time in the late 1950s. While the algorithm is less efficient than some other shortest path algorithms like *Dijkstra's algorithm*, $\mathcal{O}(|N| \cdot |E|)$ vs. $\mathcal{O}(|E| + |N| \log|N|)$, (using heaps), it has the very nice property, as opposed to Dijkstra's, that it works on graphs with negative costs. This property will be important later on when we look at penalties for flights deviating from their schedule. The problem it solves is this: Given a graph G and a root node s, find the shortest path from s to all other nodes in N.

In order to solve this, the algorithm takes in a graph represented as a list of nodes and a list of edges, and returns two lists representing the distance and previous node of each node. It starts off by setting the distance to all nodes to infinity, except the root node which is set to zero. Then, for all nodes v it loops through all edges e_{uv} and checks if (3.11) is satisfied.

$$dist(v) < dist(u) + c_{uv} \tag{3.11}$$

Here $dist(i)$ represent the distance from s to node i. If this is not satisfied, $dist(v)$ is set to be equal to $dist(u) + c_{uv}$, and the parent of v is set to u. While doing this, the algorithm keeps track of the number of iterations, if this number exceeds $|N| \cdot |E|$ The graph contains a negative cycle, and the problem is infeasible. Here is the resulting code:

```
input: a graph $G$ and a root node $s$
output: a list distance and a list parents

Maximum iterations = $|N| * |E|$
Iterations completed = $0$

% Step 1: Set distances
for each node $n$
        distance[$n$] = infinity
distance[$s$] = $0$

%Step 2: Finding shortest path
for each node $n$
        for each edge $e_{uv}$ with weight $c_{uv}$
                if distance[$v$] > distance[$u$] + $c_{uv}$
                        distance[$v$] = distance[$u$] + $c_{uv}$
                        parents[$v$] = $u$
                        Increase Iterations completed by $1$
                if Iterations completed > Maximum iterations
                    error: "G contains a negative cycle."
return distance and parents
```

The network simplex algorithm

The network simplex algorithm is based on the famous *simplex method* developed by George Bernard Dantzig who published it in 1947 (see Vanderbei, 2008 [13] for a detailed description), and is used to solve the minimum cost flow problem described in the previous chapter. It consists of two parts, the primal network simplex algorithm, and the dual network simplex algorithm, which are used when the problem is primal feasible, and dual feasible, respectively.

Given a connected graph G, we choose a root node s, and find a spanning tree $\mathcal{T} \subset E$. We then proceed to calculate the flows x_{uv}, the dual variables y_u starting from $s = 0$, and all the dual slacks z_{uv} for all $u, v \in N$, according to the constraints of the problem. It follows from the complementary slackness conditions that for all edges $e_{uv} \in \mathcal{T}$, $z_{uv} = 0$, and for all $e_{uv} \notin \mathcal{T}$, $x_{uv} = 0$. If for all $e_{uv} \in \mathcal{T}$, $x_{uv} \geq 0$, the initial solution is primal feasible, and we apply the primal network simplex algorithm. If not, but for all $e_{uv} \notin \mathcal{T}$, $z_{uv} \geq 0$, the initial solution is dual feasible, and we apply the dual network simplex algorithm.

The primal network simplex algorithm

If we are in the situation where the solution is primal feasible but not dual feasible, we feed all the information calculated above into the primal simplex algorithm. The algorithm then starts looking at edges $e_{ij} \notin \mathcal{T}$ to see if adding this edge to $\mathcal{T}$, and removing an edge $e_{uv} \in \mathcal{T}$ from $\mathcal{T}$, results in a directing of flow which is at a lower cost than before. However, just randomly searching for edges to add and remove is very inefficient, and so rules from the simplex method are applied in order to ensure efficiency:

- The edge e_{ij} entering $\mathcal{T}$ must be an edge which is dual infeasible, that is, $z_{ij} < 0$.

When e_{ij} is added to $\mathcal{T}$, it generates a cycle. The flow on the edges in this cycle have to change in order to accommodate the increased flow on e_{ij}. The flows going in the same direction in the cycle as e_{ij} is increased, while the flows going in opposite direction is decreased. Continuing decreasing these flows, eventually one becomes zero. The edge belonging to this flow is the one which will be removed from $\mathcal{T}$. This gives us the following rule:

- The edge e_{uv} being removed from $\mathcal{T}$ must be oriented along the cycle in opposite direction of e_{ij}, and of all those edges, it must have the smallest flow.

The next step is to update the variables. As mentioned, some flows in the cycle is increased while others decrease. The amount of this change is equal to x_{uv}.

In order to update the dual and dual slack variables, we look at what would happen to the graph if we removed e_{uv} from $\mathcal{T}$ without adding e_{ij}. This would split $\mathcal{T}$ into two disjoint subtrees $\mathcal{T}'$ and $\mathcal{T}''$. One of these subtrees, let us say $\mathcal{T}'$, will contain the root node. Recalling that the calculation of dual variables started from the root node, it is clear that the dual variables of nodes contained in $\mathcal{T}'$ remains the same. The dual variables in $\mathcal{T}''$ will however change, and we have the following rule:

- If e_{ij} crosses from $\mathcal{T}'$ to $\mathcal{T}''$, all dual variables in $\mathcal{T}''$ is increased by z_{ij}. Otherwise, decrease these dual variables by the same amount.

The final variable update needed are the dual slacks. Only the edges bridging between $\mathcal{T}'$ and $\mathcal{T}''$ (in either direction) will have their dual slacks changed. The reason for this is that they are the only ones where only one of the incident nodes has had its dual value changed. We have the following rule:

- The dual slack variables corresponding to edges bridging between the subtrees in the same direction as e_{ij} is decreased by the old dual slack of e_{ij}. Those that correspond to edges bridging in the opposite direction is increased by the same amount.

After all the variables have been updated, e_{ij} is added to $\mathcal{T}$ and e_{uv} is removed from $\mathcal{T}$.
The algorithm has now completed its first iteration. This will be repeated until either we are back at the previous visited tree meaning we have a degenerate solution[13], or there are no more dual infeasible edges to be added to $\mathcal{T}$, and we have a primal dual feasible solution, meaning we have an optimal solution. In order to obtain the objective value, all that is needed is to multiply the costs of the edges in $\mathcal{T}$ with their respective flows. Below is the code for the primal network simplex algorithm:

```
input: a graph $G$, a root node $s$, and a tree $T$
output: a tree $T$

$e_{ij}$ = edge with minimum dual slack

while $e_{ij}$.DualSlack $<$ $0$
        e$_{uv}$ = edge in cycle in opposite direction of $e_{ij}$ with minimum flow
        Update flows in cycle
        Remove $e_{uv}$ from $T$
        Update dual variables
        Update dual slack variables
        Add $e_{ij}$ to $T$
        Update $e_{ij}$

return $T$
```

The dual network simplex algorithm

We now turn our attention to the situation where the initial solution is dual but not primal feasible. This means that for at least one edge $e_{uv} \in \mathcal{T}$, $x_{uv} \leq 0$. The basic idea of the dual network simplex algorithm is to remove this edge from $\mathcal{T}$, and look for an edge $e_{ij} \notin \mathcal{T}$ to take its place. Again the algorithm follows certain rules to ensure efficiency.

- The edge e_{uv} being removed from $\mathcal{T}$ must be an edge which is primal infeasible, that is, $x_{uv} < 0$.

Observe that when removing e_{uv} from $\mathcal{T}$, $\mathcal{T}$ is split up into two disjoint subtrees, which we denote $\mathcal{T}'$ and $\mathcal{T}''$. It is clear that in order to obtain a feasible solution again, e_{ij} must be bridging across these two subtrees. First we consider the possibility that e_{ij} is bridging in the same direction as e_{uv}. This means that in order for x_{ij} to be increased, x_{uv} must be decreased. This leads to x_{uv} not being increased to zero, and e_{uv} won't be able to leave $\mathcal{T}$. And so it is clear that e_{ij} must bridge in the opposite direction of e_{uv}. When an edge is added to $\mathcal{T}$, its dual slack variable drops to zero. Also, all the other edges bridging in the same direction as this edge, will have their dual slack reduced by the same amount. And so, in order to ensure dual feasibility, of all the edges bridging in the opposite direction of e_{uv}, e_{ij} must have the smallest dual slack, giving us the following rule:

- The entering edge e_{ij} must bridge across the two subtrees $\mathcal{T}'$ and $\mathcal{T}''$ in the opposite direction from the leaving edge e_{uv}, and among all such edges, it must have the smallest dual slack.

Having decided both the leaving and entering edge, the new spanning tree has been determined. All that is left, is to update the variables. This is done according to the same rules as in the primal network simplex algorithm. The first iteration is now complete, if there are more edges with negative flow, the algorithm will repeat itself until either we are back at a previous spanning tree, meaning we have a degenerate solution, or there are no more primal infeasible edges, and we have found the optimal solution.

3.6 Minimum cost flow sensitivity analysis

In this section we take a closer look at the sensitivity analysis of the minimum cost flow problem when a new constraint is added to the dual problem. The theory here is based on Bertsimas and Tsitsiklis (1997)[14].

Given our primal/dual problem pair for the graph G:

$$\begin{aligned} \text{maximize} \quad & c'x & \qquad \text{minimize} \quad & -t'b \\ \text{subject to} \quad & Ax = -b & \qquad \text{subject to} \quad & t'A \leq c' \\ & x \geq 0 & & \end{aligned}$$

Where A is the node-edge incidence matrix of G. Let $\mathbf{B}$ be the optimal basis corresponding to the optimal solution $\mathbf{x}^*$. Here is what happens to our problem when another inequality constraint is introduced in the dual. The dual problem becomes

$$\begin{aligned} \text{minimize} \quad & -\mathbf{t}'\mathbf{b} \\ \text{subject to} \quad & \mathbf{t}'\mathbf{A} \leq \mathbf{c}' \\ & \mathbf{t}'\mathbf{A}_{n+1} \leq c_{n+1} \end{aligned}$$

The effect on the primal problem can be seen by taking the dual again, which results in this primal problem

$$\begin{aligned} \text{maximize} \quad & \mathbf{c}'\mathbf{x} + c_{n+1}x_{n+1} \\ \text{subject to} \quad & \mathbf{A}\mathbf{x} + \mathbf{A}_{n+1}x_{n+1} = -\mathbf{b} \\ & \mathbf{x} \geq \mathbf{0}, x_{n+1} \geq 0 \end{aligned}$$

We see that adding an inequality to the dual is equivalent to introducing a new variable to the primal. This follows the negative transpose property [13] of $\mathbf{A}$.

The question is, will this affect the solution? Or in other words, is the current basis still optimal? Note that $(\mathbf{x}, x_{n+1}) = (\mathbf{x}^*, 0)$ is still a basic feasible solution to the new problem associated with the basis $\mathbf{B}$, and thus we need only check the optimality conditions. For the basis $\mathbf{B}$ to remain optimal, w ehave from [14] that it is necessary and sufficient that the cost change of x_{n+1} is nonpositive, that is,

$$\bar{c}_{n+1} = c_{n+1} - \mathbf{c}'_B \mathbf{B}^{-1} \mathbf{A}_{n+1} \leq 0$$

Using what we know from the strong duality theorem (see proof [14]) we write the condition as

$$\bar{c}_{n+1} = c_{n+1} - \mathbf{t}' \mathbf{A}_{n+1} \leq 0$$

If this is satisfied, $(\mathbf{x}^*, 0)$ is an optimal solution to the new problem. However, if the condition is not satisfied, $(\mathbf{x}^*, 0)$ might not be the optimal solution. In this case we apply the primal network simplex algorithm on the new problem, starting from the current basis $\mathbf{B}$. Usually, this approach leads to an optimal solution with a small number of iterations, and it is usually faster than solving the new problem from scratch.

If we look at the dual problem, what this says is that, if by adding a new edge we obtain some new way of pushing flow through the graph so that the dual variables decrease, that is, the cost increases, then the basis $\mathbf{B}$ is not optimal for the new problem. We let this new edge enter the basis and remove an edge according to the rules in the primal network simplex algorithm. On the airport, this corresponds to the situation where two flights want to access the runway at the same time.

Chapter 4

Modelling airport flight events as a graph

In this chapter we present some terminology about flights and airports, and formulate the problem we want to solve. We then see how this relates to our discussion in the previous chapter, before we introduce what we call the events graph which lays the foundation for our solution algorithm.

4.1 Description and terminology

4.1.1 Airports

The surface of an airport can be represented as a graph $G = (P, S)$, where P is a set of points and S is a set of directed segments joining these points together. Figure 7.3 shows how this is done for Stockholm Arlanda Airport. In this thesis we will consider the airport to have one operative runway, but the problem can be extended to include multiple runways. The runway will be represented by two adjacent points, the *runway entry point* and the *runway exit point*, and an incident directed segment. As the names suggests, flights enter the runway at the entry point, either by landing or taxiing in, and leave the runway at the exit point, either by taking off or taxiing off. We will assume that the flights land and take off in the same direction, and so the entry and exit points will be fixed for all flights. However, this can easily be extended so that each flight has its own entry and exit point. A runway used in this fashion is said to be in *mixed mode*. The term *taxiway* is used when describing the surface on which the flights travel.

There are two types of points $p \in P$. If at p, flights are allowed to stop and wait for their taxi route to clear of traffic, p is called a *holding area* or *holding point*, while the points where f is not allowed to stop are known as *pass through points*. If p is adjacent to more than two segments, it is called an *intersection*. Note that a point may be both a holding area and an intersection, or a pass through point and an intersection.

4.1.2 Flights

Associated with every flight f, there is a vital set of information used by air traffic controllers to handle the flow of traffic. Below follows some definitions of this information that are essential when modelling the events of each flight.

When moving around on the airport, the path of points and segments a flight f travels along, is called the *taxi route* of f. In this thesis, both the runway and the gates will be included in the taxi route. If the flight f starts off at a gate, and ends in the air, f is called a *departure* or *departing* flight. Otherwise, it is called an *arrival* or *arriving* flight. Flights merely traveling on the taxiway i.e. on its way to maintenance will not be considered. Given that f is a departure, f is said to have a *target off-block time*. That is the time when f will be ready (desires) to leave the gate, and start taxiing. The target off-block time is given in advance by a schedule, but may change as the schedule are updated, for instance when there are missing passengers. The actual time at which f leaves the gate is referred to as the *actual off-block time* of f. This is the moment when f is starts moving along its taxi route. As with the off-block times, given that f is a departure, the *target take-off time* and the *actual take-off time* of f represent the time at which f desires to take off, and when it actually takes off. The target take-off time however is usually given as a time window, which we refer to as the *departure window* of f.

Looking at an arriving flight g, g is said to have a *desired landing time*, and an *actual landing time*. The desired landing time is given in advance, and is often a time window bought from the airport by airliners, referred to as the *arrival window* of g. Where departing flights have off-block times, g has *in-block* times. The *target in-block time* is the time at which g would like to park at the gate, while the *actual in-block time* is the actual time at which g parks. A final thing to mention are the *total taxi times*. For departing flights, this is the time difference between actual off-block time and actual take-off time, while for arriving flights is the time difference between actual landing time and actual in-block time. As mentioned in chapter 2, previous research has focused on minimizing the total of this for all flights.

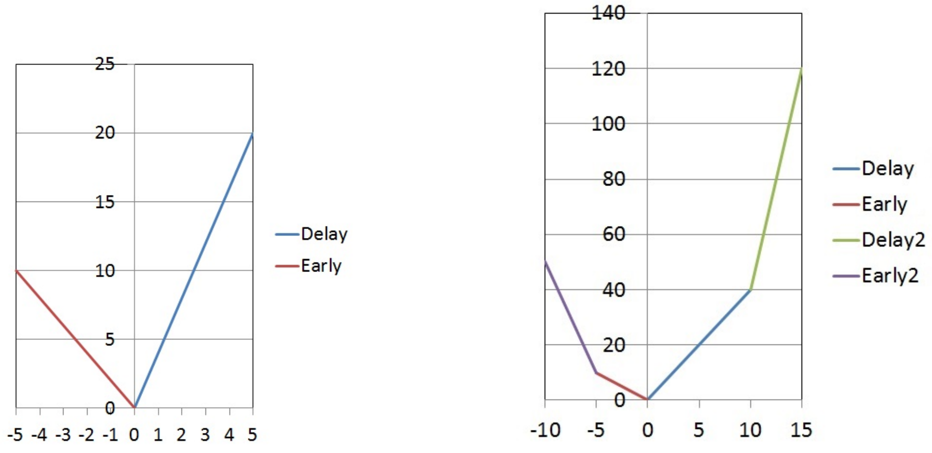
4.1.3 Deviations

After a flight has left the system, either by taking off, or parking at its gate, we need to determine whether or not that flight has deviated from its schedule, that is whether it is early or late compared to the original plan. To simplify the discussion, given a flight f, we use the term *exit time* as a common name for both in-block time and take-off time. We also say that a flight *deviates* from its schedule if there is a difference between its target exit time, and its actual exit time. This deviation can be divided in two depending on whether or not the actual exit time of a flight is later than the target exit time, in which case the flight is *late* or *delayed*, or if it is earlier, in which case the flight is *early*.

A flight's deviation is penalized by a cost for each time unit it deviates from the original plan. There are different ways to represent these costs, for instance there might be a time window for which it is acceptable that the flight deviates i.e. the cost is zero. However, in this thesis we will assume that the costs are linear and starting from zero deviation. Later we will see that this can easily be adapted. We will however allow the marginal cost of being early differ from the marginal cost of being delayed, typically being delayed is penalized harder than being early, see figure 4.1a. This is based on the fact that the points of an airport are a shared resource of the flights, and so it is better for a departure to be sent off early than for it to occupy a point longer and thus affecting other flights on the taxiway. One might think that since the points are a shared resource, would it not be better to send off the flights as early as possible instead of penalizing it? However, the optimal would be for all flights to be able to exit at the desired time, and hence a small cost for being early is added. This can be seen as a penalty for having too many flights active on one part of the taxiway at a given time, making the air traffic controllers job more difficult.

Piecewise linear costs

According to [6], the Central Flow Management Unit of Eurocontrol in Brussels assigns a 'Calculated Time Of Take-off' to a flight which will enter congested airspace or go to congested airports. The intention is to smooth the traffic flow by limiting the times at which a flight can enter these congested areas. Flights cannot take off more than five minutes before this calculated time or more than ten minutes after the calculated time, giving the fifteen minute *take-off window* or *time-slot*. To comply with this, an additional penalty for taking off or parking outside this window can be added. This penalty is an increased cost added to the existing cost of deviation. One could force the flight to be cancelled, but we have modeled it just as an increase, still allowing flights to take off or park, see figure 4.1b.



(a) Deviation costs

(b) Piecewise linear convex costs

Figure 4.1: Deviation costs

4.1.4 Sequence and conflicts

The term *sequence of flights* or just *sequence*, is used about the order of which the flights accesses the runway, and the problem of finding such a sequence is known as the *runway sequencing problem*. This sequence may be fixed, or be subject to change depending on the objective of the problem. Air traffic controllers typically want little to no change in the sequence as this increases the communication needed to aircrafts, and thus the workload of each controller. However, keeping the sequence fixed, may lead to increased delay of each flight, as one flight's tardiness will affect the other flights.

We define a *conflict* as an event which occurs when two flights tries to access the same point on the airport within a given time window. Solving a conflict simply means deciding which of the flights goes first, and we say that this flight has *precedence* over the other flight. The decision taken by the air traffic controllers is based on various criteria such as what types of aircrafts are involved, are the flights arriving or departing, the sequence, where on the taxiway and what the traffic situation is at the time the conflict occurs. The two outcomes of can be seen in figure 4.2, where the conflict point is highlighted. Having made this decision, a minimum *time separation*, or a *separation constraint*, is imposed between the two flights. A special case is when the conflict is located on the runway. When flights take off a turbulence is created in the air space behind it and so it is not safe for another flight to take off right away. This safety margin here varies depending on what types of aircrafts are involved e.g. a smaller aircraft taking off after a larger aircraft requires a larger safety margin than a large aircraft taking off after a smaller aircraft. Also, it is not safe for a flight to land while another flight is occupying some part of the runway, and so some separation must be imposed.

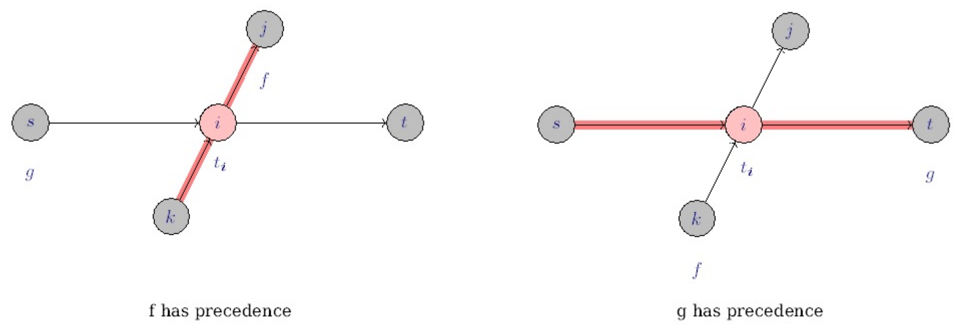


Figure 4.2: Outcomes of taxiway conflicts

4.2 Formulating the problem

Having set the scene, it is now time to formulate the problem we want to solve. Given an airport with a single mixed mode runway, and a predetermined sequence on the runway, assume that all flights have been given a taxi route for instance obtained by the shortest path algorithm. Also assume that flights are not allowed to stop on any segments. Our goal is to solve all occurring conflicts and obtain a schedule for each flight, minimizing the total cost of deviations at the airport. We will see that this can be formulated as an LP problem on standard form, and that it is in fact the equivalent to the minimum cost flow dual problem.

Taxi route

In order to obtain the LP formulation, we start off by introducing the variables associated with the taxi route of a flight f. Let i be a point or a segment in the route of f. We then denote by t_i the time f enter i. If j is a point and i is the preceding segment in the route of f, then we let

$$c_{ij} = \ t_j - t_i \tag{4.1}$$

In other words c_{ij} is the time needed for f to travel through segment j. If j is a point and q is the next segment on the route of f, then if f was allowed to stop at j, the time difference $t_q - t_j$ is greater or equal to zero with inequality if f stopped, giving the following constraint

$$t_j - t_q \leq 0 \tag{4.2}$$

If f was not allowed to stop at j we have

$$t_j - t_q = 0 \tag{4.3}$$

These are the only types of constraints needed in order to describe the taxi route of f. Figure 4.3 shows the associated variables on the airport.

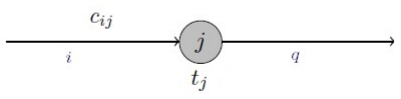


Figure 4.3: Taxi route variables

Deviations

The next variables to be presented are those associated with a flight's deviation. We denote the variables representing how delayed or how early the flight is as t_{delay} and t_{early} respectively. In order to obtain the inequalities for deviations, we start off with the following observations:

If a flight is early, its delay is zero. If a flight is delayed, the deviation must be equal to the actual exit time, t_{exit}, minus the target exit time.

$$t_{delay} = max\{0,\ t_{exit} - \text{target exit time}\}$$

If a flight is delayed, its earliness is zero, and its deviation must be equal to the target exit time minus the actual exit time.

$$\hat{t}_{early} = max\{0, \text{ target exit time} - t_{exit}\}$$

This gives us the following inequality for a delayed flight:

$$t_{exit} - \text{ target exit time } \leq t_{delay}$$

Which in standard form writes

$$t_{exit} - t_{delay} \leq \text{ target exit time} \tag{4.4}$$

We also see from the observation that the delay is greater than zero, which gives us

$$-t_{delay} \leq 0 \tag{4.5}$$

From the second observation, if a flight is early, we get the following inequality:

$$\text{target exit time} - t_{exit} \leq \hat{t}_{early}$$

This leaves us with

$$-\hat{t}_{early} - t_{exit} \leq -\text{target exit time}$$

However, this is not on the standard form, and so we have to use a "trick". By letting $\hat{t}_{early} = t_{early}$ and substituting, we obtain

$$t_{early} - t_{exit} \leq -\text{target exit time} \tag{4.6}$$

$$t_{early} \leq 0 \tag{4.7}$$

The conditions for piecewise linear costs are similar. We introduce the variables t_{delay2} and t_{early2} representing how much outside its time slot a flight exit, and the quantities *time slot delay* and *time slot early* representing the deviations this time slot allows. In other words, the time slot is an interval [-time slot early, time slot delay] with a given cost if the flight exit outside this interval. We observe that the delay outside the time slot must be at least the delay inside the time slot minus the value of the time slot delay. This gives us

$$t_{delay} - \text{ time slot delay } \leq t_{delay2}$$

which we rewrite to get on the same form as the previous conditions

$$t_{delay} - t_{delay2} \leq \text{ time slot delay} \tag{4.8}$$

Again for the early flights we have the opposite, the earliness outside the time window must be at most the earliness inside the time window minus the value of the time window, giving us in rewritten form

$$t_{early2} - t_{early} \leq \ -\text{time slot early} \tag{4.9}$$

As with the linear costs, the delay outside the time slot is nonnegative, and the earliness is nonpositive, giving us

$$\begin{aligned} -t_{delay2} &\leq 0 \\ t_{early2} &\leq 0 \end{aligned} \tag{4.10}$$

It is easy to see that this can be extended to include several time slots by introducing two new variables t_{delayi} and t_{earlyi} for each time slot i, and substituting t_{delay} with $t_{delayi-1}$ and t_{delay2} with t_{delayi} in (4.8), and the same for the early variables in (4.9).

Conflicts

As mentioned previously, a conflict occurs when two or more flights want to access the same point within a given time window. The separation constraints imposed to avoid this can be represented as a disjunctive pair of constraints. Assume there is a conflict between flight f and g at point p_i. Let t_m represent the time f enter p_i and t_n the time g enter p_i. Then either the constraint $t_n \geq t_m + d_i^{fg}$ or the constraint $t_m \geq t_n + d_i^{gf}$, where d_i^{fg} and d_i^{gf} are time separations, must be satisfied in order for the conflict to be resolved. The variables involved in the constraints varies depending on whether the points in question are pass through points or holding points.

Let us first look at the pass through points. Assume that f and g want to access the point p_i at the same time. Assume also that some decision has been made, giving f precedence over g. Let t_k and t_l represent the time variables for f and g entering p_i respectively, and let d_i^{fg} denote the time separation between f and g at p_i. Then the condition

$$t_l \geq t_k + d_i^{fg}$$

state that g cannot enter p_i until at least the time f leaves the p_i plus the time separation. Rewritten on the standard form we get

$$t_k - t_l \leq d_i^{fg} \tag{4.11}$$

Now let p_j be a holding point. In this case the time at which a flight enter p_j may differ from when it exits, and so if the separation constraint is represented in the previous way, we could end up in a situation where flight f enter the point, stops longer than the separation time, and flight g comes crashing into it. In order to avoid this, instead of using the time f enter p_j, we use the time f *leaves* p_j, that is, the time when f enter the incident segment, which we denote t_s. This gives the following condition rewritten to the standard form

$$t_s - t_l \leq d_j^{fg} \tag{4.12}$$

These are the only conditions needed to avoid flights colliding on the taxiway. In fact equations (4.1) - (4.12) are all the conditions we need, and so we can turn our attention to the objective function.

Objective function

As stated, our objective is to find the minimum total cost of deviation for all flights $f \in F$ where F is a set of flights . We introduce, for each flight, the costs of deviations denoted $b_{delay}, b_{early}, b_{delay2}, b_{early2}$ associated with the deviation variables, and let $b_i = 0$ for the other variables t$_i$. By letting the delay costs be nonnegative, and the early costs be nonpositive, we end up with the following objective function

$$\text{minimize} \quad \sum_{f \in F} b^f_{delay} t^f_{delay} + b^f_{early} t^f_{early} + b^f_{delay2} t^f_{delay2} + b^f_{early2} t^f_{early2} \tag{4.13}$$

Recalling the signs of the deviation variables it is clear that this sum is nonnegative.

The complete formulation

Combining the conditions (4.1) - (4.12) together with the objective function, letting p_h be holding points, p_p be pass through points, C_p be conflicts in pass through points, and C_h be conflicts in holding points, we obtain the formulation

$$
\begin{array}{lll}
\text{minimize} & \mathbf{b}'\mathbf{t} & \\
\text{subject to} & t_j - t_i = c_{ij} & \forall\ (i,j) \in p_p \\
& t_j - t_q \leq 0 & \forall\ (i,j) \in p_h \\
& t^f_{exit} - t^f_{delay} \leq \text{target exit time} & \forall\ f \in F \\
& t^f_{early} - t^f_{exit} \leq -\text{target exit time} & \forall\ f \in F \\
& t^f_{delay} - t^f_{delay2} \leq \text{time slot delay} & \forall\ f \in F \\
& t^f_{early2} - t^f_{early} \leq \text{time slot early} & \forall\ f \in F \\
& -t^f_{delay} \leq 0 & \forall\ f \in F \\
& t^f_{early} \leq 0 & \forall\ f \in F \\
& -t^f_{delay2} \leq 0 & \forall\ f \in F \\
& t^f_{early2} \leq 0 & \forall\ f \in F \\
& t_k - t_l \leq d^{fg}_i & \forall\ (k,l) \in C_p \\
& t_s - t_l \leq d^{fg}_j & \forall\ (s,l) \in C_h
\end{array}
\tag{4.14}
$$

Which clearly is an LP formulation.

4.3 Minimum cost flow dual formulation

Having shown that our problem can be formulated as an LP problem, the next step is to show that it is in fact equivalent to the dual of the minimum cost flow problem.

We rewrite the equality constraints from (4.14) as inequalities,and note that an important assumption is missing. Recall from section 3.4.1 that the sum of demands must equal zero. In (4.14), the demands are the costs $\mathbf{b}$, and since these can be set so they do not add up to zero, this is not ensured. In addition, the nonnegativity and nonpositivity conditions on the delay and early variables are not on standard form. In order to overcome this, we introduce a new variable t_0 representing the initial time for all flights, that is the moment time starts running. We then assume that this initial time is zero giving us the condition

$$t_0 = 0 \tag{4.15}$$

We now write the equations previously not on standard form as

$$\begin{aligned} t_0 - t_{delay} &\leq 0 \\ t_{early} - t_0 &\leq 0 \\ t_0 - t_{delay2} &\leq 0 \\ t_{early2} - t_0 &\leq 0 \end{aligned} \tag{4.16}$$

Our problem is now on the form

$$\begin{aligned} \text{minimize} \quad & \mathbf{b}'\mathbf{t} \\ \text{subject to} \quad & t_0 = 0 \\ & A\mathbf{t} \leq \mathbf{c} \end{aligned} \tag{4.17}$$

In order to handle the sum of demands, we associate the cost

$$b_0 = -\sum_{f \in F} b^f_{delay} + b^f_{early} + b^f_{delay2} + b^f_{early2}$$

with t_0, thus fulfilling the sum of demands assumption.

However, this leaves us with another problem, namely the assumption that $t_0 = 0$. We proceed by augmenting the objective function by including this constraint with a weight of the sum of costs giving

$$\begin{aligned} \text{minimize} \quad & \mathbf{b}'\mathbf{t} \ \ -\sum_i b_i t_0 \\ \text{subject to} \quad & A\mathbf{t} \leq \mathbf{c} \end{aligned} \tag{4.18}$$

Finally we need to show that solving (4.18) is equivalent to solving (4.17).

Proof. Assume $\hat{T}$ is a feasible solution to (4.17). It is clear that $\hat{T}$ satisfies all the constraints of (4.18), and that the objective value is the same. Hence $\hat{T}$ is also a feasible solution to (4.18), with the same objective value. Assume now that $\tilde{T}$ is a feasible solution to (4.18), and let $\hat{T} = \tilde{T} - t_0\mathbb{1}$. The following is then true for the constraints

$$\begin{aligned} A(\tilde{T} - t_0\mathbb{1}) &\leq \mathbf{c} \\ \Rightarrow A\tilde{T} - At_0\mathbb{1} &\leq \mathbf{c} \\ \Rightarrow \qquad A\tilde{T} &\leq \mathbf{c} \end{aligned}$$

where the final step follows from the fact that A is the node-arc incidence matrix. And so $\hat{T}$ is a feasible solution to (4.17). Due to the assumption that $\sum_i b_i = 0$, the objective value is also the same. □

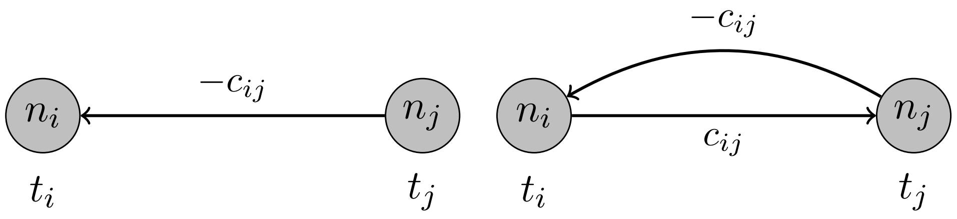
4.4 The events graph

In order to solve the problem formulated in the previous section by using optimization theory, we will introduce a graph, referred to as the *events graph*, which represents the flight events happening at the airport. While being a single graph, the events graph can be "divided" into two parts, one part representing events happening on the taxiway, and the other the deviation of flights.

4.4.1 Taxi route

We start by looking at the taxi route of a flight f. For each of the variables $t_i, \ldots, t_k$ associated with f entering a point on the taxiway, we associate nodes $n_i, \ldots, n_k \in N$. Depending on whether the point is a holding point or pass through point, the nodes will be referred to as a *holding node* or a *pass through node*. Note that no nodes have been associated with the variables representing f entering a segment. We solve this by letting t_q in (4.2) represent the time f enter the next point instead of the next segment, and replacing 0 in the same equation with $-c_{jq}$. This reduces the number of nodes and segments required by k for each flight.

Using this, along with the other taxi route constraints (written on standard form), we associate a directed edge $e_{ji} \in E$ with each constraint. Depending on whether the node n_i is a holding node or a pass through node, it will look like figure 4.4 in the events graph.



(a) Holding node (b) Pass through node

Figure 4.4: Events graph taxi route example

A complete taxi route example

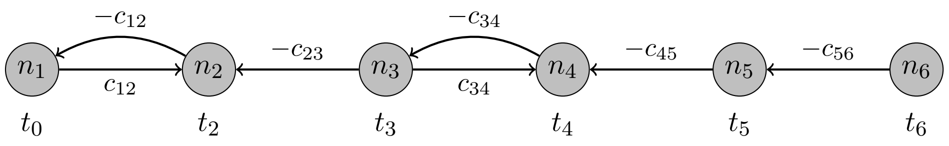


Figure 4.5: Single flight taxi route

We now show, complete taxi route for a flight. Given the sample airport in figure 4.6, let f be a departing flight starting from gate 2, and let $(p_2, p_3, \ldots, p_6)$ be the points in the taxi route. We also include the start node n_1 referring to the initial time t_0. Node 2 represents gate 2, and t_2 is the target off-block time of f. On its way towards take-off, f arrives the intersection, represented by node 3, at time t_3. Assuming this is a pass through node, f continues towards holding area 1, node 4, and arrives at t_4. Here it may wait, or continue directly to node 5, the runway entry, where it arrives at t_5. We assume in this example that departing flights are allowed to wait at the runway entry for the airspace to clear. Taxiing on the runway f arrives at the runway exit a.k.a. the take-off point, or node 6 at t_6, and take off immediately. All in all the taxi route of f gives us these constraints

$$
\begin{aligned}
t_2 - t_1 &\le c_{12} \\
t_1 - t_2 &\le -c_{12} \\
t_2 - t_3 &\le -c_{23} \\
t_4 - t_3 &\le c_{34} \\
t_3 - t_4 &\le -c_{34} \\
t_4 - t_5 &\le -c_{45} \\
t_5 - t_6 &\le -c_{56}
\end{aligned}
$$

The resulting events graph for this taxi route is shown in figure 4.5. The time difference $t_6 - t_3 - c_{23}$, is the total taxi time of f, in other words the time it spends on the taxiway. As mentioned in chapter 2 some of the previous studies has focused on minimizing this difference.

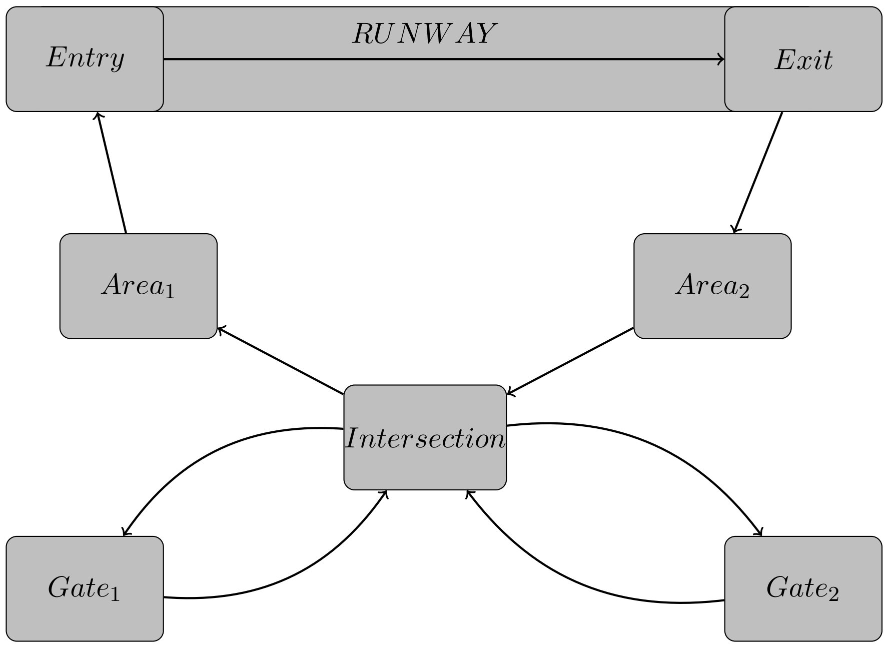


Figure 4.6: Sample airport

4.4.2 Deviations

Next we turn our attention to the "second part" of the events graph, representing the deviation of flights. In order to represent this, for each flight we associate a new node, n_{delay} with the variable t_{early} and a new node n_{early} with the variable t_{early}. In addition we associate an edge with the constraints (4.4),(4.6), and the first two in (4.16). Figure 4.7 shows the extended events graph. For improved readability, the figure highlights the deviations, so the taxi route part of the graph is displayed as a single edge, labelled taxi route. .

Piecewise linear costs

In order to represent if a flight exits within the given time slot, for each flight we add new nodes n_{delay2} and n_{early2} which is associated with the variables t_{delay2} and t_{early2}. We also introduce edges associated with the constraints (4.8), (4.9), and the final two in (4.16). The events graph is now extended to look like figure 4.8. It is easy to see how multiple time windows can be represented in the events graph by adding new nodes and edges. .

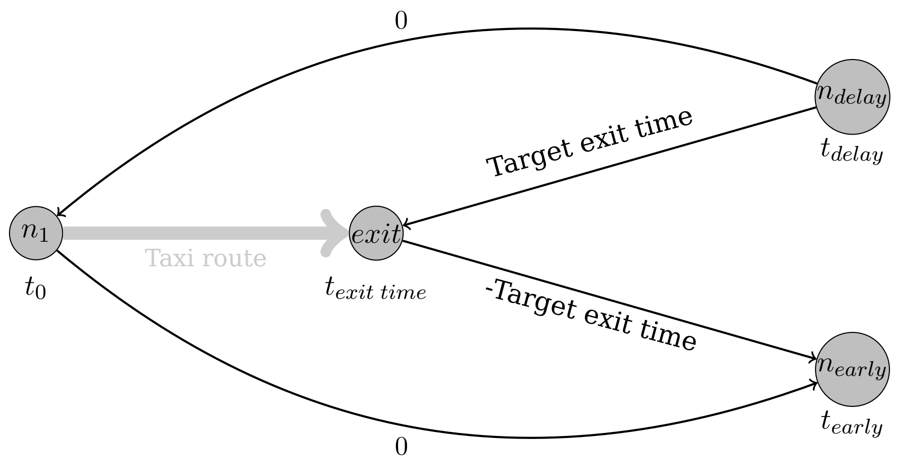


Figure 4.7: Deviation

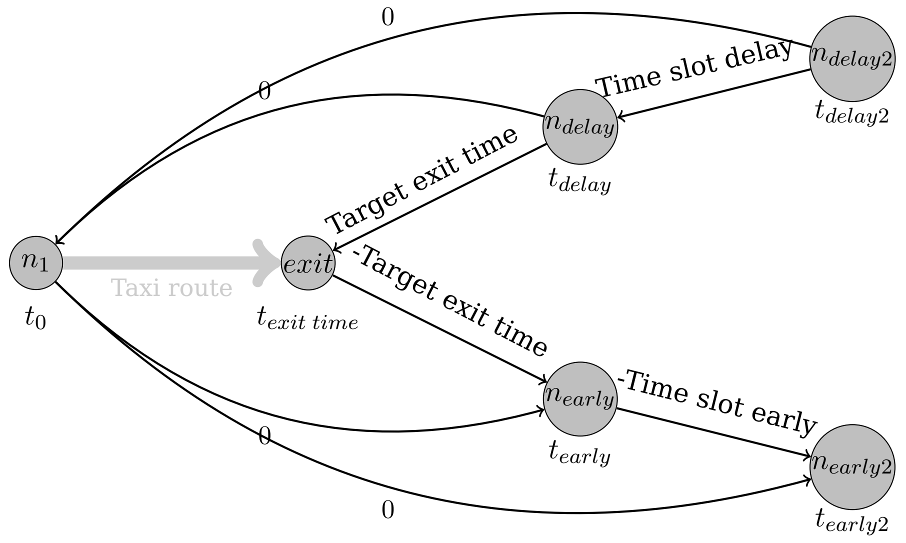


Figure 4.8: Deviation with piecewise linear convex costs

4.4.3 Graph for a single flight

Combining the previous discussions regarding the taxiway and the deviations, we now present the events graph for the single flight described in the previous section. Figure 4.9 shows this graph, where node 6 represents the exit (parking or take-off), node 7 and 8 represents *delay* and *early*, and node 9 and 10 represents *delay2* and *early2*. .

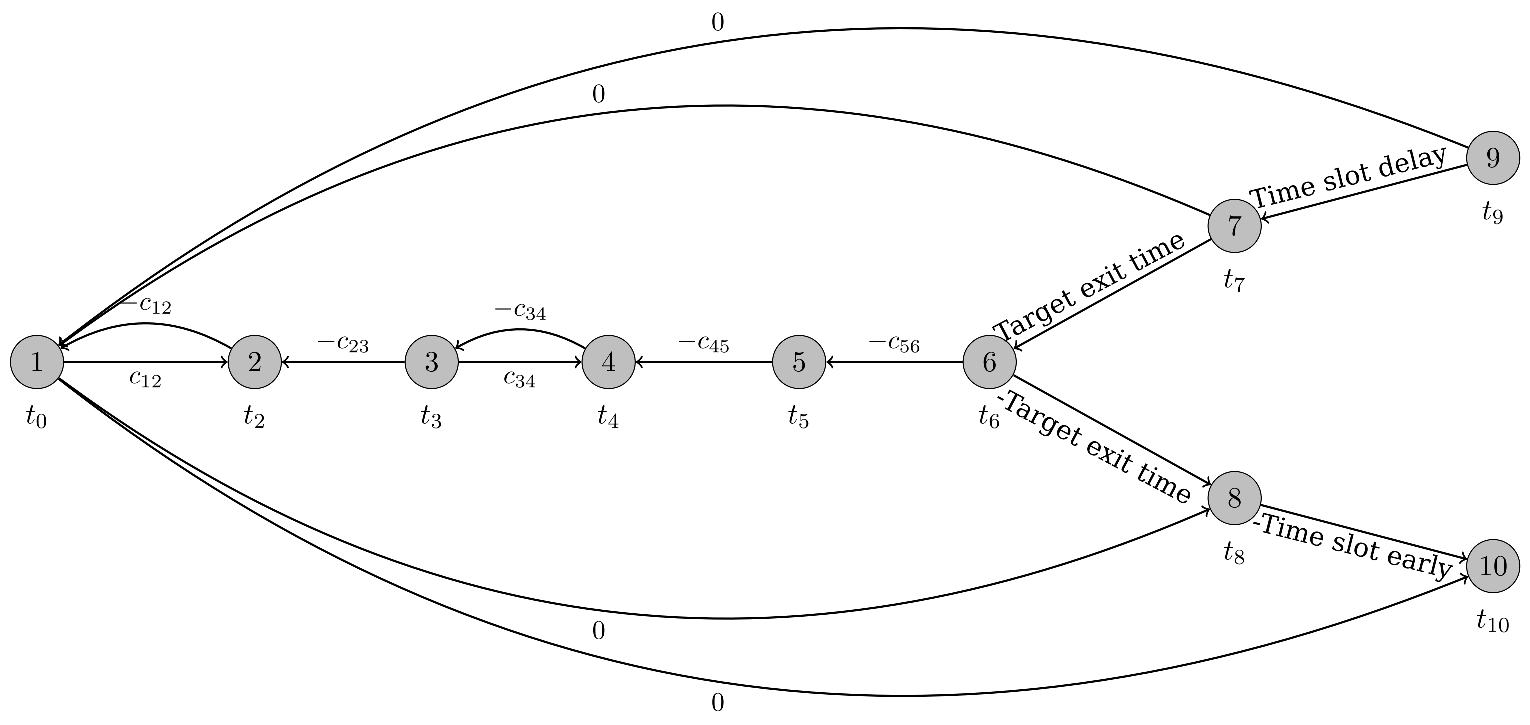


Figure 4.9: Single departing flight

4.4.4 Multiple flights and separation constraints

So far in this section we have limited the events graph to contain events for a single flight only. We now extend this to a more realistic situation by introducing a second flight, and describe how this can be represented in the graph. The goal is to, at the end of this subsection, have a clear idea of how the events graph looks when containing multiple flights.

We continue to look at the sample airport from this section, but in addition to the departing flight we now imagine that there is a flight arriving on the runway, traveling through holding area 2 and the intersection before parking at gate 1. The nodes representing events happening along the arriving flight's taxi route is generated in the same way as for the departing flight, and as with the first flight, the second flight has some deviation nodes associated with it. It is also clear that the arriving flight's constraints are of the same type as the departing flight's constraints. In order to represent the two flights in a single events graph we note that the flights share the same initial time, that is node 1 where t_0 is zero, and that all nodes representing taxiway events and deviations are distinct.

When dealing with multiple flights we have seen that the introduction of separation constraints is needed if there are any potential conflicts. In our sample airport, we see that both flights wants so access the intersection and the runway entry. Following the discussion in section 4.1, two separation constraints will be needed. However, since we no longer have variables for flights entering segments, we need to adapt the separation constraints to represent this. Following the notation from section 4.1, assume two flights f and g want to access the holding point p_j, since p_j is a holding point, we need to set a separation between the events when f leaves p_j and when g enter. Since, we no longer have an event representing f entering the incident segment, we instead replace the time variable t_s by the variable $t_t - c_{st}$ representing the event that f enter the next point on the taxi route minus the time difference between the two events. We see that this can also be applied to separation constraints between events regarding pass through points, and so all separation constraints can be written as

$$t_t - t_l \leq d_j^{fg} + c_{st} \tag{4.19}$$

Returning to our two flight example, assume a decision has been made that the departing flight will be given precedence at the intersection, and that the arriving flight will get precedence at the runway entry. In the events graph, this is represented by adding two edges associated with the constraints

$$t_3 - t_{15} \leq d^{fg}_{intersection} + c_{14,15}$$
$$t_{12} - t_5 \leq d^{fg}_{runway\ entry} + c_{11,12}$$

Figure 4.10 shows the events graph for this example, and in chapter 5 we discuss how to add separation constraints between flights, and the effects this has. It is now easy to see how the events graph can be extended to include as many flights as we want, and how avoiding conflicts between them is represented.

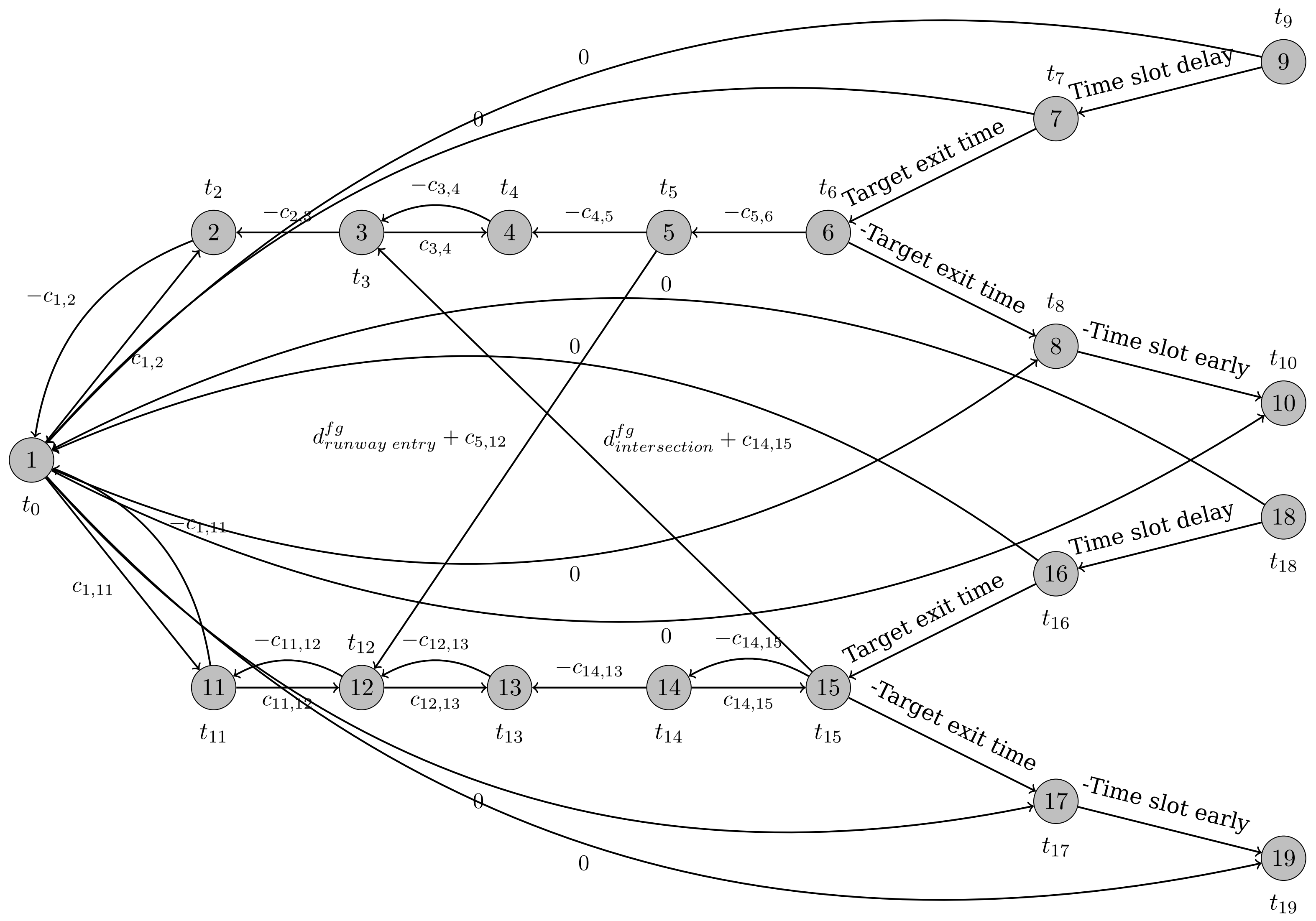


Figure 4.10: Events graph of two flights

Chapter 5

Conflict resolution and sequencing

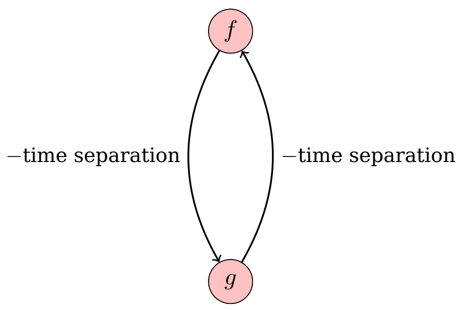
In chapter 4 we briefly touched on the topic of conflict resolution by giving a definition, and introducing separation constraints between flights. This chapter presents a more detailed discussion, and a proposal for how this can be implemented in a solution algorithm. The first section describes the concept of solving a conflict in the events graph, while section two and three presents two algorithms for detecting and solving conflicts between flights in the events graph. In section four, we present a discussion on where to start solving conflicts, while section five describes how to implement a sequence in the events graph. The final section shows why we cannot rely on Dantzig's rule in the network simplex algorithm in order to resolve conflicts.

5.1 Introduction

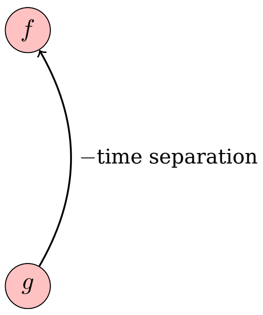
Before anything, an air traffic controllers most important job is to make sure that no aircrafts collide. This is done by keeping a minimum safety distance between flights at all times. An alternative method would be to use the taxi speeds of flights, and combine them with the distances to get a time separation. Previous research [6] have used algorithms resolving conflicts within certain areas of the airport. It is however, reasonable to believe that the solution would improve if flights can be located anywhere on the taxiway when the conflict is resolved.

5.1.1 Solving a conflict in the events graph

As we have seen in chapter 4, resolving a conflict between two flights f and g simply means deciding which flight goes first. Having made



(a) Choices in a conflict



(b) Solution where f has precedence

a choice, this is modelled in the events graph by adding an edge representing the precedence constraints between the events of the flights involved. The weight on this edge represent the minimum separation between f and g at this particular area of the airport. Figure 5.1a shows the decision which has to be made in the events graph, where one of the two edges is to be added, and figure 5.1b shows the solution where f has precedence over g.

5.1.2 Making the precedence decision

The decision on which flight is to be given precedence can be based on many variables. However, the air traffic controllers at Arlanda told us that their decisions are mainly based on whether the flights are arrival or departures, and which flight has the earliest scheduled exit time, and so those are the criteria used in our solution. Arrivals will always get precedence over departures, and if the conflict is between two arrivals or two departures, the flight with the earliest exit time will get precedence.

Other aspects that could be considered when deciding on precedence are for instance

- *Where the conflicts occurs.* Perhaps one of the flights are close to their exit point, and so should get precedence in order to minimize the number of active flights on the taxiway regardless whether it is a departure or not.
- *The current deviation of flights.* Perhaps flights that are already delayed should get precedence so to increase the chances of it exiting within the time window and reducing the total cost. In the same manner, flights that are early might not need precedence in order to exit on time.

- *Size and airline.* The size of flights may be important when it comes to wake vortex after take-off. Allowing a larger aircraft to take off before a smaller one may increase the wait time by an unnecessary large amount. It may also be important to spread the precedence of flights among the airliners in such a way that no company gets an unfair advantage.

There may be additional concerns that should be considered in addition to the ones mentioned here, and those that are might not have a positive effect on the solution. However, in order to test this, additional real life test cases other than the one we have been provided with are needed.

5.2 Simple conflict resolution

Given two flights f and g with a predetermined taxi routes, the perhaps most intuitive way of avoiding conflicts is to check each point in the taxi route for both flights, and see if they coincide. Then, if some do, impose a separation constraint between the flights at these points. Here is how this can be done in the events graph.

Given an events graph G and a flight f with a predetermined taxi route. Assume nodes have been added to N in compliance with the discussion in chapter 4, and let $N_f \subseteq N$ be the set containing these nodes. Imagine now that information about a new flight g becomes available, and that based on this information, g is given a taxi route. In order to avoid conflicts between f and g, we have to for each node n that is to be be added to N as part of the taxi route of g, check if the event n refers to, happens at the same point as any of the nodes in N_f. If it does, some decision on which flight gets precedence is made, and an edge representing the separation constraint between the two flights is added incident to the nodes. The code for finding and resolving conflicts in this manner is presented below. An example of the taxi route of two flights where conflicts have been resolved can be seen in figure 5.2. Here we have highlighted the nodes referring to the same points. However, as we can see from the figure, even for few flights with short taxi routes, the problem becomes much more complex by resolving conflicts in this manner, as one additional edge is required for each shared point in the routes. Another approach is therefore needed.

```
input: a graph: G = (N,E)
       a flight route: route_g
       a list of flight routes: routes

% Compare new flight route nodes with existing flight route nodes
for each node i in route_g
        for each element e in routes
                 for each node j in e
% Check nodes refer to the same point
                        if (i = j)
                           Make precedence decision
                           Add precedence constraint to E
```

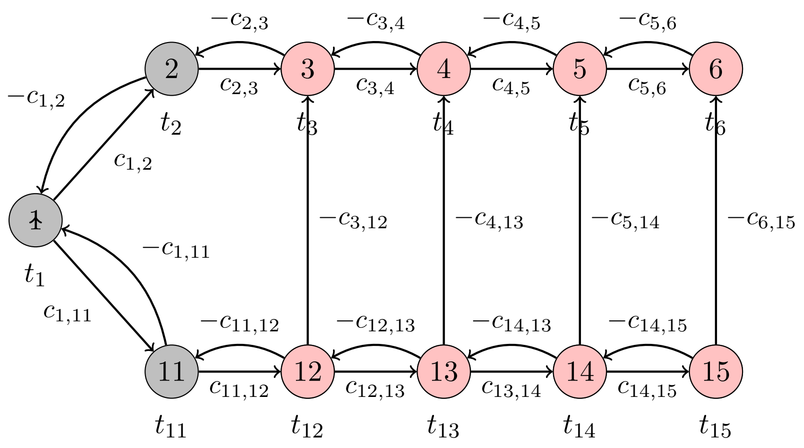


Figure 5.2: Simple conflict resolution between two flights

5.3 Shared paths

Resolving conflicts at every shared point on the taxiway between every flight, results in a large number of constraints added to the problem. In order to avoid this situation, we now introduce the concept of *shared paths*.

5.3.1 Description

A *shared path* between two flights f and g is a pair of ordered sets of one or more adjacent points on the taxiway where the taxi route of f and g coincide, i.e. a shared path is a set of two paths. All shared paths are undirected, and so if (p_i, p_{i+1}, p_{i+2}) is part of the taxi route of f and (p_{i+2}, p_{i+1}, p_i) is the same for g, the set consisting of (p_i, p_{i+1}, p_{i+2}) and (p_{i+2}, p_{i+1}, p_i) is a shared path between f and g. It is also possible to have multiple shared paths between two flights which occurs when the taxi routes coincide at some point, then diverge before they converge again.

5.3.2 Shared paths and the events graph

Since the shared paths are undirected, two different methods of implementing them are used depending on whether or not the flights travel in the same or in the opposite direction.

Same direction

Assume that a shared path for f and g has been discovered, and that the flights enter this path at the same point. In order to resolve this conflict, a decision on who gets precedence is made, and a separation constraint between the flights at the first point is imposed. Assuming that f and g travels at the same speed, and that the time separation is constant throughout the shared path, the conflict has been resolved. If this is not the case, the conflict at the first point is still resolved, and so we move to the next point in the shared path and impose a separation constraint. Repeating this process eventually the conflict for the entire shared path will be resolved. Note that in the second situation, the required number of constraints to be added is the same as for the method in the previous subsection.

Opposite direction

While flights traveling in the same direction may be located inside the shared path at the same time, flights traveling in opposite directions have to wait for the entire path to be clear before entering it. That is, when a shared path is detected between two flights traversing it in opposite direction, a decision has to be made regarding precedence, then a separation constraint has to be imposed between the flights at the first and last point in the shared path. Doing this results in resolving the conflict for the entire shared path, with only two additional constraints to be added.

In the events graph, shared paths are represented by the nodes referring to the events of flights entering the points in the shared path. Figure 5.3 shows the implementation of shared paths for flights in the same direction with the same speed, and figure 5.4 the same for opposite direction. The nodes included in the shared path has been highlighted. We see that it is possible in some cases to reduce the number of additional edges required to 1 for same directional conflicts, and 2 for all opposite directional conflicts. Below we present the code for finding a shared path:

```
input:  a flight route: route_f
        a flight route: route_g
        an airport: A
output: a shared path: SP

% Detect conflict
while no conflict is detected
     for each node i in route_f
             for each node j in route_g
                     if i and j refers to the runway
                        Continue
                     if i and j refers to the same point in A
                        Conflict is detected
                        i and j are contained in SP

% Get the shared path
while (i <= route_f.length && j >= 0)
        if (i+1 = j-1)
                i+1 and j-1 are contained in SP
                f and g are traveling in opposite direction
                return SP
while (i <= route_f.length && j <= route_g.length)
        if (i+1 = j+1)
                i+1 and j+1 are contained in SP
                f and g are traveling in the same direction
                return SP
```

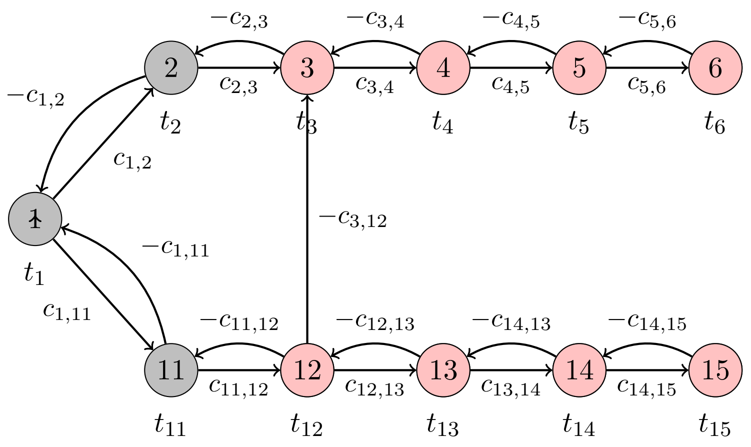


Figure 5.3: Result of conflict resolution in same direction and equal speed

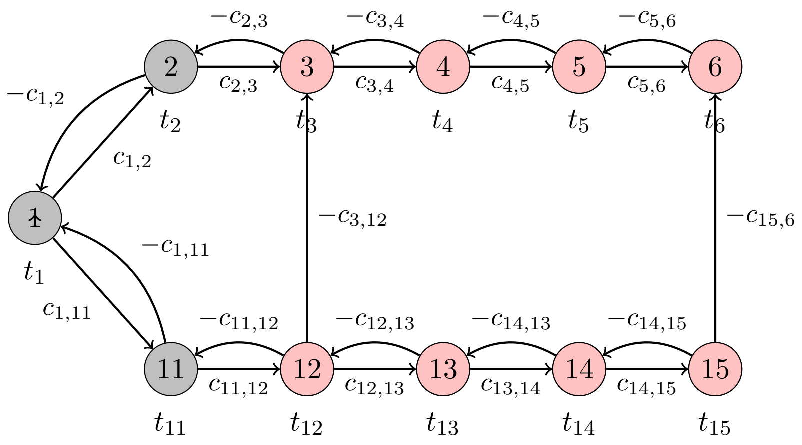


Figure 5.4: Result of conflict resolution in opposite direction

5.4 Deciding which conflict to solve

In section 5.2 we showed how to solve the first conflict detected in the events graph. However, since solving a conflict affects the solution, see section 3.6, and therefore also the other current conflicts, it is reasonable to assume that making a "smart" decision on which conflict to resolve first may reduce the number of occuring conflicts as opposed to just solving the first one detected. As with the task of deciding precedence, several thing could be considered when making this decision. We mention a two suggestions:

- *Solve conflicts on the most conflicted node.* This might be a good place to start, depending on how the airport is structured. There is however a risk that the conflicts will just shift to the nearest holding area.

- *Solve conflicts for the most conflicting flight.* The solution of solving a conflict for this flight have the highest potential for number of conflict resolved by solving a single conflict. There is however the possibility of just shifting them in time generating the same number or even more new ones.

In the end, we decided to start by solving the conflict which occurs first in time. The motivation for this choice is that solving these conflicts may only affect the conflicts occuring later, while solving a later conflict, may affect the earlier conflicts which again affects the later ones generating a cycle of negative effects. We may however still end up in the situation that solving a conflict generates one or more new conflicts occuring both earlier and later that the one we solved.

5.5 Sequencing

The runway sequencing problem can be viewed as a subproblem of conflict resolution where the goal is to find the best sequence of flights accessing the runway. In order to find the optimal sequence, there are several aspects that needs to be considered such as the arriving flight's time window, and the desired take-off times for departures. While the simple conflict resolution algorithm described in the previous section is too inefficient for solving all conflicts relating to the taxiway, it can be used for implementing the sequence solution, since the only nodes in the events graph implicated in the sequence are the ones referring to either the entering the runway entry or the runway exit, and therefore only a single edge needs to be added. Figure 5.5 shows how the sequencing between two arriving flights and one departing flight looks in the events graph. Node 2,6 and 8 represents flights entering the runway entry, node 3,7 and 9 the runway exit, and node 4,5 and 10 the gates. .

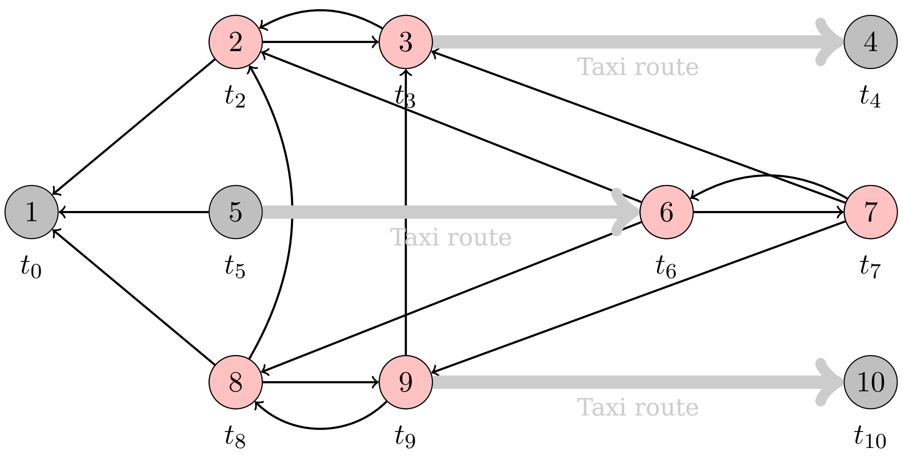


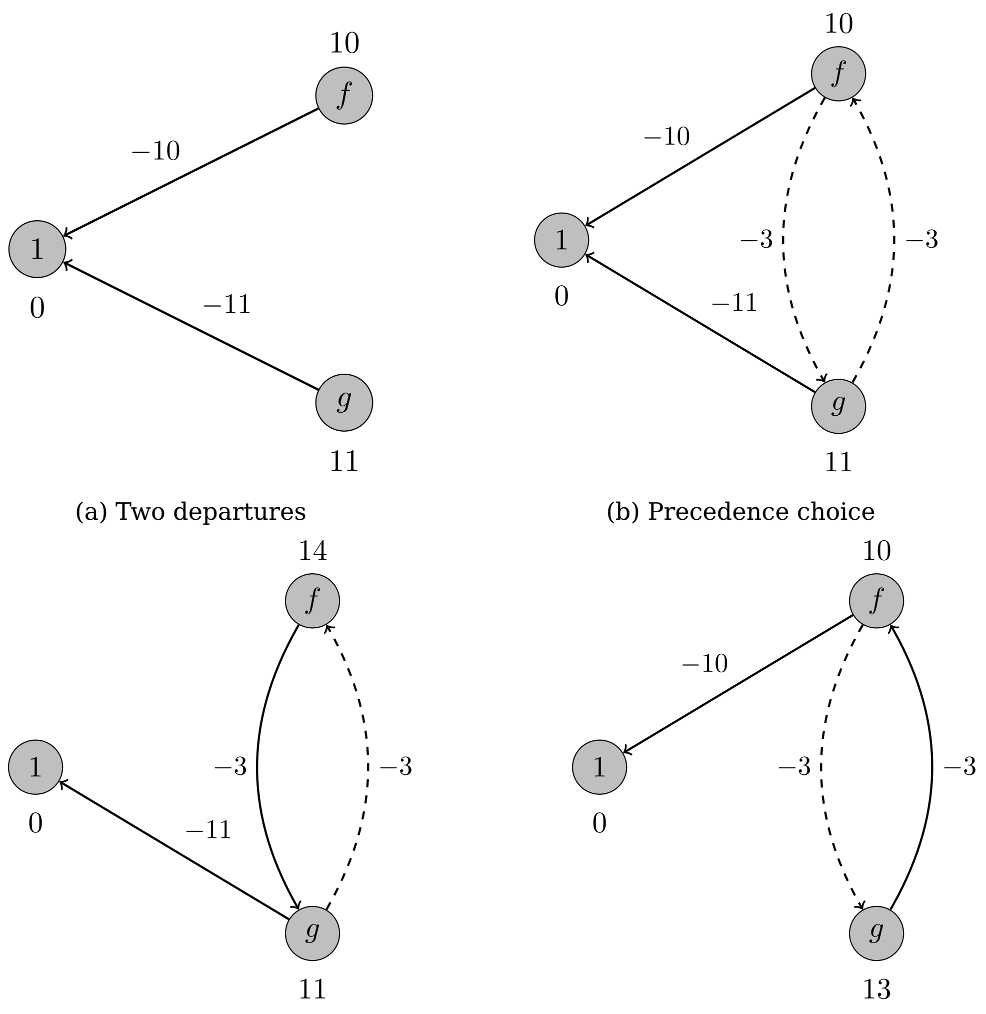
Figure 5.5: Sequencing

5.6 Network simplex based solution to conflicts

In section 5.1 we discussed various criteria for solving conflicts. In this section we will show why the network simplex method using Dantzig's pivot rule [7] cannot be used to this purpose.

From section 3.6 we have seen the changes to a problem P of adding a separation constraint to resolve a conflict. It is obvious, due to the nature of conflicts that the optimal solution of P is no longer optimal for our new problem $\hat{P}$. The question becomes, which of the two new edges is to be added to our tree solution? Dantzig's pivot rule states that the edge to enter the tree is the one with the maximum violation i.e. the edge with the most negative dual slack for the primal iterations. However, as we will see, following this rule may lead to a solution where the objective value is larger than by choosing the competing edge. We illustrate this with an example:

Assume there are two departing flights f and g that wishes to take off as early as possible, and let the cost of being delayed be 4 per minute. Given a taxi route, the taxi time of f is 10 minutes while the taxi time of g is 11 minutes. The events graph corresponding to this situation is shown in figure 5.6a, where we have contracted each taxi route to a single edge. Assume now that, due to safety regulations, the separation time between take-off is set to 3 minutes. This results in a conflict occurring at the runway, depicted in figure 5.6b. Using the formula $z_{ij} = y_i - y_j + c_{ij}$ we calculate the associated dual slack variables to be $z_{23} = -4$ and $z_{32} = -2$, and so according to Dantzig's rule, edge e_{23} should be added to the tree. In other words, g should get precedence over f. Since f ideally could have taken off at 10 minutes, but now instead have to wait until 14 minutes, the increase in cost equals 16. We see however, that had f been given precedence over g, g would only have had to wait an additional 2 minutes, and the increased cost would only have been 8. The figures 5.6c and 5.6d shows the two possible solutions. With this, we conclude that using the simplex method with this rule may not generate the solution with the lowest objective value.



(a) Two departures

(b) Precedence choice

(c) g has precedence

(d) f has precedence

Figure 5.6: Dantzig's rule

Chapter 6

Finding an optimal solution of the events graph

Even if a proof that the simultaneous ground movement problem is NP-hard has not been published, with regards to the proof that the simplified version (see section 2.4.2) is, we find it highly likely that the simultaneous ground movement problem is also NP-hard. The idea in this chapter is to find a fast heuristic which solves 4.18. In order to do this, we implement a specialized version of the simplex method. Before we describe our heuristic, we give a proof showing how, given any connected digraph with no negative cycles, it is possible to find an initial dual feasible solution to the minimum cost flow problem, and describe how we have calculated the values of the variables in this solution. We then move on to the events graph, and describe our heuristic step by step from finding an initial solution to finding the optimal solution. Note that in the following, with the term *solution*, we mean a set of edges that form a spanning tree, and the term *tree solution* also includes the variables associated with this tree.

6.1 Dual feasible initial tree

Given a connected digraph G, our goal is to obtain a dual feasible tree solution. In order to achieve this, we have developed an algorithm, based on the Bellman-Ford algorithm, which in the absence of negative directed cycles guarantees a dual feasible solution even if the graph is not strongly connected. If the structure of the events graph was unknown, this would be required due to the implementation of delay nodes which are unreachable from the root node. We later discovered that due to the nature and structure of the events graph, a more efficient method could be used to find an initial tree, see section 6.2, and this algorithm was in fact not necessary. Nonetheless, we present it for future use where graphs may have a different structure than our events graph, and since some may find the theoretical concept interesting.

Theorem 6.1.1. *If a connected digraph G does not contain any negative directed cycles, we can always find an initial dual feasible solution to the minimum cost flow problem.*

Proof. Let $G = (N, E)$ be a connected digraph, with no negative directed cycles. We want to show how to construct a dual feasible solution to the minimum cost flow problem associated with G. Choose a node $r \in N$. Let $N_r \subseteq N$ be the set of nodes reachable from r in G and let $\bar{N} = N \setminus N_r$ be the remaining nodes. Let $\bar{G} = (\bar{N}, \bar{E})$ be the subgraph of G induced by $\bar{N}$, i.e. $\bar{G} = G[\bar{N}]$. Assume we have at hand a feasible tree $\bar{T}$ and the corresponding dual tree solution $q(\bar{T})$ for the dual problem associated with $\bar{G}$, which we know exist since $\bar{G}$ does not contain any negative directed cycles. Such a solution can be extended to a feasible solution for the minimum cost flow problem associated with G in the following way:

Apply the Bellman-Ford algorithm $G_r = G[V_r]$ with root r and let $d(T_r)$ be the dual tree solution produced by the algorithm and the associated tree T_r, respectively. Let $\tilde{E} \subseteq E$ be the set of edges with start node in $\bar{N}$ and end node in N_r. Observe that there are no edges $e_{uv} \in E$ with $u \in N_r$ and $v \in \bar{N}$, otherwise v would be reachable from r which is a contradiction. For all edges $e_{uv} \in \tilde{E}$ calculate the associated dual slack value $z^e_{uv} = c^e_{uv} + y^e_u - y^e_v$. If all these z^e_{uv} are positive, decrease all y^e_u until one $z^{\hat{e}}_{uv}$ becomes zero. If not all z^e_{uv} are positive, let $\hat{e}$ be the edge with the most negative z_{uv}, and increase $y^{\hat{e}}_u$ until $z^{\hat{e}}_{uv}$ is zero. Recalculate the variable values in $q(\bar{T})$ with u as root. The tree T corresponding to the dual feasible solution of the minimum cost flow problem associated with G is then $T_r \cup \bar{T} \cup \hat{e}$. Which is dual feasible as T_r and $\hat{T}$ are dual feasible, and the constraint associated with $\hat{e}$ is an equality (tight).

Finally, to construct a feasible solution for $\bar{G}$ we can reason inductively, noting that $\bar{V} \subseteq V - \{r\}$, i.e. $|V_r| < |V|$ and the induction terminates. $\square$

Below we include the code for finding this initial dual feasible solution.

```
input: a directed graph $G$, a list of distances to each node $y$
       a list representing if a node has been visited
       a list representing parent relations,
       a list representing child relations
output: a dual feasible solution $\hat{G}$
%Step 1: Initialize
Set all distances to infinite
Decide on a root node
Set distance to the root node to zero, and mark it as visited
%Step 2: First subset
Starting from the root node:
for each node $n$ in $N$
        for each edge $e_{uv}$ in $E$
                if $y_u + c_{uv} < y_v$
                        Mark $v$ as visited
                        Set $y_v = y_u + c_{uv}$
                        Set $u$ to be the parent of $v$
                        Add $v$ to the children of $u$
                        Add edge $e_{uv}$ to $\hat{G}$
                        if $v$ is already a child to a node $j$
                                Remove that connection
                                Remove edge $e_{jn}$ from $\hat{G}$
%Step 3: Second subset
if all nodes have been visited
     return $\hat{G}$
else
     Repeat step 2 with the first unvisited node as root
     excluding edges with end nodes that have been visited
     for each edge $e_{uv}$ between the first and second subset.
                Calculate the dual variables
                if $y_v - y_u \leq c_{uv}$
                  Decrease the $y_u$s until at least one of the
                  inequalities becomes an equality
                  Add this edge to $\hat{G}$
                  Update distances in the second subset
                else
                  Increase the $y_u$s with the largest violation
                  so that they are all satisfied
                  Add the edge representing the equality to $\hat{G}$
                  Update distances in the second subset
Repeat step 3
```

We also include an example to make sure everything is clear before moving to the next step. Here we will make use of some new terms. Given a tree T and an edge $e_{uv} \in T$, we say that the *parent* of v is u, and that v is a *child* of u.

Let G be a directed graph with $N = \{n_1, n_2, n_3, n_4, n_5, n_6, n_7, n_8\}$ and $E = \{e_{12}, e_{13}, e_{23}, e_{34}, e_{41}, e_{52}, e_{53}, e_{56}, e_{57}, e_{61}, e_{86}\}$ as shown in figure 6.1 where the distances are shown above the nodes. We begin by using the standard Bellman-Ford algorithm, deciding on n_1 as the root node, the distance to n_1 is set to zero, and the node is marked as visited. We begin by looking at the edges outgoing of n_1, finding n_2 and n_3, which we both mark as visited. Checking the distances we find that, for n_2, clearly $2 < \infty$, and we set $y_2 = 2$. Updating the child-parent relations we see that n_1 is now that parent of n_2, and n_2 is a child of n_1. We add the edge e_{12} to the tree solution, and check if n_2 had any previous child-parent relations. Since it did not, we move on to n_3. We find the same for n_3 seeing that $3 < \infty$, so we set $y_3 = \ 3$, update the child-parent relations, and add $e_{1,2}$ to the tree solution. No more nodes are reachable from n_1 and so we move on to n_2. Here we see that the only reachable node is n_3, however $2 + 5 > 3$, and so no distances are updated. Proceeding to n_3, we see that n_4 is reachable, and so we set $y_4 = y_3 + 7 = 10$, and update the child-parent relations. For n_4 only n_1 is reachable, and no more distances can be updated. All the reachable nodes have now been visited, and so we go to step 3 in the algorithm, choosing n_5 as the root node for the second subset. Running Bellman-Ford with n_5 as the root node and excluding e_{52} and e_{53} we find that n_6 and n_7 are children of n_5, and that the distances are $y_5 = 0, y_6 = 4$ and $y_7 = \ -2$. We update the child-parent relations and add e_{56} and e_{57} to the tree solution. This leaves us with three subgraphs where $\{n_1, n_2, n_3, n_4\} \in S_1$, $\{n_5, n_6, n_7\} \in S_2$, and $\{n_8\} \in S_3$ which we will come back to later. Looking at the subtrees S_1 and S_2, we wish to connect them while still ensuring a dual feasible solution. Figure 6.2 shows the situation with the node variables and edges already included in the tree solution highlighted, and the edges spanning the two trees in grey. The next step in the algorithm is to check the bridging edges e_{52}, e_{53} and e_{61} to find out which one to add to the solution, connecting the two subgraphs. Checking the inequalities $y_v - y_u \leq c_{uv}$, we see that the largest violation is for edge $e_{6,1}$ with 6, and so we increase y_5, y_6, and y_7 by 6. Edge e_{61} is added to the tree solution, and the child-parent relations are updated so that n_6 is the parent of n_5 and n_5 is the parent of n_7. The two subtrees are now connected. Going back to step 3 in the algorithm, we find that n_8 is still unvisited. We mark n_8 as visited, set $y_8 = 0$, and run Bellman-Ford with n_8 as root. No nodes are reachable from n_8, and so we are left with two subtrees $S_1 \cup S_2$ and S_3, with only the edge e_{86} bridging between them see figure 6.3a. Clearly the inequality $y_6 - y_8 \leq 0$ is violated (by a value of 10), and so we increase y_8 by 10. Having made the inequality into an equality, we add the respecting edge to the tree solution, set n_6 as the parent of n_8, and add n_8 to the children of n_6. All the nodes have now been visited, and we have found a dual feasible initial solution for the network simplex method which is highlighted in figure 6.3b.

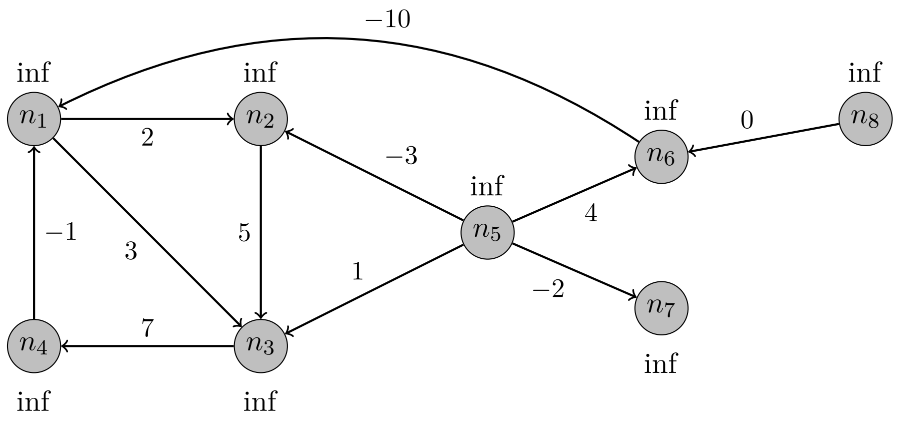


Figure 6.1: Initial graph

.

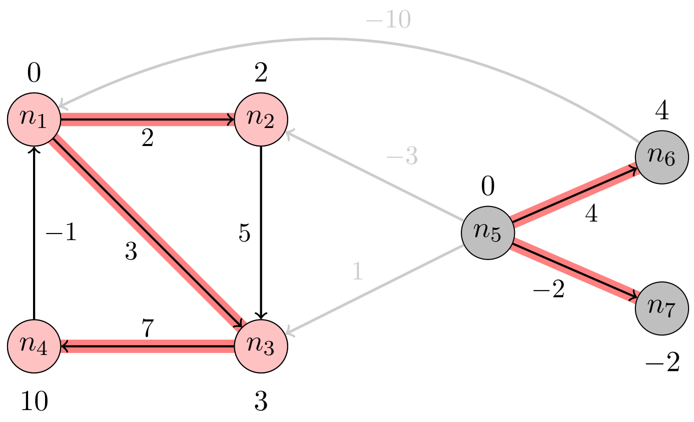
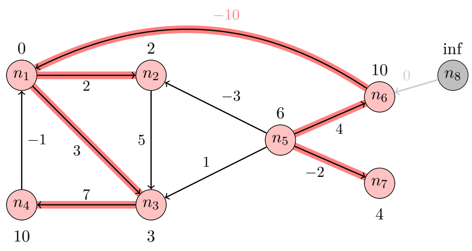
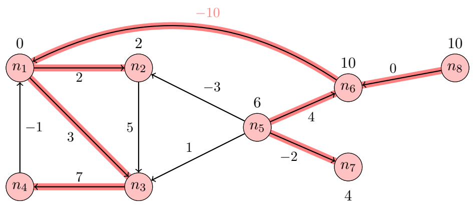


Figure 6.2: Disconnected subsets



(a) Second iteration of step 2



(b) Dual feasible initial solution

Efficiency and worst case scenario

We have seen that in addition to run Bellman-Ford from the root node, the algorithm runs Bellman-Ford on each unreachable subsets from the root containing subset, and so the running time is $\mathcal{O}(|N| \cdot |E| \cdot |G|)$ where $|G|$ is the number of subsets. From this we see that the worst case scenario is obtained when the graph looks like figure 6.4 and n_1 is chosen as the root node. The algorithm then has to run Bellman-Ford n times to find an initial solution as none of the other nodes are reachable, making $|G| = n$. However, the graph also represents the *best* case scenario! By choosing n_n as the root node, the algorithm only have to run Bellman-Ford once, as all the nodes are then reachable from n making $|G| = 1$. And so we see that knowing the structure of the graph is vital to the algorithms performance. It is also worth noting that the algorithm does not require any temporary edges to be added in order to find a solution, and so the complexity does not increase. This is important if $|E| \approx |N|$ as an increase in the number of edges could lead to a great increase in running time.

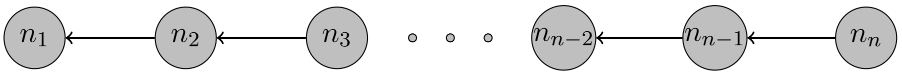


Figure 6.4: Worst case scenario

6.2 Optimal solution of the events graph

In this section we go through our solution approach for finding an optimal conflict free solution to the problem formulated in chapter 4. The underlying idea is to apply the network simplex algorithm on an initial dual feasible solution in order to improve it. The initial solution is calculated disregarding the sequence and all possible conflicts, then the sequence is applied, before conflicts are detected and resolved.

Initial solution of the events graph

In order to find an initial solution, we begin by making the following observation:
Given two flights f and g, the initial solution for f is independent of the initial solution of g.
This follows from the fact that the initial solution is calculated by disregarding all conflicts and sequencing, and so the events of g does not affect the events of f, as each flight simply travels as it wishes. We use this to our advantage when finding the initial solution.

Given information about a set of flights F containing their desired entry and exit times, and a given taxi route for each flight, we construct the initial solution T following these steps: For each flight $f \in F$ add nodes and edges to a graph G in accordance with the discussion in chapter 4. Then obtain the initial solution of f, T_f, by adding all the backward edges from the initial root node to the final node in the taxi route of f, check if f is early or delayed, and add edges to T_f accordingly. Finally add T_f to T. Figure 6.5 shows the initial tree for two flights where the first flight is delayed and the second is early.

Our observation also holds great value in the situation where an optimal, conflict free, solution for a set of flights has been found and information about a new flight becomes available. In this case we only have to add the initial solution of the new flight to the initial solution of the existing set of flights instead of calculating a new initial solution for the entire set from scratch.

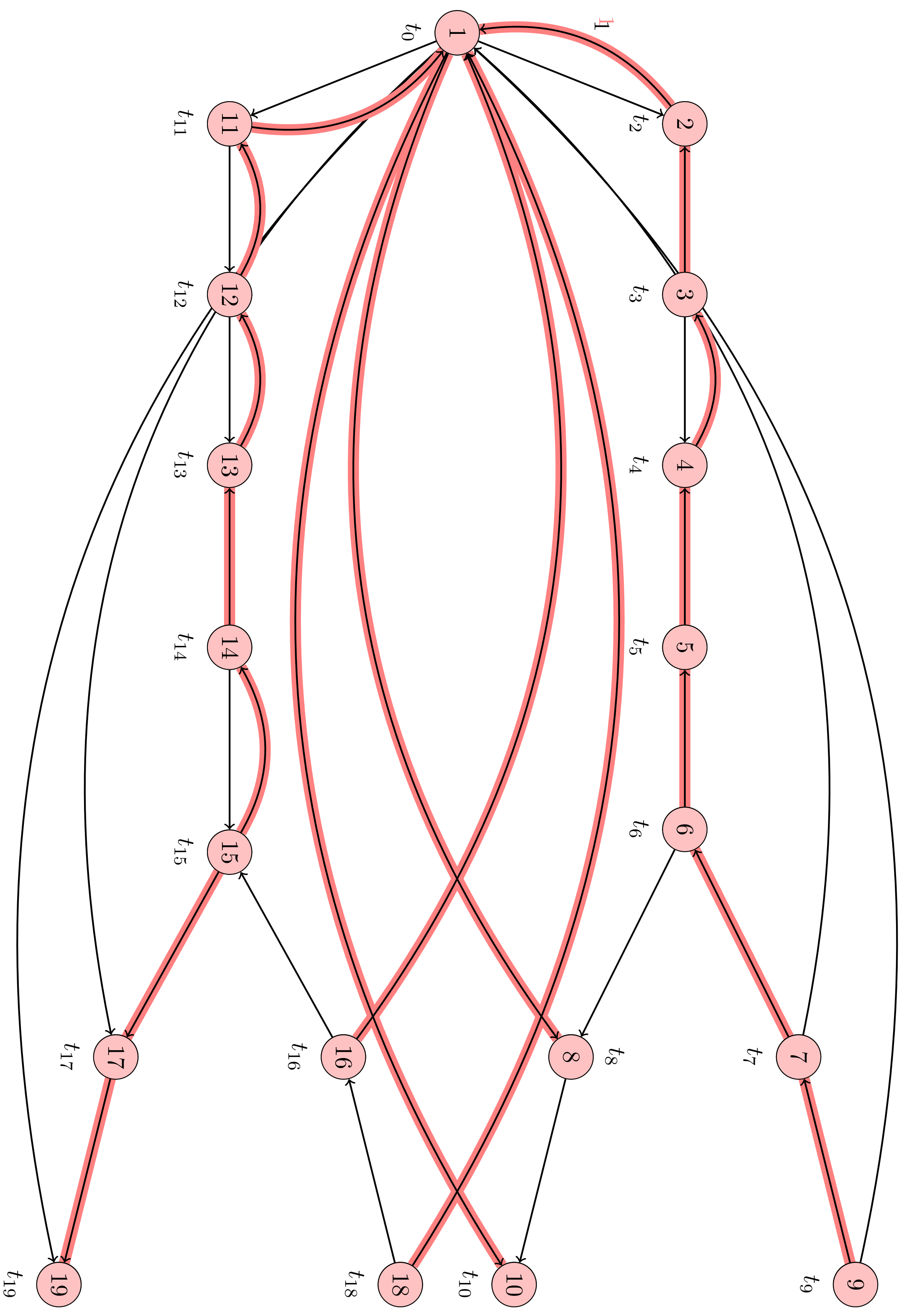


Figure 6.5: Initial tree of two flights

Calculating the variables

Having shown how to find the initial tree, we now go through the algorithms we used for calculating the variables in the initial solution. We begin by noting that from [14] we have the following: *In every tree of n > 1 nodes, there are at least two two leaf nodes.*

Proof. Assume for contradiction that G' = (N,E') is a tree of n > 1 nodes that has either one or zero leaf nodes. Start a walk at this node, or any node if there are no leaves, exiting each node on a different edge than by the one you entered. This is possible since every node except the single leaf have at least two incident edges. Since there are a finite number of nodes, eventually you will arrive at an already visited node on this walk. This creates a cycle, but since G' contains a cycle it cannot be a tree, and we get a contradiction. □

It is at these leaf nodes we begin calculating the flow values. By looking at the demand of these nodes, we immediately find the flow of the incident edge. Sending flow along this edge changes the demand of the leaf nodes to zero, and the adjacent nodes have their demands changed by the equivalent of the flow value. We then "remove" the leaf nodes and edges with flow calculated from the tree, giving us new leaf nodes. Repeating this process until all the edges in the tree have had their flow calculated completes the algorithm.

In order to calculate the dual and dual slack variables, we traverse all nodes, and edges not in the tree, using the equations $y_v = y_u + c_{uv}$ and $z_{uv} = y_u + c_{uv} - y_v$ from [13]. The complete code for finding the initial dual feasible tree solution is presented below.

```
input: a set of flights $F$, an events graph $G$
output:a tree solution $T$

%Step 1: Find initial tree
for each flight $f$ in $F$
        add backward edges on the taxi route to $T$
        check if $f$ is early or delayed
        if early
            add edges for $f$ being early to $T$
        if delayed
            add edges for $f$ being delayed to $T$add $t_f$ to $T$
        if on time
            add edges for $f$ being on time to $T$

%Step 2: Calculate dual variables
for each edge $e_{ij}$ in $T$
        $i$.dual $=$ $j$.dual + $e_{ij}$.weight

%Step 3: Calculate flow
Mark all edges as not visited
for each node $n$ in $T$
          $n.nbr$ $=$ number of adjacent nodes

while maximum of $nbr$ $>$ $0$
       for each edge $e_{ij}$ in $T$
                if $i$ or $j.nbr$ $=$ $1$ and $e_{ij}$ is not visited
                        Mark edge as visited
                        Set flow of edge according to demands
                        Update demands on incident nodes
                        Reduce $nbr$ of incident nodes by $1$
%Step 4: Calculate dual slack
for each edge $e_{ij}$ not in $T$
        $e_{ij}$.dual slack $=$ $i$.dual $-$ $j$.dual $+$ $e_{ij}$.weight
return $T$
```

Having found the initial tree solution representing how the flights ideally wish to travel, it is time to check whether or not this is possible. We run the conflict detection algorithm from section 5.3 to see if there are any conflicts in this solution. If there are any, solve them based on the discussion in chapter 5.3, and update the solution using the primal network simplex method. Below we go through all the steps required in order to find a conflict free optimal solution.

```
input: a set of flights $F$, an events graph $G$
output: an events graph $G$, a tree $T$

% Step 1: Obtain initial solution
for each flight $f$ in $F$
        add nodes and edges to $G$ according to taxi route events
        add nodes and edges to $G$ representing deviations
        obtain initial dual feasible tree $t_f$ for $f$
        add $t_f$ to $T$

Calculate the initial solution

%Step 2: Update the solution
while some flow $< 0$
        Update solution using the dual network simplex method

%Step 3: Detect and solve conflicts
Detect conflicts

while one or more conflicts are detected
        Make a decision on which conflict to solve first
        Add edges to $G$ representing precedence constraints
        Update solution using the primal network simplex method

return $G$, $T$
```

Chapter 7

Test runs and results

Having covered the theory needed to implement our solution method, it is time to put this theory to use. In this chapter we will present the different test runs done in order to test the algorithm, and the results we obtained. Our goal was to see how well the algorithm preformed in time when faced with increasingly complex problems, and to compare the different conflict resolution approaches. We start off by introducing the instances and settings used for this purpose.

Remark: All test runs were performed on a $Dell\ Latitude\ E6400$ using an $Intel\ Core2\ Duo\ T9600$ processor running at $2.8\ GHz$. The code has been written in C#, and is rather extensive (around 2400 lines), so it is not presented here. However, the code for finding an initial dual feasible solution can be found in the appendix.

7.1 Scenarios

In order to test our algorithm, we have used three different scenarios originating from the same data set obtained from the air traffic controllers at Stockholm Arlanda Airport. The data set contained information on each flights entry time, desired exit time, if it was an arrival, and which gate it was allocated. Table 7.1 gives an overview of the flights entry and desired exit times. We see from the table that many of the departures have ben scheduled to take off at the same time and so we can expect some conflicts to occur. We also make a note that all flights have a taxi time of 10, even arrivals parking at different gates, which is unrealistic, and so some deviation is expected.

Scenario 1: 10 Flights				Scenario 2: 20 Flights				Scenario 3: 30 Flights			
Arrivals		Departures		Arrivals		Departures		Arrivals		Departures	
Entry time	Exit time	Entry time	Exit time	Entry time	Exit time	Entry time	Exit time	Entry time	Exit time	Entry time	Exit time
305	315	300	310	305	315	300	310	305	315	300	310
320	335	300	310	320	335	300	310	320	335	300	310
308	318	300	310	308	318	300	310	308	318	300	310
310	320	305	315	310	320	305	315	310	320	305	315
313	315	305	315	313	323	305	315	313	323	305	315
				317	327	305	315	317	327	305	315
				318	328	305	315	318	328	305	315
				319	329	305	315	319	329	305	315
				320	330	305	315	320	330	305	315
						310	320	323	333	310	320
						310	320	324	334	310	320
								325	335	315	325
								327	337	320	330
								328	338		
								330	340		
								331	341		
								332	342		

Table 7.1: Scenarios

For every scenario, each flight was given a predetermined taxi route obtained from running the shortest path algorithm on a graph based on Stockholm Arlanda Airport. Figure 7.2 Shows an overview of the airport, while figure 7.3 shows a section of the airport represented as a graph. We used a fixed time window $[-5, 10]$ for the exit times, and the cost of deviating within this time window were set to be 4 for being delayed, and -2 for being early. For flights deviating outside this time window, the costs were set to be 16 and -8 respectively. The various time separations on conflicts and sequencing were set to 6 for conflicts, 2 for arrival separation, and 6 for departure separation. It was assumed that flights could wait at any point on the taxiway except the runway exit and also the runway entry for arrivals. We also assume that flights travel at the same speed.

7.2 Results

Before we present the results from the different scenarios, we make an important remark to be remembered when comparing the numbers: Conclusions about values of taxi times, deviations and total costs cannot be drawn from the scenarios, as some of the input data such as travel time of segments, arrival, departure, and conflict separations does not reflect real life scenarios. The values can however be used to compare the results of the scenarios with each other.

First we ran the algorithm with the conflict resolution approach of solving the first detected conflict (using the principle of shared paths), and then reoptimizing. The results are shown in table 7.2. We then changed the conflict resolution approach to the one described in section 5.4, giving us the results shown in table 7.3. Finally we did a run using 20 flights representing how air traffic controllers would handle the flights, by releasing them from the gate as early as possible. The result of this is also shown in the tables.

Scenario (First detected conflict)	Scenario 1	Scenario 2	Scenario 3	ATC
Number of flights	10	20	30	20
Number of arrivals	5	9	17	9
Number of departures	5	11	13	11
Number of nodes	440	825	1260	825
Number of edges	479	904	1379	915
Number of conflicts detected	10	33	83	33
Iterations required to reoptimize after conflicts	10	39	106	39
Separation edges required	14	42	100	42
Running time of code (seconds)	0.436	4.878	30.031	4.855
Running time of code After initial solution (seconds)	0.367	4.76	29.773	4.730
Total Desired taxi time	202	382	580	382
Total actual taxi time	211	424	753	445
Total wait time on taxiway	9	42	174	63
Total deviation (Base: Travel time)	0.6	0.65	0.754	0.6
Total cost	13560	27720	51740	27720
Average cost per time unit of deviation	9.58	9.59	9.66	9.59

Table 7.2

Scenario (First conflict in time)	Scenario 1	Scenario 2	Scenario 3	ATC
Number of flights	10	20	30	20
Number of arrivals	5	9	17	9
Number of departures	5	11	13	11
Number of nodes	440	825	1260	825
Number of edges	479	904	1379	915
Number of conflicts detected	10	27	68	27
Iterations required to reoptimize after conflicts	10	31	80	31
Separation edges required	14	35	84	35
Running time of code (seconds)	0.463	4.11	24.324	4.166
Running time of code After initial solution (seconds)	0.392	3.985	24.066	3.959
Total Desired taxi time	202	382	580	382
Total actual taxi time	211	424	753	445
Total wait time on taxiway	9	42	174	63
Total deviation (Base: Travel time)	0.6	0.65	0.754	0.6
Total cost	13560	27720	51740	27720
Average cost per time unit of deviation	9.58	9.59	9.66	9.59

Table 7.3

Comments

Complexity of the events graph: We see that the number of edges is very close to the number of nodes for all scenarios. Relating this to the discussion in section 4.4 this is the result of the assumption that flights can stop on all points. The same discussion also reveals that the number of edges can at most be $\mathcal{O}(2n) + c$ where n is the number of nodes and c the number of edges needed to resolve conflicts. This is the case when flights are not allowed to stop at any point. We also see that the average number of nodes required for each flights events is 42.4, while the average number of edges is 46.4. This gives an understanding how how the complexity increases for each added flight.

Conflict resolution: We see that the number of conflicts increase as the number of flights increase which is natural. In addition, the increase from 20 to 30 flights is bigger than the increase from 10 to 20 flights due to the increased complexity. The same goes for the number of primal iterations needed in order to obtain an optimal solution after resolving the conflicts, however we see that even for 30 flights, the number of iterations required is not much bigger than the number of conflicts, keeping in mind that the minimum number of iterations required equals the number of conflicts as the definition of conflicts requires at least one iteration to resolve. The number of required separation edges in order to resolve the conflicts are, as a result of flights traveling at the same speed, close to one for each conflict. The tables also show us how the two conflict resolution approaches compares to each other. We observe that the second method of resolving conflicts, by starting with the first in time, results in fewer conflicts needed to be resolved in total, and fewer iterations needed to obtain a conflict free optimal solution. This difference increases along with the increase in number of flights. Due to this reduction in conflicts, the number of required separation edges naturally also decreases.

Running time: Our first observation regarding the running time is that it increases quite drastically when the number of flights increases. However, finding the initial solution is done within 0.3 seconds even for 30 flights, and so we conclude that it is the conflict detection and resolution that is the major time consumer with 84% of the total running time for 10 flights, and 99% for 30 flights. Our second observation is that the second conflict resolution approach leads to a 3.7 second decrease in running time for scenario 3 due to resolving fewer conflicts.

Optimal solution: The optimal solution of the problem appeared to be identical regardless of which conflict resolution approach we used, perhaps due to using the same precedence conditions, however, we regard this as a coincidence, and testing on other data sets should disprove that this is always the case.

Comparing with air traffic controllers: Our algorithm tries to hold flights at the gate as long as possible before releasing them onto the taxiway. Comparing this with the air traffic controllers, who releases flights when they are ready, we see that the overall taxi time is reduced, without compromising the solution, by following the algorithms suggestion. This is due to flights having to wait ON the taxiway rather than at the gate in order to avoid conflicts, when following the controllers approach. As a result of this, the taxiway has fewer active flights over a longer period of time. We see this from figure 7.1 where the controllers release flights earlier than the algorithm, and since the optimal solution was the same, none of the flights exit earlier when directed by controllers.

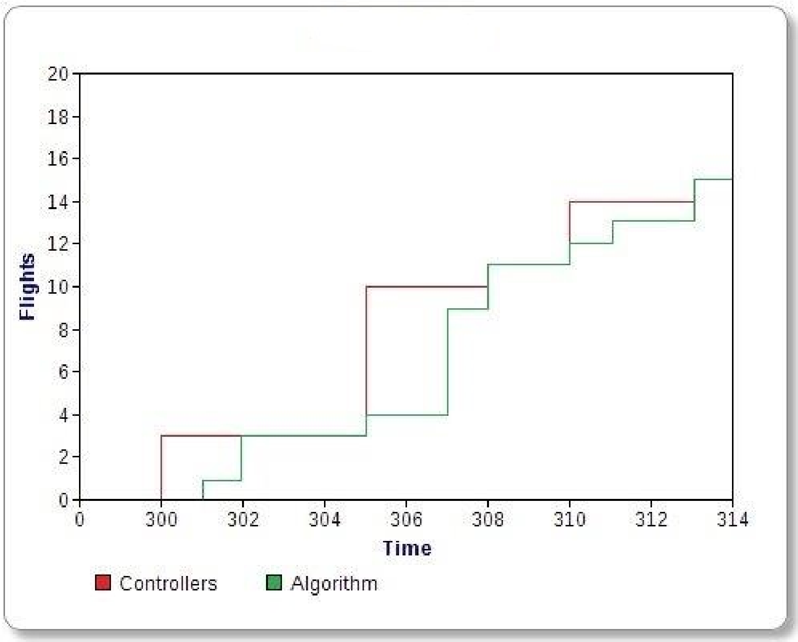
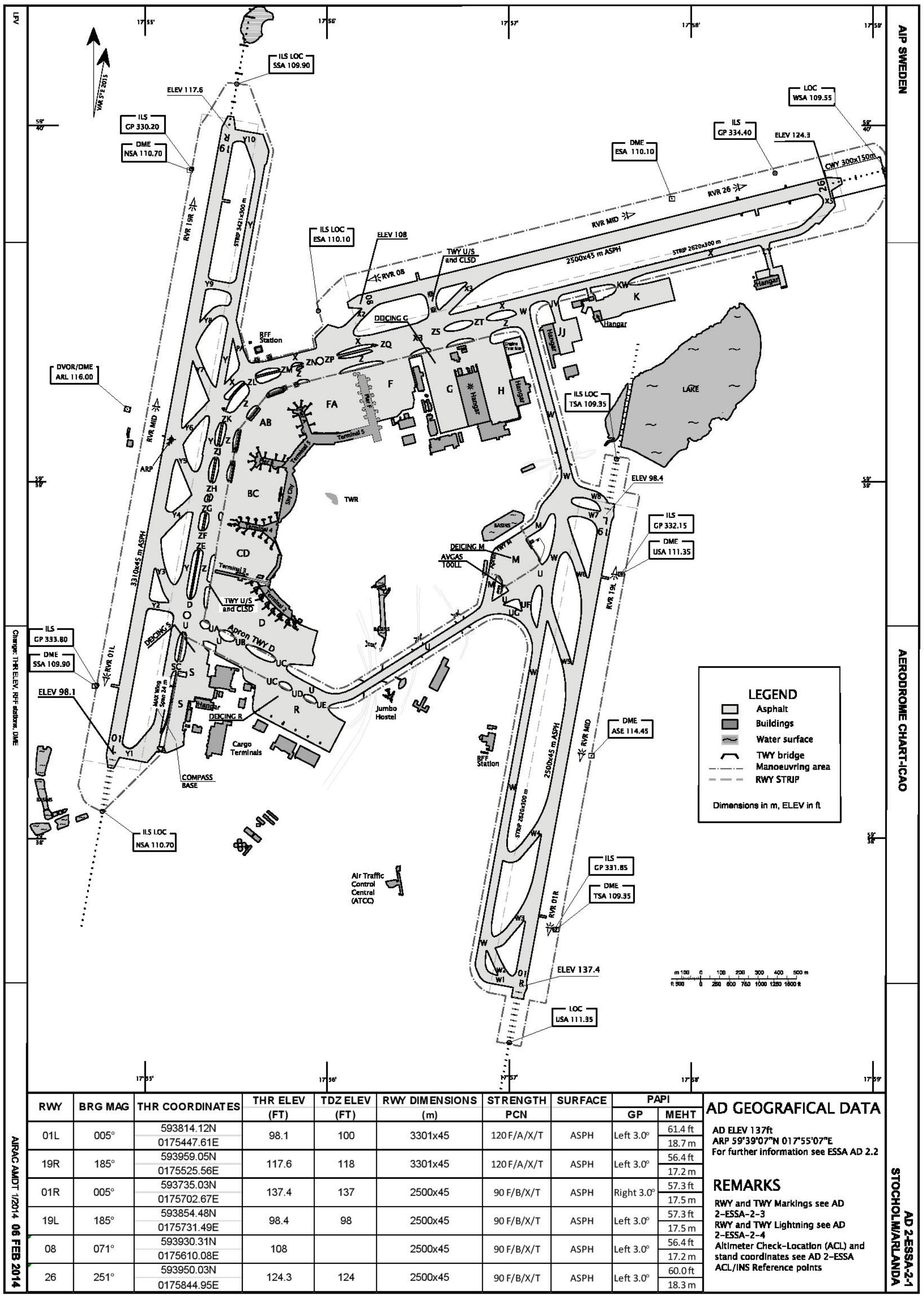


Figure 7.1: Active flights



RWY	BRG MAG	THR COORDINATES	THR ELEV (FT)	TDZ ELEV (FT)	RWY DIMENSIONS (m)	STRENGTH PCN	SURFACE	PAPI GP	PAPI MEHT
01L	005°	593814.12N 0175447.61E	98.1	100	3301x45	120 F/A/X/T	ASPH	Left 3.0°	61.4 ft 18.7 m
19R	185°	593959.05N 0175525.56E	117.6	118	3301x45	120 F/A/X/T	ASPH	Left 3.0°	56.4 ft 17.2 m
01R	005°	593735.03N 0175702.67E	137.4	137	2500x45	90 F/B/X/T	ASPH	Right 3.0°	57.3 ft 17.5 m
19L	185°	593854.48N 0175731.49E	98.4	98	2500x45	90 F/B/X/T	ASPH	Left 3.0°	57.3 ft 17.5 m
08	071°	593930.31N 0175610.08E	108		2500x45	90 F/B/X/T	ASPH	Left 3.0°	56.4 ft 17.2 m
26	251°	593950.03N 0175844.95E	124.3	124	2500x45	90 F/B/X/T	ASPH	Left 3.0°	60.0 ft 18.3 m

AD GEOGRAFICAL DATA

AD ELEV 137ft
ARP 59°39'07"N 017°55'07"E
For further information see ESSA AD 2.2

REMARKS

RWY and TWY Markings see AD 2-ESSA-2-3
RWY and TWY Lightning see AD 2-ESSA-2-4
Altimeter Check-Location (ACL) and stand coordinates see AD 2-ESSA ACL/INS Reference points

Figure 7.2: Stockholm Arlanda Airport

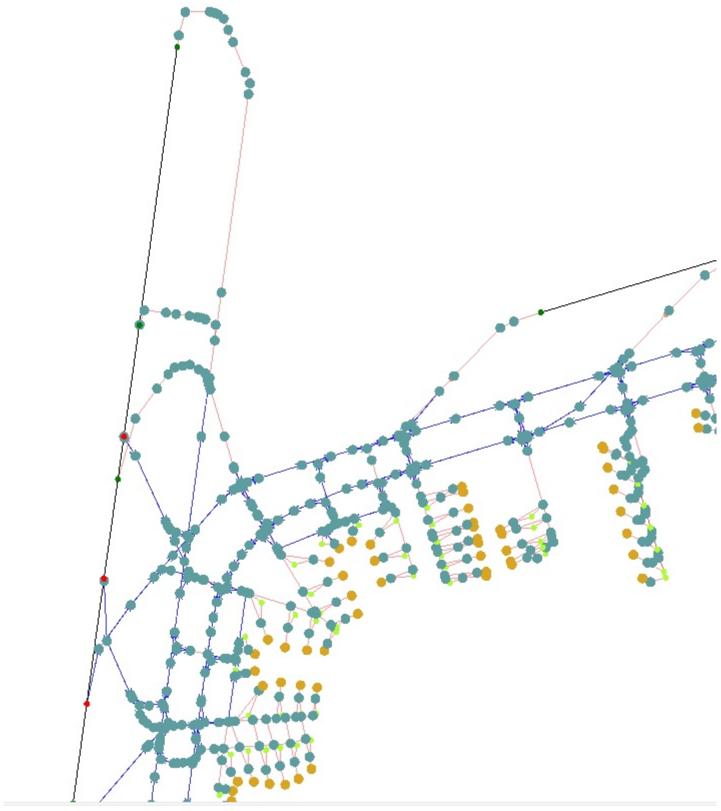


Figure 7.3: Graph of Stockholm Arlanda Airport

7.3 Conclusions

In this thesis we have shown how the ground movement problem can be formulated as a minimum cost flow problem to which the network simplex method can be applied. Given a sequence on the and taxi route, we are able to handle both arriving and departing flights, obtaining a conflict free solution. Even with our naive implementation of this algorithm, for instance we have not implemented dynamic tree algorithms, an initial solution can be found and updated within a second even for instances containing 30 flights. We have seen that the most time consuming part of our solution approach is detecting conflicts, however, making a "smart" choice on which conflict to solve first reduces the running time by up to 19% for our test cases by reducing the number of conflicts occuring. Still the rapid increase in total running time with the increase in number of flights are of concern, especially for airports with more traffic.

Comparing our algorithm to the air traffic controllers, we see that the algorithms optimal solution allows flights to wait longer at the gate before they start taxiing, reducing the overall taxi time. This could prove important as passenger delays are less likely to affect the actual off-block time of the individual flights. It also has an environmental and economical impact, as less fuel is spent du to not needing to have the engines running while at the gate. In addition, even less fuel is consumed, as the flights are less likely to have to slow down and then accelerate while taxiing, and research, (Stettler, Eastham, and Barrett, 2011)[16], has shown that accelerating aircrafts have close to twice the fuel consumption as non-accelerating aircrafts. As a result of this, airliners may tank up less fuel before a flight, reducing the weight and overall fuel consumption.

We have also shown how it is possible to find an initial dual feasible solution on any connected digraph. This method requires running the Bellman-Ford algorithm on multiple subsets when the graph is not strongly connected. Therefore, knowledge of how these subsets are connected can greatly improve the running time of the algorithm. A discussion on different approaches to conflict resolution has also been given, and results for resolving the first detected conflict and results for resolving the first conflict in time on a data set has been given. This thesis will be presented at the 2014 VeRoLog (EURO Working Group on Vehicle Routing and Logistics Optimization) conference in Oslo.

Chapter 8

Future work

A consequence of research will always be that the answers to the posed questions, give rise to new questions. The research area of this thesis is still very young, and vast resources are spent on it every year, just take the SESAR project as an example with an estimated budget of €2.1 billion for its 2020 horizon. This chapter gives some suggestions on topics to be focused on extending the work done in this thesis.

Additional data sets

The first natural step would be to obtain additional data sets from real life data, and run simulations on them. This should be done both for Stockholm Arlanda Airport, but also for different airports in order to get more varied situations. Doing this would improve the understanding of results, and additional conclusions may be drawn.

Efficiency

We have by no means claimed that our implementation of the algorithm is the most efficient one, in fact it is rather naive, and the overall running time may benefit greatly by improving the implementation. It would be interesting to see an implementation using for instance the *LEMON Graph Library* (www.http://lemon.cs.elte.hu/trac/lemon), as this is open source, and seems to contain some interesting implementations.

Another way to improve efficiency is to investigate different approaches for detecting conflicts. As we have seen, this is the major time consumer in our approach, and so have great potential for improvement. An idea here would be to somehow use the previous solution in order to guide the algorithm towards the next solution.

Stability

This is an aspect of conflict resolution which we have not discussed. In theory, conflict resolution could be used to obtain the optimal sequence of flights on the runway, however, careful considerations about changes affecting flights both in time and sequence from one solution to the next must be taken into account. If a solution requires a complete rearrange on the sequence of flights from the previous solution, this might not be possible as extensive communication with pilots would be required. Therefore, investigation into how much change can be tolerated, and the implications of these changes to each flight should be considered.

Integration with other airport operations

Improving stability of the solutions would lead to more reliable input to other airport operational problems, such as gate allocation, baggage handling and deicing sequencing. It is not unlikely that in the future solutions to these problems combined form the foundation to reduce the overall deviations of flights at an airport. Improvements in all these areas would give the passenger more reliable information on their flights departure.

Integration with other airports

Even though a lot of research is needed before a study of integration with other airports is done, we still mention it as a final topic. When stability of solutions have been improved, more reliable data on flights can be sent to different airports, improving the estimation of arrival times. This gives both air traffic controllers, and optimization algorithms a better chance of planning ahead, improving their choices, and obtaining a better solution.

Appendices

Appendix A

Class objects

Node objects

```
public Node(T t, double demand, bool isHoldingNode,
           double dual)
{
        Number = t;
        Demand = demand;
        DemandLeft = demand;
        Dual = dual;
        IsHoldingNode = isHoldingNode;
        OutgoingEdges = new List<Edge>();
        IncomingEdges = new List<Edge>();
}
```

Edge objects

```
public Edge(Node n1, Node n2, double weight, bool isDirected,
            U associatedObject)
{
        DualSlack = 0;
        marked = false;
        Flow = 0;
        IsDirected = isDirected;
        Node1 = n1;
        Node2 = n2;
}
```

Flight objects

```
public Flight(String id, int flightNumber, String startPoint,
              String endPoint, double entryTime,
              double exitTime, boolarrival, Airport airport )
{
        ID = id;
        FlightNumber = flightNumber;
        StartPoint = airport.Network.FindWaypoint(startPoint);
        EndPoint = airport.Network.FindWaypoint(endPoint);
        EntryTime = entryTime;
        ExitTime = exitTime;
        Arrival = arrival;
        RouteNodes = new List<Graph<Node, Edge>.Node>();
        RoutePoints = new List<Point>();
}
```

Appendix B

Finding initial dual feasible solution

Adding nodes and edges for taxiing and sequence

```
public void TaxiAndSequence
input: List<Point> taxiPoints
       List<DirectedSegment> taxiSegments
       Flight flight
       List<List<Point>> taxiPlan
       Dictionary<Node, Point> dictArrival
       Dictionary<Node, Point> dictDeparture
       Graph<int, string> graph
output:

int counter = 0;
foreach(Point point in taxiPoints)
{
    % First point in taxi route
    if (counter == 0)
       if(flight.arrival == true)
         Node newNode =
                graph.AddNode(graph.Nodes.Count + 1, 0, false);
          flight.RouteNodes.Add(newNode);
          flight.RoutePoints.Add(point);
          dictArrival.Add(newNode, point);
       else
         Node newNode = graph.AddNode(graph.Nodes.Count + 1, 0,
                                              point.IsHoldingPoint));
         flight.RouteNodes.Add(newNode);
         flight.RoutePoints.Add(point);
         dictDeparture.Add(newNode, point);
      graph.AddEdge(Nodes.Count, 1, - flight.EntryTime, true,
                    "e" + graph.Nodes.Count + "_" +1);
    % Other points in taxi route
    else
        Node newNode =
                graph.AddNode(graph.Nodes.Count + 1, 0, true);
                flight.RouteNodes.Add(newNode);
                flight.RoutePoints.Add(point);
       if (flight.Arrival == true)
          dictArrival.Add(newNode, point);
       if (flight.Arrival == false)
          dictDeparture.Add(newNode, point);
       if (newNode.IsHoldingNode == false)
          graph.AddEdge(graph.Nodes.Count - 1, graph.Nodes.Count,
             taxiSegments[counter -1].Segment.Distance, true,
             "e" + graph.Nodes.Count -1 + "_" + graph.Nodes.Count);
          graph.AddEdge(graph.Nodes.Count, graph.Nodes.Count - 1,
             taxiSegments[counter-1].Segment.Distance, true,
             "e" + graph.Nodes.Count + "_" + graph.Nodes.Count-1);
      else if (newNode.IsHoldingNode == true)
          graph.AddEdge(graph.Nodes.Count, graph.Nodes.Count - 1,
             taxiSegments[counter-1].Segment.Distance, true,
             "e" + graph.Nodes.Count + "_" + graph.Nodes.Count-1);
counter += 1;
}
```

Include edges from taxi route in the initial tree

```
public void InitialFlightTaxiTree
input: Node startNode
       List<Node> routeNodes
       List<Edge> Tree
       List<int> parents

% Add nodes and edges
foreach (Node node in routeNodes)
        foreach (Edge edge in node.OutgoingEdges)
                parents[edge.Node1.Number - 1] = edge.Node2.Number;
                Tree.Add(edge);
                node.Dual = edge.Node2.Dual - edge.Weight;
```

Include edges from flight deviation in the initial tree

```
public void InitialFlightDeviationTree
input: Node startNode
       List<Node> DeviationNodes
       List<Edge> Tree
       Flight flight
       List<int> parents
       double delayWindow
       double earlyWindow

% If flight is early
if (flight.RouteNodes[last index].Dual < flight.DesiredExitTime)
  parents[flight.RouteNodes[last index].OutgoingEdges[1]
        .Node2.Number1] =
        flight.RouteNodes[flight.RouteNodes.Count - 1]
        .OutgoingEdges[1].Node1.Number;
   Tree.Add(flight.RouteNodes[last index].OutgoingEdges[1]);
   for (int i = 0; i <= 3; i += 2)
   {
      foreach (Edge edge in DeviationNodes[i].OutgoingEdges)
      {
          if(edge.Node2.Number-1] = edge.Node2.Number)
            parents[edge.Node1.Number - 1] = edge.Node2.Number;
            Tree.Add(edge);
      }
   }
% If flight is very early
if (flight.RouteNodes[last index].Dual <=
                        flight.DesiredExitTime - earlyWindow)
   parents[DeviationNodes[1].OutgoingEdges[0].Node2.Number-1] =
                 DeviationNodes[1].OutgoingEdges[0].Node1.Number;
   Tree.Add(DeviationNodes[1].OutgoingEdges[0]);

%If flight is delayed
if (flight.RouteNodes[last index].Dual > flight.DesiredExitTime)
  foreach (Edge edge in DeviationNodes[0].OutgoingEdges)
  {
    if(edge.Node2.Number == flight.RouteNodes[last index].Number)
       parents[edge.Node1.Number - 1] = edge.Node2.Number;
       Tree.Add(edge);
  }
  for (int i = 1; i <= 3; i += 2)
  {
     foreach (Edge edge in DeviationNodes[i].IncomingEdges)
     {
       if (edge.Node1.Number == 1)
         parents[edge.Node2.Number-1] = edge.Node1.Number;
         Tree.Add(edge);
     }
  }
% If flight is very delayed
if (flight.RouteNodes[last index].Dual >=
                       flight.DesiredExitTime + delayWindow)
   parents[DeviationNodes[0].IncomingEdges[0]
          .Node1.Number - 1] = DeviationNodes[0]
                            .IncomingEdges[0].Node2.Number;

% If flight is on time
if (flight.RouteNodes[last index].Dual ==
                                     flight.DesiredExitTime)
  parents[DeviationNodes[3].IncomingEdges[0]
                            .Node2.Number - 1] =
                         DeviationNodes[3].IncomingEdges[0]
                                           .Node1.Number;
  parents[DeviationNodes[1].IncomingEdges[0]
                           .Node2.Number - 1] =
                         DeviationNodes[1].IncomingEdges[0]
                                            .Node1.Number;
  parents[DeviationNodes[0].OutgoingEdges[0].
                            Node1.Number - 1] =
                         DeviationNodes[0].OutgoingEdges[0]
                                           .Node2.Number;
  parents[DeviationNodes[2].OutgoingEdges[0].
                            Node1.Number - 1] =
                         DeviationNodes[2].OutgoingEdges[0]
                                           .Node2.Number;
  Tree.Add(DeviationNodes[3].IncomingEdges[0]);
  Tree.Add(DeviationNodes[1].IncomingEdges[0]);
  Tree.Add(DeviationNodes[0].OutgoingEdges[0]);
  Tree.Add(DeviationNodes[2].OutgoingEdges[0]);
```

Calculate variables and update solution

```
public void CalculateDualSolution
input: Node startNode
```

```
       Graph<Node, Edge> airport
       List<Edge> Tree
       List<int> parents
output: List<int>[] Children

% Initialize required lists
Children = new List<int>();
foreach (Node node in airport.Nodes)
{
  Children[node.Number-1] = new List<int>();
}
for (int i = 1; i <= parents.Count-1; i++)
{
   Children[parents[i]-1].Add(i+1);
}
List<bool> Explored = new List<bool>();
foreach (Node node in airport.nodes)
{
   Explored.Add(false);
}

% Calculate variables
airport.InitialFlow(Tree)
airport.dualCalc(sartNode, Explored, Tree);
airport.dualSlackCalc(Tree);

% Update solution
while (airport.MinFlow(Tree).Flow < 0)
{
   airport.dualStep(Tree, parents, Children)
}
return Children
```

Bibliography

[1] International Federation of Air Line Pilots' Associations, *Vision statement: The future of air navigation. Revision 2.0*, February 2011.

[2] SESAR, *Milestone deliverable D1. Air transport situation: The current situation. Edition 3, EUROCONTROL*, July 2006.

[3] SESAR, *European ATM master plan. Edition 2*, October 2012.

[4] Schüpbach K, Zenklusen R, *Approximation Algorithms for Conflict-Free Vehicle Routing*, 640-651, 2011.

[5] Rathinam S, Montoya J, Jung Y, *An optimization model for reducing aircraft taxi times at the Dallas Fort Worth International Airport*, 2008.

[6] Atkin J.A.D., *On-line decision support for take-off runway scheduling at London Heathrow airport*, Ph. D. Thesis, University of Nottingham, 2008.

[7] Ahuja, Magnanti, Orlin, *Network Flows - Theory, algorithms, and applications*, Prentice hall, 1993.

[8] Balakrishnan H, Jung Y, *A framework for coordinated surface operations planning at Dallas-Fort Worth International Airport*, 2007.

[9] Ravizza S, *Enhancing decision support systems for airport ground movement*, Ph. D. Thesis, University of Nottingham, 2013.

[10] Marín A, Codina E, *Network design: Taxi planning*, 2008.

[11] Bianco L, Dell'Olmo P, Giordani S, *Scheduling models for air traffic control in terminal areas*, 2006.

[12] Atkin J.A.D., Burke EK, Ravizza S, *The airport ground movement problem: Past and current research and future directions*, 2010.

[13] Vanderbei R, *Linear Programming Foundations and Extensions. Edition 3*, 2008.

[14] Bertsimas D, Tsitsiklis J *Introduction to Linear Optimization*, 1997

[15] Kirsner S, Acquisition puts Amazon rivals in awkward spot, Article in Boston Globe, December 2013.

[16] Stettler MEJ, Eastham S, Barrett SRH, *Air quality and public health impacts of UK airports. Part I: Emissions. Atmospheric Environment*, 2011.