Miriam Lobato da Rosa

Airport Ground Movement Scheduling

Miriam
Lobato da Rosa

Airport Ground Movement Scheduling
Sequenciamento de Movimentos Terrestres nos Aeroportos

Internship Report submitted to the University of Aveiro to fulfill the requirements for obtaining a Master's degree in Mathematics with a minor in Statistics and Operational Research, conducted under the scientific guidance of Professor Dr. Agostinho Miguel Mendes Agra, Assistant Professor, Department of Mathematics, Aveiro University.

– *à minha avó*

the jury

president

Professor Doctorate Isabel Maria Simões Pereira
Auxiliary Professor of Mathematics Department of Aveiro University

examiners committee

Professor Doctorate Rui Jorge Soares Borges Lopes
Auxiliary Professor of Economic, Management and Industrial Engineering Department of Aveiro University

Professor Doctorate Agostinho Miguel Mendes Agra
Auxiliary Professor of Mathematics Department of Aveiro University (guidance)

acknowledgements

I would like to thank to Dr. Agostinho Miguel Mendes Agra, who I worked with during all Master's Degree, in particular during the Internship, for his support, work and dedication, and above all for believing in what I was capable of doing.

To Wide Scope and its staff, for accepting the Internship and specially to Dr. Filipe Carvalho for teaching, comprehending and helping me through all difficulties along the internship.

To all my family, without them I wouldn't have conquered this goal in my life and specially to my dear friend Sandra Coutinho, who stood by me and supported me in many ways while I was in Aveiro.

To everyone that, directly or indirectly, helped me.

Keywords airport operations; ground movement; scheduling.

Abstract

Worldwide air traffic tends to increase and for many airports it's no longer an option to expand terminals and runways, so airports are trying to maximize their operational efficiency. Many airports already operate near their maximum capacity. Peak hours imply operational bottlenecks and cause chained delays across flights impacting passenger, airlines and airports. Therefore there is a need for the optimization of the ground movements at the airports.

There are three major problems concerning airport operations: the departures and arrivals sequencing on the runways; the staff management operations preceding the green light for aircraft to leave the gate; and the ground movement between the gate and the runway (and reverse).

The scope of this work is the ground movement problem that interacts with the other two scheduling problems mentioned and provides decisions in real-time.

The ground movement problem consists of routing the planes from the gate to the runway for takeoff or on reverse path, and to schedule their movements.

Our approach proposes a fast optimization system that considers a set of planes moving to and from a set of runways along a given road network conditioned by the airport ground layout. It considers constraints such as the route constraints, separation between aircrafts due to jet blast, aircraft movement speeds, timing constraints for arrivals and departures in a constantly changing environment.

The objective is to minimize fuel consumptions on the ground (from the airline perspective) and to minimize the time spent on the time window slot for occupying the airport ground (from the airports perspective) while granting all safety regulations at all times. Also passengers and the environment benefit from an optimized ground movement.

The optimization approach proposed provides a fast heuristic solution for each real-time event generated respecting all the rules established by Advanced Surface Movement, Guidance and Control Systems (A-SMGCS) of the International Civil Aviation Organization (ICAO).

Palavras Chave: operações nos aeroportos; movimentos em terra; sequenciamento.

Resumo

O tráfego aéreo no mundo está em crescimento e para a maioria dos aeroportos não é uma opção expandir os terminais ou as pistas, fazendo com que estes tentem maximizar a eficiência operacional. Muitos aeroportos estão a operar perto da sua capacidade máxima. Horas de ponta implicam engarrafamentos e causam simultâneos atrasos ao longo de toda a cadeia de operações com consequências para passageiros, companhias aéreas e aeroportos. Por estes motives há uma necessidade de otimização dos movimentos no solo que ocorrem nos aeroportos.

Existem três grandes problemas no que diz respeito às operações dos aeroportos: o sequenciamento das partidas e chegadas; a gestão das operações que precedem a "luz verde" para que o avião possa sair do *stand*; e os movimentos no solo entre o *stand* e a pista (e o oposto).

O âmbito deste trabalho enquadra-se nos movimentos no solo que interagem com os dois outros problemas de sequenciamento mencionados e fornece decisões em tempo real.

O problema dos movimentos terrestres consiste em estabelecer o roteamento dos aviões desde o *stand* até à pista para levantarem voo, ou no caminho inverso, e sequenciar os seus movimentos.

A nossa abordagem consiste numa otimização rápida que considera um conjunto de aviões a moverem-se de, e para, a pista, e uma rede condicionada pela planta do aeroporto. Considera, ainda, restrições tais como: de rota; separações entre aviões devido ao *jet blast*; velocidade de cada avião; de tempo para chegadas e partidas, num ambiente em constante mudança.

O objetivo é minimizar o consumo de combustível enquanto os aviões estão no solo (da perspetiva das companhias aéreas) e minimizar o tempo despendido em cada *slot* de janela temporal na ocupação do espaço terrestre do aeroporto, garantindo todas as regras de segurança. Também os passageiros e o ambiente beneficiam de um conjunto de movimentações em terra otimizadas.

A otimização proposta fornece uma solução heuristica rápida para cada evento em tempo real respeitando todas as regras estabelecidas no *Advanced Surface Movement, Guidance and Control Systems (A-SMGCS)* da Organização Internacional de Aviação Civil (ICAO).

– *Miriam Lobato da Rosa*

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This Internship report was conducted under the Master's degree in Mathematics with a minor in Statistics and Operations Research at Aveiro University. The main purpose was to apply all knowledges acquired over the time passed in Master Degree, pursue new techniques and learn about professional applications as well. The Internship was carried out at Wide Scope in Lisbon.

Wide Scope's was founded in 2003 from the belief in "determination to win". One important detail about Wide Scope is that the company is 100% Portuguese. Wide Scope operates not only in Portugal, but is also present in multiple countries from different continents through partners.

Wide Scope is a consultant in information technology specialized in combinatorial optimization solutions, applied mathematics and artificial intelligence.

The company is operates in the three market segments (Optimization, Consulting and Mobile) which it operates in.

Regarding optimization, Wide Scope designs and implements solutions of combinatorial optimization. Concerning Consulting segment the company implements solutions based on JAVA enterprise technology and it is the first official partner of Liferay. Respecting Mobile segment, Wide Scope has a unit to develop mobile applications for iPhone, Android, Blackberry and Windows Mobile, which represent, already, relevant references in financial sector, entertainment and publishing.

Companies like EMSA, TAP, SIBS, Jeronimo Martins and Fiat are some of Wide Scope's clients.

Simultaneously, Wide Scope also invests in research and development, from which the best optimization technologies rise. All their researches, innovation and best optimization techniques were developed by Wide Scope's research team.

During the Internship I was able to be part of a case study concerning airport ground movement.

The main objective of the project was to improve and optimize the ground movement of

airports, treating the problem from a scheduling point of view.

With air traffic growing fast, airports have to respond to a fast growth of demand and as consequence airlines have to increase their fleet, destinations and number of connections. To meet these needs, airports have to operate in their maximum capacities and in order to do that it is necessary to optimize all operations. For example, ground movement is one of the operations taking place at airports, optimizing it will help by making less delays, provoked by moving from a place to another at the airport and prevent conflicts as well.

The objective is to minimize: time spent taxiing, for the airport's point of view; fuel consumption, for airlines point of view, which could have a good environmental impact; and time spent at airplane for passengers.

The ground movement problem begins each time an airplane is ready to go from its origin to its destination. If it is a departing airplane then the origin is at the gate, and destination is the runway. If it is an arriving airplane, the origin is the runway, and the destination is the gate. Every time an airplane is ready, it is necessary to schedule the path along taxiways and besides respecting the airport's layout, it is also necessary to respect some restrictions.

The results found with this study were promising and allowed airplanes to save about 5% of taxiing time. This way delays and conflicts at the ground are reduced as well.

This report was divided in 7 chapters and an appendix. After the introduction, the problem description is made at Chapter 2. Here, all basic concepts are introduced and some assumptions are explained. In the next chapter, Chapter 3, is described a mathematical model to solve the problem giving an exact solution, furthermore it is explained all input data an decision variables. Since exact methods aren't always capable of finding the solution in a reasonable time, heuristic methods were implemented and applied to the problem. Chapter 4 explains the Local Search Heuristics and meta-heuristic, Tabu Search, used to improve the initial solution. Next, at Chapter 5 all results are presented and compared. The last chapter, Chapter 6, conclusions are discussed and some future work in this study is suggested.

Chapter 2

Problem Description

With the world's development and globalization, people need to travel around the globe as quickly and safest as they can. Everyone is our neighbor and the fastest, cheapest, and sometimes, the only way to reach destination, is by airplane. Most frontiers are nowadays open, so catching a flight and enter some country is very easy. One of the consequences is the increasing air traffic.

EUROCONTROL Seven-Year Forecast states "*At European level, the traffic forecasts remain mostly unchanged: a moderate growth of 1.2%* ($\pm$1.2%) *for 2014 and more steady growth of 2.7%* ($\pm$1%) *for 2015. From 2015 onwards, growth is expected to be back at around 2.7% per year, on average. The 2008 peak of traffic of 10.1 million flights is forecasted to be reached again in 2016; (...). In the first part of the horizon (2015-2018), growth rates will average at around 2.5%, falling off to 2.2% in 2018 when capacity constraints will increasingly affect the demand in Europe. In the last two years 2019-2020 will see traffic growth rates averaging at around 3% as additional capacity brought in Turkey will lift the pressure on the whole network. For the whole 2014-2020 period, flight growth averages 2.5% per year in the base scenario.*"[5].

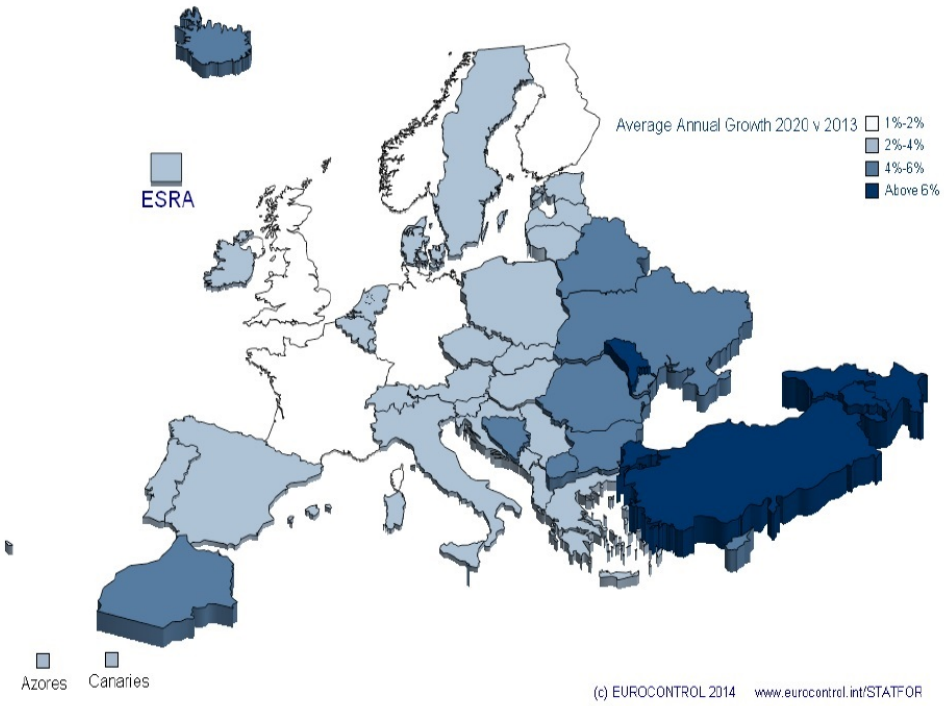


Figure 2.1: Average annual flight growth 2013-2020 per State (source:EUROCONTROL)

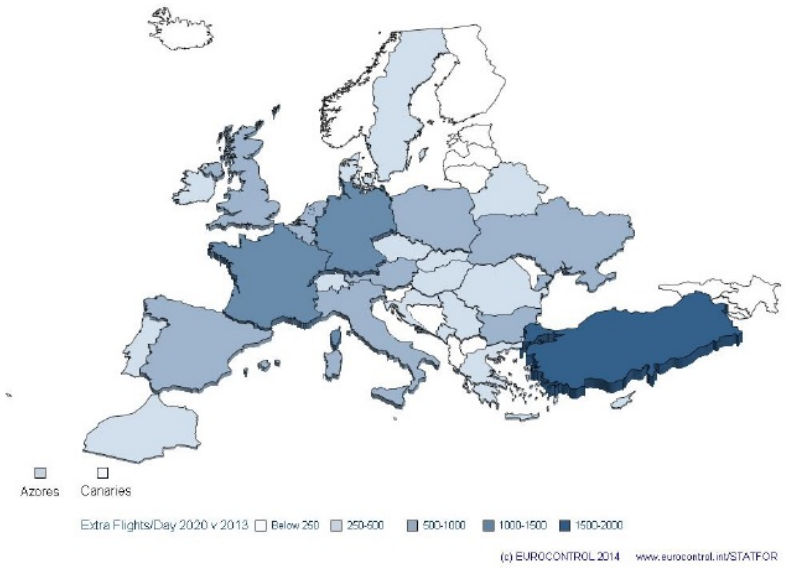


Figure 2.2: Number Of Additional Movements per Day of Each State (source:EUROCONTROL)

In what concerns to the world, the growth was equally high.

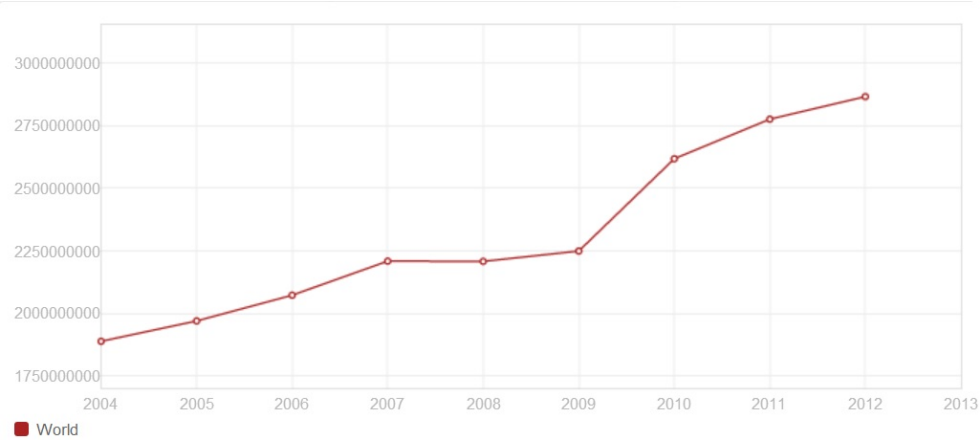


Figure 2.3: Air transport, passengers carried (source: World Bank)

Due to the fast growth of air traffic, airports have to maximize their capacities. For many of them, it's no longer an option to expand terminals and runways. In order to satisfy the demand, airports must optimize their operations maintaining high security levels while minimizing conflicts. Facing this new reality International Civil Aviation Organization (ICAO) proposed an Advanced Surface Movement, Guidance and Control Systems (A-SMGCS).

Currently the Surface Movement, Guidance and Control Systems (SMGCS) procedures are based on the principle of "see and be seen". According to the manual, the number of accidents during surface movements is increasing, because of air traffic growth, the increasing number of operations that take place in low visibility conditions and the complexity of airports layout. Facing these problems ICAO proposed an upgrade to the system. The new manual specifies some system objectives and functions. An A-SMGCS should support some primary functions, for example, surveillance, routing, guidance and control. The same A-SMGCS should be capable of assisting authorized aircraft and vehicles to maneuver safely and efficiently on the movement area [8].

Ground movement is critical since it links all airport operations. Airport operations are: the passenger and baggage management; the gate assignment; runway aircraft assignment (when there is more then one); departing sequences; arrival sequences and taxiway planning.

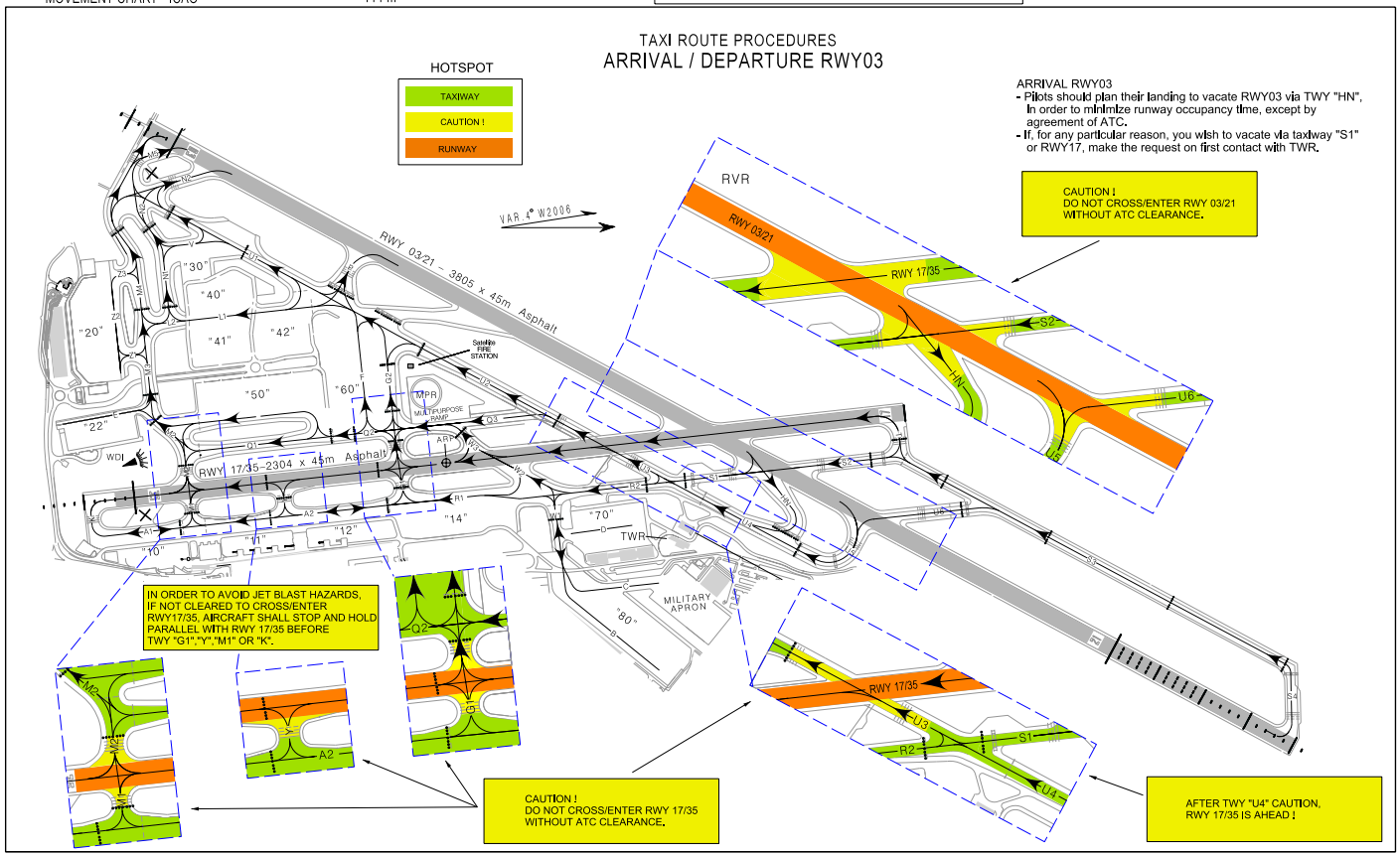
Before proceeding, it is important to understand some concepts about airports layout. As an example is Lisbon airport layout (fig. 2.4).

AERODROME GROUND
MOVEMENT CHART - ICAO

APRON ELEV
114 m

ATIS – 124.150	TWR – 118.100 /118.500
CLD – 118.950 /118.500	GND – 121.750 /118.500

LISBOA / LPPT



Text Boxes updated

Figure 2.4: Lisbon Airport Diagram

A **runway** is "*the paved surface design for aircraft take-off and landing. Runways have different designated orientations (...) and generally some distance from the terminal buildings. The runways may be parallel, offset or intersecting*"[4]. Furthermore is important to know some particularities. Near the runway entrance and exit exists ***Holding points***. Holding points are "*a place indicated by painted ground markings illuminated signage and (often) stop bars where aircraft stop until they are authorized to enter the runway*".

A **taxiway** is "*paved way for aircraft to move to and from the terminals and different parts of the airport*"[4]. At taxiways exists ***Stop bars*** and ***Holding bars***. At Holding bars aircrafts must wait until receive instructions to proceed, while a stop bar is "*a series of lights indicating whether access to a runway is authorized or not*"[4] . The aircraft can proceed when it changes to green.

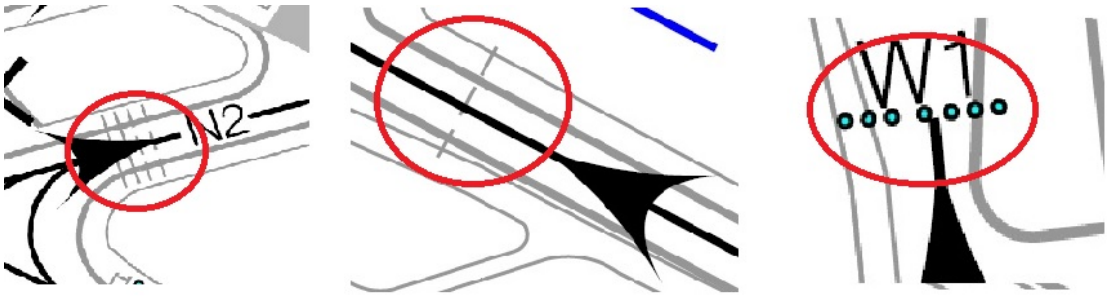


Figure 2.5: (From left to right) Holding Point near the Runway, Holding Point at Taxiway and Stop Bar at Taxiway

Conflicts can be divided in three categories, with the following priority, according A-SMGSC manual [8]:

1. Runway conflicts;
2. Taxiway conflicts; and
3. Apron/stand/gate conflicts.

More details and examples of conflicts are given in appendix.

In line with previous research [1] some assumptions were made:

a) In this problem the routes for each plane are not pre-determined, so every time an aircraft enters the system (receives green-light or reaches a holding point area at the runway after landing) routes are redefined for every plane;

b) It is possible to ensure that two airplanes don't conflict with each other by applying constraints while they are taxiing, maintaining a certain distance between them. All parameters are in agreement with what is set in A-SMGCS manual;

c) Movement speeds are set by A-SMGCS manual and depend, not only, on type and size of each aircraft but also on which taxiway the airplane is maneuvering;

d) The gate is already assigned when the aircraft arrives;

e) The sequence of runway users is, also, known;

f) For the departing aircrafts is also important to have into account the order they reach the runway, because it must respect the departing sequence. So the first to take-off must be the first to attend to the runway.

The ground movement problem is, in other words, a routing and scheduling problem. These problems can be easily solved for low-activity airports or for low-activity periods. But that is not what happens for most international airports. There are several aircrafts moving from and to the runways. The complexity of the problem relies on safety, meaning that aircrafts cannot conflict with one another.

The problem starts either when an aircraft is authorized to move from its parking position, to a position where it can start taxiing or when it lands at the runway.

From the departing aircraft's point of view, start engine represents the beginning of taxi time. In real time cases there are two kinds of actions for departing aircrafts. These airplanes have a start engine, when they turn on the engine and the green light when they can start taxiing. So the main objective is to optimize the time between the moment the airplane receives green light and the moment it starts taxiing. When the instruction is given, the goal is to reach the runway as soon as possible, without any conflicts. Obviously the problem becomes more complicated and more complex the higher the number of airplanes maneuvering at taxiway are.

For the arriving aircrafts the problem only begins when they reach the holding point that gives access to the taxiway. The goal is to reach the gate as soon as possible.

2.1 Related Literature

In 2001 at the Genetic and Evolutionary Computation Conference a Genetic Algorithm [11] was applied to the airport ground movement problem. In this article, a taxi optimization tool was introduced and tested on Roissy Charles De Gaulle Airport. The authors reflect on the fact that an optimal path doesn't mean that is the shortest one. The definition of optimal path depends on the goal of the project. They also reinforce the idea that is very difficult to predict an airplane's future positions because of all the delays and exact landing time. In order to have a better accuracy it is necessary to update the situation from time to time. So every δ minutes there is an update. Also, the time window considerate is $T_w > \delta$. In this case only aircrafts landing or taking-off in the time window will be considered. The Genetic Algorithm is used every δ minutes.

The Fitness function was used to ensure that solutions without any conflicts were the best.

A local fitness value, F_i is defined for each aircraft i of a population element. In what concerns the crossover operator, first two population elements are randomly chosen. For each parent A and B their fitness function is compared $(A_i; B_i)$ and the children will take aircraft i of the best parent.

In order to prevent the algorithm from a premature convergence a sharing process is implemented.

The experimental results are done with three hypotheses: random hypothesis; exact hypothesis and middle hypothesis. All of these have to due with the allocation of aircraft to the runway.

The authors were able to conclude that the Genetic Algorithm is very efficient.

Later in 2004, a mixed-integer programming (MIP) formulation was introduced to represent the movement of an aircraft on the surface of the airport [13]. Real data from Amsterdam Airport Schipol was used to demonstrate that the algorithms lead to significant improvements of efficiency, with reasonable computational effort. The route is given and an arrival or departure time for each aircraft is given as well. The decision relays on the time that each airplane is going to leave a particular point at the airport such that no conflicts occur, and all airplanes meet the time requirements. A mathematical formulation as MIP was presented.

For this problem three different variants of rolling horizon algorithms were implemented. For the first variant, the planning period is split in a series of disjunct time intervals of equal length. Aircrafts are assigned to an interval, based on the earliest possible time they can start taxiing. In each iteration the aircrafts belonging to an interval are scheduled using CPLEX optimizer based on a MIP formulation. The second variant considers that, if an aircraft that was scheduled in a previous interval but will reach some nodes of its route in the current interval, the part of the route of this aircraft that lies within this time interval is now allowed to be rescheduled. The third approach has a sliding window that tries to spread the undesirable effects among all aircrafts. The number of airplanes considered depends on the length of the sliding window. So the problem also relies on the optimal window.

Using all three methods the authors came into the conclusion that the optimal time window was fifteen minutes. It was possible to reduce the average delay from 20% to 2%.

In 2008 Keith and Richards presented an "Optimization of Taxiway Routing and Runway Scheduling" using Mixed Integer Linear Programming (MILP)[9]. The scope of the paper is a method for globally optimizing runway and taxiway operations together. Their objective function is a weighted combination of the final time at the runway, the total of taxi time, a terminal penalty based on a distance left to travel and the total taxi time. In order to prevent some conflicts they introduced some constrains, for example, Routing Constraints to ensure that valid routes are considered. Taxi Timing constrains granting that separation between two airplanes is maintained, and runway timing that are used to enforce take-off separation, because of the vortex wake.

Thought, their approach was not applied to real data. They only replicate London Heathrow Airport's north runway east holding point structure. To compare results, the time of the last movement of the aircraft was used as a metric and they were able to quantify all possible benefits that the optimizer could bring. They could save an average of 20%.

Another study done in 2008 by Roling and Visser [12] had on basis the work developed by Smelting et al. [13] and also used a MILP formulation. The major difference between these two works is that the method implemented by Smelting allowed range speeds and did not permit holding or rerouting. Another different aspect is that Smelting et al. did not used discretized time, so it was not possible to update planning. The authors considered an horizon planning of T and they assume that at the beginning of each planning update, a complete set of schedule taxi movements is available for the planning interval $[t_0; t_0 + T]$. They used a discrete time-space network to represent taxi-planning system.

Rolling and Visser present their Taxi Planning Model as a MILP model on which the objective function equals a weighted combination of the total taxi time and total holding time. They considered as constraints the node occupancy, the link occupancy, route and delay choice, and waiting times.

To test their model and theory they used an hypothetical airport layout. They were able to conclude that, in order to avoid conflicts, some airplanes should be rerouted from their shortest path.

In 2010 a survey was conducted comparing some researches made about this problem [1]. The survey states that the first approach was to develop a Mixed Integer Linear Programming (MILP). Models were formulated, but the MILP solver wasn't capable of finding a solution in a reasonable time, so heuristics methods were applied. So far the only ones used were Genetic Algorithm (GA), as according to the survey.

This survey reviewed MILP - related research and then the Genetic Algorithms.

Some of the MILP approaches considered a predefined set of routes for each aircraft and others, besides the scheduling problem had a routing problem too. Also, some of the researches considered only one objective, minimize taxi time. As for others considered a multi-objective function, minimize taxi time, reducing delays of departures, reducing delays of arrivals, attempt to maximize the number of arrivals and take-offs and minimize taxi distance.

In what concerns GA, this survey also reviewed related research. For example, in some researches it was allowed only one delay for each airplane, and some other one generated a space-time network and the aircraft was routed in order of priority level. After the aircraft had been routed, the network was adjusted by removing the allocated route along with potentially conflicting edges, preventing the next aircraft to conflict with the first one. Some other research considered a two stage approach where the aircrafts with conflicts were solved in the first stage before the different clusters were unified and solved in combination in the second stage. Also, there was a two-phase approach, considering the runway

sequence in the first stage and the ground movement in the second stage.

When comparing different approaches the authors were able to notice some major differences, such as objectives and constraints, optimality vs. execution time, here they defend that the solution approach adopted should depend upon the airport, since exact solutions approaches becomes less practical as loads increase. Because of these differences it is difficult to determine the gap between the exact approach and heuristic approaches, states the survey.

Chapter 3

Mathematical Formulation

To formulate the problem, besides taking into account the assumptions, some modeling decisions are necessary. The first one is to decide whether to work with continuous time or discrete time. Similarly to the approach done in [12], a discrete time network representation is proposed. Time is treated as discrete by dividing the considered planning interval into periods of equal length, $k= 1, 2, ..., K$, where K is the total number of periods. The approach followed is to minimize taxiing time plus a penalty for delays for takeoffs.

The initial aircraft positions, holding points at intersections, holding positions of taxiway, and the runway are considered as nodes.

In each time period an aircraft can be taxiing or waiting. An aircraft can wait at the origin, or at each node, preventing conflicts from happening (for example having two aircrafts reaching the same node or having two aircrafts disrespecting the safety margin). An aircraft is not obligated to wait at any node if there is no conflict ahead.

It is also important to see that some taxiways intersect with each other. To respect constraints and to solve the problem, artificial nodes were introduced. At this nodes airplanes cannot wait. The other nodes represent holding points, holding bars or stop bars.

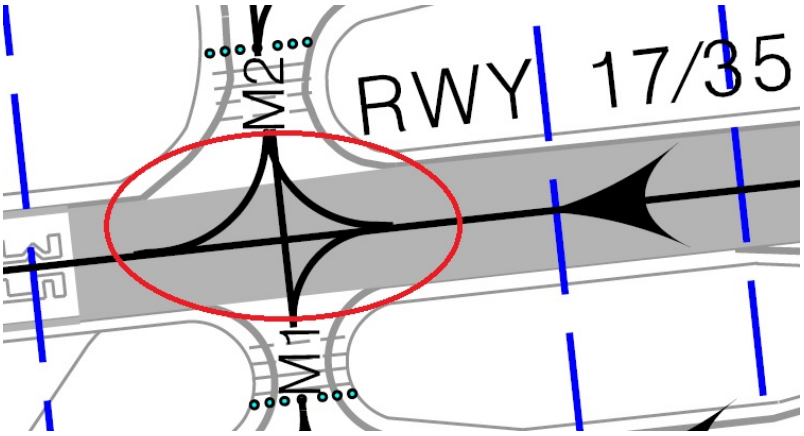


Figure 3.1: Example a taxiway intersection

Figure 3.2 shows how the situation was solved.

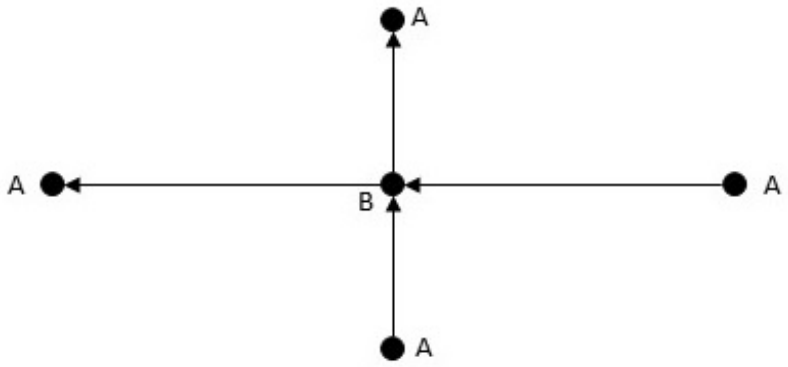


Figure 3.2: Artificial Node

The ground movements of an airport can be modeled as flows on an expanded directed graph, $G = (V, A)$, where V is the set of nodes which represent a copy of the original set of nodes (the initial aircraft positions, holding points at intersections, holding positions of taxiway, artificial nodes, and the runway) for each time period. Set A is the set of arcs, representing the existing connections between the nodes.

An example of a movement of an aircraft is illustrated in figure 3.3. The aircraft starts at its origin node "O" taxiing to node 1. Here the aircraft waits one period until proceeds to node 2. This happens because the aircraft is going to conflict with an other aircraft. At node 2 it hasn't to wait, proceeding immediately to the next node, 3. The aircraft reaches its destination, after 15 periods.

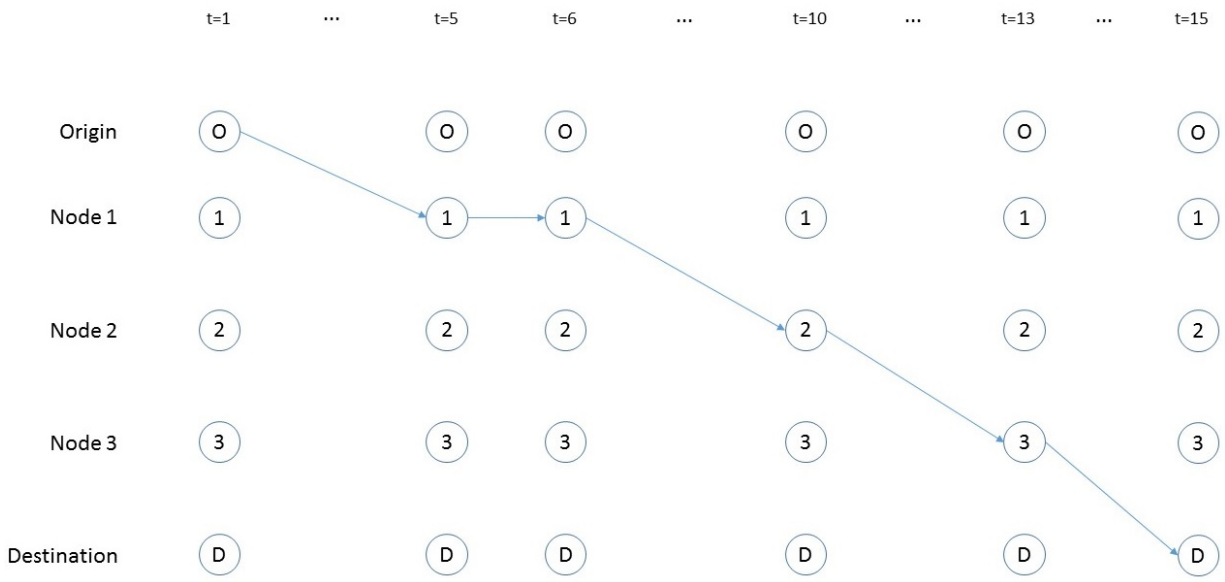


Figure 3.3: Example of the movement of an aircraft in a time expanded network.

3.1 Notation

The input data is the following:

- $G = (V, A)$ is the input direct graph. Indexes i, j are used to identify nodes belonging to V.
- $P = \{1, ..., |P|\}$ is the set representing the airplanes in the system. The set of planes is composed by two subsets, P^D representing departing aircrafts and P^A representing arriving airplanes. $P^D \cup P^A = P$. The indexes used to identify aircrafts in P are p and q;
- PT_p is the scheduled time for airplane $p \in P$ to use the runway, it also defines the runway sequence implicitly;
- $T = \{1, ..., |T|\}$ is the set representing the number of periods considered;
- $C = \{1, ..., \bar{c}\}$ are the different categories of airplanes;
- PC_p represents the airplane type/category;
- β is penalty per time unit of delay at runway;
- a_p is the predecessor plane $p \in P$.;
- sp_p is the separation time based on the category of airplane $p \in PC$;

- $o_p \in V$ is the initial position (origin) of aircraft $p \in P$;
- $d_p \in V$ is the final position (destination) of aircraft $p \in P$;
- Set $K = \{0, ..., k\}$ of periods. To identify periods t, l index are used;
- $k_{i,j,p}$ is the taxi time from i to j for aircraft p;
- ST_p is the time at which airplane $p \in P$ is available at origin.

Furthermore, some decision variables were defined:

- Binary variable $x_{ijpt}, p \in P, (i,j) \in A, t \in T$ is 1 if aircraft p passes through arc $(i,j) \in A$ at period t and zero otherwise;
- Binary variable $y_{ipt}, p \in P, i \in V, t \in T$ is 1 if aircraft p holds in position $i \in V$ at period t and zero otherwise.
- Binary variable $z_{itc}, i \in V, t \in T, c \in C$ is 1 if an aircraft of type c is in position i at period t and zero otherwise.

Only variables related to important nodes and arcs are defined.

3.2 Model

The problem can, now, be formulated. The objective function is given as follows.

$$\min_{z} \quad \underbrace{\sum_{p \in P^D} \beta_p \left(\left(\sum_{i \in V} \sum_{t \in T | t > k_{i,d_p,p}} t x_{i,d_p,p,t-k_{i,d_p,p}} \right) - PT_p \right)}_{\text{penalty for late arrival at the runway}} + \underbrace{\sum_{i \in V} \sum_{j \in V} \sum_{p \in P} \sum_{t \in T} x_{ijpt} k_{ijp}}_{\text{time spend taxiing}} + \underbrace{\sum_{j \in V} \sum_{p \in P} \sum_{t \in T} y_{jpt}}_{\text{time spend holding}} \tag{3.1}$$

In the objective function, if in the first term $x_{i,d_p,p,t-k_{i,d_p,p}} = 1$, the penalty for late arrival is $\beta(t - PT_p)$, where $t - PT_p$ is the difference between the time t the airplane reaches the runway and the period it should had arrived, PT_p. The second term takes into account taxi time for each airplane. When $x_{ijpt} = 1$ it sums the cost of going from i to j for airplane p, k_{ijp}. The waiting times of aircrafts are given by the third term when $y_{jpt} = 1$.

Subject to,

$$\sum_{i \in V} x_{o_p,ip,ST_p} + y_{o_p,p,ST_p} = 1 \qquad \forall p \in P \tag{3.2}$$

$$\sum_{i \in V} \sum_{i \in T} x_{i,d_p,p,t} = 1 \qquad \forall p \in P \tag{3.3}$$

$$\sum_{\substack{i \in V \\ (t-k_{ijp}) \geqslant ST_p}} x_{ijp,t-k_{ijp}} + y_{jp,t-1} = \sum_{i \in V} x_{jipt} + y_{jpt} \qquad \begin{matrix} \forall j \in V, p \in P, t \in T, \\ t > ST_p, j \neq d_p \end{matrix} \tag{3.4}$$

$$\sum_{i \in V} \sum_{\substack{t \in T \\ t > k_{i,d_p,p}}} tx_{i,d_p,p,t-k_{i,d_p,p}} \geqslant \sum_{i \in V} \sum_{\substack{t \in T \\ t > k_{i,d_{a_p},a_p}}} tx_{i,d_{a_p},a_p,t-k_{i,d_{a_p},a_p}} \qquad \begin{matrix} \forall p \in P^D, \\ a_p > 0, a_p \in P^D \end{matrix} \tag{3.5}$$

$$\sum_{i \in V} \sum_{\substack{t \in T \\ t > k_{i,d_p,p}}} tx_{d_p,i,p,t-k_{i,d_p,p}} > \sum_{i \in V} \sum_{t \in T} tx_{o_{a_p},i,a_p,t} \qquad \begin{matrix} \forall p \in P^D, \\ a_p > 0, a_p \in P^A \end{matrix} \tag{3.6}$$

$$\sum_{p \in P} \sum_{i \in V} \sum_{\substack{t_1 \in \{t,\ldots,t+sp_c-2\} \\ t_1 \leqslant nt \\ t_1 > k_{ijp}}} x_{ijp,t_1-k_{ijp}} \leqslant z_{jtc} + M(1 - z_{jtc}) \qquad \begin{matrix} \forall c \in C, t \in T, j \in V, \\ t > 1 \end{matrix} \tag{3.7}$$

$$\sum_{c \in C} z_{itc} \leqslant 1 \qquad \forall i \in V, t \in T \tag{3.8}$$

$$\sum_{p \in PC_c} \sum_{i \in V | t > k_{ijp}} x_{ijp,t-k_{ijp}} + \sum_{p \in PC_c} y_{jp,t-1} = z_{jtc} \qquad \forall j \in V, t \in T, c \in C \tag{3.9}$$

$$x_{ijpt} \in \{0,1\} \qquad \begin{matrix} \forall i \in V, j \in V, \\ \forall p \in P, t \in T \end{matrix} \tag{3.10}$$

$$y_{jpt} \in \{0,1\} \qquad \begin{matrix} \forall j \in V, p \in P, \\ \forall t \in T \end{matrix} \tag{3.11}$$

$$z_{jtc} \in \{0,1\} \qquad \begin{matrix} \forall j \in V, t \in T, \\ \forall c \in C \end{matrix} \tag{3.12}$$

Constraints are divided in **flow conservation constraints** (3.2 - 3.4), **runway sequence restrictions** (3.5 - 3.6), **separation time between airplanes** (3.7), **conflict restrictions** (3.8 - 3.9) and **binary constraints** (3.10 - 3.12).

Equations (3.2) determine that each aircraft waits at origin or starts taxiing. Equations (3.3) assure that the airplane p always reaches its destination. Constraints (3.4) guarantees flow conservation at each expanded node. To respect this constraint either the airplane is at j in period $t-1$ and waits at that position one period, or it is at node i at period $t - k_{i,j,p}$ ($k_{i,j,p}$ is the time that aircraft p spend taxiing from i to j) and reaches node j at time t, respecting the equation.

Referring to the runway sequence, inequalities (3.5) assure that any two departing aircrafts using the runway respect the given sequence. If there are two departing aircrafts using the runway $p \in P^D$, $a_p \in P^D$ and $x_{i,d_{a_p},a_p,t-k_{i,d_{a_p},a_p}} = 1$ the restriction guarantees that airplane p only reaches the runway at time t later than the preceding airplane. Similarly (3.6) ensures that if there is an departing aircraft ($x_{o_p,i,p,t} = 1$, $p \in P^D$) leaving before an arriving one ($x_{i,d_{a_p},a_p,t-k_{i,d_{a_p},a_p}} = 1$, $a_p \in P^A$) the departing time of the first is lower then

the arriving time of the later one.

The separation constraints (3.7) guarantee that two aircrafts maintain a safety margin which depends on the type airplane that goes ahead. Inequality (3.7) assures that if $x_{i,j,p,t_1-k_{i,j,p}} = 1$ then the aircraft p will reach j in one of the following time periods, t, $t+1$, ..., or $t + sp - 2$, where sp is the separation time between two airplanes. It is important to refer that the sequence of arriving airplanes ir guaranteed, known and it doesn't change.

Inequalities (3.8) guaranty that either the node i is occupied or not. Equations (3.9) together with 3.8 assure that only one airplane at a time is in node j at period t.

Restrictions (3.10), (3.11) and (3.12) guarantee that variables are binary.

Chapter 4

Heuristic Methods

The airport ground movement problem is a NP-Hard problem [7]. The particularities of the problem make it very difficult to find an optimal solution through exact methods within an acceptable runtime.

Hence, heuristics methods were implemented, in JAVA language, to obtain good solutions in a short runtime. For example, an instance with 36 airplanes found a solution within 0.82 seconds.

In order to evaluate the quality of the solutions provided by the heuristic algorithms, they are compared against the optimal solutions obtained by the exact approach (Branch and Cut).

Turning the ground movement problem in a scheduling problem is simple. Taxiways represent resources that are used to fulfill a task. Each airplane needs a set of taxiways to reach its destination. So the set of taxiways that each airplane uses are seen as tasks. Airplanes can use different resources to reach destination, so what combination of resources and times is the best to minimize costs?

Summarizing, scheduling consists in mapping the tasks of airplanes to resources and process times, given some restrictions.

It is possible to divide the problem into two parts: first, for each plane, it is necessary to calculate all possible routes between an origin and a destination. This first part is exact. The second part begins after having all possible routes for every plane in the system, and the problem is to choose a route for each plane and schedule the airplanes through the taxiways.

In order to obtain an initial solution there is a preprocessing of data that can be seen as the first part of the problem. In preprocessing the k-shortest path between every two nodes are collected. This solution is exact and obtained through the implementation of Depth-First Search Algorithm (DFS). Still in the preprocessing of data the sequence of runway user is determined, having into account the time each plane has to use the runway.

After this preprocessing of data there were two constructive algorithms implemented. The

difference between these two constructive algorithms has do to with the way to choose the path each airplane will use. One of the algorithms chooses the shortest path for each airplane whereas the other chooses randomly between the k-shortest paths.

In order to improve the initial solutions obtained with the constructive algorithms, local search, iterated Local Search (ILS) and Tabu Search (TS) were implemented.

In what concerns ILS two different processes of local search were implemented, Steepest Ascent and First Ascent, which will be explain ahead.

In what respects TS, three different types of search were implemented that can be used individually or at the same time, as it will be explained further ahead.

In the end the goal is to compare the results obtained with constructive heuristics, the results obtained with improvements and meta-heuristics.

4.1 Preprocessing Data

The first part, preprocessing data, not only takes care of determining all possible routes between two different nodes, but also includes the determination of the runway sequence. By receiving the time that each airplane is supposed to use the runway, it is possible to sort airplanes from the earliest to the latest and make a sequence. Each airplane has an origin node, a destination node, a plane number and a time at runway. Also, information about the airport network is given. After receiving this information, data is pre-processed.

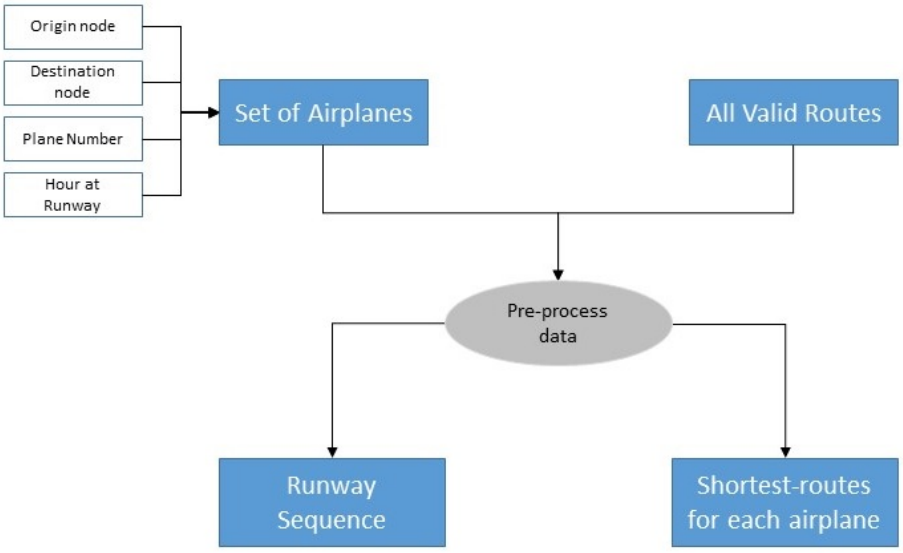


Figure 4.1: Pre-process Data

4.1.1 Depth-First Search Algorithm

The Depth-First Search (DFS) is an algorithm to cross or search through a graph. The algorithm is based on given an initial vertex (research center) v and, from that vertex on, search a new research center w, which is adjacent to v and was not yet used as a research center. In case there is no such vertex, the process returns to the previous vertex (in case there is one), repeating the process recursively. This procedure ends when there are no more vertices's to be used as research centers [2].

Here DFS algorithm is going to be used to search all possible routes between two vertices and using a directed graph.

It is important to notice that for each airplane the destination is a fixed node, but the origin is dynamic, because if an airplane enters the system when an aircraft is already taxiing, the origin of this airplane is going to be the node where it was standing. That is why it is relevant to know all routes between two different nodes.

```
1 Input: A graph G and two vertex (v(o), v(d)), origin and destination of G
2   procedure DFS(G,v(o),v(d))
3     label v(o) as visited
4     for all edges from v(o) to w in G.adjacentEdges(v(o)) do
5       if vertex w is not labeled as visited then
6         recursively call DFS(G, v(o), w)
7         if w equals v(d) add w to List of path
8 Output: All paths from origin to destination.
```

Figure 4.2: DFS - Pseudo-Code

All possible routes, between two vertices's, are storage data.

After generating all possible routes between two nodes, the paths are sorted from the shortest to the longest.

At this point a decision is made, only some of the routes are considered. First the 10-shortest routes are considered. Additionally, given the estimated taxi time for the shortest route, only those routes which taxi time is less than 50% of the shortest time are considered. This should be a parameter, chosen by the user that depends on the number of airplanes and the airport layout.

4.2 Constructive Heuristics

This method starts from an empty solution and repeatedly extends the current solution until a complete one is constructed. The solution is only completed if all aircrafts tasks are labeled. The constructive heuristic uses the hour that each plane uses the runway (runway user sequence) to decide which one uses taxiways first. When there is more then

one runway, all hours are compared and an unique list is built, from the earliest plane on the runway to the latest one.

Figure 4.3 illustrates how to reach the initial solution. And for each origin-destination pair k-shortest routes are assigned to the airplane.

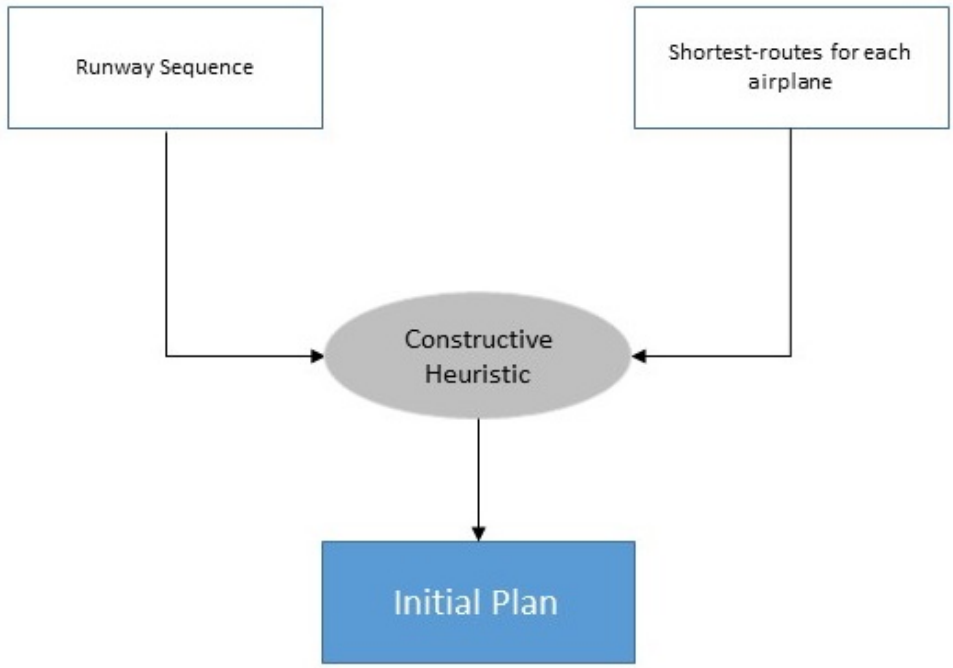


Figure 4.3: Constructive Scheme

As it was referred before the initial solution is obtained through a constructive algorithm. Two constructive heuristics were implemented, one uses the shortest path between an origin and a destination. By shortest path is meant the one with less taxi time. The other constructive heuristic chooses the path randomly between the k-shortest paths.

These will be the constructive heuristic in use.

4.2.1 Constructive Heuristic Using Shortest path

The path chosen for each airplane is the shortest path, has the name implies. The initial solution is built from there.

```
1 Input: All aircrafts and runway sequence
2   List of aircrafts sorted by the hour at respective runway, from the
        earliest to latest
3   from i <-- 1 to last user
4     choose the shortest path of plane i
5     schedule all tasks of the route of plane i
6 Output: Schedule for all aircrafts in taxiways
```

Figure 4.4: Constructive shortest-path Pseudo-Code

4.2.2 Constructive Heuristic Using Random path

By random search is meant that instead of choosing the shortest path for the route of every plane, that choice is random, based on the number of available routes. A random generator is implemented, in order to choose the route for aircrafts, and then the same constructive heuristic is applied.

```
1 Input: All aircrafts and runway sequence
2   List of aircrafts sorted by the hour at respective runway, from the
        earliest to latest
3   from i <-- 1 to last user
4     choose randomly a path of plane i
5     schedule all tasks of the route of plane i
6 Output: Schedule for all aircrafts in taxiways
```

Figure 4.5: Constructive shortest-path Pseudo-Code

4.2.3 Validation - Scheduling

Some considerations are necessary in what concerns scheduling airplanes in taxiways. For example, looking at the problem as a scheduling one, tasks are the taxiways that airplanes have to travel from the origin to its destination. In this sense, each airplane has a list of tasks to fulfill. To accomplish a task a resource will be needed, so taxiways are also, resources. Therefore, two airplanes cannot use the same resource at the same time and besides that, they cannot alter the order that each airplane fulfills its list of tasks. So when an airplane begins its path not only has to respect the order of tasks but also has to respect the establish order of resource utilization.

This way the goal is to delay or anticipate the start engine of a departing airplane, so the time spend taxiing is minimum and to prevent congestion in taxiways or waiting for take-off on runway before the scheduled time. For airplanes that arrive at airport the objective is to take them to the gate as soon as possible. The longer they stay at taxiways waiting

is more likely to provoke bottlenecks.

The initial solution is not necessarily the optimal one. Despite all aircrafts are using the shortest path between origin and destination, if there is a common taxiway to several routes of airplanes is possible to create a bottleneck near that area. Situations like this one can make taxi time much bigger than the estimated, and using other route may cause a reduction in taxi time. Finding all neighbors of the initial solution, and their own neighbors makes possible to choose the best plan.

4.3 Local Search Heuristics

4.3.1 Iterated Local Search

The iterated Local Search method uses the constructive heuristic with random paths, and then uses a local search to find better solutions. In other words, the random search is performed and then a local search is made based on the solution found.

Two different approaches of the ILS method were used, First Ascent and Deepest Ascent. The main difference between both methods is that in First Ascent method when a better solution is found the search stops while in Deepest Ascent, despite finding a better solution the search goes on until covers this neighborhood.

```
1 Input: Aircrafts, Runways Sequence
2     for all aircrafts
3       choose a random route
4     contructive
5     validation - schedule
6     local search
7 Output: Plan
```

Figure 4.6: Iterated Local Search Pseudo-Code

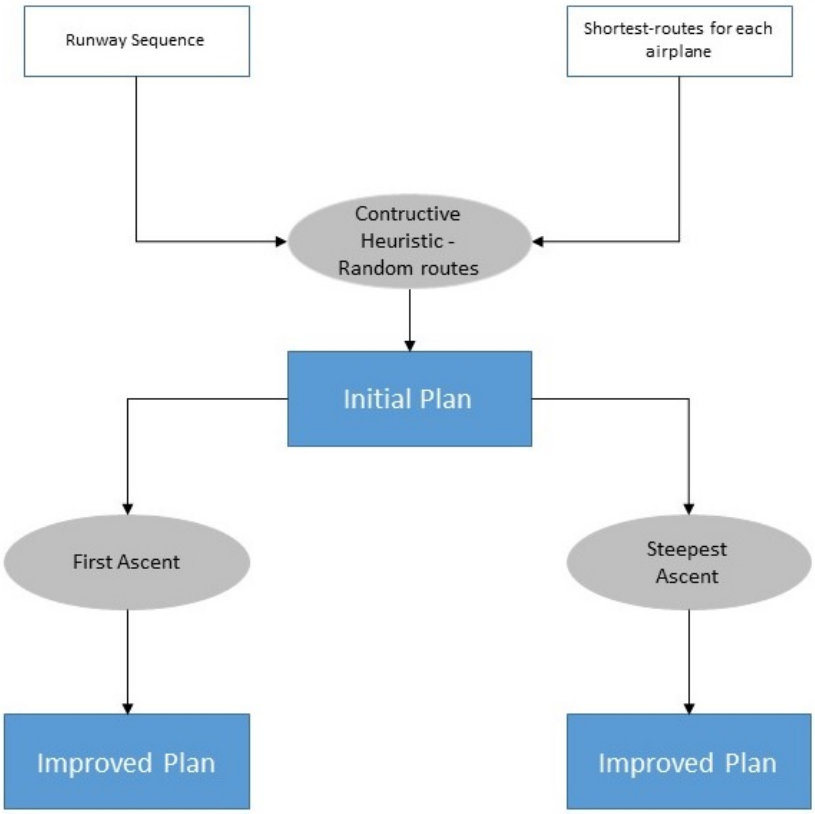


Figure 4.7: Iterated Local Search Scheme

4.3.1.1 Local Search - First Ascent Method

First Ascent method is a local search based on the first best neighbor found. The Random Search heuristic gives a solution and then First Ascent method searches for the first better neighbor. This search is made by changing the route, chosen randomly by the Random Search, for one plane at a time. For example, given 3 planes the random search determined that plane one has route 2, plane two has route 3, and plane 3 has route 1. First Ascent, searches, in plane one, all routes, one by one. When a route that improves the current solution is found the search stops. And so on for the other planes.

```
Input: Plan
  for i<--1 to last aircraft
    for j<--1 to last route of aircraft i
      change to route j
      if new plan better
        plan = new plan
        STOP
Output: Plan
```

Figure 4.8: First Ascent Pseudo-Code

4.3.1.2 Local Search - Steepest Ascent Method

Steepest Ascent method is a local search based on search through every neighbor, given some criteria, and chooses the best one. Using Random Search to find the initial solution, the steepest ascent looks for the best neighbor. This search is made by changing the route chosen for one plane at a time. For example, given 3 planes the Random Search determined that plane one has route 2, plane two has route 3, and plane 3 has route 1. Steepest Ascent, searches, in plane one, all routes, one by one, choosing the route that improves solution. If there is no route that improves solution, the solution found by random search maintains. And so on for the other planes. The major difference between Steepest Ascent and First Ascent is that the First Ascent stops in the first better neighbor, while Steepest Ascent keeps searching despite having found a better solution already.

```
Input: Plan
  for i<--1 to last aircraft
    for j<--1 to last route of aircraft i
      change to route j
      if new plan better
        plan = new plan

Output: Plan
```

Figure 4.9: First Ascent Pseudo-Code

4.4 Tabu Search

When standing at a local minimum, it is not always possible to reach other solutions by conducting a simple local search. For that reason a meta-heuristic may be used. Although meta-heuristics do not guarantee that the globally optimal solution can be found, it allows to search over a substantial set of feasible solutions with a computational effort much lower than exact algorithms, in general.

Tabu Search is a meta-heuristic that guides a local heuristic search procedure to explore the solution space beyond local optimality [6]. The local procedure is a search that uses an operation, known as *"move"* to define the neighborhood of any solution. Tabu Search imposes or introduces some restrictions in order to point some direction in the search process to reach difficult regions of the problem. These directions work in different ways, such as to exclude alternative paths and as consequence alternatives solutions, or to impose a different path based on evaluation and probability of selections. Another particular aspect of Tabu Search is its adaptive memory, providing a flexible search. Many restrictions are enforced by making reference to memory, which means the same solution cannot be taken into account twice.

Collecting information during the search process allows Tabu Search to be more effective.

Figure 4.10 illustrates how Tabu Search is working. Receiving a plan a *"move"* is made to choose a neighbor. When a neighbor is chosen it is important to verify if the new plan is still valid. As explained earlier changes can cause invalid plans. If it has no invalid moves, the new plan is compared with the input plan. If it is better a new solution is found. If not, it is necessary to skip this local minimum using a local search. Mildest Descent was the chosen one and it will be explained further in the report.

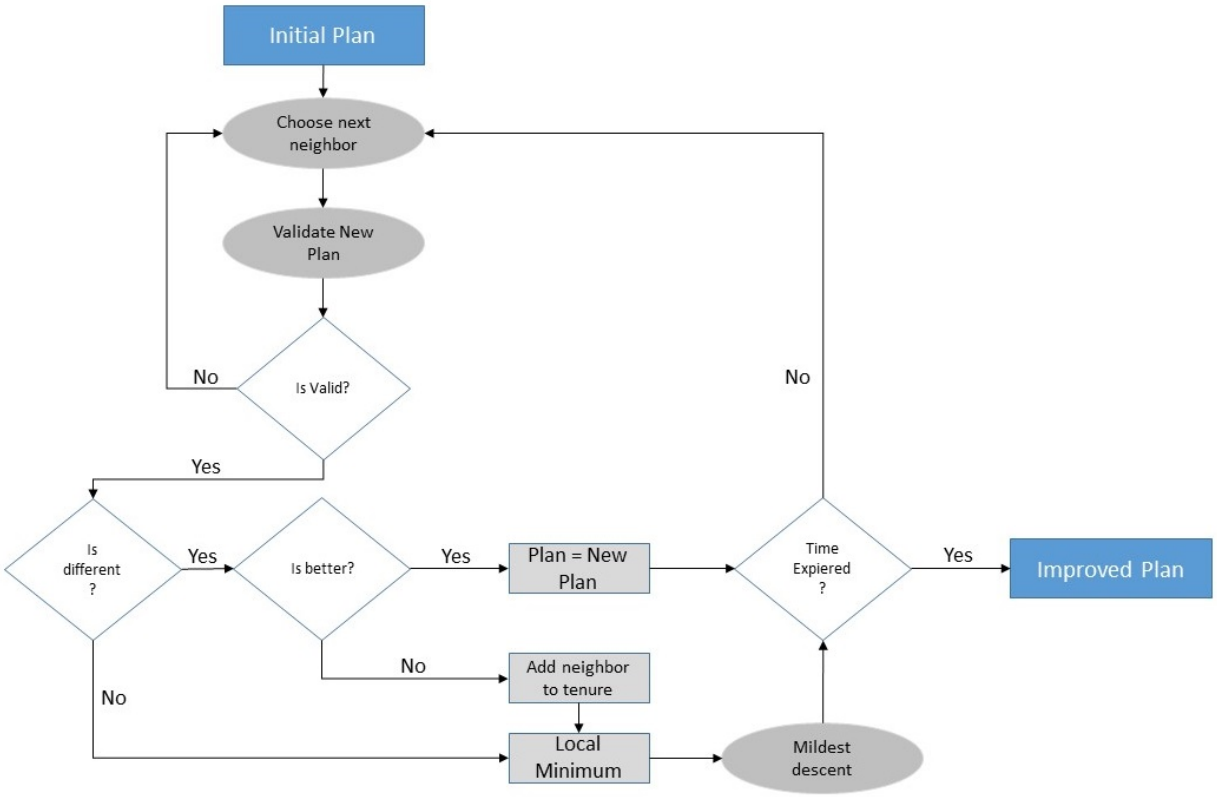


Figure 4.10: Tabu Search Scheme

It is important that the search is intelligent to be more effective than random search or Iterative Local Search, so intelligent inhibitions or restrictions are necessary.

In this problem there are three types of *moves*: changing routes, changing the sequence aircrafts use taxiways and when there is more then one runway and a unique sequence is made based on the time of use of runway, change the sequence but only for aircrafts that use different runways at same time.
Combining all three *"move"* it is possible to go from a solution to another.

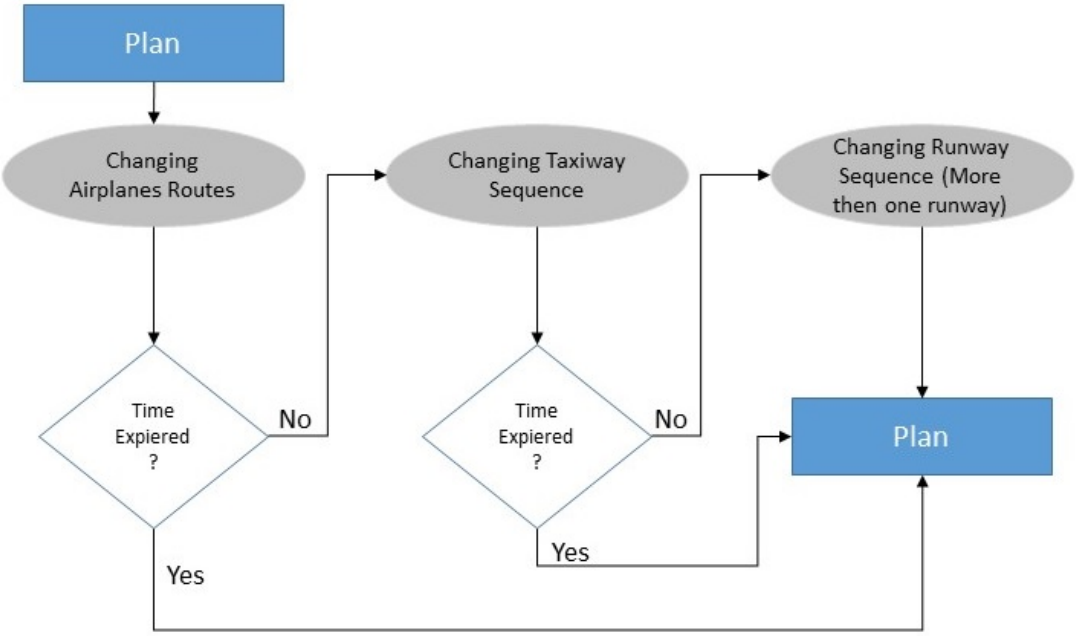


Figure 4.11: Tabu Search Moves Scheme

4.4.1 Changing Routes

When the solution is analyzed it's important to understand if the estimated taxi time for each aircraft is significantly lower then the effective taxi time. So for each aircraft is compared the time spend taxiing with the estimated taxi time of other routes. The route is only changed if it improves the solution.

```
Input: Plan
  for all airplanes
    for all airplane routes
    if taxi time > taxi time estimated next route
      route = next route
    Contructive considering these routes
    evaluate solution
    if is better
      plan = new plan
Ouput: Plan
```

Figure 4.12: TS - Changing Routes Pseudo-Code

4.4.2 Changing Sequence in Taxiways

The initial plan uses the runways users sequence to establish the sequence to be used at taxiways. Without knowing which sequence is the optimal one it is possible to try every sequences, granting that the hour at runway is respected and the sequence of aircrafts at the runway is also respected. These are the types of moves that can cause cycles.

```
Input: Plan
  for all taxiways
    List taxiwayUser
    for i <-- 1 to last aircraft using taxiway
      temp = taxiwayUser(i+1)
      taxiwayUser(i+1) = taxiwayUser(i)
      taxiwayUser(i) = temp
      validate new plan
      if new plan is valid
        if new plan is better
          plan = new plan
Output: Plan
```

Figure 4.13: TS - Changing Sequence in Taxiways Pseudo-Code

Figure 4.14 shows an example of changing sequence at taxiways. At each "move" two tasks at same taxiway are changed, until all taxiways are covered.

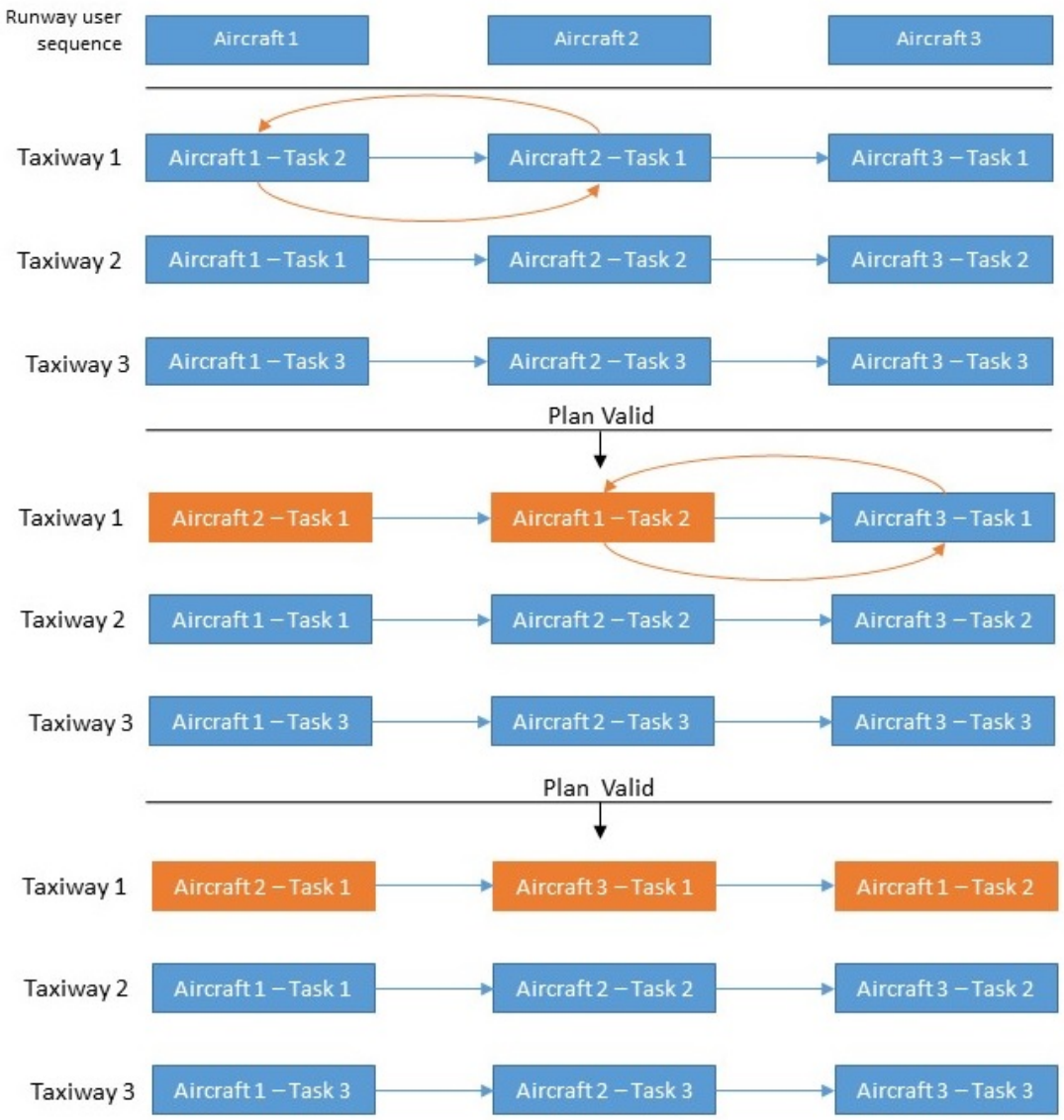


Figure 4.14: Tabu Search - Changing Sequence at Taxiways

4.4.3 Change Taxiway Sequence - Aircrafts using different runways at same hour.

Many airports operate with more then one runway. The only rule they have to respect is that when there is more then one runway operating at same time, the runways must be parallel. The runways that are operating are beyond the scope of the project, since that is input data. Since Constructive Heuristic uses the runway users sequence to establish the sequence of taxiways users, when there is more then one runway and only one sequence is made, is possible to change the sequence, by changing airplanes with the same our at runway.

```
1 Input: Plan and Sequence at each Runway
2   Form an unique sequence sorted by hour at runways
3   for all aircrafts in sequence
4     if hour(aircraft(i)) == hour(aircraft(i+1))
5       change sequence
6     Contruct new plan
7   if new plan is better
8   plan = new plan
9 Output: Plan
```

Figure 4.15: TS - Change Taxiway Sequence: Aircrafts using different runways at same hour Pseudo-Code

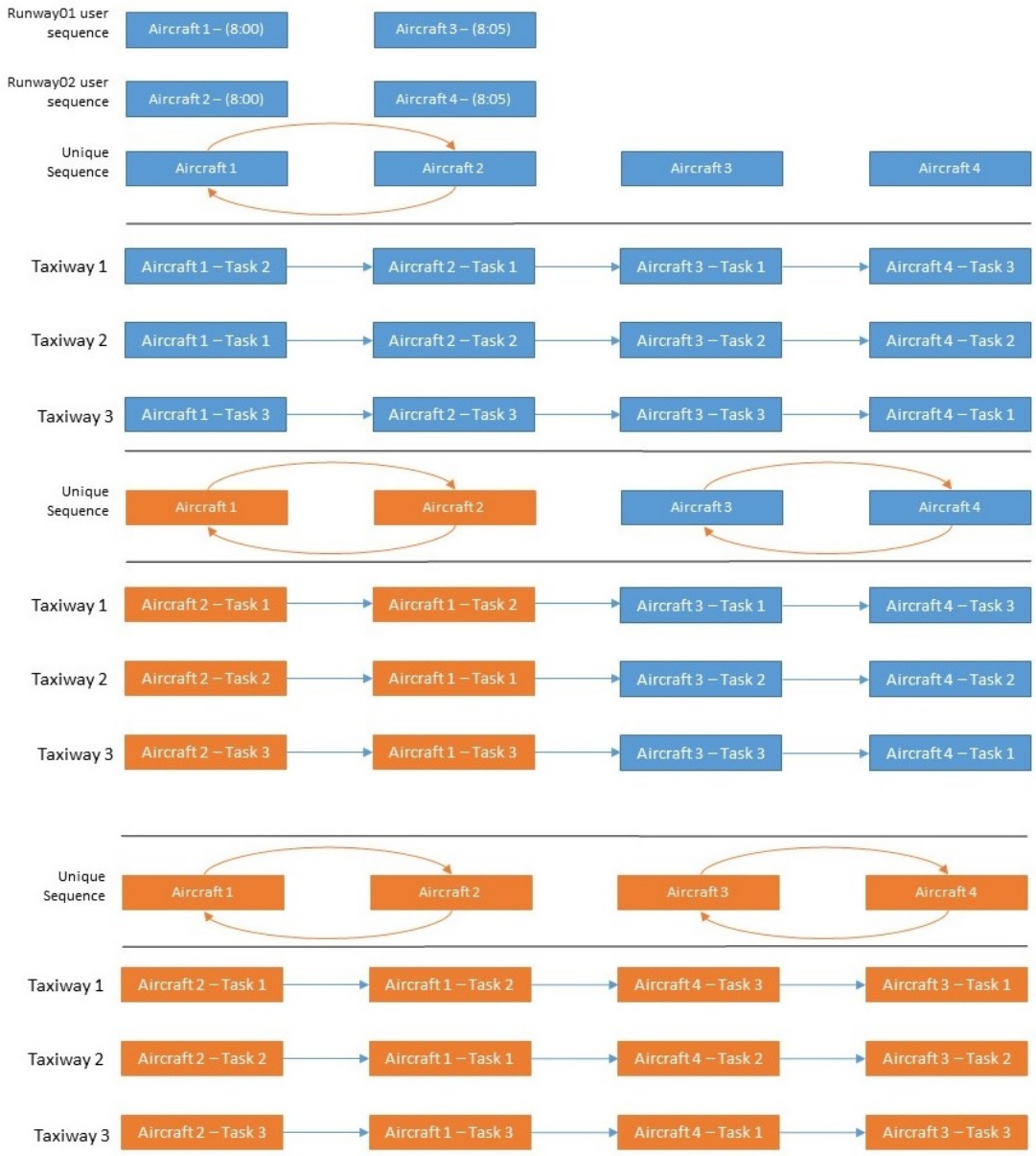


Figure 4.16: Tabu Search - Changing Sequence

Figure 4.16 shows how these "moves" are made. First aircraft 1 changes with aircraft 2 in every taxiway, then the next "move" is to change between the other two aircrafts with the same hour at runway.

4.4.4 Mildest Descent

Every once in a while it is possible that the solution has reached a local minimum, and searching in similar neighbor will not help to improve the solution. To get out of this local minimum instead of continuing searching with first ascent end steepest ascend (methods

explained above), it is necessary to apply Mildest Descent. This method consists in, given all possible neighbors choose the less worse, in other words chose the second best plan. To the solution found it is applied the TS algorithm which will generate different neighbors, as the input plan is going to be different.
It is important to say that the solution will not improve by applying this method alone.

This method only allows, when standing at a local minimum, to unlock new possible neighbors.

```
1  Input: Plan
2    for i<--1 to last aircraft
3      for j<--1 to last route of aircraft i
4        change to route j
5        if new plan better
6          plan = new plan
7          add plan to list
8    sort list
9    Choose second on list
10 Output: Plan
```

Figure 4.17: Mildest Descent Pseudo-Code

4.4.5 Validation - Syntax

Validation doesn't make sense for the initial solution, but once local changes are made, they can create cycles, turning impossible for airplanes complete all tasks.

An example of plan with no cycles is represented in figure 4.18:

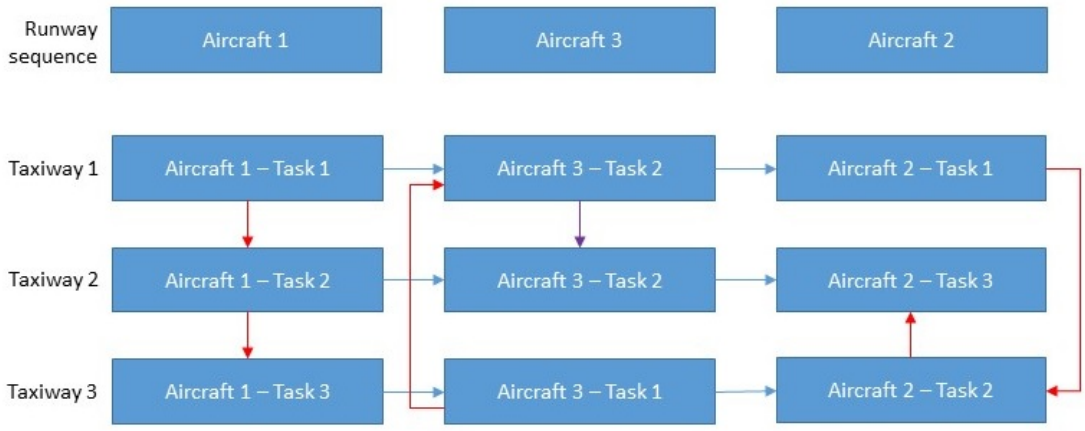


Figure 4.18: Constructive with no cycle

In figure 4.19 when *Aircraft 2 - Task 3* is completed is not possible to complete any other:

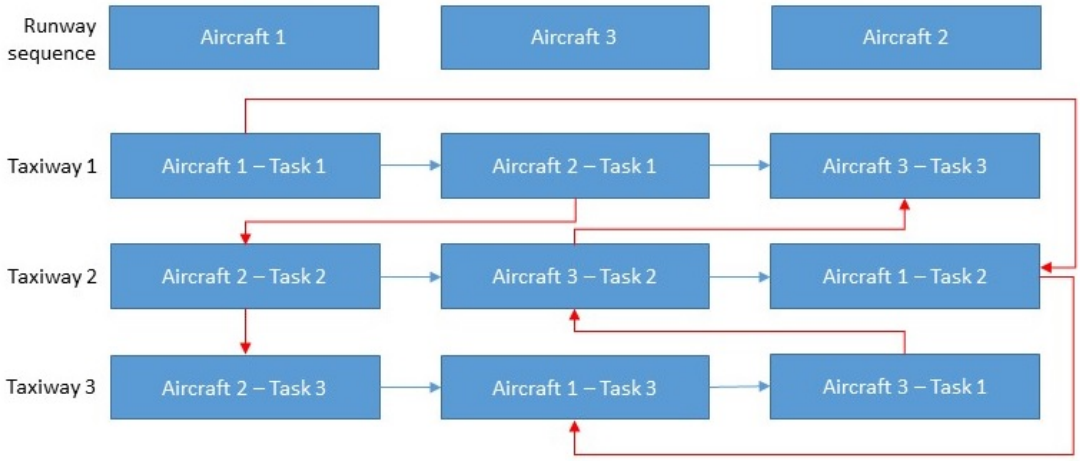


Figure 4.19: Constructive with cycle

In order to avoid situations like 4.19 is important to validate the plan for possible cycle existence.

```
1 Input: Plan
2   List of all tasks from plan
3   do while list is not empty
4     from i <— 1 to last task of list
5       if i has no predecessor
6       remove from list
7 Output: Valid or Not valid
```

Figure 4.20: Validation Syntax Pseudo-Code

So every time a movement is made it is necessary to search for cycles in the solution. Because of this, the syntax validation is applied along with the improvement heuristics and meta-heuristics.

Chapter 5

Results

All results were achieved using a PC with an Intel(R) core (TM) 2 duo CPU T8300 @ 2.4 GHZ.

During this research the results were obtained by using constructive heuristics, iterated Local Search, Tabu Search and an exact method, Branch and Cut.

The first results that were compared were between the two constructive heuristics. Then the results obtained with ILS, TS and Branch and Cut were analyzed.

The best heuristic procedure will be compared with the exact method, Branch and Cut.

The comparison between all methods was done with real data from Lisbon Airport.

5.1 Constructive Heuristics

First of all it is important to understand the differences between the two constructive methods. As stated before one uses the shortest path for each airplane and the other one uses a random path chosen between the k-shortest paths of each airplane.

From the observation of results it was possible to conclude that using the shortest paths was always better, despite the difference is not that big.

Results	
Hour	**Difference**
8:00AM	75s
9:00AM	70s
10:00AM	95s
11:00AM	105s
12:00PM	60s
13:00PM	115s
15:00PM	80s
16:00PM	75s
17:00PM	80s
18:00PM	95s
19:00PM	60s

Table 5.1: Comparative Table - Constructive Heuristic

5.2 Comparing iterated Local Search with Tabu Search

The first thing done, when analyzing both results with the same running time is to understand which method presents the best solution. The running time chosen was one second. It was possible to verify that ILS always had worst solutions, so the next step was to verify how long would take ILS to reach the same solution as TS.

To find the same solution ILS needed an average of 6 hundredths of a second more.

Results		
Hour	**ILS**	**TS**
8:00AM	1.20s	0.82s
9:00AM	1.06s	0.34s
10:00AM	1.02s	0.17s
11:00AM	1.03s	0.29s
12:00PM	1.02s	0.28s
13:00PM	1.03s	0.31s
15:00PM	1.04s	0.38s
16:00PM	1.09s	0.33s
17:00PM	1.05s	0.16s
18:00PM	1.03s	0.27s
19:00PM	1.03s	0.27s

Table 5.2: Comparative Table - ILS versus Tabu Search

5.3 Comparing Tabu Search with Branch and Cut

After concluding that TS was the best method, this meta-heuristic will be compared with Branch and Cut (BC), the exact method, using XPRESS. Not only results will be compared but also the running time of each method.

The first limitation in what concerns runtime and memory, was that XPRESS optimizer could only process short time intervals. So the problem was divided in time windows, with a restricted number of airplanes and time periods, therefore the planning horizon was only fifteen minutes. Analyzing all different time windows the one that had more planes had 11 planes.

So, the solutions found by both methods need to be compared in order to understand if the meta-heuristic result is close, or not, to the optimal solution. Accordingly, the time windows were, also, applied to the TS algorithm.

It was possible to notice that TS always found the same solution as BC except in one case.

With such short time windows the number of planes considered in each iteration is very low. In fact the simulation with the largest number of airplanes had only 11 planes. With such low number of airplanes using taxiways there aren't any conflicts to manage, therefore it is not possible to make reliable conclusions.

The pick hours where chosen. The pick hour hour in the morning from 8am to 9am and in the afternoon the pick hour is from 19pm to 20pm.

The next table shows some comparisons:

Results		
Hour	**BC**	**TS**
8:00AM - 8:14AM	481s	0.13s
8:15AM - 8:29AM	248s	0.09s
8:30AM - 8:44AM	770s	0.09s
8:45AM - 8:59AM	662s	0.11s
19:00PM - 19:14PM	445s	0.09s
19:15PM - 19:29PM	53s	0.03s
19:30PM - 19:44PM	169s	0.07s
19:45PM - 19:59PM	262s	0.07s

Table 5.3: Comparative Table - Branch and Cut versus Tabu Search

As table 5.3 shows the heuristic methods, in particular Tabu search, were much faster then the exact one. One second was the time used by Tabu Search to find the best solution between all the admissible ones and it found the optimal solution.

At the time window 8:30AM to 8:44AM the solution found in both methods is the optimal

solution, but one airplane uses an alternative path. This happens, because a few paths have the same taxi time and this was one of those cases. For the remaining time windows the results were exactly the same.

5.4 Comparing the Solution from TS Heuristic with the Real

With so short time windows it was not possible to have real conclusions about the time saved for each airplane. So to accomplish some reasonable conclusion Tabu Search was applied to an entire day, with 1h time windows. With these time windows it was possible to simulate conflicts and understand how to solve them. At appendix C it's an example of how long an airplane took with TS versus Real.

In one day at Lisbon Airport (LPPT), airplanes spend taxiing 3251 minutes. A departing aircraft spends a minimum of 4 minutes taxiing and a maximum of 27 minutes, while arriving aircrafts spend a minimum of 2 minutes and a maximum of 10 minutes taxiing as it is shown in table (5.4).

Comparing with the Tabu Search solution, a total of 3072 minutes spend taxiing, a departing aircraft used taxiways with a minimum of 1 minute and a maximum of 10 minutes. In what respects to arriving aircrafts, they spend a minimum of 3 minutes and a maximum of 24 minutes taxiing.

Results						
Movement	**Min LPPT**	**Min TS**	**Max LPPT**	**Max TS**	**Average LPPT**	**Average TS**
Arriving	2	3	10	24	5	12
Departing	4	1	27	10	11	4

Table 5.4: Comparative Table

Tabu Search allowed to save 5.5% in taxi time.

The next table, table (5.5) presents a summary of results.

Airplanes	410
Taxi Time Spend LPPT	3251 minutes
Taxi Time Estimated TS heuristic	3072 minutes
Savings	5.5%

Table 5.5: Aggregate Table

Analyzing Lisbon Airport data allowed to understand that departing aircrafts were taking longer paths to reach destination (runway) than arriving aircrafts (gate/apron). This

happens because from taxi planning point of view is easier to make departing aircrafts form a queue (minimizing the number of conflicts) than make arriving aircrafts stop in strategic points so they don't create bottlenecks near the runway exits or on their ways to aprons.

Tabu search not only granted a saving of 5.5% of taxi time, but also moved waiting times to arriving airplanes from departing ones.

A consequence of these decisions is that airplanes spend less fuel. A departing aircraft besides the weight of passengers and luggage has also the weight of fuel, and an arriving aircraft has the same weight in what concerns passengers and luggage but less in fuel. So departing airplanes are heavier than the arriving ones. These decision allows airplanes to spend less fuel. Noticing that an heavier airplane spends more fuel, if it also spends more time taxiing then they will spend even more fuel. Transferring the waiting times to lighter airplanes allows to save in fuel consumption as well.

Chapter 6

Conclusion

With the increasing of the air traffic and the problems that come with it, it became important and crucial to optimize the airport ground movement. So the scope of this project was the ground movement problem at the airports.

After explaining the problem, its particularities and some concepts, a mathematical formulation is presented in order to solve the problem with an exact method. However, this is a NP-hard problem, so exact methods do not find the optimal solution in an acceptable time. Therefore, after analyzing what was the best method to find a better solution in short runtime, it was used a constructive heuristic, using the shortest path, and a meta-heuristic to reach the best solution possible.

The exact method was used to understand how good were the solutions found by heuristics and meta-heuristics, given a short running time.

After applying real data to the exact method and a meta-heuristic method it was possible to reach some conclusions.

The exact methods used in XPRESS optimizer had some limitations. Those limitations lead to the division of the problem in several time windows and each time window had only a few airplanes. So comparing the results with real data didn't make sense, because there were no conflicts and airplanes reach their destination without any problems, making taxi time much lower.

Instead, the exact model was compared with a meta-heuristic, Tabu Search. It was possible to conclude that the meta-heuristic implemented to solve the problem had satisfying results. For so short time windows, Tabu Search always found the optimal solution.

Hence, it was good enough to experiment with real data from Lisbon Airport and compare results with the real time aircrafts spent taxiing and the estimated taxi time by Tabu Search.

Tabu Search allowed to save about 5.5% of time spending taxiing. This save was possible because of the type of the decision made by Tabu Search. Tabu search moved waiting times to arriving airplanes from departing ones. These decision, besides being for the airport,

are good news to airline companies because heavier airplanes spends more fuel, implying that departing airplanes spend more fuel, so this are the ones that should have less taxi time. Moving waiting times to lighter airplanes saved in fuel consumption.

As for future work the implementation should be ready to receive online information about airplanes and their movements in the airport along with where they are every second. This way, the origin must be a dynamic input and Tabu Search must be improved to take less then one second to reach a solution.

It is important to minimize other objectives. Considering the same problem but instead minimizing the taxiing time per airplane, minimize the number of conflicts at taxiways or the fuel consumption, for example.

Bibliography

[1] Atkin, Jason A. D., Burke, Edmund K., Ravizza, Stefan, *"The Airport Ground Movement Problem: Paste and Current Research and Future Directions"*, 4th International Conference on Research in Air Transportation, 2010;

[2] Cardoso, D., Szymanski, J., Rostami, M., *"Matematica Discreta"*, Escolar Editora, 2009;

[3] Clare, G. and Richards, A.,*"Receding Horizon Iterative Optimization of Taxiway Routing and Runway Scheduling"*, American Institute of Aeronautics and Astronautics, 2009;

[4] Cambridge Professional English, *"Flightpath. Aviation English for Pilots and ATCOs - Glossary of Aviation Terms"*, Cambridge University Press 2011;

[5] European Organization for the Safety of Air Navigation (EUROCONTROL, *"EUROCONTROL Seven-Year Forecast"*, 2014;

[6] Glover, Fred., Laguna, Manuel, *"Tabu Search"*, Kluwer Academic Publishers, 1997;

[7] Godbole, P. J., Ranade, A. G., Pant, R.S. *"Routing and Scheduling Algorithm for Aircraft Ground Movement Optimization"*, American Institute of Aeronautics and Astronautics;

[8] International Civil Aviation Organization (ICAO), *"Advanced Surface Movement Guidance and Control Systems (A-SMGCS) Manual"*, 1^{st} edition, 2004;

[9] Keith, G., Richards, A., *"Optimization of Taxi Routing and Runway Scheduling"*, in Proceeding of the AIAA Guidance, Navigation and control conference, Honolulu, USA, 2008;

[10] Marin, A., Codina, E., *Network Design - Taxi Planning*, 2008;

[11] Pesic, B., Durand, N., Alliot, J., *"GECCO 2001 conference: Real-World Applications Aircraft Ground Traffic Optimization using a Genetic Algorithm"*, 2001;

[12] Roling, P. C., Visser, H. G., *"Optimal airport surface traffic planning using mixed-integer linear programming"*, International Journal of Aerospace Engineering, vol. 2008, no.1, pp.1-11, 2008;

[13] Smeltink, J. W., Soomer, M. J., Wall, P. R. de, Mei, R. D. van der, *"An optimization model for airport taxi scheduling"*,in Proceedings of the INFORMS Annual Meeting, Denver, USA, 2004.

Appendix A

According to the A-SMGCS there are several conflicts at airports that must be considered in order to avoid accidents. Most of them were taken into account in the project so it would as real as possible.

Runway conflicts can be defined as [8]:

- aircraft arriving to, or departing aircraft on, a closed runway;
- arriving or departing aircraft with traffic on the runway;
- arriving or departing aircraft with moving traffic to or on converging or intersecting runway;
- arriving or departing aircraft with opposite direction arrival to the runway;
- arriving or departing aircraft with traffic crossing the runway;
- arriving or departing with taxiing traffic approaching the runway;
- arriving aircraft exiting runway at speed with converging taxiway traffic;
- arriving aircraft with traffic in the sensitive area;
- aircraft exiting runway at unintended or non-approved locations; and
- unidentified traffic approaching the runway.

In what concerns taxiway conflicts, they can be defined as follows [8]:

- aircraft on a closed taxiway;
- aircraft approaching stationary traffic;
- aircraft overtaking same direction traffic;
- aircraft with opposite direction traffic;
- aircraft approaching taxiway intersections with converging traffic;
- aircraft taxiing with excessive speed;

- aircraft exiting the taxiway at unintended or non-approved locations;
- unauthorized traffic on the taxiways;
- unidentified traffic on the taxiways; and
- crossing of a lit stop bar.

Some examples of a conflicts at taxiways are illustrated in next figures:

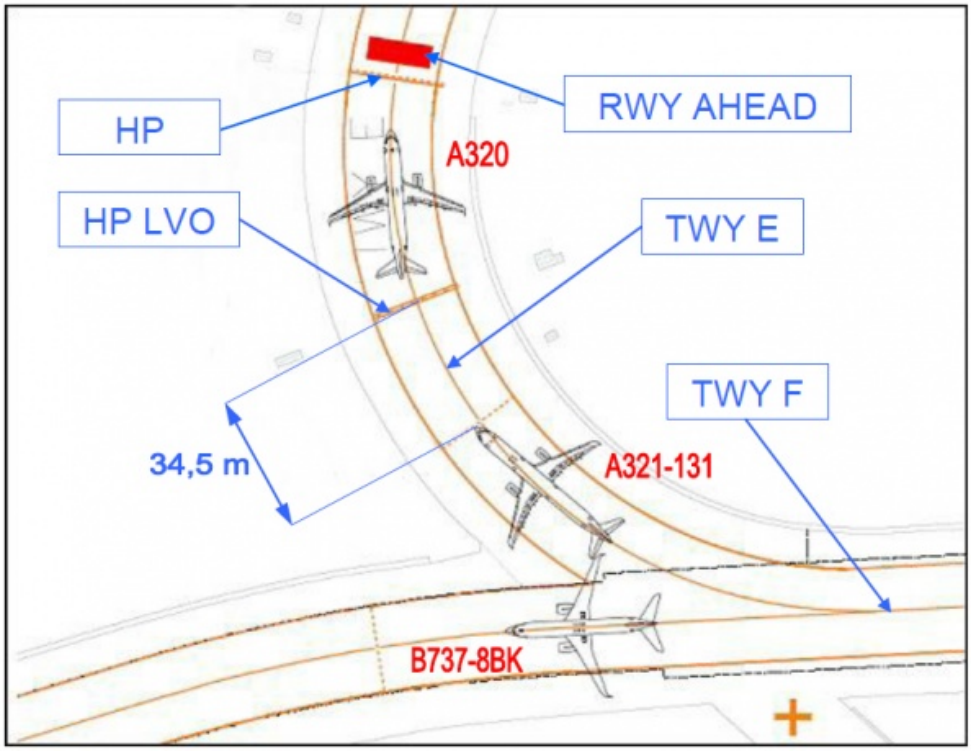


Figure 1: Taxiway Conflict

Figure 2: (From left to right) Aircraft approaching taxiway intersections with converging traffic; Aircraft overtaking same direction traffic and Aircraft with opposite direction traffic

The last one, apron/stand/gate conflicts are defined as [8]:

- aircraft movement with conflicting traffic;
- aircraft movement with conflicting stationary objects;
- aircraft exiting the apron/stand/gate area at unintended or non-approved locations; and
- unidentified traffic in the apron/stand/gate area.

In what concerns safety the A-SMGCS it is important to take in consideration the "Runway Incursion Detection Scenario" as shown in figure 3 or the "Longitudinal Spacing Parameters" in figure 4.

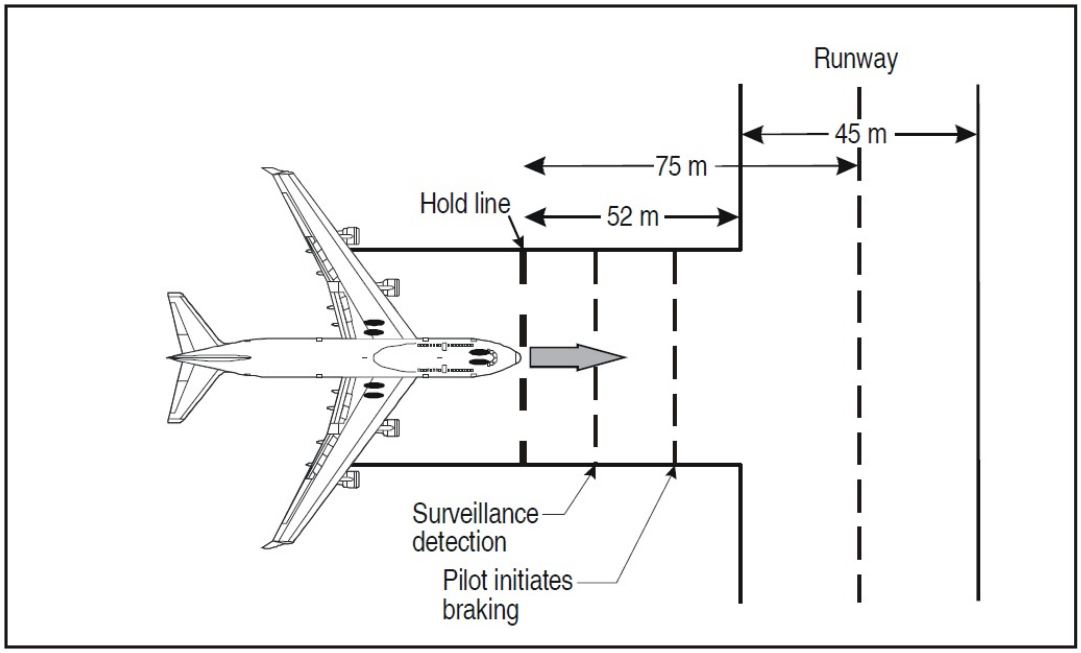


Figure 3: Runway Incursion Detection Scenario (source: A-SMGCS)

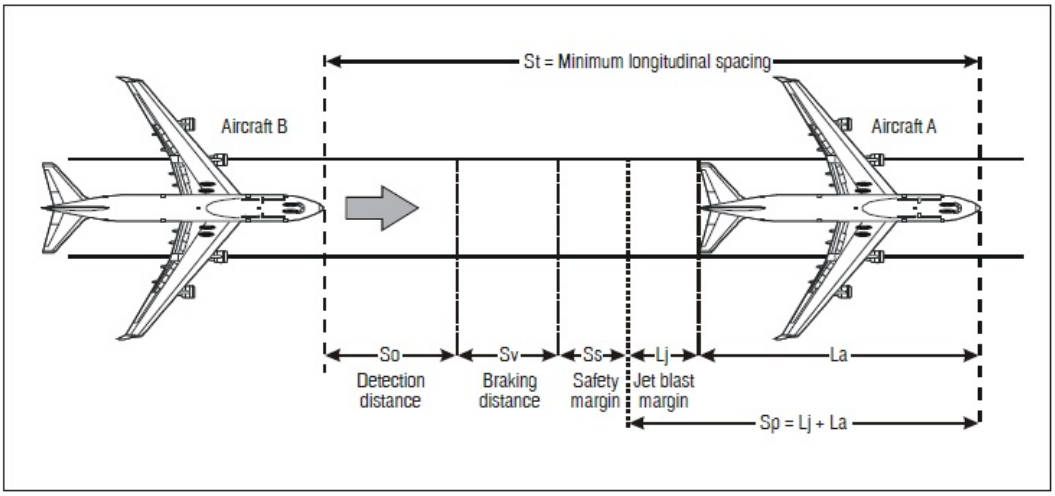


Figure 4: Longitudinal Spacing Parameters (source: A-SMGCS)

Appendix B

Here it will be presented the JAVA code implemented.

Constructive:

```
1  public class GreedyConstructive {
2    public Plan construct(Simulation simulation, int p, int r){
3
4      Plan plan = new Plan(simulation);
5
6      Map<Track, ArrayList<Task>> gantGreedy = new HashMap<>();
7
8      for(Job job: plan.getJobs()){
9          Plane plane = job.getPlane();
10           if(plane.equals(plan.getJobs().get(p).getPlane())){
11             job.setScheduledRoute(job.getAllScheduledRoute().get(r));
12             for(Task task: job.getScheduledRoute()){
13               Track track = task.getResource();
14               task.setPlane(plane);
15               if(gantGreedy.get(track) == null){
16                 gantGreedy.put(track, new ArrayList<Task>());
17               }
18
19               gantGreedy.get(track).add(task);
20             }
21
22           } else {
23             for(Task task: job.getScheduledRoute()){
24               Track track = task.getResource();
25               task.setPlane(plane);
26               if(gantGreedy.get(track) == null){
27                 gantGreedy.put(track, new ArrayList<Task>());
28               }
29
30               gantGreedy.get(track).add(task);
31             }
32
33           }
34     }
35
36     plan.setSequences(gantGreedy);
37
```

```
38      plan.validate();
39
40      return plan;
41    }
42  }
```

Validation:

```
1
2  public class Validator {
3
4    public Validator() {
5    }
6
7    public void validate(Plan plan) {
8
9      if (this.validateSyntax(plan)) {
10       this.evaluateTimes(plan);
11       this.cost(plan);
12       estimatedTaxiTime(plan);
13     } else {
14       log.error("Validation failed");
15     }
16   }
17
18   protected boolean validateSyntax(Plan plan) {
19     // validate precedences
20     HashSet<Task> taskToLabel = new HashSet<>();
21     HashSet<Task> labelled = new HashSet<>();
22     HashSet<Task> candidates = new HashSet<>();
23
24     // all tasks to label
25     for (Job job : plan.getJobs()) {
26       taskToLabel.addAll(job.getScheduledRoute());
27     }
28
29     // initiate the candidates for the first time
30     for (Task step : taskToLabel) {
31       Task predecessorInPlane = getPredecessorInPlane(step, plan);
32       Task predecessorInTrack = getPredecessorInTrack(step, plan);
33
34       if (predecessorInPlane == null && predecessorInTrack == null) {
35         candidates.add(step);
36       }
37     }
38
39     while (!candidates.isEmpty()) {
40       HashSet<Task> candidatesCopy = new HashSet<>(candidates);
41       for (Task step : candidatesCopy) {
42         Task predecessorInPlane = getPredecessorInPlane(step, plan);
43         Task predecessorInTrack = getPredecessorInTrack(step, plan);
44         boolean freePlane = (predecessorInPlane == null || labelled
```

```
45              .contains(predecessorInPlane));
46          boolean freeTrack = (predecessorInTrack == null || labelled
47              .contains(predecessorInTrack));
48
49          if (freePlane && freeTrack) {
50            candidates.remove(step);
51            labelled.add(step);
52            if (getSucessorInPlane(step, plan) != null) {
53              if (labelled.contains(getPredecessorInTrack(
54                  getSucessorInPlane(step, plan), plan))
55                  || getPredecessorInTrack(
56                      getSucessorInPlane(step, plan), plan) == null) {
57                candidates.add(getSucessorInPlane(step, plan));
58              }
59            }
60            if (getSucessorInTrack(step, plan) != null) {
61              candidates.add(getSucessorInTrack(step, plan));
62            }
63          }
64        }
65        if (candidatesCopy.equals(candidates)) {
66          return false;
67        }
68      }
69
70      return labelled.size() == taskToLabel.size();
71
72    }
73
74    protected Task getPredecessorInPlane(Task task, Plan plan) {
75
76      Task result = null;
77
78      for (Job job : plan.getJobs()) {
79
80        List<Task> route = job.getScheduledRoute();
81
82        for (Task step : route) {
83
84          if (step.equals(task)) {
85
86            int i = route.indexOf(step);
87
88            if (i == 0) {
89
90              result = null;
91              return result;
92
93            } else {
94
95              result = route.get(i - 1);
```

```
96              return result;
97            }
98          }
99        }
100     }
101
102     return result;
103   }
104
105   protected Task getPredecessorInTrack(Task task, Plan plan) {
106
107     Task result = null;
108
109     Track track = task.getResource();
110
111     Map<Track, ArrayList<Task>> sequences = plan.getSequences();
112
113     for (Task trackTask : sequences.get(track)) {
114       if (task.equals(trackTask)) {
115
116         int taskIndex = sequences.get(track).indexOf(task);
117
118         if (taskIndex < 1) {
119
120           result = null;
121           return result;
122         } else {
123           result = sequences.get(track).get(taskIndex - 1);
124           return result;
125         }
126       }
127     }
128
129     return result;
130   }
131
132   protected Task getSucessorInPlane(Task task, Plan plan) {
133
134     Task result = null;
135
136     List<Job> jobs = plan.getJobs();
137
138     for (Job job : jobs) {
139
140       List<Task> route = job.getScheduledRoute();
141
142       for (Task step : route) {
143
144         if (step.equals(task)) {
145
146           int i = route.indexOf(step);
```

```
147
148             int numberOfRoutes = route.size();
149
150             if (i > numberOfRoutes - 2) {
151
152               result = null;
153               return result;
154
155             } else {
156
157               result = job.getScheduledRoute().get(i + 1);
158               return result;
159             }
160           }
161         }
162       }
163
164       return result;
165     }
166
167     protected Task getSucessorInTrack(Task task, Plan plan) {
168
169       Task result = null;
170
171       Track track = task.getResource();
172
173       Map<Track, ArrayList<Task>> sequences = plan.getSequences();
174
175       for (Task trackTask : sequences.get(track)) {
176
177         if (task.equals(trackTask)) {
178
179           int taskIndex = sequences.get(track).indexOf(task);
180
181           if (taskIndex > sequences.get(track).size() - 2) {
182
183             result = null;
184             return null;
185
186           } else {
187
188             result = sequences.get(track).get(taskIndex + 1);
189             return result;
190           }
191         }
192       }
193
194       return result;
195     }
196
197     /*
```

```
198     * evaluate syntax-feasible plan and calculate start and end times for
            each
199     * task
200     */
201
202   protected void evaluateTimes(Plan plan) {
203
204     forwardPath(plan);
205     backwardPath(plan);
206
207   }
208
209   protected void forwardPath(Plan plan) {
210
211     HashSet<Task> taskToLabel = new HashSet<>();
212     HashSet<Task> timedTask = new HashSet<>();
213
214     // all tasks to label
215     for (Job job : plan.getJobs()) {
216       taskToLabel.addAll(job.getScheduledRoute());
217     }
218
219     for (Task step : taskToLabel) {
220
221       Task predecessorInPlane = getPredecessorInPlane(step, plan);
222       Task predecessorInTrack = getPredecessorInTrack(step, plan);
223
224       if (predecessorInPlane == null && predecessorInTrack == null) {
225
226         boolean equalResource = false;
227         for (Track runwayTrack : plan.getRunways()) {
228
229           if (step.getResource().equals(runwayTrack)) {
230
231             equalResource = true;
232             break;
233           }
234         }
235         if (equalResource) {
236           step.setStartTime(step.getPlane().getTimeAtRunway());
237           step.setEndTime(step.getStartTime()
238               + step.getResource().getTimeNeed());
239           timedTask.add(step);
240         } else {
241           step.setStartTime(0L);
242           step.setEndTime(step.getResource().getTimeNeed());
243           timedTask.add(step);
244         }
245       }
246     }
247
```

```
248     for (Task next : timedTask) {
249
250       taskToLabel.remove(next);
251     }
252
253     while (!taskToLabel.isEmpty()) {
254
255       HashSet<Task> taskToLabelCopy = new HashSet<Task>(taskToLabel);
256
257       for (Task task : taskToLabelCopy) {
258
259         Long startTime = (long) 0;
260
261         Task predecessorInPlane = getPredecessorInPlane(task, plan);
262         Task predecessorInTrack = getPredecessorInTrack(task, plan);
263
264         boolean freePlane = predecessorInPlane == null
265             || timedTask.contains(predecessorInPlane);
266         boolean freeTrack = predecessorInTrack == null
267             || timedTask.contains(predecessorInTrack);
268
269         if (freePlane && freeTrack) {
270           timedTask.add(task);
271
272           boolean equalResource = false;
273
274           for (Track runwayTrack : plan.getRunways()) {
275
276             if (task.getResource().equals(runwayTrack)) {
277
278               equalResource = true;
279               break;
280             }
281           }
282           if (equalResource) {
283             task.setStartTime(task.getPlane().getTimeAtRunway());
284             task.setEndTime(task.getStartTime()
285                 + task.getResource().getTimeNeed());
286             taskToLabel.remove(task);
287
288           } else {
289
290             if (predecessorInTrack == null) {
291               startTime = predecessorInPlane.getEndTime();
292             }
293             if (predecessorInPlane == null) {
294               startTime = predecessorInTrack.getEndTime();
295             }
296
297             if (predecessorInTrack != null
298                 && predecessorInPlane != null) {
```

```
299               startTime = Math.max(
300                   predecessorInPlane.getEndTime(),
301                   predecessorInTrack.getEndTime());
302             }
303             if (predecessorInTrack != null) {
304
305               Map<Plane, Map<Plane, Integer>> separation = plan
306                   .getSeparationTime();
307
308               Plane predecessorPlane = predecessorInTrack
309                   .getPlane();
310               Plane plane = task.getPlane();
311
312               if (startTime + task.getResource().getTimeNeed() < startTime
313                   + separation.get(predecessorPlane).get(
314                       plane)) {
315                 task.setStartTime(startTime
316                     + separation.get(
317                         predecessorInTrack.getPlane())
318                         .get(task.getPlane()));
319               } else {
320                 task.setStartTime(startTime);
321               }
322             } else {
323               task.setStartTime(startTime);
324             }
325             task.setEndTime(task.getStartTime()
326                 + task.getResource().getTimeNeed());
327             taskToLabel.remove(task);
328           }
329         }
330       }
331
332     }
333
334   }
335
336   protected void backwardPath(Plan plan) {
337
338     List<Task> finalTimedTask = new ArrayList<>();
339     List<Task> finalTaskToLabel = new ArrayList<>();
340
341     for (Job job : plan.getJobs()) {
342       finalTaskToLabel.addAll(job.getScheduledRoute());
343     }
344
345     for (Task step : finalTaskToLabel) {
346       Task sucessorInPlane = getSucessorInPlane(step, plan);
347       Task sucessorInTrack = getSucessorInTrack(step, plan);
348       if (sucessorInPlane == null && sucessorInTrack == null) {
349         finalTimedTask.add(step);
```

```
350         }
351       }
352
353       for (Task next : finalTimedTask) {
354         finalTaskToLabel.remove(next);
355       }
356
357       while (!finalTaskToLabel.isEmpty()) {
358
359         HashSet<Task> finalTaskToLabelCopy = new HashSet<Task>(
360             finalTaskToLabel);
361
362         for (Task task : finalTaskToLabelCopy) {
363
364           Task sucessorInPlane = getSucessorInPlane(task, plan);
365           Task sucessorInTrack = getSucessorInTrack(task, plan);
366
367           boolean freePlane = sucessorInPlane == null
368               || finalTimedTask.contains(sucessorInPlane);
369           boolean freeTrack = sucessorInTrack == null
370               || finalTimedTask.contains(sucessorInTrack);
371
372           if (freePlane && freeTrack) {
373
374             finalTimedTask.add(task);
375             Long finalStartTime = task.getStartTime();
376             Long finalEndTime = task.getEndTime();
377             boolean equalResource = false;
378
379             for (Track runwayTrack : plan.getRunways()) {
380
381               if (task.getResource().equals(runwayTrack)) {
382                 equalResource = true;
383                 break;
384               }
385             }
386             if (!equalResource) {
387
388               if (sucessorInPlane != null) {
389
390                 Long folga = sucessorInPlane.getStartTime()
391                     - task.getEndTime();
392                 task.setStartTime(finalStartTime + folga);
393                 //task.setEndTime(finalEndTime + folga);
394                 task.setEndTime(task.getStartTime()+task.getResource().
                        getTimeNeed());
395
396               }
397
398             }
399             finalTaskToLabel.remove(task);
```

```
400         }
401       }
402     }
403
404   }
405
406   protected void cost(Plan plan) {
407
408     Long result = 0L;
409
410     List<Job> jobs = plan.getJobs();
411
412     for (Job job : jobs) {
413
414       List<Task> route = job.getScheduledRoute();
415
416       int index = route.size();
417
418       job.getPlane()
419           .setTimeSpendTaxiing(
420               ((job.getScheduledRoute().get(index - 1)
421                   .getEndTime() - job.getScheduledRoute()
422                   .get(0).getStartTime())));
423
424       result = result
425           + job.getScheduledRoute().get(index - 1).getEndTime()
426           - job.getScheduledRoute().get(0).getStartTime();
427     }
428
429     plan.setQuantitativeSolution(result);
430
431   }
432
433   protected void estimatedTaxiTime (Plan plan){
434
435     List<Job> jobs = plan.getJobs();
436
437     for(Job job: jobs){
438
439       Long result = 0L;
440
441       List<Task> planeRoute = job.getScheduledRoute();
442
443       for(Task task: planeRoute){
444         Long taskTime = task.getResource().getTimeNeed();
445         result = result + taskTime;
446       }
447
448       job.getPlane().setEstimatedTaxiTime((result));
449     }
450
```

```
451
452   }
453
454 }
```

Iterated Local Search code:

```
1   public Plan randPlan(Plan plan, Simulation simulation){
2
3     Random rand = new Random();
4
5     int p = 0;
6
7     int numberOfJobs = plan.getJobs().size();
8
9     while(p < numberOfJobs){
10
11      int r = rand.nextInt(plan.getJobs().get(p).getAllScheduledRoute().size
            ());
12      GreedyConstructive gredy = new GreedyConstructive();
13      plan = gredy.construct(simulation, p, r);
14      ++p;
15    }
16
17    return plan;
18  }
```

First Ascent code:

```
1 public void improve(Plan plan, Simulation simulation){
2
3     int p = 0;
4
5     Plan alternativePlan = new Plan(simulation);
6
7     while( p < plan.getPlanes().size()){
8
9       int r = 0;
10
11      int numberOfRoutes = plan.getJobs().get(p).getAllScheduledRoute().size
            ();
12
13      while(r < numberOfRoutes){
14
15        ArrayList<Task> originalScheduledRoute = plan.getJobs().get(p).
              getScheduledRoute();
16        GreedyConstructive gredy = new GreedyConstructive();
17        alternativePlan = gredy.construct(simulation, p, r);
18
19          if(plan.getQuantitativeSolution() <= alternativePlan.
                getQuantitativeSolution()){
20            plan.getJobs().get(p).setScheduledRoute(originalScheduledRoute);
21            ++r;
```

```
22            } else {
23              plan = alternativePlan;
24              break;
25            }
26
27        }
28        ++p;
29      }
30
31
32    }
```

Steepest Ascent code:

```
1  public class SteepestAscent {
2
3    /*
4     * Searches every route in every plane, one at a time, for a better route.
5     */
6
7    public void improve(Plan plan, Simulation simulation, Timer t) {
8
9      int p = 0;
10
11     Plan alternativePlan = new Plan(simulation);
12     Plan originalPlan = plan;
13
14     int numberOfPlanes = plan.getPlanes().size();
15
16     while (p < numberOfPlanes){
17       int r = 0;
18
19       int numberOfRoutes = plan.getJobs().get(p).getAllScheduledRoute()
20           .size();
21
22       while (r < numberOfRoutes) {
23
24         GreedyConstructive gredy = new GreedyConstructive();
25         alternativePlan = gredy.construct(simulation, p, r);
26         if (originalPlan.getQuantitativeSolution() <= alternativePlan
27             .getQuantitativeSolution()) {
28
29           ++r;
30
31         } else {
32           originalPlan = alternativePlan;
33           ++r;
34         }
35       }
36       ++p;
37     }
38
```

```
39
40    }
41
42 }
```

Tabu Search code:

```
1  public class TabuSearch {
2
3    public void improveOne(Plan plan, Simulation simulation, Timer t) {
4
5      Plan alternativePlan = new Plan(simulation);
6      HashSet<Plan> tenure = new HashSet<Plan>();
7
8      while (!t.isExpired()) {
9        boolean newPlan = true;
10
11
12         Plan initialPlan = plan;
13
14         List<Job> jobs = plan.getJobs();
15
16         for (Job job : jobs) {
17
18           Long taxiTimeNeed = 0L;
19
20           List<Task> scheduledRoute = job.getScheduledRoute();
21
22           int index = scheduledRoute.size();
23
24           Long taxiTimeUsed = scheduledRoute.get(index - 1)
25               .getEndTime()
26               - scheduledRoute.get(0).getStartTime();
27
28           for (Task task : scheduledRoute) {
29             taxiTimeNeed = taxiTimeNeed
30                 + task.getResource().getTimeNeed();
31           }
32
33           if (taxiTimeUsed > taxiTimeNeed) {
34
35             for (ArrayList<Task> tasks : job.getAllScheduledRoute()) {
36
37               Long otherRouteTimeNeed = 0L;
38
39               if (!tasks.equals(scheduledRoute)) {
40
41                 for (Task next : tasks) {
42                   otherRouteTimeNeed = otherRouteTimeNeed
43                       + next.getResource().getTimeNeed();
44                 }
45
```

```
46                if (taxiTimeUsed > otherRouteTimeNeed) {
47
48                  GreedyConstructive gredy = new GreedyConstructive();
49                  int p = jobs.indexOf(job);
50                  int r = job.getAllScheduledRoute().indexOf(
51                      tasks);
52                  alternativePlan = gredy.construct(
53                      simulation, p, r);
54
55                  if (plan.getQuantitativeSolution() <= alternativePlan
56                      .getQuantitativeSolution()) {
57                    tenure.add(alternativePlan);
58                  } else {
59                    plan = alternativePlan;
60                    break;
61                  }
62                }
63
64              }
65
66            }
67          }
68        }
69
70        newPlan = plan.equals(initialPlan);
71
72        if (newPlan) {
73          MildestDescent localSearchMildestDescent = new MildestDescent();
74
75          localSearchMildestDescent.improve(plan, simulation, t);
76        }
77      }
78
79    long stop = 1 * 5 * 1000; //5seg
80    Timer s = new Timer(stop);
81
82    improveTwo(plan,s);
83
84
85    log.debug(plan);
86
87  }
88
89  public void improveTwo(Plan plan, Timer t) {
90
91    Plan bestPlan = plan;
92
93    HashSet<Plan> tenure = new HashSet<Plan>();
94
95    while (!t.isExpired()) {
96
```

```
97        Map<Track, ArrayList<Task>> sequences = plan.getSequences();
98
99        List<Track> runways = plan.getRunways();
100
101       for (Track track : sequences.keySet()) {
102
103         for (Track runwayTrack : runways) {
104           if (!runwayTrack.equals(track)) {
105
106             List<Task> trackSeq = sequences.get(track);
107
108             for (int i = 1; i < trackSeq.size() - 2; ++i) {
109
110               if (trackSeq.get(i).getPlane().getPlaneTask() == "A"
111                   && trackSeq.get(i + 1).getPlane()
112                       .getPlaneTask() == "D") {
113                 Task temp = trackSeq.get(i);
114                 trackSeq.set(i, trackSeq.get(i + 1));
115                 trackSeq.set(i + 1, temp);
116                 if (plan.getQuantitativeSolution() > bestPlan
117                     .getQuantitativeSolution()) {
118                   tenure.add(plan);
119                 } else {
120                   tenure.add(bestPlan);
121                   bestPlan = plan;
122                 }
123
124               }
125             }
126           }
127         }
128       }
129
130       plan.validate();
131
132
133     }
134
135   }
136
137   public void improveThree(Plan plan, Timer t, Simulation simulation) {
138
139     List<Plan> tenure = new ArrayList<Plan>();
140
141     int j = 0;
142
143     while (!t.isExpired()) {
144
145
146       Plan alternativePlan = plan;
147       List<Plane> alternativeSequence = alternativePlan
```

```
148             .getRunwayUserSequence();
149         int sequenceSize = alternativeSequence.size();
150 
151         boolean equalTimeAtRunway = false;
152 
153         for (int i = j; i < sequenceSize - 1; ++i) {
154 
155           Plane first = alternativeSequence.get(i);
156           Plane second = alternativeSequence.get(i + 1);
157 
158           equalTimeAtRunway = first.getTimeAtRunway().equals(
159               second.getTimeAtRunway());
160 
161           if (equalTimeAtRunway) {
162 
163             alternativePlan.getRunwayUserSequence().set(i, second);
164             alternativePlan.getRunwayUserSequence().set(i + 1, first);
165             break;
166           }
167         }
168 
169         GreedyConstructive greedy = new GreedyConstructive();
170         alternativePlan = greedy.construct(simulation, 0, 0);
171 
172         if (plan.getQuantitativeSolution() <= alternativePlan
173             .getQuantitativeSolution()) {
174           tenure.add(alternativePlan);
175         } else {
176           tenure.add(plan);
177           plan = alternativePlan;
178         }
179         ++j;
180 
181       }
182 
183 
184     }
185 
186 }
```

Mildest Descent code:

```
1  public class MildestDescent {
2 
3    public void improve(Plan plan, Simulation simulation, Timer t){
4 
5      List <Plan> mildestPlans = new ArrayList<Plan>();
6 
7      Plan originalPlan = plan;
8 
9      int p = 0;
10 
```

```
11      Plan alternativePlan = new Plan(simulation);
12      Plan mildestPlan = alternativePlan;
13
14      mildestPlans.add(originalPlan);
15
16      int numberOfPlanes = plan.getPlanes().size();
17
18      while( p < numberOfPlanes){
19
20        int r = 0;
21
22        int numberOfRoutes = plan.getJobs().get(p).getAllScheduledRoute().size
              ();
23
24        while(r < numberOfRoutes){
25
26          ArrayList<Task> originalScheduledRoute = plan.getJobs().get(p).
                getScheduledRoute();
27          GreedyConstructive gredy = new GreedyConstructive();
28          alternativePlan = gredy.construct(simulation, p, r);
29          mildestPlans.add(alternativePlan);
30          if(plan.getQuantitativeSolution() <= alternativePlan.
                getQuantitativeSolution()){
31
32            plan.getJobs().get(p).setScheduledRoute(originalScheduledRoute);
33
34            if(r == 0 && p ==0){
35              mildestPlan = alternativePlan;
36            }
37
38            if(mildestPlan.getQuantitativeSolution() > alternativePlan.
                  getQuantitativeSolution()){
39               mildestPlan = alternativePlan;
40            }
41
42            ++r;
43
44          } else {
45            mildestPlan = plan;
46            plan = alternativePlan;
47            ++r;
48          }
49        }
50        ++p;
51      }
52
53      Collections.sort(mildestPlans, new PlanComparator());
54
55      plan = mildestPlans.get(1);
56
57
```

```
58      }
59
60
61  }
```

Appendix C

The next table presents the difference of taxi time between the real data and what is predicted by TS. This comparison is made for the time window of 8 AM.

Results		
Airplane	**Real Data**	**TS**
Airplane 1	4min	8min
Airplane 2	9min	7min
Airplane 3	4min	8min
Airplane 4	4min	7min
Airplane 5	8min	6min
Airplane 6	6min	7min
Airplane 7	7min	7min
Airplane 8	5min	8min
Airplane 9	12min	5min
Airplane 10	3min	6min
Airplane 11	9min	6min
Airplane 12	5min	7min
Airplane 13	16min	7min
Airplane 14	5min	8min
Airplane 15	13min	7min
Airplane 16	5min	7min
Airplane 17	10min	7min
Airplane 18	4min	7min
Airplane 19	5min	6min
Airplane 20	3min	6min
Airplane 21	10min	7min
Airplane 22	6min	7min
Airplane 23	5min	10min
Airplane 24	10min	6min
Airplane 25	11min	7min
Airplane 26	3min	6min
Airplane 27	10min	7min
Airplane 28	7min	6min
Airplane 29	7min	7min
Airplane 30	3min	8min
Airplane 31	10min	8min
Airplane 32	4min	7min
Airplane 33	8min	5min
Airplane 34	15min	7min
Airplane 35	10min	8min
Airplane 36	15min	8min

Table 1: Comparative Table - TS versus Real Data